Programmatic file downloads in the browser

May 14, 2019 · 12 min read

Blobs and object URLs exposed

File downloading is a core aspect of surfing the internet. Tons of files get downloaded from the internet every day ranging from *binary files* (like applications, images, videos, and audios) to files in plain text.

Fetching files from the server

Traditionally, the file to be downloaded is first requested from a *server* through a *client*— such as a user's web browser. The server then returns a response containing the *content* of the file and some instructional *headers* specifying how the client should download the file.

Schematic of Client-Server communication in fetching a file via HTTP

In this diagram, the green line shows the flow of the request from the client to the server over HTTP. The orange line shows the flow of the response from the server back to the client.

Though the diagram indicates the communication flow, it does not explicitly show what the request from the client looks like or what the response from the server looks like.

Here is what the response from the server could possibly look like:

Sample HTTP Response for a GIF image — the asterisks(*) represent the binary content of the image

In this response, the server simply serves the raw content of the resource (*represented with the asterisks—* `*`) which will be received by the client.

The response also contains some headers that give the client some information about the nature of the content it receives—in this example response, the **Content-Type** and **Content-Length** headers provide that information.

When the client (web browser in this case) receives this HTTP response, it simply displays or renders the GIF image—which is not the desired behavior. **The desired behavior is that the image should be downloaded not displayed.**

Enforcing file download

To inform the client that the content of the resource is not meant to be displayed, the server must include an additional header in the response. The **Content-Disposition** header is the right header for specifying this kind of information.

The `Content-Disposition` header was originally intended for mail user-agents—since emails are multipart documents that may contain several file attachments. However, it can be interpreted by several HTTP clients including web browsers. This header provides information on the **disposition type** and **disposition parameters**.

The **disposition type** is usually one of the following:

1. **inline**—The body part is intended to be displayed automatically when the message content is displayed
2. **attachment**—The body part is separate from the main content of the message and should not be displayed automatically except when prompted by the user

The **disposition parameters** are additional parameters that specify information about the body part or file such as filename, creation date, modification date, read date, size, etc.

Here is what the HTTP response for the GIF image should look like to enforce file download:

Sample HTTP Response for downloading a GIF image — the asterisks(*) represent the binary content of the image

Now the server enforces a download of the GIF image. Most HTTP clients will prompt the user to download the resource content when they receive a response from a server like the one above.

Click to download in the browser

Let's say you have the URL to a downloadable resource. When you try accessing that URL on your web browser, it prompts you to download the resource file — whatever the file is.

The scenario described above is not feasible in web applications. For web applications, the desired behavior will be — **downloading a file in response to a user interaction**. For example, *click to save a photo* or *download a report*.

Achieving such a behavior in the browser is possible with HTML **anchor elements** (`<a></a>`). Anchor elements are useful for adding hyperlinks to other resources and documents from an HTML document. The URL of the linked resource is specified in the `href` attribute of the anchor element.

Here is a conventional HTML anchor element linking to a PDF document:

A basic HTML anchor element (<a></a>)

The download attribute

In HTML 5, a new `download` attribute was added to the anchor element. The `download` attribute is used to inform the browser to download the URL instead of navigating to it — hence a prompt shows up, requesting that the user saves the file.

The `download` attribute can be given a valid filename as its value. However, the user can still modify the filename in the save prompt that pops-up.

There are a **few noteworthy facts** about the behavior of the **`download`** attribute:

1. In compliance with the *same-origin policy*, this attribute only works for same-origin URLs. Hence, it cannot be used to download resources served from a different origin
2. Besides HTTP(s) URLs, it also supports `blob:` and `data:` URLs — which makes it very useful for downloading content generated programmatically with JavaScript
3. For URLs with a HTTP `Content-Disposition` header that specifies a filename — the header filename has a higher priority than the value of the `download` attribute

Here is the updated HTML anchor element for downloading the PDF document:

HTML anchor element (<a></a>) for resource download

Programmatic content generation

With the advent of HTML5 and new Web APIs, it has become possible to do a lot of complex stuff in the browser using JavaScript without ever having to communicate with a server.

There are now Web APIs that can be used to programmatically:

- draw and manipulate images or video frames on a canvas — **Canvas API**
- read the contents and properties of files or even generate new data for files — **File API**
- generate object URLs for binary data — **URL API**

to mention only a few.

In this section, we will examine how we can programmatically generate content using Web APIs on the browser.

Let's consider two common examples.

Example 1 — CSV generation from JSON array

In this example, we will use the **Fetch API** to asynchronously fetch JSON data from a web service and transform the data to form a string of *comma-separated-values* that can be written to a CSV file. Here is a breakdown of what we are about to do:

- fetch an array collection of JSON objects from an API
- extract selected fields from each item in the array
- reformat the extracted data as CSV

Here is what the CSV generation script could look like:

```
function squareImages({ width = 1, height = width } = {}) {
  return width / height === 1;
}

function collectionToCSV(keys = []) {
  return (collection = []) => {
    const headers = keys.map(key => `"${key}"`).join(',');
    const extractKeyValues = record => keys.map(key =>
`"${record[key]}"`).join(',');

    return collection.reduce((csv, record) => {
      return (`${csv}\n${extractKeyValues(record)}`).trim();
    }, headers);
  }
}

const exportFields = [ 'id', 'author', 'filename', 'format', 'width',
'height' ];

fetch('https://picsum.photos/list')
```

Here we are fetching a collection of photos from the Picsum Photos API using the global `fetch()` function provided by the **Fetch API**, filtering the collection and converting the collection array to a CSV string. The code snippet simply logs the resulting CSV string to the console.

First, we define a `squareImages` *filter function* for filtering images in the collection with equal width and height.

Next, we define a `collectionToCSV` *higher-order function* which takes an array of keys and returns a function that takes an array collection of objects and converts it to a CSV string extracting only the specified keys from each object.

Finally, we specify the fields we want to extract from each photo object in the collection in the `exportFields` array.

Here is what the output could look like on the console:

Example 2 — Image pixel manipulation using the Canvas API

In this example, we will use the **Canvas API** to manipulate the pixels of an image, making it appear grayscale. Here is a breakdown of what we are about to do:

- set the canvas dimensions based on the image
- draw the image on a canvas
- extract and transform the image pixels on the canvas to grayscale
- redraw the grayscale pixels on the canvas

Let’s say we have a markup that looks pretty much like this:

<div id=”image-wrapper”>
<canvas></canvas>
<img src=”https://example.com/imgs/random.jpg&#8221; alt=”Random Image”>
</div>

Here is what the image manipulation script could look like:

```
const wrapper = document.getElementById('image-wrapper');
const img = wrapper.querySelector('img');
const canvas = wrapper.querySelector('canvas');

img.addEventListener('load', () => {
  canvas.width = img.width;
  canvas.height = img.height;

  const ctx = canvas.getContext('2d');

  ctx.drawImage(img, 0, 0, width, height);

  const imageData = ctx.getImageData(0, 0, width, height);
  const data = imageData.data;

  for (let i = 0, len = data.length; i < len; i += 4) {
    const avg = (data[i] + data[i + 1] + data[i + 2]) / 3;

    data[i]     = avg; // red
    data[i + 1] = avg; // green
```

Here is a comparison between an actual image and the corresponding grayscale canvas image.

Blobs and object URLs

Before we proceed to learn how we can download content generated programmatically in the browser, let's take some time to look at a special kind of object interface called `Blob`, which is already been implemented by most of the major web browsers. You can learn about Blobs here.

Blobs are objects that are used to represent raw immutable data. Blob objects store information about the type and size of data they contain, making them very useful for storing and working file contents on the browser. In fact, the `File` object is a special extension of the `Blob` interface.

Obtaining blobs

Blob objects can be obtained from a couple of sources:

- Created from non-blob data using the `Blob` constructor
- Sliced from an already existing blob object using the `Blob.slice()` method
- Generated from Fetch API responses or other Web API interfaces

Here are some code samples for the aforementioned blob object sources:

```
const data = {
  name: 'Glad Chinda',
  country: 'Nigeria',
  role: 'Web Developer'
};

// SOURCE 1:
// Creating a blob object from non-blob data using the Blob constructor
const blob = new Blob([ JSON.stringify(data) ], { type: 'application/json'
});
```

```
const paragraphs = [
  'First paragraph.\r\n',
  'Second paragraph.\r\n',
  'Third paragraph.'
];
const blob = new Blob(paragraphs, { type: 'text/plain' });

// SOURCE 2:
// Creating a new blob by slicing part of an already existing blob object
const slicedBlob = blob.slice(0, 100);
```

```
// SOURCE 3:
// Generating a blob object from a Web API like the Fetch API
// Notice that Response.blob() returns a promise that is fulfilled with a
blob object
fetch('https://picsum.photos/id/6/100')
  .then(response => response.blob())
  .then(blob => {
    // use blob here...
  });
```

Reading blob content

It is one thing to obtain a blob object and another thing altogether to work with it. One thing you want to be able to do is to read the content of the blob. That sounds like a good opportunity to use a **`FileReader`** object. You can learn about `FileReader` objects here.

A `FileReader` object provides some very helpful methods for asynchronously reading the content of blob objects or files in different ways. The `FileReader` interface has pretty good browser support and supports reading blob data as follows *(as at the time of this writing)*:

- **as text** — `FileReader.readAsText()`
- **as binary string** — `FileReader.readAsBinaryString()`
- **as base64 data URL** — `FileReader.readAsDataURL()`

- **as array buffer** — `FileReader.readAsArrayBuffer()`

Building on the Fetch API example we had before, we can use a `FileReader` object to read the blob as follows:

```
fetch('https://picsum.photos/id/6/240')
  .then(response => response.blob())
  .then(blob => {
    // Create a new FileReader innstance
    const reader = new FileReader;

    // Add a listener to handle successful reading of the blob
    reader.addEventListener('load', () => {
      const image = new Image;

      // Set the src attribute of the image to be the resulting data URL
      // obtained after reading the content of the blob
      image.src = reader.result;

      document.body.appendChild(image);
    });

    // Start reading the content of the blob
    // The result should be a base64 data URL
    reader.readAsDataURL(blob);
```

Object URLs

The `URL` interface allows for creating special kinds of URLs called *object URLs*, which are used for representing blob objects or files in a very concise format. Here is what a typical object URL looks like:

```
blob:https://cdpn.io/de82a84f-35e8-499d-88c7-1a4ed64402eb
```

Creating and releasing object URLs

The `URL.createObjectURL()` static method makes it possible to create an object URL that represents a blob object or file. It takes a blob object as its argument and returns a `DOMString` which is the URL representing the passed blob object. Here is what it looks like:

```
const url = URL.createObjectURL(blob);
```

It is important to note that, this method will always return a new object URL each time it is called, even if it is called with the same blob object.

Whenever an object URL is created, it stays around for the lifetime of the document on which it was created. Usually, the browser will release all object URLs when the document is being unloaded. However, it is important that you release object URLs whenever they are no longer needed in order to improve performance and minimize memory usage.

The `URL.revokeObjectURL()` static method can be used to release an object URL. It takes the object URL to be released as its argument. Here is what it looks like:

```
const url = URL.createObjectURL(blob);
URL.revokeObjectURL(url);
```

Using object URLs

Object URLs can be used wherever a URL can be supplied programmatically. For example:

- they can be used to load files that can be displayed or embedded in the browser such as images, videos, audios, PDFs, etc — for example, by setting the `src` property of an `Image` element
- they can be used as the `href` attribute of an `<a></a>` element, making it possible to download content that was extracted or generated programmatically

Downloading generated content

So far, we have looked at how we can download files that are served from a server and sent to the client over HTTP — which is pretty much the **traditional flow**. We have also seen how we can programmatically extract or generate content in the browser using Web APIs.

In this section, we will examine how we can download programmatically generate content in the browser, leveraging all we have learned from the beginning of the article and what we already know about blobs and object URLs.

Creating the download link

First, let's say we have a **blob object** by some means. We want to create a helper function that allows us to create a download link (`<a></a>` element) that can be clicked in order to download the content of the blob, just like a regular file download.

The logic of our helper function can be broken down as follows:

- Create an object URL for the blob object
- Create an *anchor* element (`<a></a>`)
- Set the `href` attribute of the anchor element to the created object URL
- Set the `download` attribute to the filename of the file to be downloaded. This forces the anchor element to trigger a file download when it is clicked
- If the link is for a one-off download, release the object URL after the anchor element has been clicked

Here is what an implementation of this helper function will look like:

```
function downloadBlob(blob, filename) {
  // Create an object URL for the blob object
  const url = URL.createObjectURL(blob);

  // Create a new anchor element
  const a = document.createElement('a');

  // Set the href and download attributes for the anchor element
  // You can optionally set other attributes like `title`, etc
  // Especially, if the anchor element will be attached to the DOM
  a.href = url;
  a.download = filename || 'download';

  // Click handler that releases the object URL after the element has been
clicked
  // This is required for one-off downloads of the blob content
  const clickHandler = () => {
    setTimeout(() => {
      URL.revokeObjectURL(url);
      this.removeEventListener('click', clickHandler);
```

That was a pretty straightforward implementation of the download link helper function. Notice that the helper triggers a *one-off automatic* download of the blob content whenever it is called.

Also notice that the helper function takes a filename as its second argument, which is very useful for setting the default filename for the downloaded file.

The helper function returns a reference to the created anchor element (`<a></a>`), which is very useful if you want to attach it to the DOM or use it in some other way.

Here is a simple example:

```
// Blob object for the content to be download
const blob = new Blob(
  [ /* CSV string content here */ ],
  { type: 'text/csv' }
);

// Create a download link for the blob content
const downloadLink = downloadBlob(blob, 'records.csv');

// Set the title and classnames of the link
downloadLink.title = 'Export Records as CSV';
downloadLink.classList.add('btn-link', 'download-link');

// Set the text content of the download link
downloadLink.textContent = 'Export Records';

// Attach the link to the DOM
document.body.appendChild(downloadLink);
```

Revisiting the examples

Now that we have our download helper function in place, we can revisit our previous examples and modify them to trigger a download for the generated content. Here we go.

1. CSV generation from JSON array

We will update the final promise `.then` handler to create a download link for the generated CSV string and automatically click it to trigger a file download using the `downloadBlob` helper function we created in the previous section.

Here is what the modification should look like:

```
fetch('https://picsum.photos/list')
  .then(response => response.json())
  .then(data => data.filter(squareImages))
  .then(collectionToCSV(exportFields))
  .then(csv => {
    const blob = new Blob([csv], { type: 'text/csv' });
    downloadBlob(blob, 'photos.csv');
  })
  .catch(console.error);
```

Here we have updated the final promise .then handler as follows:

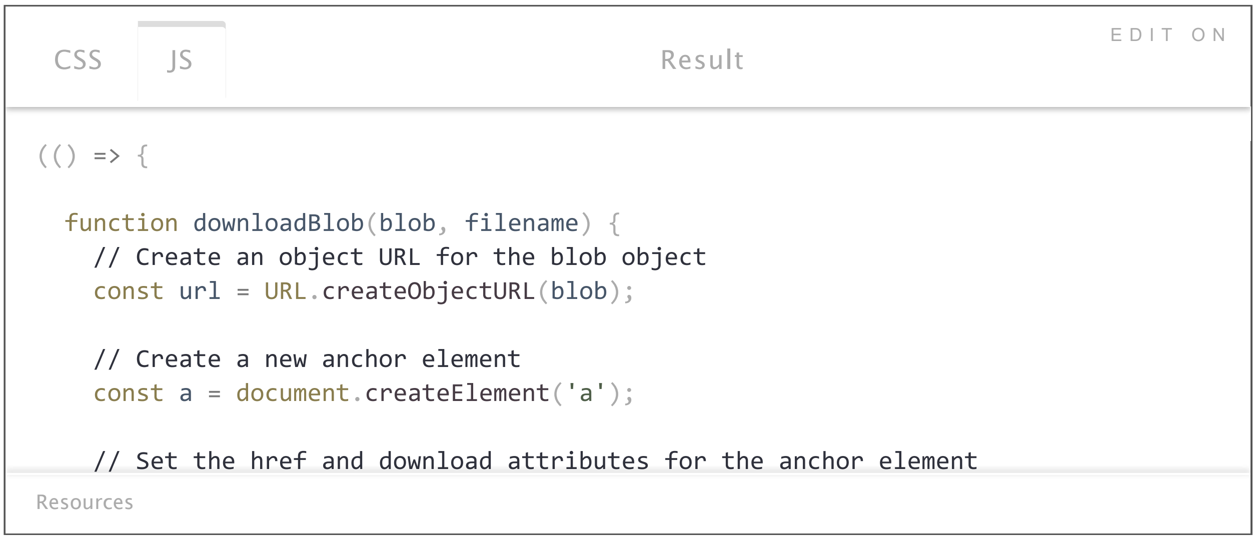
- create a new blob object for the CSV string, also setting the correct type using:

```
{ type: 'text/csv' }
```

- call the `downloadBlob` helper function to trigger an automatic download for the CSV file, specifying the default filename as `“photos.csv”`
- move the promise rejection handler to a separate `.catch()` block:

```
.catch(console.error)
```

Here is a working and more advanced example of this application on **Codepen**:



```
(() => {

  function downloadBlob(blob, filename) {
    // Create an object URL for the blob object
    const url = URL.createObjectURL(blob);

    // Create a new anchor element
    const a = document.createElement('a');

    // Set the href and download attributes for the anchor element
```

2. Image pixel manipulation

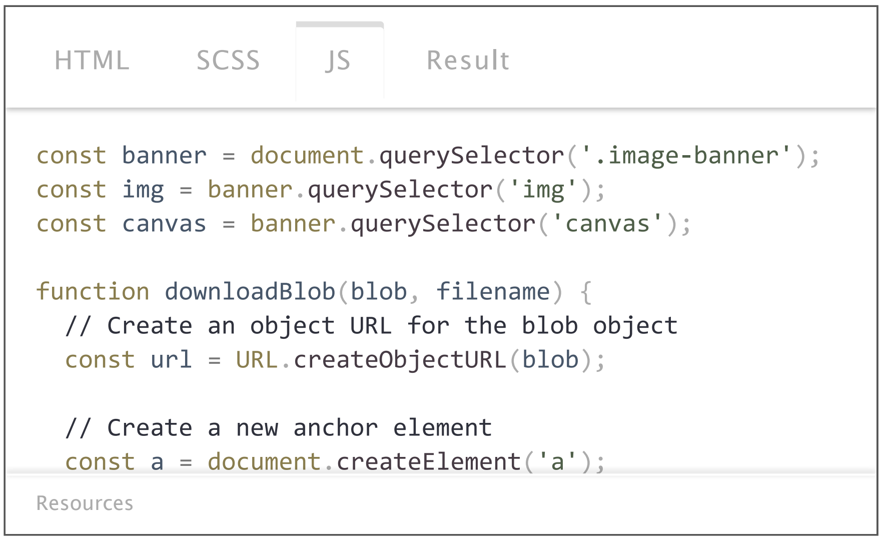
We will add some code to the end of the `load` event listener of the `img` object, to allow us:

- create a blob object for the grayscale image in the `canvas` using the `Canvas.toBlob()` method
- and then create a download link for the blob object using our `downloadBlob` helper function from before
- and finally, append the download link to the DOM

Here is what the update should look like:

```
img.addEventListener('load', () => {

  /* ... some code have been truncated here ... */

  ctx.putImageData(imageData, 0, 0);

  // Canvas.toBlob() creates a blob object representing the image
contained in the canvas
  // It takes a callback function as its argument whose first parameter is
the
  canvas.toBlob(blob => {
    // Create a download link for the blob object
    // containing the grayscale image
    const downloadLink = downloadBlob(blob);

    // Set the title and classnames of the link
    downloadLink.title = 'Download Grayscale Photo';
    downloadLink.classList.add('btn-link', 'download-link');

    // Set the visible text content of the download link
```

Here is a working example of this application on **Codepen**:



```
const banner = document.querySelector('.image-banner');
const img = banner.querySelector('img');
const canvas = banner.querySelector('canvas');

function downloadBlob(blob, filename) {
  // Create an object URL for the blob object
  const url = URL.createObjectURL(blob);

  // Create a new anchor element
  const a = document.createElement('a');
```

Conclusion

We’ve finally come to the end of this tutorial. While there could be a lot to pick from this tutorial, it is glaring that Web APIs have a lot to offer as regards building powerful apps for the browser. Don’t hesitate to be experimental and adventurous.

Thanks for making out time to read this article. If you found this article insightful, feel free to give some rounds of applause if you don’t mind — as that will help other people find it easily on Medium.

Share this:

Twitter Reddit LinkedIn Facebook

Glad Chinda Follow

Full-stack web developer learning new hacks one day at a time. Web technology enthusiast. Hacking stuffs @theflutterwave.

Leave a Reply

Enter your comment here...