Week 2 Intermediate JavaScript – Files, Patterns and Flags

# **Day 2: Form Data**

## Blob Object

What is a Blob

A Blob is a data type representing a collection of binary data.

It's used for storing files, images, audio, video, and other binary data.

Blobs are immutable, raw data objects.

They're created using the Blob constructor or APIs that return Blob objects.

Blobs are useful when data doesn't need to be processed immediately but must be stored or transmitted in its raw form.

JavaScript provides the FileReader object to read Blob content.

Blobs can be converted into temporary URLs using URL.createObjectURL() for referencing in a browser.

**let content = ['Hello, World!'];**

**let blob = new Blob(content, { type: 'text/plain' });**

Blobs are essential for handling binary data in web applications, enabling efficient storage, manipulation, and transmission of files and media.

## Simple Form

Sending HTML forms with or without files, additional fields etc.

Constructor

**let formData = new FormData([form]);**

FormData is an object to store and send form data. Network methods, such as fetch, can accept a FormDataobject as a body.

It’s encoded and sent out with Content-Type: form/multipart. So, from the server point of view, that looks like a usual form submission.

The server accepts the POST request with the form and replies “User saved”.

**<form id=”formElem”>**

**<input type=”text” name=”name” value=”John”>**

**<input type=”text” name=”surname” value=”Mike”>**

**<input type=”submit”>**

**</form>**

**<script>**

**formElem.onsubmit = async (e) => {**

**e.preventDefault();**

**let response = await fetch(‘/article/formdata/post/ user’, {**

**method: ‘POST’,**

**body: new FormData(formElem)**

**});**

**Let result = await response.json();**

**Alert(result.message);**

**};**

**</script>**

## FormData Methods

**formData.append(name, value):** Adds a form field with the given name and value.

**formData.append(name, blob, fileName):** Adds a field as if it were <input type="file">. The third argument fileName sets the file name.

**formData.delete(name):** Removes the field with the given name.

**formData.get(name):** Gets the value of the field with the given name.

**formData.has(name):** Checks if a field with the given name exists.

**formData.set(name, value) / formData.set(name, blob, fileName):** Removes all fields with the given name, then appends a new field.

Iterating over formData fields can be done using a for..of loop.

**let formData = new FormData();**

**formData.append(‘key1’, ‘value1’);**

**formData.append(‘key2’, ‘value2’);**

**for(let [name, value] of formData) {**

**alert(`${name} = ${value}`);**

**}**

## Sending a form with a file

The form is always sent as Content-Type: form/multipart.

This encoding allows to send files. So, <input type="file"> fields are sent also, similar to a usual form submission.

**<form id=”formElem”>**

**<input type=”text” name=”firstName” value=”John”>**

**Picture: <input type=”file” name=”picture” accept=”image/\*”>**

**<input type=”submit”>**

**</form>**

**<script>**

**formElem.onsubmit = async (e) => {**

**e.preventDefault();**

**let response = await fetch(‘/article/formdata/post/user-avatar’, {**

**method: ‘POST’,**

**body: new Formdata(formElem)**

**});**

**let result = await response.json();**

**alert(result.message);**

**};**

**</script>**

## \*Sending a form with Blob data

Sending Images with Form Data

Dynamically generated Blobs, like images, can be directly sent with a fetch request.

Convenience of Form Data

It's often more convenient to include an image as part of a form along with additional fields and metadata.

Server Compatibility

Servers are typically designed to handle multipart-encoded forms, making them a preferred choice for submitting images.

Submitting Images from Canvas

To submit an image from a <canvas> along with other fields, FormData can be employed.

Adding an Image Blob

Use **formData.append("image", imageBlob, "image.png")** to mimic the submission of a file named "image.png" as if it were from the visitor's filesystem.

(code)

**FormData** objects are used to capture HTML form and submit it using fetch or another network method.

We can either create new FormData(form) from an HTML form, or create an empty object, and then append fields with methods:

* formData.append(name, value)
* formData.append(name, blob, fileName)
* formData.set(name, value)
* formData.set(name, blob, fileName)

Two peculiarities here:

The set method removes fields with the same name, append doesn’t.

To send a file, 3-argument syntax is needed, the last argument is a file name, that normally is taken from user filesystem for **<input type="file">**

Other methods are:

* formData.delete(name)
* formData.get(name)
* formData.has(name)

## Fetch: Download Progress

Tracking Download Progress with fetch()

The fetch method provides a way to monitor download progress.

It currently does not support tracking upload progress.

Reading Response as a Stream

To track download progress, we utilize the response.body property. This represents a "readable stream" that provides the response body in chunks.

Managing Stream Chunks

We can control the reading process by handling the chunks of the response. Each chunk is an object with properties done (indicating if the reading is complete) and value (a typed array of bytes).

Reading Progress

**let response = await fetch(url);**

**let contentLength = +response.headers.get('Content-Length');**

**let receivedLength = 0;**

**let reader = response.body.getReader();**

**let chunks = [];**

**while (true) {**

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

**if (done) break;**

**chunks.push(value);**

**receivedLength += value.length;**

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

**}**

**let chunksAll = new Uint8Array(receivedLength);**

**let position = 0;**

**for (let chunk of chunks) {**

**chunksAll.set(chunk, position);**

**position += chunk.length;**

**}**

**let result = new TextDecoder().decode(chunksAll);**

**let data = JSON.parse(result);**

Handling Binary Content

If binary content is needed instead of JSON, it's simpler. Replace steps 4 and 5 with the following:

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

This will give you the binary content as a Blob.

This process is for tracking download progress, not upload progress. Upload progress tracking is not currently supported with fetch. For upload progress, consider using XMLHttpRequest.

## Fetch: Abort

Fetch returns a promise. And JavaScript generally has no concept of “aborting” a promise. So how can we cancel a fetch?

Aborting a fetch request can be achieved using the AbortController object. This is especially useful when we want to cancel an ongoing request.

Steps to Implement Aborting a Fetch Request:

1. Create an AbortController:

**let controller = new AbortController();**

The AbortController has two main properties: **abort()** method and **signal**.

2. Pass the signal Property to Fetch:

**fetch(url, {**

**signal: controller.signal**

**});**

This associates the controller's signal with the fetch request.

3. To Abort, Call controller.abort():

**controller.abort();**

This will trigger the abort event on the AbortController's signal.

Handelling Abort Error:

**try {**

**let response = await fetch(url, { signal: controller.signal });**

**let data = await response.json();**

**// Process the data**

**} catch (err) {**

**if (err.name === 'AbortError') {**

**console.log('Fetch aborted');**

**} else {**

**console.error('Error:', err);**

**}**

**}**

Aborting Multiple Fetches

**let controller = new AbortController();**

**let urls = ['url1', 'url2', 'url3'];**

**for (let url of urls) {**

**fetch(url, { signal: controller.signal })**

**.then(response => response.json())**

**.then(data => console.log(data))**

**.catch(err => {**

**if (err.name === 'AbortError') {**

**console.log('Fetch aborted');**

**} else {**

**console.error('Error:', err);**

**}**

**});**

**}**

**// To abort all fetches**

**controller.abort();**