Week 2 Intermediate JavaScript – Files, Patterns and Flags

# **Day 1: File Objects**

## 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.

## File Object

Inherits from Blob and extends with filesystem capabilities.

Ways to obtain:

1. Constructor

**new File(fileParts, fileName, [options])**

* **fileParts:** Array of Blob/BufferSource/String values.
* **fileName:** Name of the file.
* **options (optional):**

lastModified: Timestamp (integer date) of last modification.

1. User interaction

**<input type="file">**

**drag-and-drop** or other browser interfaces.

Inherits file information from the operating system.

Getting a File object from **<input type="file">**

When **<input type="file">** is used, input.files is an array-like object containing selected files.

For a single file, it can be accessed as **input.files[0]**

The input.files property may contain multiple files if the input allows it. In this case, you can access each file using an **index**.

File object properties (in addition to Blob properties):

**name:** The file name.

**lastModified:** Timestamp of the last modification.

**<input type=”file” onchange=”showFile(this)”>**

**<script>**

**Function showFile(input) {**

**let file = input.files[0];**

**alert(`File name: ${file.name}`);**

**alert(`Last modified: ${file.lastModified}`);**

**}**

**</script>**

## File Reader

FileReader is designed for **reading data** from Blob (and File) objects. It **handles data delivery through events** due to potential time-consuming disk operations.

**let reader = new FileReader();**

Methods:

1. **readAsArrayBuffer(blob)**

Reads data in binary format as an ArrayBuffer.

For binary files and low-level binary operations. For high-level operations like slicing, File can be directly used, as it inherits from Blob.

1. **readAsText(blob, [encoding])**

Reads data as a text string with an optional encoding parameter (defaults to utf-8).

Suitable for text files when a string representation is desired.

1. **readAsDataURL(blob)**

Reads binary data and encodes it as a base64 data URL.

Used when embedding data in src attributes for tags like img. An alternative method for this operation is discussed in the Blob chapter: URL.createObjectURL(file)

1. **abort()**

Cancels the ongoing operation.

Events to occur as read proceeds

* loadstart: Indicates the start of loading.
* progress: Occurs during the reading process.
* load: No errors, reading is successfully completed.
* abort: Triggered when abort() is called.
* error: Signifies an error during the operation.
* loadend: Marks the end of reading, whether successful or not.
* reader.result: Contains the result (if successful).
* reader.error: Holds the error (if the operation fails).

Common events used: **Load** and **error**

**<input type=”file” onchange=”readFile(this)”>**

**<script>**

**function readFile(input) {**

**let file = input.files[0];**

**let reader = new FileReader();**

**reader.readAsText(file);**

**reader.onload = function() {**

**console.log(reader.result);**

**};**

**reader.onerror = function() {**

**console.log(reader.error);**

**};**

**}**

**</script>**

Additional Notes

FileReader is used to **read not only files but any blobs.**

* It converts a blob to various formats:
* readAsArrayBuffer(blob) converts to ArrayBuffer.
* readAsText(blob, [encoding]) converts to a string (an alternative to TextDecoder).
* readAsDataURL(blob) converts to a base64 data URL.

For Web Workers, there's a synchronous variant called FileReaderSync which returns results without generating events.

File objects inherit from Blob and include additional properties like name and lastModified.

FileReader objects can read from a file or a blob in three formats: String, ArrayBuffer, and Data URL (base64 encoded).

In many cases, there's no need to read file contents. Creating a short URL with URL.createObjectURL(file) allows easy referencing in HTML elements like <a> or <img>.

Sending a File over a network is straightforward as network APIs like XMLHttpRequest or fetch natively accept File objects.

## Fetch

JavaScript Network Requests

JavaScript allows sending network requests for tasks like submitting orders, fetching user info, or receiving updates without page reload.

The term "AJAX" (Asynchronous JavaScript And XML) refers to this, though XML isn't mandatory.

**fetch()** is a modern method for this task, offering versatility and solid browser support.

**let promise = fetch(url, [options])**

* **url: The target URL.**
* **options: Optional parameters**

How it works:

The browser initiates the request and returns a promise. Obtaining a response involves two stages. Firstly, a promise resolves with a Response object containing headers but not the body.

The promise rejects if there are network issues or the site doesn't exist.

**HTTP errors (e.g., 404) normal, won't cause rejection.**

Key response properties:

**ok:** true if HTTP status is 200-299.

**status:** HTTP status code.

## POST vs GET

A **POST** request is one of the HTTP methods used when a client (e.g., a browser) wants to send data to a server to create or update a resource on the server. This data is typically sent in the body of the request.

Unlike a **GET** request which appends data to the URL (and is primarily used for retrieving information), a POST request includes the data in the body of the request, which makes it suitable for sending larger amounts of data, like form submissions or file uploads.

For example, when you submit a form on a webpage, it's often sent as a POST request. This way, the data isn't visible in the URL, which is important for sensitive information like passwords.

## POST Requests

Making a POST Request with Fetch

To perform a POST request, or any other HTTP method, we employ fetch options:

**Method**: Specifies the HTTP method, e.g., POST.

**Body**: Can be:

* A string (e.g., JSON).
* FormData object, for submitting data as form/multipart.
* Blob/BufferSource for sending binary data.
* URLSearchParams, for submitting data in x-www-form-urlencoded encoding (rarely used).

**let user = { name: “John”, surname: “Smith” };**

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

**method: ‘POST’,**

**headers: {**

**‘Content-Type’: ‘application/json;charset=utf-8’**

**},**

**body: JSON.stringify(user)**

**});**

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

**alert(result.message);**

When the body is a string, the default Content-Type is set to text/plain;charset=UTF-8. To send JSON-encoded data, we utilize the headers option to set it to application/json, which is the correct content type for JSON data.

This is the process of making a POST request using the fetch API and the associated options.

## Sending an Image

We can also submit binary data directly using Blob or BufferSource.

(code)

Response Properties

**response.status** – HTTP code of the response.

**response.ok** – True if the status is in the range of 200-299.

**response.headers** – A Map-like object containing HTTP headers.

Methods for Retrieving Response Body

**response.json()** – Parses the response as a JSON object.

**response.text()** – Returns the response as text.

**response.formData()** – Returns the response as a FormData object (for form/multipart encoding).

**response.blob()** – Returns the response as a Blob (binary data with type).

**response.arrayBuffer()** – Returns the response as an ArrayBuffer (pure binary data).

Fetch Options Recap

**method** – Specifies the HTTP method.

**headers** – An object with request headers (note that not all headers are allowed).

**body** – Can be a string, FormData, BufferSource, Blob, or UrlSearchParams object for sending data.

## GitHub GetUsers Program

Instructions

Create an async function getUsers(names), that gets an array of GitHub logins, fetche the users from GitHub and returns an array of GitHub users.

The GitHub url with user informaiton for the given USERNAME is: <https://api.github.com/users/USERNAME>.

Important details:

There should be one fetch request per user. And requests shouldn’t wait for each other. So that the data arrives as soon as possible.

If any request fails, or if there’s no such user, the function should return null in the resulting array.

**const usernames = ['octocat', 'nonexistentuser', 'anotheruser'];**

**const users = await getUsers(usernames);**

**console.log(users);**

**async function getUsers(names) {**

**const promises = names.map(async (name) => {**

**const response = await fetch(`https://api.github.com/users/${name}`);**

**if (response.ok) {**

**const user = await response.json();**

**return user;**

**} else {**

**return null;**

**}**

**});**

**const users = await Promise.all(promises);**

**return users;**

**}**

How it works

async function getUsers(names): This defines an asynchronous function named getUsers that takes an array of GitHub logins (names) as an argument.

const promises = names.map(async (name) => { ... }: Here, we're using map to create an array of promises. For each name, we perform an asynchronous operation.

const response = await fetch(https://api.github.com/users/${name}`);`: This line fetches the user information from GitHub for the provided username.

if (response.ok) { ... } else { ... }: Checks if the response is successful (status code 200-299). If it is, we proceed to parse the JSON response.

const user = await response.json();: This line extracts the JSON data from the response.

return user;: If everything is successful, we return the user information.

return null;: If there's an error or the user doesn't exist, we return null.

const users = await Promise.all(promises);: We wait for all the promises to resolve using Promise.all. This ensures that we get an array of users or null values.

return users;: Finally, we return the array of users.