**File Objects**

A file object inherits from Blob and is extended with filesystem related capabilities. There are two ways to obtain it.

First there is a constructor, similar to Blob:

new File(fileParts, fileName, [options])

* + **fileParts -** is an array of Blob/BufferSource/String values.
  + **fileName -** file name string.
  + **options -**optional object:
  + **lastModified** - the timestamp (integer date) of last modification.

Second, more often we get a file from <input type="file"> or drag’n’drop or other browser interfaces. In that case, the file gets this information from OS

As File inherits from Blob, File objects have the same properties, plus:

name – the file name,

lastModified – the timestamp of last modification

**File Reader**

FileReader is an object with the sole purpose of reading data from Blob (and hence File too) objects. It delivers the data using events as reading from disk may take time.

The constructor is as follows:

let reader = new FileReader(); // no arguments

The main method is as follows:

* + **readAsArrayBuffer(blob)** – read the data in binary format ArrayBuffer.
  + **readAsText(blob, [encoding])** – read the data as a text string with the given encoding (utf-8by default).
  + **readAsDataURL(blob)** – read the binary data and encode it as base64 data url.
  + **abort()** – cancel the operation.
  + error – error has occurred.
  + loadend – reading finished with either success or failure.
  + When the reading is finished, we can access the result as:
  + reader.result is the result (if successful)
  + reader.error is the error (if failed).
  + The most widely used events are for sure load and error.

**Fetch**

In JavaScript, the term "fetch" typically refers to making an HTTP request to retrieve data from a server or an external resource, such as an API. The Fetch API is a built-in JavaScript feature that allows you to make network requests. Here's a basic overview of how you can use the Fetch API in JavaScript:

fetch('https://api.example.com/data')

.then(response => {

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json(); // Parse the response body as JSON

})

.then(data => {

// Do something with the data

console.log(data);

})

.catch(error => {

console.error('Fetch error:', error);

});

In JavaScript, the term "fetch" typically refers to making an HTTP request to retrieve data from a server or an external resource, such as an API. The Fetch API is a built-in JavaScript feature that allows you to make network requests. Here's a basic overview of how you can use the Fetch API in JavaScript:

1. Basic Fetch Request:

fetch('https://api.example.com/data') .then(response => { if (!response.ok) { throw new Error('Network response was not ok'); } return response.json(); // Parse the response body as JSON }) .then(data => { // Do something with the data console.log(data); }) .catch(error => { console.error('Fetch error:', error); });

Save to grepper

In this example:

* We use the **fetch()** function to make an HTTP GET request to '<https://api.example.com/data>'.
* We check if the response is successful (status code 200-299).
* If the response is successful, we parse the response body as JSON using **.json()**.
* We then handle the data in the second **.then()** block.
* If there's an error in the request, it's caught in the **.catch()** block.

**Response**

In JavaScript, a "response" typically refers to the data received as a result of making an HTTP request to a server using the Fetch API or another mechanism. A response object contains information about the HTTP response, including status, headers, and the response body. Here's how you can work with HTTP responses in JavaScript using the Fetch API:

**response.json()** – parse the response as JSON object,

* + **response.text()** – return the response as text,
  + **response.formData()** – return the response as FormData object (form/multipart encoding, explained in the next chapter),
  + **response.blob()** – return the response as Blob(binary data with type),
  + **response.arrayBuffer()** – return the response as ArrayBuffer (pure binary data),

additionally, response.body is a ReadableStreamobject, it allows to read the body chunk-by-chunk, we’ll see an example later.

**Post Request**

In JavaScript, you can make POST requests to a server using various methods and libraries. One of the most common ways to make POST requests is by using the **fetch** API, which is available in modern browsers. Here's an example of how to make a POST request using.

**FormData Methods**

FormData is an object in JavaScript that allows you to create and manipulate data in HTML forms. It is often used when working with AJAX requests to send form data to a server. FormData provides several methods and properties to work with form data. Here are some of the commonly used methods of the FormData object:

1. FormData() Constructor:
   * You can create a new FormData object by calling the constructor with or without an HTML form element as an argument. If you pass a form element, it will automatically populate the FormData object with the form's data.

**Fetch Cross- Origin Request**

Cross-Origin Request refers to a web browser security feature that restricts web pages from making requests to a different domain than the one that served the web page. This security feature is known as the Same-Origin Policy, and it helps prevent potential security vulnerabilities, such as cross-site request forgery (CSRF) and cross-site scripting (XSS) attacks.

To fetch data from a different domain (cross-origin), you typically need to use mechanisms that the browser allows, such as Cross-Origin Resource Sharing (CORS) or JSONP (JSON with Padding). Here's a basic example of how to make a cross-origin request using JavaScript and the Fetch API, assuming the server on the target domain has CORS enabled:

**Using Forms**

We use forms as a way to communicate with another server was to submit <form>

**Cors For Simple Request**

CORS, or Cross-Origin Resource Sharing, is a security feature implemented by web browsers to control which web pages can access resources from different origins (domains). CORS is relevant when a web page loaded from one domain (the "origin") makes a request for resources (e.g., APIs, images, or fonts) hosted on a different domain.

A "simple request" in the context of CORS is a type of HTTP request that meets specific criteria and is subject to less stringent CORS restrictions. To be considered a simple request, the request must meet the following conditions:

1. The HTTP method used in the request is one of the following:
   * GET
   * HEAD
   * POST
2. The only allowed request headers (in addition to the standard headers like Accept, Accept-Language, and Content-Type) are:
   * Accept
   * Accept-Language
   * Content-Language
   * Content-Type (with some limitations, see below)
   * DPR (Device Pixel Ratio)
   * Downlink
   * Save-Data
   * Viewport-Width
   * Width
3. The Content-Type header is limited to the following values:
   * application/x-www-form-urlencoded
   * multipart/form-data
   * text/plain

If an HTTP request meets these conditions, it is considered a "simple request," and the browser will automatically include the necessary CORS headers in the response. These headers typically include Access-Control-Allow-Origin, Access-Control-Allow-Methods, and Access-Control-Allow-Headers.

**Credentials**

In JavaScript, the term "credentials" often refers to user authentication and authorization mechanisms. JavaScript can interact with credentials in various ways, primarily when working with web applications and APIs. Here are some key aspects of handling credentials in JavaScript.

**Patterns and Flags**

In JavaScript, you can work with patterns and flags primarily when using regular expressions. Regular expressions, often referred to as regex or regexp, are powerful tools for pattern matching and text manipulation. Flags are used to modify how regular expressions behave when matching patterns in strings.

Here's an overview of patterns and flags in JavaScript regular expressions:

1. Patterns: Patterns in regular expressions are used to define the specific sequence of characters you want to match within a string. You can use various metacharacters and quantifiers to create complex patterns. Here are some commonly used elements within patterns:
   * Literal Characters: Any literal characters in the pattern will be matched exactly. For example, the pattern /apple/ will match the word "apple" in a string.
   * Metacharacters: These are characters with special meanings in regular expressions, like . (matches any character), \* (matches zero or more occurrences), + (matches one or more occurrences), ? (matches zero or one occurrence), etc.
   * Character Classes: You can use square brackets [] to define a character class, which matches any single character from the provided set. For example, [aeiou] matches any vowel.
   * Anchors: Anchors like ^ (start of the string) and $ (end of the string) are used to specify where a match should begin or end.
2. Flags: Flags are used to modify how a regular expression pattern is applied. They are added after the closing slash / of a regular expression literal. Here are some commonly used flags in JavaScript:
   * i (Case-Insensitive): When you use the i flag, the regular expression becomes case-insensitive, meaning it will match characters regardless of their case. For example, /apple/i will match "apple," "Apple," "aPpLe," etc.
   * g (Global): The g flag indicates that the regular expression should find all matches in the input string, not just the first one. It's useful when you want to find all occurrences of a pattern.
   * m (Multiline): The m flag enables multiline mode, which changes how ^ and $ anchors work. In multiline mode, ^ matches the start of each line within a multiline string, and $ matches the end of each line.
   * s (Dot-All): The s flag, also known as "dot-all" mode, makes the dot (.) match any character, including newline characters (\n).

Here's an example of using flags with regular expressions in JavaScript:

const text = "Apple is a fruit. Apples are tasty."; const pattern = /apple/gi; // Matches "apple" case-insensitively and globally const matches = text.match(pattern); console.log(matches); // Outputs: ["Apple", "Apples"]

In this example, the g flag makes the regular expression find all matches in the text, and the i flag makes it case-insensitive, so it matches both "Apple" and "Apples."