```
<input type="file" onchange="showFile(this)">

<script>
function showFile(input) {
  let file = input.files[0];

  alert(`File name: ${file.name}`); // e.g my.png
  alert(`Last modified: ${file.lastModified}`); // e.g 1552830408824
}
</script>
```

Week 7 Files, Patterns and Flags

Day 1 – File Objects

A file object is derived from Blob and enhanced with features related to file systems. There are two methods for acquiring it.

1. Constructor like Blob:
   new File(filesParts, 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.

2. Most commonly, we obtain a file from sources like `<input type="file">` elements, drag-and-drop interactions, or other browser interfaces. In such cases, the file inherits its information from the operating system.

   Since File inherits from Blob, File objects possess the same properties as Blob objects, in addition to some extra properties specific to File objects:

   name – the file name,
   lastModified – the timestamp of last modification

   This is how we can obtain a File object from an `<input type="file">` element:

```
<input type="file" onchange="showFile(this)">

<script>
function showFile(input) {
  let file = input.files[0];

  alert(`File name: ${file.name}`); // e.g my.png
  alert(`Last modified: ${file.lastModified}`); // e.g 1552830408824
}
</script>
```

If the input allows for the selection of multiple files, `input.files` will be an array-like object containing those files. In this case, with only one file selected, we can simply access it using `input.files[0]`.

File Reader

FileReader is an object primarily designed to extract data from Blob objects, which also includes File objects. It uses events to provide data because reading from disk can be time-consuming.

FileReader (JavaScript):

- FileReader is an object used to read data from Blob and File objects asynchronously.
- It offers methods like readAsArrayBuffer, readAsText, and readAsDataURL for reading data in various formats.
- Key events include load (successful completion), error (error during reading), and progress (reading in progress).
- After reading, result contains the loaded data (if successful), and error holds information about any errors encountered.

Fetch

- fetch() Method: `fetch()` is a modern and versatile method in JavaScript used to send network requests to a server and retrieve data asynchronously without the need to reload the page.

- Basic Syntax: The basic syntax of the `fetch()` method is as follows:

JavaScript

```
let promise = fetch(url, [options]);
```

- `url`: The URL that the request should be made to.
- `options` (optional): An object containing parameters like the HTTP method, headers, etc., to customize the request.

- Promise-Based: `fetch()` returns a promise immediately upon invocation, allowing for asynchronous handling of the request and response.

- Two-Stage Process:
- First Stage: The promise resolves with a `Response` object when the server responds with headers. This stage provides information such as HTTP status and headers but doesn't include the response body.
- `ok`: A boolean property that is `true` if the HTTP status code falls within the 200-299 range, indicating a successful response.
- `status`: The HTTP status code, which indicates the outcome of the request, even if it's an error status (e.g., 404 or 500).

- Error Handling: Promises returned by `fetch()` can also reject if there are network problems, the site doesn't exist, or if there are HTTP errors. Handling these errors is an integral part of using `fetch()`.

- HTTP Errors: It's important to note that HTTP errors, including 404 (Not Found) or 500 (Internal Server Error), are considered a normal part of the response process and do not cause promise rejection by default.

`fetch()` is a powerful tool for making asynchronous network requests in JavaScript, and it provides flexibility for handling responses and errors, making it a fundamental component for web developers when interacting with servers and APIs.

Post Requests

To perform a POST request or use a different HTTP method when making a request, you must utilize fetch options:

- **method –** HTTP-method, e.g Post,
- **body-** one of:
- a string(e.g. JSON),
- FormData object, to submit the data as form/mutlipart,
- Blob/BlufferSource to send binary data,
- URLSearchParams, to submit the data in x-www-form-urlencoded encoding, barely used

Code snippet submitting user object as JSON:

It's important to be aware that when the request body is a string, the default `Content-Type` is automatically set to `text/plain;charset=UTF-8`. To send data in JSON format, which is the appropriate content type for JSON-encoded data, you should specify `application/json` using the `headers` option.

Sending an Image

We can submit binary data directly using Blob or BufferSource. As an example, consider a <canvas> element where users can draw by moving their mouse. When the "submit" button is clicked, the image is sent to the server.

```
<input type="file" onchange="showFile(this)">

<script>
function showFile(input) {
  let file = input.files[0];

  alert(`File name: ${file.name}`); // e.g my.png
  alert(`Last modified: ${file.lastModified}`); // e.g 1552830408824
}
</script>
```

Here we also didn't need to set Content-Typemanually, because a Blob object has a built-in type (here image/png, as generated by toBlob).

The submit() function can be rewritten without async/await like this:

A typical fetch request consists of two await calls:

```
let response = await fetch(url, options); // resolves with response headers
let result = await response.json(); // read body as json
```

Or, promise-style:

```
fetch(url, options)
  .then(response => response.json())
  .then(result => /* process result */)
```

Response properties:

response.status – HTTP code of the response,

response.ok – true is the status is 200-299.

response.headers – Map-like object with HTTP headers.

Methods to get response body:

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, see the next chapter),

response.blob() – return the response as Blob(binary data with type),

response.arrayBuffer() – return the response as ArrayBuffer (pure binary data),

Fetch options so far:

method – HTTP-method,

headers – an object with request headers (not any header is allowed),

body – string, FormData, BufferSource, Blob or UrlSearchParams object to send.

DAY 2 – FormData

- *HTML Forms*: HTML forms are used to collect user input. They consist of various input elements like text fields, radio buttons, checkboxes, etc.

- *Accessing Form Elements*: You can access form elements in JavaScript using their `name` attribute or by using their `id`. For example, `document.forms['myForm']` or `document.getElementById('username')`.

- *Form Submission*: To handle form submission, you can attach an event listener to the form's `submit` event using `addEventListener`. This allows you to intercept the form data before it's sent to the server.

- *Preventing Default Behavior*: Inside the submit event handler, you often want to prevent the default form submission behavior using `event.preventDefault()`. This allows you to handle the submission with JavaScript.

- *Accessing Form Values*: You can access the values entered in form elements using the `value` property. For example, `document.forms['myForm'].elements['username'].value` gets the value of an input element with the name "username."

- *Form Validation*: You can use JavaScript to validate form data before submission. Check if the data is in the expected format and meets any required criteria. Provide feedback to the user if validation fails.

- *Serializing Form Data*: To send form data to the server via AJAX or fetch, you can serialize the form data into a format like JSON or URL-encoded data using JavaScript. You can manually collect form element values and construct the data object.

- Form Reset To reset a form to its initial state, you can call the `reset()` method on the form element. For example, `document.forms['myForm'].reset()`.

- *Working with Form Elements*: JavaScript provides various methods and properties to manipulate form elements, such as setting values, disabling/enabling, and hiding/showing elements based on user interactions.

- *Security*: Be cautious when handling form data in JavaScript. Always validate and sanitize user input on the server-side to prevent security vulnerabilities like SQL injection and Cross-Site Scripting (XSS).

- *Modern Libraries*: Consider using modern JavaScript libraries and frameworks like React, Angular, or Vue.js, which offer more structured and efficient ways to handle forms in web applications.

Day 3- Fetch: Cross-Origin Requests

- *Cross-Origin Requests*: Cross-origin requests occur when a web page makes a request to a different domain, protocol, or port than the one it originated from. These requests are subject to the Same-Origin Policy for security reasons.

- *Fetch API*: The Fetch API is a modern JavaScript API for making HTTP requests. It provides a more flexible and powerful way to work with network requests compared to the older XMLHttpRequest.

- *Fetch Syntax:* The basic syntax for making a fetch request:

javascript

```
fetch(url, options)
  .then(response => {
    // Handle the response
  })
  .catch(error => {
    // Handle errors
  });
```

Cross-Origin Requests with Fetch: To make a cross-origin request, you can specify the URL of the resource on a different domain in the `fetch` function. By default, fetch follows the Same-Origin Policy.

CORS (Cross-Origin Resource Sharing): To enable cross-origin requests, the server hosting the resource needs to include appropriate CORS headers in its response. Common CORS headers include `Access-Control-Allow-Origin` and `Access-Control-Allow-Methods`.

Example of a GET Request: Making a cross-origin GET request to retrieve JSON data:

```
javascript

fetch('https://api.example.com/data.json')
  .then(response => response.json())
  .then(data => {
    // Handle the JSON data
  })
  .catch(error => {
    // Handle errors
  });
```

Example of a POST Request: Making a cross-origin POST request with JSON data:

```
javascript

const url = 'https://api.example.com/save-data';

const data = { key: 'value' };



fetch(url, {

  method: 'POST',

  headers: {

    'Content-Type': 'application/json',

  },

  body: JSON.stringify(data),

})

  .then(response => {

    // Handle the response

  })

  .catch(error => {

    // Handle errors

  });
```

Handling Response: Use `.then()` to handle the successful response and `.catch()` to handle errors. You can check the response status code and headers, and parse the response data as needed.

- *Promises*: Fetch returns Promises, which allows you to use modern asynchronous code patterns like async/await for cleaner and more readable code.

- *Security Considerations*: Be aware of security concerns when making cross-origin requests, as sensitive data may be exposed. Always validate and sanitize the data on the server-side.

When working with cross-origin requests, you should have permission from the target server and ensure that CORS headers are properly configured to allow the requests from your domain.

Day 4-Patterns and Flags

- *Regular Expressions (Regex):* Regular expressions are powerful patterns used for text matching and manipulation in JavaScript.

- *Pattern Creation:* Patterns are defined using `/` delimiters. For example: `/pattern/`.

- *Flags*: Flags modify the behavior of a regular expression.

- Common flags include:

  - `i`: Case-insensitive matching.

  - `g`: Global search (matches all occurrences, not just the first).

  - `m`: Multi-line matching (treats `^` and `$` as line boundaries).

- *Example of Using Flags*: Matching a pattern case-insensitively:

javascript

```
const text = 'Hello World';

const pattern = /hello/i;

console.log(pattern.test(text)); // true
```

- ***Basic Patterns***:

- *Literal Characters*: Match specific characters. Example: `/abc/` matches "abc".

- *Character Classes*: Match a set of characters. Example: `/[aeiou]/` matches any vowel.

- *Quantifiers*: Specify the number of occurrences. Example: `/a{2,4}/` matches "aa", "aaa", or "aaaa".

- *Metacharacters*: Special characters with special meanings, like `.` (matches any character) and `|` (alternation).

- *Example with Metacharacters*: Matching an email address:

javascript

```
const pattern = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
```

- *Anchors: Anchors*: like `^` (start of string) and `$` (end of string) specify where a match must occur.

- *Example with Anchors*:

Matching a string that starts with "Hello":

javascript

```
const pattern = /^Hello/;
```

- *Groups and Capturing*:

- Parentheses `()` are used for grouping and capturing parts of a match.

- *Example with Groups and Capturing*:

- Capturing a date in a string:

javascript

```
const pattern = /(\d{2})\/(\d{2})\/(\d{4})/;
const result = pattern.exec('01/15/2023');
// result: ["01/15/2023", "01", "15", "2023"]
```

- *Escape Character*:

- Use `\` to escape metacharacters when you want to match them as literal characters. For example, `/a\*/` matches "a*".

- *Regex Methods:*

- JavaScript provides methods like `test()`, `match()`, `exec()`, `search()`, and `replace()` for working with regular expressions.

- *Regex Object*:

- You can create a `RegExp` object for dynamic pattern generation. Example: `const pattern = new RegExp('pattern', 'flags');`.

- *Regex Validation*:

- Regular expressions are commonly used for input validation, data extraction, and text manipulation tasks in JavaScript applications.

Regex is a versatile tool for handling complex text patterns in JavaScript, and mastering its usage can greatly enhance your ability to work with textual data.