**Week 2 JavaScript Intermediate.**

**Day 1. File and FileReader**

A file is blob that represent a file from the filesystem.

First there is a constructor, similar to Bob:

new File(fileParts, filename, [options])

* FileParts – is an array of Blob/BufferSource/String values.
* filename – file string.
* Options – optional object:”
* lastModified – the timestamp (integer date) of last modification

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

The constructor is as follows:

Let reader = new FileReader();

* readAsArrayBuffer(blob) – read the data in binary 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.
* Arbort()- cancel the operation.

**Fetch**

JavaScript can send network requests to the server and load new information whenever is needed.

The fetch method is modern and versatile, so we’ll start with it. It evolved for several years and continues to improve, right now the support is pretty solid among browsers.

The basic syntax is:

Let promise = fetch(url, [options])

url- the URL to access.

Options -optional parameters: method, headers etc. The browser starts the request right away and returns a promise. Getting a response is usually a two-stage process.

Response provides multiple promise-based methods to access the body in various formats:

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

Response.body is a readableStreamobject, it allows to read the body chunk-by-chunk.

**POST REQUESTS**

To make a POST request, or a request with another method, we need to use fetch options:

* Method – HTTP-method, e.g POST,
* Body – one of:

1. A string (e.g JSON)
2. FormData object, to submit the data as form/multipart.
3. Blob/BufferSource to send binary data.
4. URLSearchParams, to submit the data in x-www-form-urlencoded encoding, rarely used.

**Sending An Image**

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

**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)

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

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

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**

**FORM DATA**

The Contractor is: let formData = new FormData([form)];

If HTML form element is provided is provided, it automatically captures the its fields. As you may have already guessed,

FormData is an object to store and send form data.

The special thing about FormData is that network method such fetch, can accept a FormDataobject as body.

FormData Methods

FormData can be modify with methods:

* formData.append(name, value) – add a form field with the given name and value.
* formData.append(name, blob, fileName) – add a field as if it were <input type=”file”, the third argument filename sets file name (not form field name), as it were a name of the file in user’s filesystem.
* formData.delete(name) – remove the field with the given name,
* formData.get(name) – get the value of the field with given name.
* formData.has(name) - if there exists a field with the given name, returns true, otherwise false.
* formData.set(name, value),
* formData.set(name, blob, fileName).

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, similar to a usual form submission.

**Sending a form with blob data**

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">.

**FETCH: DOWNLOAD PROGRESS**

To track download progress, we can use response.boy property. It’s a “readable stream” a special object that provides body chunk-by-chunk, as it comes.

The result of await reader.read() call is an object with two properties:

* Done – true when the reading is complete.
* Value – a typed array of bytes: Uinit\*8Array.

**FETCH: ABORT**

Aborting a fetch is a little bit trickly.

Controller is an extremely simple object. It has a single method abort(), and Single property signal. When abort is called, the abortvent triggers on controller.signal.

**AbortController** is scalable, it allows to cancel multiple fetches at once.

**FETCH: CROSS-ORIGIN REQUESTS**

The core concept here is origin – a domain/port/protocol triplet.

Cross-origin requests-those sent to another domain (even a subdomain) or protocol or port-require special headers from the side.

Try{

Await fetch(‘http://example.com’);

}catch(err){

Alert(err); failed to fetch

Using form

GET/POST requests could be made to another site without networking methods, but accessing content from another site is forbidden, making it impossible to read the response

Simple Requests

There are two types of cross-domain requests:

Simple request is a request that satisfies two conditions:

Simple method: GET, POST, or HEAD

Simple headers – the only allowed custom headers:  
accept, Accept-language, content-language,

Content-Type with the value application/x-www-form-urlencoded, multipart/form-data or text/plain.

Any other request is considered “non-simple. For Instance, request with PUT method or with an API-Key HTTP-header does not fit the limitations.

The essential difference is that a “simple request” can be made with a <form> or a <script> without any special methods.

CORS for Simple Requests

If the browser is cross-origin, the browser always adds Origin header to it.

if we request https://anywhere.com/request from https://javascript.info/page, the headers will be like:

GET/request

Host: anywhere.com

Origin: <https://javascript.info>

RESPONSE HEADERS

For cross-origin request, by default JavaScript may only access “simple response headers”.

Cache-Control

Content-Language

Content-Type

Expires

Last-Modified

Pragma

There is no content length header in the list. If we are downloading something and would like to track the percentage of progress, then additional permission is requires to access that header.

Non-simple requests

We can use any HTTP-method: not just GET/POST, but also PATCH, DELETE and others

A preflight request uses method OPTIONS and has nobody.

* Access-Control-Request-Method header has the requested method.
* Access-Control-Request-Headers header provides a comma-separated list of non-simple HTTP-headers

If the server agrees to serve the requests, then is should respond respond with status 200, without body.

* The response header Access-Control-Allow-Methods must have allowed method.
* The response header Access-Control-Allow-Headers must have a list of allowed headers.
* Additionally, the header Access-Control-Max-Age may specify a number of seconds to cache the permissions. So, the browser won’t have to spend a preflight for subsequent requests that satisfy given permissions.

**CREDENTIALS**

A cross=-origin request by default does not bring any credentials (cookies or HTTP authentication). That is uncommon for HTTP-requests. a request to http://site.com is accompanied by all cookies from that domain.  But cross-domain requests made by JavaScript methods are an exception.

Fetch does not send any cookies, even those that belong to another.com domain.

Why?

That’s because a request with credentials is much more powerful that an anonymous one. If allowed, if grants JavaScript the full power to act and access sensitive information on behalf of a user.

fetch sends cookies originating from another.com without request to that site.

**FETCH API**

[referrer, referrerPolicy](https://javascript.info/fetch-api#referrer-referrerpolicy)

These options govern how fetch sets HTTP Refererheader.

That header contains the URL of the page that made the request. In most scenarios, it plays a very minor informational role, but sometimes, for security purposes, it makes sense to remove or shorten it.

The **referrer** option allows to set any**Refererwithin** the current origin) or disable it.

Patterns: Patterns in JavaScript often refer to regular expressions, which are powerful tools for pattern matching and manipulation of strings. You can create regular expressions using the RegExp constructor or by using regular expression literals, denoted by forward slashes (/).

USAGE

Let str = “I Love JavaScript!”;

Let regexp = /love/;

Alert(str.search(regexp));

Flags

Flags in regular expressions are used to modify the behavior of the pattern match. They are typically placed after the closing slash in a regular expression literal or passed as a string to the RegExp constructor.

Commonly used flags include:

i: Case-insensitive matching.

g: Global matching (matches all occurrences, not just the first).

m: Multi-line matching (treats the input as multiple lines).

For example, to create a case-insensitive regular expression to match "example" globally in a string, you can use the gi flags like this:

const pattern = /example/gi;