**JAVASCRIPT WEEK 7 SUMMARY**

**Day 1 – File Objects**

**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.

The choice of read method depends on which format we prefer, how we are going to use data.

readAsArrayBuffer – for binary files, to do low-level binary operations. For high-level operations, like slicing, File inherits from Blob, so we can call them directly, without reading.

readAsText – for text files when we’d like to get a string.

readAsDataURL – when we’d like to use this data in src for img or another tag. There’s an alternative to reading a file for that, as discussed in chapter Blob: URL.createObjectURL(file).

As the reading proceeds, there are events:

* loadstart – loading started.
* progress – occurs during reading.
* load – no errors, reading complete.
* abort – abort() called.
* error – error has occurred.
* loadend – reading finished with either success or failure.

When the reading is finished, you 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 –** JavaScript can send requests to the server and load new information whenever is needed. For example, you can: Submit an order, load user information, receive latest updates from the server, etc. All of that without reloading the page!

There’s an umbrella term “AJAX” (abbreviated Asynchronous Javascript And Xml) for that. We don’t have to use XML though: the term comes from old times, that’s that word is there. There are multiple ways to send a network request and get information from the server. 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. First, the promise resolves with an object of the built-in Response class as soon as the server responds with headers. So, we can check HTTP status, to see whether it is successful or not, check headers, but don’t have the body yet. The promise rejects if the fetch was unable to make HTTP-request, e.g. network problems, or there’s no such site. HTTP-errors, such as 404 or 500, are considered a normal flow. We can see them in response properties:

ok – boolean, true if the HTTP status code is 200-299.

status – HTTP status code.

**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:
  + a string (e.g. JSON)
  + FormData object, to submit the data as form/multipart
  + Blob/BufferSource to send binary data
  + URLSearchParams, to submit the data in x-www-form-urlencoded encoding, rarely used.

**Sending an Image –** You 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, 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**

**FormData methods –** You can modify fields in FormData 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 if 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 the given name
* formData.has(name) – if there exists a field with the given name, returns true, otherwise false

A form is technically allowed to have many fields with the same name, so multiple calls to append add more same-named fields. There’s also method set, with the same syntax as append. The difference is that .set removes all fields with the given name, and then appends a new field. So it makes sure there’s only one field with such a name:

* 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 also, similar to a usual form submission.

**Sending a form with Blob data –** Sending a dynamically generated Blob, e.g. an image, is easy. You can supply it directly as fetch parameter body. In practice though, it’s often convenient to send an image not separately, but as a part of the form, with additional fields, such as “name” and other metadata. Also, servers are usually more suited to accept multipart-encoded forms, rather than raw binary data. Please note how the image Blob is added:

formData.append("image", imageBlob, "image.png");

That’s same as if there were <input type="file" name="image"> in the form, and the visitor submitted a file named image.png (3rd argument) from their filesystem.

**Fetch: Download progress –** The fetch allows you to track download progress. Please note: there’s currently no way for fetch to track upload progress. For that purpose, please use XMLHttpRequest. To track download progress, we can use response.body property. It’s a “readable stream” – a special object that provides body chunk-by-chunk, as it comes. Unlike response.text(), response.json() and other methods, response.body gives full control over the reading process, and we can count how much is consumed at any moment.

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: Uint8Array.

You wait for more chunks in the loop, until done is true. To log the progress, you just need for every value to add its length to the counter.

**Fetch:abort -** Aborting a fetch is a little bit tricky. Remember, fetch returns a promise. And JavaScript generally has no concept of “aborting” a promise. So how can you cancel a fetch? There’s a special built-in object for such purposes: AbortController. The usage is pretty simple:

* Step 1: create a controller: let controller = new AbortController(); A controller is an extremely simple object. It has a single method abort(), and a single property signal. When abort() is called, the abortevent triggers on controller.signal:
* Step 2: pass the signal property to fetch option: let controller = new AbortController();

fetch(url, {

signal: controller.signal

});

Now fetch listens to the signal.

* Step 3: to abort, call controller.abort(): We’re done: fetch gets the event from signal and aborts the request.

**Day 3 – Fetch: Cross-Origin requests**

If you make a fetch from an arbitrary website, that will probably fail. 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 remote side. That policy is called “CORS”: Cross-Origin Resource Sharing. For instance, try fetching <http://example.com>:

Try{

Await fetch(‘http://example.com’);

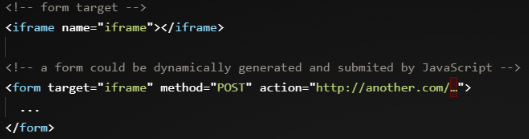
} catch(err) {

Alert(err); // failed to fetch

}

Fetch fails, as expected. Why? Because cross-origin restrictions protect the internet from evil hackers. For many years a script from one site could not access the content of another site. That simple, yet powerful rule was a foundation of internet security. E.g. a script from the page hacker.com could not access the user’s mailbox at gmail.com. People felt safe. JavaScript also did not have any special methods to perform network requests at that time. It was a toy language to decorate a web page. But web developers demanded more power. A variety of tricks were invented to work around the limitation.

**Using Forms -** One way to communicate with another server was to submit a <form> there. People submitted it into <iframe>, just to stay on the current page, like this:



So, it was possible to make a GET/POST request to another site, even without networking methods. But as it’s forbidden to access the content of an <iframe>from another site, it wasn’t possible to read the response. As we can see, forms allowed to send data anywhere, but not receive the response.

**Simple Requests –** There are two types of cross-domain requests: Simple requests and all the others. Simple requests are easier to make, so let’s start with them. A simple request is a request that satisfies two conditions:

* Simple method: GET, POST or HEAD
* Simple headers – the only allowed custom headers are: 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, a 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. So, even a very old server should be ready to accept a simple request.

Contrary to that, requests with non-standard headers or e.g. method DELETE can’t be created this way. For a long time, JavaScript was unable to do such requests. So an old server may assume that such requests come from a privileged source, “because a webpage is unable to send them”.

When we try to make a non-simple request, the browser sends a special “preflight” request that asks the server – does it agree to accept such cross-origin requests, or not? And, unless the server explicitly confirms that with headers, a non-simple request is not sent. Now we’ll go into details. All of them serve a single purpose – to ensure that new cross-origin capabilities are only accessible with explicit permission from the server.

**CORS for simple requests -** If a request is cross-origin, the browser always adds Origin header to it. For instance, if you request https://anywhere.com/request from https://javascript.info/page, the headers will be like:

GET /request

Host: anywhere.com

Origin: <https://javascript.info>

As you can see, Origin contains exactly the origin (domain/protocol/port), without a path. The server can inspect the Origin and, if it agrees to accept such a request, adds a special header Access-Control-Allow-Origin to the response. That header should contain the allowed origin (in our case https://javascript.info), or a star \*. Then the response is successful, otherwise an error. The browser plays the role of a trusted mediator here: It ensures that the correct Origin is sent with a cross-domain request. If checks for correct Access-Control-Allow-Origin in the response, if it is so, then JavaScript access, otherwise forbids with an error.

Here’s an example of a permissive server response:

200 OK

Content-Type:text/html; charset=UTF-8

Access-Control-Allow-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

Any other response header is forbidden.

Please note: there’s no Content-Length header in the list.

This header contains the full response length. So, if we’re downloading something and would like to track the percentage of progress, then additional permission is required to access that header (see below).

To grant JavaScript access to any other response header, the server must list it in the Access-Control-Expose-Headers header. For example:

200 OK

Content-Type:text/html; charset=UTF-8

Content-Length: 12345

API-Key: 2c9de507f2c54aa1

Access-Control-Allow-Origin: https://javascript.info

Access-Control-Expose-Headers: Content-Length,API-Key

With such Access-Control-Expose-Headersheader, the script is allowed to access Content-Length and API-Key headers of the response.

**Non-simple Requests -** We can use any HTTP-method: not just GET/POST, but also PATCH, DELETE and others. Some time ago no one could even assume that a webpage is able to do such requests. So there may exist web services that treat a non-standard method as a signal: “That’s not a browser”. They can take it into account when checking access rights. So, to avoid misunderstandings, any “non-simple” request – that couldn’t be done in the old times, the browser does not make such requests right away. Before it sends a preliminary, so-called “preflight” request, asking for permission.

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 it should respond with status 200, without body.

* The response header Access-Control-Allow-Methods must have the 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 send a preflight for subsequent requests that satisfy given permissions.

Let’s see how it works step-by-step on example, for a cross-domain PATCH request (this method is often used to update data):



There are three reasons why the request is not simple (one is enough):

Method PATCH

Content-Type is not one of: application/x-www-form-urlencoded, multipart/form-data, text/plain.

“Non-simple” API-Key header.

Step 1 (preflight request)

Prior to sending our request, the browser, on its own, sends a preflight request that looks like this:

OPTIONS /service.json

Host: site.com

Origin: https://javascript.info

Access-Control-Request-Method: PATCH

Access-Control-Request-Headers: Content-Type,API-Key

* Method: OPTIONS.
* The path – exactly the same as the main request: /service.json.
* Cross-origin special headers:
  + Origin – the source origin.
  + Access-Control-Request-Method – requested method.
  + Access-Control-Request-Headers – a comma-separated list of “non-simple” headers.

Step 2 (preflight response)

The server should respond with status 200 and headers:

* Access-Control-Allow-Methods: PATCH
* Access-Control-Allow-Headers: Content-Type,API-Key.

That allows future communication otherwise, an error is triggered.

If the server expects other methods and headers in the future, makes sense to allow them in advance by adding to the list:

200 OK

Access-Control-Allow-Methods: PUT,PATCH,DELETE

Access-Control-Allow-Headers: API-Key,Content-Type,If-Modified-Since,Cache-Control

Access-Control-Max-Age: 86400

Now the browser can see that PATCH is in the list of allowed methods, and both headers are in the list too, so it sends out the main request. Besides, the preflight response is cached for time, specified by Access-Control-Max-Age header (86400 seconds, one day), so subsequent requests will not cause a preflight. Assuming that they fit the allowances, they will be sent directly.

Step 3 (actual request)

When the preflight is successful, the browser now makes the real request. Here the flow is the same as for simple requests. The real request has Origin header (because it’s cross-origin):

PATCH /service.json

Host: site.com

Content-Type: application/json

API-Key: secret

Origin: <https://javascript.info>

Step 4 (actual response)

The server should not forget to add Access-Control-Allow-Origin to the response. A successful preflight does not relieve from that:

Access-Control-Allow-Origin: https://javascript.info

Now everything’s correct. JavaScript is able to read the full response. Please note:

Preflight request occurs “behind the scenes”, it’s invisible to JavaScript. JavaScript only gets the response to the main request or an error if there’s no server permission.

**Credentials -** A cross-origin request by default does not bring any credentials (cookies or HTTP authentication). That’s uncommon for HTTP-requests. Usually, 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. For example, fetch('http://another.com') 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 than an anonymous one. If allowed, it grants JavaScript the full power to act and access sensitive information on behalf of a user. Does the server really trust pages from Origin that much? Then it must explicitly allow requests with credentials with an additional header. To send credentials, we need to add the option credentials: "include", like this:

fetch('http://another.com', {

credentials: "include"

});

Now fetch sends cookies originating from another.com without request to that site. If the server wishes to accept the request with credentials, it should add a header Access-Control-Allow-Credentials: true to the response, in addition to Access-Control-Allow-Origin. For example:

200 OK

Access-Control-Allow-Origin: https://javascript.info

Access-Control-Allow-Credentials: true

Networking methods for cross-origin requests can be divided into two types: "simple" and others.

Simple requests must meet these conditions:

Method: GET, POST, or HEAD.

Headers: Only Accept, Accept-Language, Content-Language, and Content-Type with values of application/x-www-form-urlencoded, multipart/form-data, or text/plain.

Simple requests are sent immediately with an Origin header. Non-simple requests require a preliminary "preflight" request for permission.

For simple requests:

The browser sends the Origin header.

For requests without credentials, the server should set Access-Control-Allow-Origin to \* or the same value as Origin.

For requests with credentials, the server should set Access-Control-Allow-Origin to the same value as Origin and Access-Control-Allow-Credentials to true.

If JavaScript needs access to non-simple response headers, the server should list them in the Access-Control-Expose-Headers header.

For non-simple requests, a preflight OPTIONS request is sent before the actual request with:

Access-Control-Request-Method: Requested method.

Access-Control-Request-Headers: List of non-simple requested headers.

The server should respond with a status of 200 and headers:

Access-Control-Allow-Methods: List of allowed methods.

Access-Control-Allow-Headers: List of allowed headers.

Access-Control-Max-Age: Number of seconds to cache permissions.

Afterward, the actual request follows the "simple" scheme.

Why do we need Origin?

As you probably know, there’s HTTP-header Referer, that usually contains a URL of the page which initiated a network request. For instance, when fetching http://google.comfrom http://javascript.info/some/url, the headers look like this:

Accept: \*/\*

Accept-Charset: utf-8

Accept-Encoding: gzip,deflate,sdch

Connection: keep-alive

Host: google.com

Origin: http://javascript.info

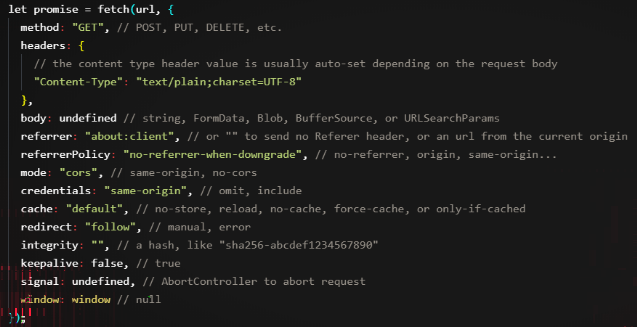
Referer: <http://javascript.info/some/url>

As you can see, both Referer and Origin are present.

Solutions:

We need Origin, because sometimes Referer is absent. For instance, when we fetch HTTP-page from HTTPS (access less secure from more secure), then there’s no Referer. The Content Security Policy may forbid sending a Referer. As we’ll see, fetch has options that prevent sending the Referer and even allow to change it (within the same site). By specification, Referer is an optional HTTP-header. Exactly because Referer is unreliable, Origin was invented. The browser guarantees correct Origin for cross-origin requests.

**Fetch API -** So far, we know quite a bit about fetch. Now let’s see the rest of API, to cover all its abilities. Here’s the full list of all possible fetch options with their default values (alternatives in comments):



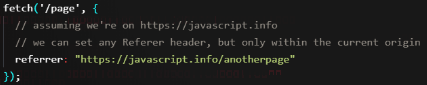
Referrer, referrerPolicy - These options govern how fetch sets HTTP Refererheader. That header contains the URL of the page that made the request. In most scenarios, it plays a very minor informational role, but sometimes, for security purposes, it makes sense to remove or shorten it. The referrer option allows to set any Refererwithin the current origin) or disable it.

To send no referrer, set an empty string:

A black background with white text

Description automatically generated

To set another URL within the current origin:



The referrerPolicy option sets general rules forReferer.

Possible values are described in the Referrer Policy specification:

* "no-referrer-when-downgrade" – default value: Referer is sent always, unless we send a request from HTTPS to HTTP (to less secure protocol).
* "no-referrer" – never send Referer.
* "origin" – only send the origin in Referer, not the full page URL, e.g. http://site.com instead of http://site.com/path.
* "origin-when-cross-origin" – send full Referer to the same origin, but only the origin part for cross-origin requests.
* "same-origin" – send full Referer to the same origin, but no referer for for cross-origin requests.
* "strict-origin" – send only origin, don’t send Referer for HTTPS→HTTP requests.
* "strict-origin-when-cross-origin" – for same-origin send full Referer, for cross-origin send only origin, unless it’s HTTPS→HTTP request, then send nothing.
* "unsafe-url" – always send full url in Referer.

To control cross-origin requests with the Fetch API:

Referrer Handling:

* To hide the referrer, use referrerPolicy: "no-referrer".
* To reveal only the origin, use referrerPolicy: "strict-origin".

Mode:

Use the mode option:

* "cors" (default) allows cross-origin requests.
* "same-origin" forbids cross-origin requests.
* "no-cors" allows only simple cross-origin requests.

Credentials:

Choose from:

* "same-origin" (default) for no credentials sent in cross-origin requests.
* "include" to send credentials, requiring Accept-Control-Allow-Credentials from the server.
* "omit" to never send credentials, even for same-origin requests.

Cache:

Configure caching with the cache option:

* "default" (default) follows standard HTTP-caching rules.
* "no-store" ignores HTTP-cache entirely.
* "reload" fetches a fresh response but populates the cache.

"no-cache" conditions the request based on cached responses.

"force-cache" uses a stale cache if available.

"only-if-cached" uses a cached response, failing if unavailable (works in "same-origin" mode).

Redirects:

Control how fetch handles HTTP-redirects:

"follow" (default) follows redirects.

"error" throws an error on redirects.

"manual" allows manual handling of redirects.

Integrity:

Ensure response integrity by specifying a checksum using the integrity option (e.g., SHA-256).

Keepalive:

Enable requests to outlive the page with the keepalive option.

Useful for background tasks but limited to a 64kb request body.

Multiple requests can circumvent the limit.

Note that the server may send empty responses for onunload requests.

These options provide fine-grained control over Fetch API behavior for cross-origin requests.

**Day 4 – Patterns and flags**

A regular expression (also “regexp”, or just “reg”) consists of a pattern and optional flags.

There are two syntaxes to create a regular expression object.

The long syntax:

regexp = new RegExp("pattern", "flags");

…And the short one, using slashes "/":

regexp = /pattern/; // no flags

regexp = /pattern/gmi; // with flags g,m and i

Slashes "/" tell JavaScript that we are creating a regular expression. They play the same role as quotes for strings.

**Usage -** To search inside a string, we can use method search.

Here’s an example:

let str = "I love JavaScript!"; // will search here

let regexp = /love/;

alert( str.search(regexp) ); // 2

The str.search method looks for the pattern /love/ and returns the position inside the string. As we might guess, /love/is the simplest possible pattern. What it does is a simple substring search.

The code above is the same as:

let str = "I love JavaScript!"; // will search here

let substr = 'love';

alert( str.search(substr) ); // 2

So searching for /love/ is the same as searching for "love".

But that’s only for now. Soon we’ll create more complex regular expressions with much more searching power.

Colors

From here on the color scheme is:

regexp – red

string (where we search) – blue

result – green

When to use new RegExp?

Normally we use the short syntax /.../. But it does not support variable insertions ${...}.

On the other hand, new RegExp allows to construct a pattern dynamically from a string, so it’s more flexible.

Here’s an example of a dynamically generated regexp:

let tag = prompt("Which tag you want to search?", "h2");

let regexp = new RegExp(< ${tag} >);

// finds <h2> by default

alert( "<h1> <h2> <h3>".search(regexp));

**Flags -** Regular expressions may have flags that affect the search.

There are only 6 of them in JavaScript:

i

With this flag the search is case-insensitive: no difference between A and a (see the example below).

g

With this flag the search looks for all matches, without it – only the first one

m

Multiline mode.

s

“Dotall” mode, allows . to match newlines.

u

Enables full unicode support. The flag enables correct processing of surrogate pairs.

y

Sticky mode.