# Lesson - 25

Fetch


**SkillUp by Dobrea Vladislav**

# Lesson Plan

- Fetch
- Fetch API
- Fetch Cross-Origin Requests
- URL Object
- XMLHttpRequest

# Fetch

JavaScript can send network requests to the server and load new information whenever it's needed.

For example, we can use a network request to:

- Submit an order,
- Load user information,
- Receive latest updates from the server,
- …etc.

…And all of that without reloading the page!

There's an umbrella term "AJAX" (abbreviated **A**synchronous **J**avaScript **A**nd **X**ML) for network requests from JavaScript. We don't have to use XML though: the term comes from old times, that's why that word is there. You may have heard that term already.

There are multiple ways to send a network request and get information from the server.

# Fetch

The `fetch()` method is modern and versatile, so we'll start with it. It's not supported by old browsers (can be polyfilled), but very well supported among the modern ones.

The basic syntax is:

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

- **`url`** – the URL to access.
- **`options`** – optional parameters: method, headers etc.

Without `options`, that is a simple GET request, downloading the contents of the `url`.

The browser starts the request right away and returns a promise that the calling code should use to get the result.

# Response

Getting a response is usually a two-stage process.

**First, the `promise`, returned by `fetch`, resolves with an object of the built-in Response class as soon as the server responds with headers.**

At this stage 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. Abnormal HTTP-statuses, such as 404 or 500 do not cause an error.

We can see HTTP-status in response properties:

- **`status`** – HTTP status code, e.g. 200.
- **`ok`** – boolean, `true` if the HTTP status code is 200-299.

For example:

```
let response = await fetch(url);

if (response.ok) { // if HTTP-status is 200-299
  // get the response body (the method explained below)
  let json = await response.json();
} else {
  alert("HTTP-Error: " + response.status);
}
```

# Response

**Second, to get the response body, we need to use an additional method call.**

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

- **`response.text()`** – read the response and return as text,
- **`response.json()`** – parse the response as JSON,
- **`response.formData()`** – return the response as `FormData` object (explained in the next chapter),
- **`response.blob()`** – return the response as Blob (binary data with type),
- **`response.arrayBuffer()`** – return the response as ArrayBuffer (low-level representaion of binary data),
- additionally, `response.body` is a ReadableStream object, it allows you to read the body chunk-by-chunk, we'll see an example later.

For instance, let's get a JSON-object with latest commits from GitHub:

```
1 fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/co
2   .then(response => response.json())
3   .then(commits => alert(commits[0].author.login));
```

# Response

⚠️ **Important:**

We can choose only one body-reading method.

If we've already got the response with `response.text()`, then `response.json()` won't work, as the body content has already been processed.

```
let text = await response.text(); // response body consumed
let parsed = await response.json(); // fails (already consumed)
```

# Response Headers

## Response headers

The response headers are available in a Map-like headers object in `response.headers`.

It's not exactly a Map, but it has similar methods to get individual headers by name or iterate over them:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/
```

```
// get one header
alert(response.headers.get('Content-Type')); // application/json; charset=utf-

// iterate over all headers
for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

# Request Headers

To set a request header in `fetch`, we can use the `headers` option. It has an object with outgoing headers, like this:

```
let response = fetch(protectedUrl, {
  headers: {
    Authentication: 'secret'
  }
});
```

...But there's a list of forbidden HTTP headers that we can't set:

- `Accept-Charset`, `Accept-Encoding`
- `Access-Control-Request-Headers`
- `Access-Control-Request-Method`
- `Connection`
- `Content-Length`
- `Cookie`, `Cookie2`
- `Date`

# POST Request

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

- **`method`** – HTTP-method, e.g. `POST`,
- **`body`** – the request body, one of:
  - a string (e.g. JSON-encoded),
  - `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.

The JSON format is used most of the time.

For example, this code submits `user` object as JSON:

```
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);
```

# Summary

```
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.text()`** – return the response as text,
- **`response.json()`** – parse the response as JSON object,
- **`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 (low-level binary data),

Fetch options so far:

- `method` – HTTP-method,
- `headers` – an object with request headers (not any header is allowed),
- `body` – the data to send (request body) as `string`, `FormData`, `BufferSource`, `Blob` or `UrlSearchParams` object.

# Fetch: Cross Origin requests

If we send a `fetch` request to another web-site, it will probably fail.

For instance, let's try fetching `http://example.com`:

```
try {
  await fetch('http://example.com');
} catch(err) {
  alert(err); // Failed to fetch
}
```

Fetch fails, as expected.

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.

# Why is CORS needed?

CORS exists to protect the internet from evil hackers.

Seriously. Let's make a very brief historical digression.

**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 the internet security. E.g. an evil script from website `hacker.com` could not access user's mailbox at website `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 and make requests to other websites.

# 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:

```
<!-- form target -->
<iframe name="iframe"></iframe>

<!-- a form could be dynamically generated and submited by JavaScript -->
<form target="iframe" method="POST" action="http://another.com/…">
  ...
</form>
```

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

To be precise, there were actually tricks for that, they required special scripts at both the iframe and the page. So the communication with the iframe was technically possible. Right now there's no point to go into details, let these dinosaurs rest in peace.

# Using Scripts

Another trick was to use a `script` tag. A script could have any `src`, with any domain, like `<script src="http://another.com/…">`. It's possible to execute a script from any website.

If a website, e.g. `another.com` intended to expose data for this kind of access, then a so-called "JSONP (JSON with padding)" protocol was used.

Here's how it worked.

Let's say we, at our site, need to get the data from `http://another.com`, such as the weather:

1. First, in advance, we declare a global function to accept the data, e.g. `gotWeather`.

```
// 1. Declare the function to process the weather data
function gotWeather({ temperature, humidity }) {
  alert(`temperature: ${temperature}, humidity: ${humidity}`);
}
```

2. Then we make a `<script>` tag with `src="http://another.com/weather.json?callback=gotWeather"`, using the name of our function as the `callback` URL-parameter.

```
let script = document.createElement('script');
script.src = `http://another.com/weather.json?callback=gotWeather`;
document.body.append(script);
```

# Using Scripts

3. The remote server `another.com` dynamically generates a script that calls `gotWeather(...)` with the data it wants us to receive.

```
// The expected answer from the server looks like this:
gotWeather({
  temperature: 25,
  humidity: 78
});
```

4. When the remote script loads and executes, `gotWeather` runs, and, as it's our function, we have the data.

That works, and doesn't violate security, because both sides agreed to pass the data this way. And, when both sides agree, it's definitely not a hack. There are still services that provide such access, as it works even for very old browsers.

After a while, networking methods appeared in browser JavaScript.

At first, cross-origin requests were forbidden. But as a result of long discussions, cross-origin requests were allowed, but with any new capabilities requiring an explicit allowance by the server, expressed in special headers.

# Simple Requests

Simple Requests are, well, simpler to make, so let's start with them.

A simple request is a request that satisfies two conditions:

1. Simple method: GET, POST or HEAD
2. 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?

# CORS for simple requests

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

For instance, 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
...
```

As you can see, `Origin` header 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:

1. It ensures that the correct `Origin` is sent with a cross-origin request.
2. It checks for permitting `Access-Control-Allow-Origin` in the response, if it exists, then JavaScript is allowed to access the response, otherwise it fails with an error.

# Response Headers

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

- `Cache-Control`
- `Content-Language`
- `Content-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

Accessing any other response header causes an error.

> **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 an additional permission is required to access that header (see below).

To grant JavaScript access to any other response header, the server must send `Access-Control-Expose-Headers` header. It contains a comma-separated list of non-simple header names that should be made accessible.

# 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 imagine that a webpage could make such requests. So there may still exist webservices 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`, no body and two headers:

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

If the server agrees to serve the requests, then it should respond with empty body, status 200 and headers:

- `Access-Control-Allow-Methods` must have the allowed method.
- `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-origin `PATCH` request (this method is often used to update data):

```
let response = await fetch('https://site.com/service.json', {
  method: 'PATCH',
  headers: {
    'Content-Type': 'application/json',
    'API-Key': 'secret'
  }
});
```

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 such 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, it 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-Co
Access-Control-Max-Age: 86400
```

Now the browser can see that `PATCH` is in `Access-Control-Allow-Methods` and `Content-Type,API-Key` are in the list `Access-Control-Allow-Headers`, 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 cached allowances, they will be sent directly.

### Step 3 (actual request)

When the preflight is successful, the browser now makes the main request. The algorithm here is the same as for simple requests.

The main 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 main response. A successful preflight does not relieve from that:

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

Then JavaScript is able to read the main server 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 initiated by JavaScript code 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-origin 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 without them. If allowed, it grants JavaScript the full power to act on behalf of the user and access sensitive information using their credentials.

Does the server really trust the script that much? Then it must explicitly allow requests with credentials with an additional header.

To send credentials in `fetch`, 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 agrees 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`.

From the browser point of view, there are two kinds of cross-origin requests: “simple” and all the others.

Simple requests must satisfy the following conditions:

- Method: GET, POST or HEAD.
- Headers – we can set only:
  - `Accept`
  - `Accept-Language`
  - `Content-Language`
  - `Content-Type` to the value `application/x-www-form-urlencoded`, `multipart/form-data` or `text/plain`.

The essential difference is that simple requests were doable since ancient times using `<form>` or `<script>` tags, while non-simple were impossible for browsers for a long time.

So, the practical difference is that simple requests are sent right away, with `Origin` header, while for the other ones the browser makes a preliminary “preflight” request, asking for permission.

**For simple requests:**

- → The browser sends `Origin` header with the origin.
- ← For requests without credentials (not sent default), the server should set:
  - `Access-Control-Allow-Origin` to `*` or same value as `Origin`
- ← For requests with credentials, the server should set:
  - `Access-Control-Allow-Origin` to same value as `Origin`
  - `Access-Control-Allow-Credentials` to `true`

Additionally, to grant JavaScript access to any response headers except `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified` or `Pragma`, the server should list the allowed ones in `Access-Control-Expose-Headers` header.

**For non-simple requests, a preliminary “preflight” request is issued before the requested one:**

- → The browser sends `OPTIONS` request to the same URL, with headers:
  - `Access-Control-Request-Method` has requested method.
  - `Access-Control-Request-Headers` lists non-simple requested headers.
- ← The server should respond with status 200 and headers:
  - `Access-Control-Allow-Methods` with a list of allowed methods,
  - `Access-Control-Allow-Headers` with a list of allowed headers,
  - `Access-Control-Max-Age` with a number of seconds to cache permissions.
- Then the actual request is sent, the previous “simple” scheme is applied.

# Fetch API

So far, we know quite a bit about `fetch`.

Let's see the rest of API, to cover all its abilities.

**Please note:**

Please note: most of these options are used rarely. You may skip this chapter and still use `fetch` well.

Still, it's good to know what `fetch` can do, so if the need arises, you can return and read the details.

# Fetch API

Here's the full list of all possible `fetch` options with their default values (alternatives in comments):

```
let promise = fetch(url, {
  method: "GET", // POST, PUT, DELETE, etc.
  headers: {
    // the content type header value is usually auto-set
    // depending on the request body
    "Content-Type": "text/plain;charset=UTF-8"
  },
  body: undefined // string, FormData, Blob, BufferSource, or URLSearchParams
  referrer: "about:client", // or "" to send no Referer header,
  // or an url from the current origin
  referrerPolicy: "no-referrer-when-downgrade", // no-referrer, origin, same-
  mode: "cors", // same-origin, no-cors
  credentials: "same-origin", // omit, include
  cache: "default", // no-store, reload, no-cache, force-cache, or only-if-ca
  redirect: "follow", // manual, error
  integrity: "", // a hash, like "sha256-abcdef1234567890"
  keepalive: false, // true
  signal: undefined, // AbortController to abort request
  window: window // null
});
```

# Fetch API

## mode

The `mode` option is a safe-guard that prevents occasional cross-origin requests:

- `"cors"` – the default, cross-origin requests are allowed, as described in Fetch: Cross-Origin Requests,
- `"same-origin"` – cross-origin requests are forbidden,
- `"no-cors"` – only simple cross-origin requests are allowed.

This option may be useful when the URL for `fetch` comes from a 3rd-party, and we want a "power off switch" to limit cross-origin capabilities.

## credentials

The `credentials` option specifies whether `fetch` should send cookies and HTTP-Authorization headers with the request.

- `"same-origin"` – the default, don't send for cross-origin requests,
- `"include"` – always send, requires `Accept-Control-Allow-Credentials` from cross-origin server in order for JavaScript to access the response, that was covered in the chapter Fetch: Cross-Origin Requests,
- `"omit"` – never send, even for same-origin requests.

# Fetch API

## cache

By default, `fetch` requests make use of standard HTTP-caching. That is, it honors `Expires`, `Cache-Control` headers, sends `If-Modified-Since`, and so on. Just like regular HTTP-requests do.

The `cache` options allows to ignore HTTP-cache or fine-tune its usage:

- **`"default"`** – `fetch` uses standard HTTP-cache rules and headers,
- **`"no-store"`** – totally ignore HTTP-cache, this mode becomes the default if we set a header `If-Modified-Since`, `If-None-Match`, `If-Unmodified-Since`, `If-Match`, or `If-Range`,
- **`"reload"`** – don't take the result from HTTP-cache (if any), but populate cache with the response (if response headers allow),
- **`"no-cache"`** – create a conditional request if there is a cached response, and a normal request otherwise. Populate HTTP-cache with the response,
- **`"force-cache"`** – use a response from HTTP-cache, even if it's stale. If there's no response in HTTP-cache, make a regular HTTP-request, behave normally,
- **`"only-if-cached"`** – use a response from HTTP-cache, even if it's stale. If there's no response in HTTP-cache, then error. Only works when `mode` is `"same-origin"`.

## redirect

Normally, `fetch` transparently follows HTTP-redirects, like 301, 302 etc.

The `redirect` option allows to change that:

- **`"follow"`** – the default, follow HTTP-redirects,
- **`"error"`** – error in case of HTTP-redirect,
- **`"manual"`** – don't follow HTTP-redirect, but `response.url` will be the new URL, and `response.redirected` will be `true`, so that we can perform the redirect manually to the new URL (if needed).

# Fetch API

## keepalive

The `keepalive` option indicates that the request may “outlive” the webpage that initiated it.

For example, we gather statistics about how the current visitor uses our page (mouse clicks, page fragments he views), to analyze and improve user experience.

When the visitor leaves our page – we’d like to save the data at our server.

We can use `window.onunload` event for that:

```
window.onunload = function() {
  fetch('/analytics', {
    method: 'POST',
    body: "statistics",
    keepalive: true
  });
};
```

Normally, when a document is unloaded, all associated network requests are aborted. But `keepalive` option tells the browser to perform the request in background, even after it leaves the page. So this option is essential for our request to succeed.

It has few limitations:

- We can’t send megabytes: the body limit for `keepalive` requests is 64kb.
  - If gather more data, we can send it out regularly in packets, so that there won’t be a lot left for the last `onunload` request.
  - The limit is for all currently ongoing requests. So we can’t cheat it by creating 100 requests, each 64kb.
- We can’t handle the server response if the request is made in `onunload`, because the document is already unloaded at that time, functions won’t work.
  - Usually, the server sends empty response to such requests, so it’s not a problem.

# URL Objects

The built-in URL class provides a convenient interface for creating and parsing URLs.

There are no networking methods that require exactly a `URL` object, strings are good enough. So technically we don't have to use `URL`. But sometimes it can be really helpful.

## Creating a URL

The syntax to create a new `URL` object:

```
new URL(url, [base])
```

- **`url`** – the full URL or only path (if base is set, see below),
- **`base`** – an optional base URL: if set and `url` argument has only path, then the URL is generated relative to `base`.

For example:

```
let url = new URL('https://javascript.info/profile/admin');
```

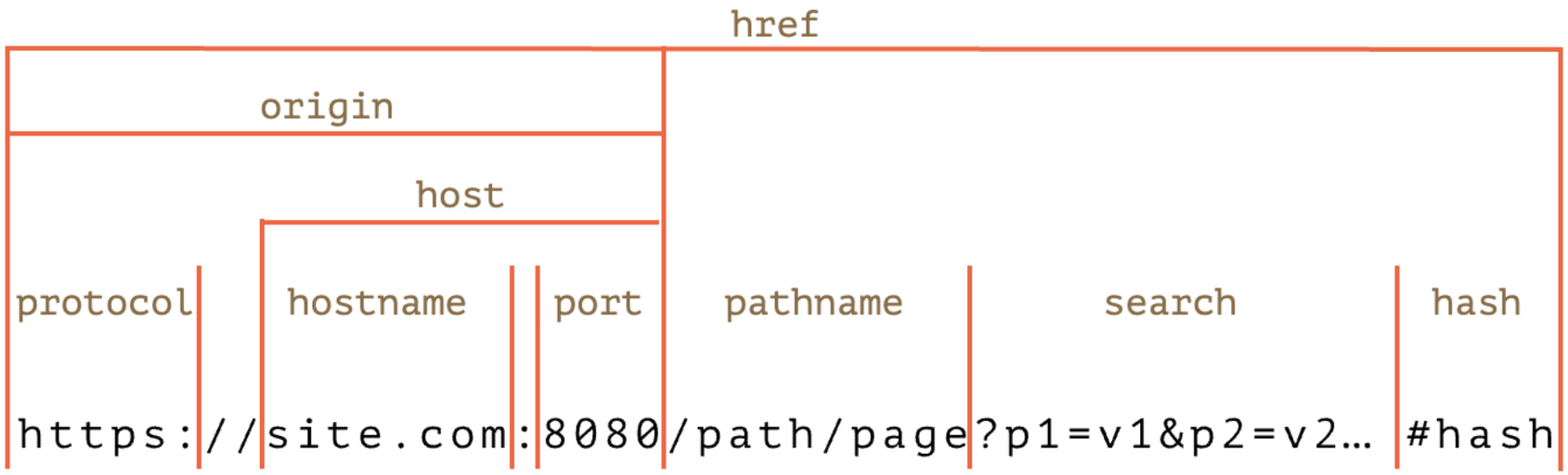
These two URLs are same:

```
let url1 = new URL('https://javascript.info/profile/admin');
let url2 = new URL('/profile/admin', 'https://javascript.info');

alert(url1); // https://javascript.info/profile/admin
alert(url2); // https://javascript.info/profile/admin
```

# URL

Here's the cheatsheet for URL components:




- `href` is the full url, same as `url.toString()`
- `protocol` ends with the colon character `:`
- `search` – a string of parameters, starts with the question mark `?`
- `hash` starts with the hash character `#`
- there may be also `user` and `password` properties if HTTP authentication is present: `http://login:password@site.com` (not painted above, rarely used).

# Search Params

Let's say we want to create a url with given search params, for instance, `https://google.com/search?query=JavaScript`.

We can provide them in the URL string:

```
new URL('https://google.com/search?query=JavaScript')
```

...But parameters need to be encoded if they contain spaces, non-latin letters, etc (more about that below).

So there's URL property for that: `url.searchParams`, an object of type URLSearchParams.

It provides convenient methods for search parameters:

- **`append(name, value)`** – add the parameter by `name`,
- **`delete(name)`** – remove the parameter by `name`,
- **`get(name)`** – get the parameter by `name`,
- **`getAll(name)`** – get all parameters with the same `name` (that's possible, e.g. `?user=John&user=Pete`),
- **`has(name)`** – check for the existance of the parameter by `name`,
- **`set(name, value)`** – set/replace the parameter,
- **`sort()`** – sort parameters by name, rarely needed,
- ...and it's also iterable, similar to `Map`.

# Encoding

There's a standard RFC3986 that defines which characters are allowed in URLs and which are not.

Those that are not allowed, must be encoded, for instance non-latin letters and spaces – replaced with their UTF-8 codes, prefixed by `%`, such as `%20` (a space can be encoded by `+`, for historical reasons, but that's an exception).

The good news is that `URL` objects handle all that automatically. We just supply all parameters unencoded, and then convert the `URL` to string:

```
// using some cyrillic characters for this example

let url = new URL('https://ru.wikipedia.org/wiki/Тест');

url.searchParams.set('key', 'ъ');
alert(url); //https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82?key=%D1%8
```

As you can see, both `Тест` in the url path and `ъ` in the parameter are encoded.

The URL became longer, because each cyrillic letter is represented with two bytes in UTF-8, so there are two `%..` entities.

# Encoding strings

In old times, before `URL` objects appeared, people used strings for URLs.

As of now, `URL` objects are often more convenient, but strings can still be used as well. In many cases using a string makes the code shorter.

If we use a string though, we need to encode/decode special characters manually.

There are built-in functions for that:

- encodeURI – encodes URL as a whole.
- decodeURI – decodes it back.
- encodeURIComponent – encodes a URL component, such as a search parameter, or a hash, or a pathname.
- decodeURIComponent – decodes it back.

A natural question is: "What's the difference between `encodeURIComponent` and `encodeURI`? When we should use either?"

That's easy to understand if we look at the URL, that's split into components in the picture above:

```
https://site.com:8080/path/page?p1=v1&p2=v2#hash
```

As we can see, characters such as `:`, `?`, `=`, `&`, `#` are allowed in URL.

...On the other hand, if we look at a single URL component, such as a search parameter, these characters must be encoded, not to break the formatting.

- `encodeURI` encodes only characters that are totally forbidden in URL.
- `encodeURIComponent` encodes same characters, and, in addition to them, characters `#`, `$`, `&`, `+`, `,`, `/`, `:`, `;`, `=`, `?` and `@`.

# XMLHttpRequest

`XMLHttpRequest` is a built-in browser object that allows to make HTTP requests in JavaScript.

Despite of having the word "XML" in its name, it can operate on any data, not only in XML format. We can upload/download files, track progress and much more.

Right now, there's another, more modern method `fetch`, that somewhat deprecates `XMLHttpRequest`.

In modern web-development `XMLHttpRequest` is used for three reasons:

1. Historical reasons: we need to support existing scripts with `XMLHttpRequest`.
2. We need to support old browsers, and don't want polyfills (e.g. to keep scripts tiny).
3. We need something that `fetch` can't do yet, e.g. to track upload progress.

Does that sound familiar? If yes, then all right, go on with `XMLHttpRequest`. Otherwise, please head on to Fetch.

# The Basics

XMLHttpRequest has two modes of operation: synchronous and asynchronous.

Let's see the asynchronous first, as it's used in the majority of cases.

To do the request, we need 3 steps:

1. Create `XMLHttpRequest`:

```
let xhr = new XMLHttpRequest();
```

   The constructor has no arguments.

2. Initialize it, usually right after `new XMLHttpRequest`:

```
xhr.open(method, URL, [async, user, password])
```

   This method specifies the main parameters of the request:

   - `method` – HTTP-method. Usually `"GET"` or `"POST"`.
   - `URL` – the URL to request, a string, can be URL object.
   - `async` – if explicitly set to `false`, then the request is synchronous, we'll cover that a bit later.
   - `user`, `password` – login and password for basic HTTP auth (if required).

   Please note that `open` call, contrary to its name, does not open the connection. It only configures the request, but the network activity only starts with the call of `send`.

# The Basics

3. Send it out.

```
xhr.send([body])
```

This method opens the connection and sends the request to server. The optional `body` parameter contains the request body.

Some request methods like `GET` do not have a body. And some of them like `POST` use `body` to send the data to the server. We'll see examples of that later.

4. Listen to `xhr` events for response.

   These three events are the most widely used:

   - `load` – when the request is complete (even if HTTP status is like 400 or 500), and the response is fully downloaded.
   - `error` – when the request couldn't be made, e.g. network down or invalid URL.
   - `progress` – triggers periodically while the response is being downloaded, reports how much has been downloaded.

# Response Type

We can use `xhr.responseType` property to set the response format:

- `""` (default) – get as string,
- `"text"` – get as string,
- `"arraybuffer"` – get as `ArrayBuffer` (for binary data, see chapter ArrayBuffer, binary arrays),
- `"blob"` – get as `Blob` (for binary data, see chapter Blob),
- `"document"` – get as XML document (can use XPath and other XML methods),
- `"json"` – get as JSON (parsed automatically).

For example, let's get the response as JSON:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/example/json');

xhr.responseType = 'json';

xhr.send();

// the response is {"message": "Hello, world!"}
xhr.onload = function() {
  let responseObj = xhr.response;
  alert(responseObj.message); // Hello, world!
};
```

# Ready States

`XMLHttpRequest` changes between states as it progresses. The current state is accessible as `xhr.readyState`.

All states, as in the specification:

```
UNSENT = 0; // initial state
OPENED = 1; // open called
HEADERS_RECEIVED = 2; // response headers received
LOADING = 3; // response is loading (a data packed is received)
DONE = 4; // request complete
```

An `XMLHttpRequest` object travels them in the order `0` → `1` → `2` → `3` → ... → `3` → `4`. State `3` repeats every time a data packet is received over the network.

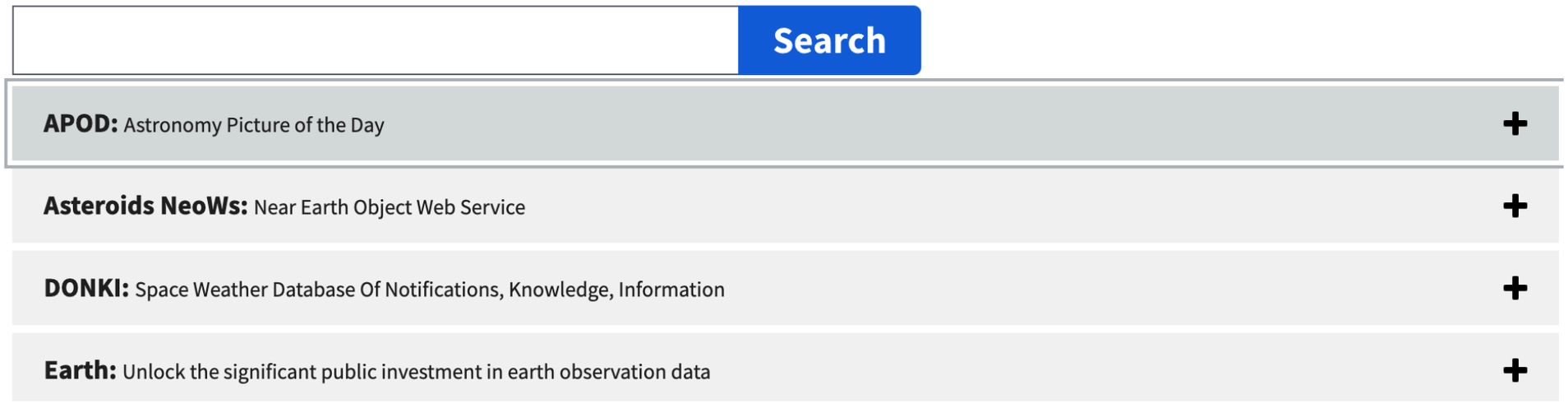
We can track them using `readystatechange` event:

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 3) {
    // loading
  }
  if (xhr.readyState == 4) {
    // request finished
  }
};
```

# Picture of Day Project

Using open API from NASA, we will build a project using it's resources

1. Go to https://api.nasa.gov/
2. Find api in list



# Picture of Day Project

Using open API from NASA, we will build a project using it’s resources

## 3. Check it’s details

### HTTP Request

`GET https://api.nasa.gov/planetary/apod`

**concept_tags are now disabled in this service. Also, an optional return parameter *copyright* is returned if the image is not public domain.**

## Query Parameters

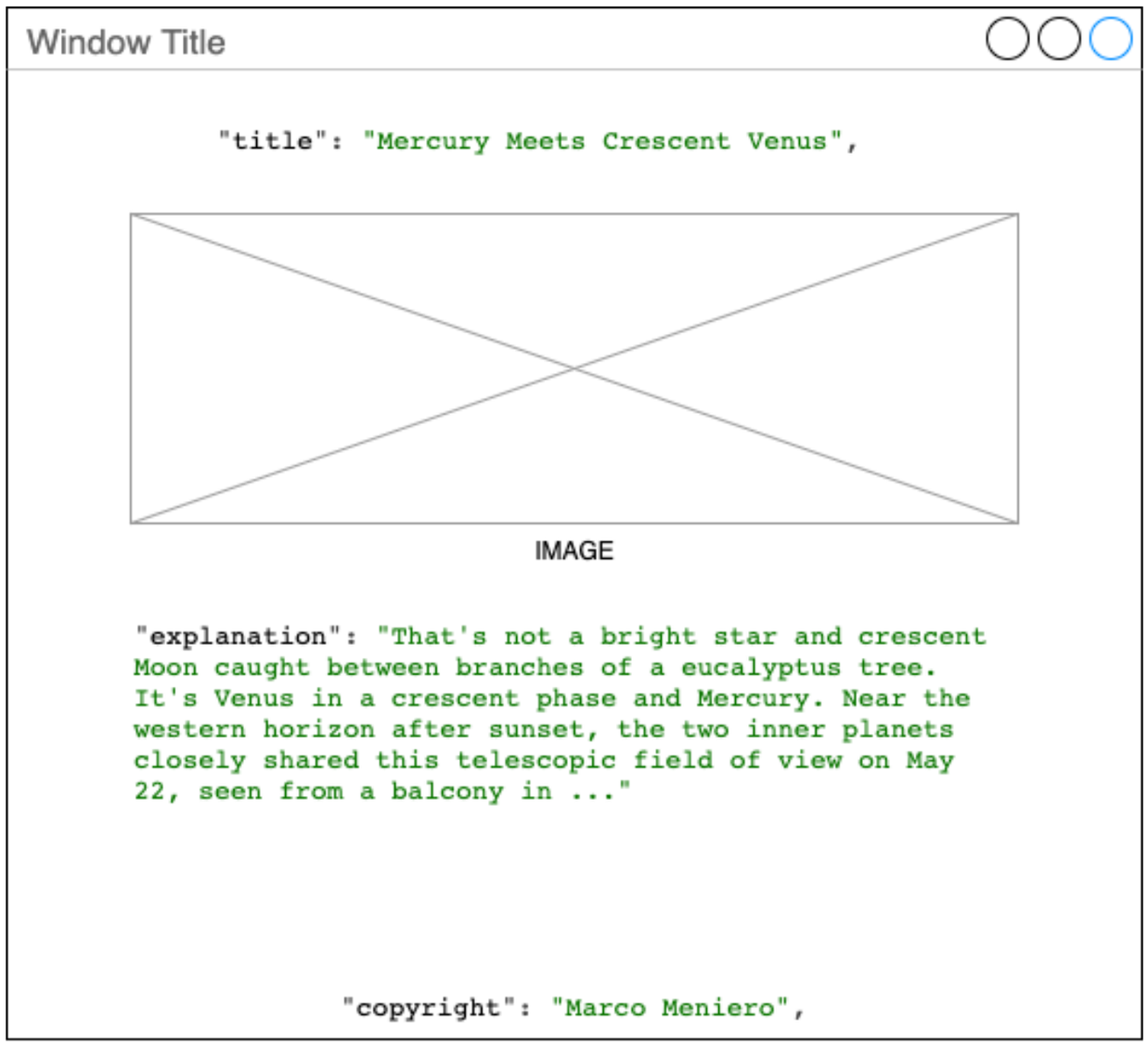
| Parameter | Type | Default | Description |
|---|---|---|---|
| date | YYYY-MM-DD | *today* | The date of the APOD image to retrieve |
| hd | bool | False | Retrieve the URL for the high resolution image |
| api_key | string | DEMO_KEY | api.nasa.gov key for expanded usage |

## Example query

`https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY`

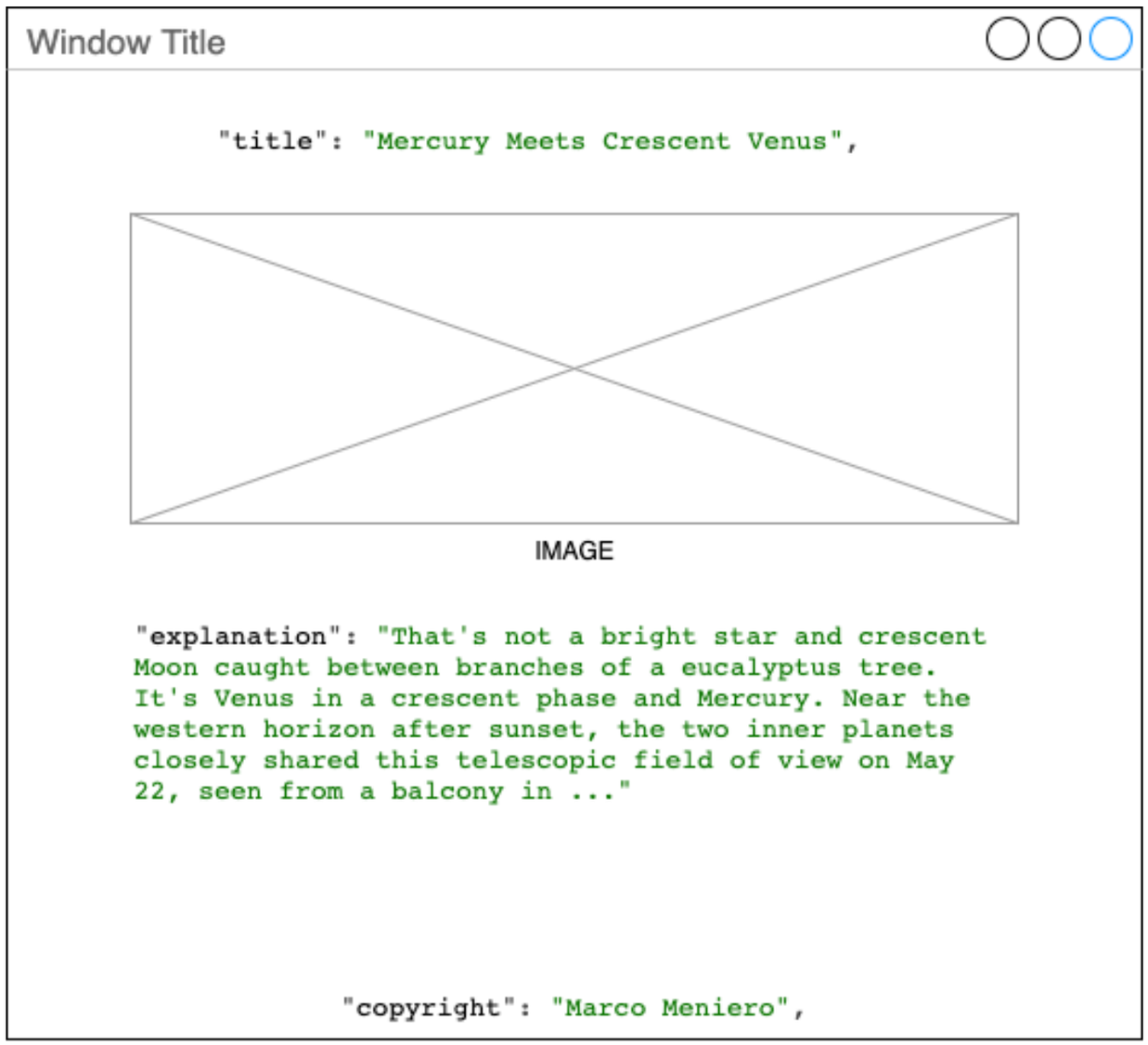
# Home Work

**Using open API from NASA, create a web page that will display the data that you get from API.**



# Home Work *

Additional, not mandatory, you can try to get data using XMLHttpRequest instead of fetch()



# Learning Resources

1. **Intro to fetch: https://javascript.info/fetch**
2. **Cross Origin: https://javascript.info/fetch-crossorigin**
3. **Fetch API: https://javascript.info/fetch-api**
4. **Working with URL: https://javascript.info/url**
5. **Old method XMLHttpRequest: https://javascript.info/xmlhttprequest**