JS Fetch

Learning Objectives

- [ ] Understanding the fetch API
  - [ ] with async/await
  - [ ] JSON
  - [ ] HTTP Response Codes
  - [ ] REST API
  - [ ] Error Handling

What is an API?

The term *API* is short for *Application Programming Interface*.

An *API* provides a way one software (the *application*) can interact with another software. Therefore, an application can define a set of features and rules on how other software can interact with it. This is called an *interface*. Think of a contract between two softwares that explains how they can work together.

APIs can be seen from different perspectives and occur on various levels.

A browser provides a lot of Web APIs. Each Web API defines a way on how a JavaScript application can use a feature given by the browser, like:

- Web Animations API
- Battery API
- Fetch API

APIs running on a server environment are a different type of API. They are provided by a *server*, opposing to the APIs provided by the browser (which is also called the *client*). A common use-case for such APIs is to read / load data. Other operations like writing or deleting data is also possible. There are common approaches regarding the architecture of a serve-side API. One such approach are REST-APIs, which is explained later in this document.

Fetch API (with async & await)

`fetch` is a web API to asynchronously fetch (load) resources from the network, like text documents.

```
async function fetchData() {
  const response = await fetch("/url/to/something");
  const data = await response.json();
  return data;
}
```

This is what happens in the example above:

1. We mark the function with the `async` keyword, because we want to use `await` inside the function.
2. We declare a variable named `response`. It stores the Response object that is returned by `fetch`.
3. Once this promise resolves (the network request is finished), we call the `.json` method on the `response` variable. This function returns another Promise.
4. This second promise resolves with the actual data (payload) converted from JSON (a formatted string) to a JavaScript value or object. This result is stored in the variable named `data`.
5. The function returns the value stored in the `data` variable.

> 💡 The `async` and `await` syntax helps with handling the two consecutive promises required to get the response data (opposing to the nested syntax using `.then()`)

> 💡 The `Response` object features other useful methods (all returning promises), among which are the following:
>
> - `response.json()` returns a Promise that resolves to the downloaded data JSON-parsed as a JavaScript value or object
> - `response.text()` returns a Promise that resolves to the downloaded data as raw text
> - `response.formData()` returns a Promise that resolves to the downloaded data parsed as FormData
> - `response.blob()` returns a Promise that resolves to the downloaded data as a raw blob (that's a machine readable format using zeros and ones)

JSON

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for transmitting data between a client (browser) and a server in web applications.

JSON is very close to being a subset of JavaScript syntax. Most valid JSON is also valid JavaScript code. This means that you can copy JSON into any `.js` file, put a `const myData =` in front, and watch Prettier make it look like *real* JavaScript. On the flip side valid JavaScript is not valid JSON. It looks like this:

```
{
  "groupName": "Students",
  "groupSize": 100,
  "students": [
    {
      "name": "John Doe",
      "age": 42,
      "location": "Pleasantville",
      "member": true,
      "groups": ["students", "citizens", "new"]
    },
    {
      "name": "Jane Doe",
      "age": 44,
      "location": "Pleasantville",
      "member": true,
      "groups": ["students", "citizens", "new"]
    },
    {
      "name": "Sam Doe",
      "age": 24,
      "location": "Pleasantville",
      "member": true,
      "groups": ["students", "citizens", "new"]
    }
  ]
}
```

Even though it closely resembles JavaScript object literal syntax, it can be used outside of JavaScript. Other programming languages often feature the ability to read (parse) JSON.

JSON exists as a string that needs to be converted to a native JavaScript object when you want to access the data. No problem! - JavaScript provides a global JSON object that features conversion methods in either direction.

> 💡 JSON Conversion Methods
>
> `JSON.parse()`
>
> This method parses a JSON string and constructs the JavaScript value or object described by the string.
>
> `JSON.stringify()`
>
> This methods converts a JavaScript value or object to a JSON string.

HTTP Response Status Codes

HTTP response status codes usually indicate whether a specific HTTP request has been successfully completed. For the most part, *any* sort of response is considered a successful completion of the request.

With the exception of custom server software, these responses are standardized. You can find a list of HTTP response status codes here.

One of the most well-known HTTP response status codes is `404 not found`, which most users of the web might have already encountered at some point.

> 💡 Do you need a cute explanation of what the HTTP response codes mean? Just search for 'HTTP status dogs' (or cats if you like them better).

Error Handling

It might be surprising that `fetch()` doesn't throw an error when the server returns a bad HTTP status, e.g. client or server errors.

```
async function fetchSomething() {
  const response = await fetch("/bad/url/oops");
  const something = await response.json();
  return something;
}
```

Assuming that in the above example `'bad/url/oops` doesn't lead to an existing location, the server would respond with the status code `404` and the text `Page not found`. This is a **completed** HTTP request.

A request will only register as rejected if a response cannot be retrieved. This may be due to network problems, such as no internet connection, the host not being found, or the server not responding.

We can separate *good* from *bad* HTTP response statuses by relying on the boolean `response.ok` property. It is set to `true` only if the HTTP response code is between `200-299`.

In our example above, `response.ok` would be set to `false` and the response code would be `404`.

If something goes wrong that causes that we do not receive a response the fetch API throws an error. Once an error is thrown execution stops and JavaScript looks for the closest `catch` block.

Any production `fetch` call should be inside a `try...catch` block:

```
async function fetchSomething() {
  try {
    const response = await fetch("/bad/url/oops");

    if (response.ok) {
      // Success (Good Response)
      const data = await response.json();
      return data;
    } else {
      // Failure (Bad Response)
      console.error("Bad Response");
    }
  } catch (error) {
    // Failure (Network error, etc)
    console.error("An Error occurred");
  }
}
```

REST API

When you create an API, you need to come up with ideas on how to structure your API, shape the features you provide and which rules to apply. This is referred to as the architecture of your API.

There are different common approaches used across the web. A very popular one is REST API or RESTful API.

REST is a set of architectural constraints, not a protocol or a standard. You as an API developer can implement REST in a variety of ways.

> 💡 REST is short for REpresentational State Transfer

The overall idea is like this:

- A *client* requests a *resource* from a *server* (like a document containing information on a specific topic)
- The *server* formulates a response that represents the resource in a format the *client* understands (like JSON - other formats like HTML, XML oder plain text are also possible)
- This transfer of data changes the state of the web application.

Each resource has a unique address. The whole communication is done via HTTP and uses different HTTP methods (like GET/POST/PUT/DELETE) to describe desired actions.

> 💡 This is a very basic and incomplete explanation. If you're interested in learning more about what makes an API RESTful, you can read about it here.