Internetværk og Web-programmering

Asynkronitet, Promises, Fetch

Forelæsning 6

Michele Albano

Distributed, Embedded, Intelligent Systems

Agenda

- Exceptions
- Timers, callbacks and events
- Promises
- Async/await
- Fetch

Exceptions and Errors

- Synonyms in JavaScript
- Meaning: an exceptional condition or error has occurred

Terminology:

- An exception is *thrown* when the error or exceptional condition happens
- *Catching* an exception means handling the exception, to execute code e.g.: to recover from the exception.

The big picture

- The `try` statement lets you test a block of code for errors
- The `catch` statement lets you handle the error
- The `throw` statement lets you create custom errors
- The `finally` statement lets you execute code, after `try` and `catch` blocks, regardless of the result

Let us `throw` an exception

- The exception can be a JavaScript String, a Number, a Boolean or an Object

```
throw "Too big";     // throw a text
throw 500;           // throw a number
```

- When the interpreter throws an error, it uses the `Error` class and its subclasses
- Properties: `name` (type of error) and `message` (holds the string passed to the constructor)
- JavaScript will actually create an Error object with two properties: name and message.

Example

```
function factorial(x) {
        // If the input argument is invalid, throw an exception!
        if (x < 0) throw new Error("x must not be negative");
        // Otherwise, compute a value and return normally
        let f;
        for(f = 1; x > 1; f *= x, x--) /* empty */ ;
        return f;
}
factorial(4)
```

Execution flow

- When an exception is thrown, JavaScript interpreter immediately stops normal program execution and jumps to the nearest exception handler
  - It checks against the current block of code
  - If no catch clause is found, next-highest enclosing block of code is considered
  - If no catch clause is found in the function, exception is propagated up to the code that invoked the function
  - If no catch clause is ever found, the exception is treated as an error and is reported to the user

How to catch en exception

- `try/catch/finally` statement:

```
try {
    // Code where the exception could be thrown
}
catch(e) {
    // Code to be executed if an exception is thrown. "e" is your exception
}
finally {
        // Code executed at the end of the tryed code if no exception is thrown,
        // and after the catched code if an exception is thrown
}
```

- The `finally` code can throw an exception, which is propagated up; it can `return` to make the method return normally (no exception)

Error handling

- Two philosophies for error handling
  - One is the fail-silent approach where you ignore errors in the code
  - The other is the fail-fast and unwind approach where errors stop the world and rewind
- Better to stop code execution and let the user know, and possibly inform the developer

Input Validation

- Important use case for exceptions:
  - Examine the input. If it is wrong, an exception is thrown
  - The exception is caught by the catch statement and a custom error message is displayed

```
<!DOCTYPE html>
<html><body>

<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick=”testIt()">Test It</button>
<p id="p01"></p>

<script>
function testIt() {
  var message, x;
  message = document.getElementById("p01");
  message.innerText = "";
  x = document.getElementById("demo").value;
  try {
    if(x == "") throw "empty";
    if(isNaN(x)) throw "not a number";
    x = Number(x);
    if(x < 5) throw "too low";
    if(x > 10) throw "too high";
  }
  catch(err) {
    message.innerText = "Input is " + err;
  }
}
</script>

</body></html>
```

Agenda

- Exceptions
- Timers, callbacks and events
- Promises
- Async/await
- Fetch

Asynchronous execution

- A program is doing something
- The current action has to wait for something before continuing
- The program wants to do something else while waiting

Use cases:

- JavaScript programs in a web browser are typically event-driven, and must wait for user input (e.g.: clicks)
- JavaScript-based servers wait for client requests to arrive over the network before they do anything

JavaScript mechanisms for asynchronous execution

- `Callbacks` are registered, and called when an event happens
- `Promises` are objects that represent not-yet available result of an asynchronous operation
- `async` and `await` provide a syntax for asynchronous programming to simplify Promise-based code

Callbacks for timers

- You can register a function
- It gets called when the Timer is fired (see previous lecture)

```
timerId = setTimeout(checkForUpdates, 60000);
```

- The callback function is called one minute after setTimeout is executed
- checkForUpdates is called once at the correct time *with no arguments*, and nothing more
- Use instead setInterval to have periodic execution. You can later stop the periodic execution with clearInterval(timerId). Don’t forget to save timerId

Example for timers

```
timerId = setTimeout(() => {
  console.log("hello later"); // runs after 2 seconds
}, 2000)
```

- I can pass it more parameters:

```
const myFunction = (firstParam, secondParam) => {
  …
}
setTimeout(myFunction, 2000, firstParam, secondParam)
```

- Similarly to the periodic execution, you can cancel the timer before the callback is called:

```
clearTimeout(timerId)
```

Callbacks for events

- You have seen this already
  - Event-driven JavaScript programs register event handler functions for specified types of events (e.g.: `'click'`) in specified contexts (e.g.: `confirmUpdateDialog`'s button)
  - The web browser invokes those functions whenever the specified events occur

```
let okay = document.querySelector('#confirmUpdateDialog button.okay');
okay.addEventListener('click', applyUpdate);
```

  - applyUpdate gets executed when the user clicks on the button of the dialog

Callbacks for file system events

- Node.js processes files, networking, etc asynchronously
  - These operations can be time consuming, and are mediated by the operating system. The Node.js program can do something else in the meantime

```
const fs = require("fs");
let options = {}; // Object to hold options, initialized with default values here

fs.readFile("config.json", "utf-8", (err, text) => {
        if (err) {
                console.warn("Could not read config file:", err);
        } else {
                Object.assign(options, JSON.parse(text));
        }
        startProgram(options);
});
```

Callbacks hell

- Imagine that you have to write callbacks for everything that can happen, and register and deregister them as needed
- Imagine also that a callback can define and register another callback, and so on and so forth
- Image having to read code from your colleagues where callbacks are indented on 5 different levels, since they are callbacks defined into callbacks that are defined into callbacks etc etc

- If it does not seem fun, you are right
  - CALLBACKS HELL
- Solution: mechanisms to write synchronous-like code that is executed asynchronously

Agenda

- Exceptions
- Timers, callbacks and events
- Promises
- Async/await
- Fetch

Promises

- A promise is a proxy for a value that will eventually become available
  - You create it “around” a function (also known as `executor`) that takes two functions as parameters. One is executed if the executor completes correctly, one if there was a failure
  - There is no way to synchronously get the value of a Promise; you can only ask the Promise to call a callback function when the value is ready
- The Promise represents a computation that will eventually produce a value or throw an exception
  - Not good for repeated computation

Benefits of Promises

- This solves two problems of “normal” callback-based programming:
  - I don’t have callbacks inside callbacks inside callbacks (callbacks hell). Instead, I have Promise chains
  - Error handling. It is impossible to just throw an exception, since there is no way for that exception to propagate back to the initiator of the asynchronous operation. Promises standardize how to handle errors

Using a Promise

- Image that getJSON(url) returns a Promise, you can call its then method to register two callbacks:

```
getJSON(url).then(
        jsonData => {
                // This is a callback function that will be asynchronously
                // invoked with the parsed JSON value when it becomes available.
        }, err => {
                // This code is executed when an exception is raised
        }
);
```

- If the function in the first then() returns a promise, you can call the then() method multiple times, each of the functions will be called on the promise returned by the previous function

Idiomatic best practices

- Call the then() method directly on the function invocation that returns a Promise, without assigning the Promise to an object
- Name the functions returning Promises with verbs
- Instead of passing a “reject” function, catch() the exception outside the then() invocation
  - As we discussed, the exception is propagated outward until it finds a catch()

```
function displayUserProfile(profile) { /* implementation omitted */ }
function handleProfileError (err) { /* implementation omitted */ }

// Notice how this line of code reads almost like an English sentence:
getJSON("/api/user/profile").then(displayUserProfile).catch(handleProfileError);
```

More terminology: States

- A Promise can be *fulfilled* (correct computation of the final value, fist callback is called)
- A Promise can be *rejected* (failure with computing the value, Exception raised, second callback / .catch() is called)
- Before that time, the Promise is *pending*. As soon as it is either *fulfilled* or *rejected*, the Promise is *settled*
- After the Promise is *settled*, it provide its *result* (either the computed value, or the Error) as soon as the .then().catch() is applied to the Promise
  - Maybe you are executing another function and you cannot yet execute the .then().catch()
  - Maybe you already did the .then().catch() and the callback will be called as soon as possible

Executing Promises sequentially

- Later we will study the fetch() method
  - Used to perform a REST request (e.g.: GET of a web page)
- Its execution returns a response with the headers of the interaction, but not the full resource
  - Large data transfer can take a lot of time, and it is better to know immediately if the HTTP transfer will fail for sure (e.g.: wrong host name)
- Calling json() on the response returns a Promise for the JSON encoding of the data
- Example without chaining, similar to callbacks hell:

```
fetch("/api/user/profile").then(response => {
        response.json().then(profile => { // Ask for the JSON-parsed body
                // When the body of the response arrives, it will be parsed as JSON and passed to this function
                displayUserProfile(profile);
        });
});
```

Chaining Promises

- A natural way to express a sequence of asynchronous operations.
- Linear chain of then() method invocations, without having to nest each operation within the callback of the previous one
- Example:

This .then() returns a Promise

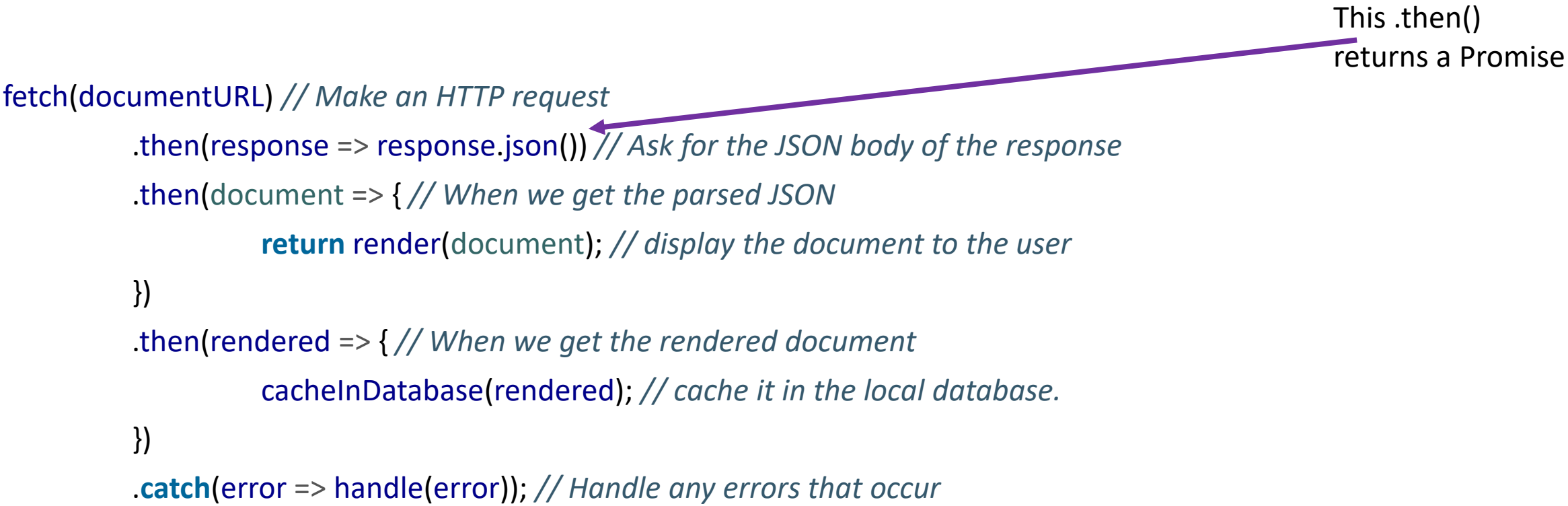
```
fetch("/api/user/profile")
        .then(response => response.json()) // Ask for the JSON-parsed body
        .then(profile => { // When the body of the response arrives, it will be parsed
                displayUserProfile(profile); // as JSON and passed to this function
        });
```

- Same meaning of:

```
.then(response => { return response.json(); } )
```

More complex example

- This example has one more step and error handling
- Each invocation of the then() method returns a new Promise, which is not fulfilled until the function passed to ITS then() is complete



More terminology: Fates

- As soon as the callbacks registered using then() / then().catch() returns, the Promise is *resolved*. We can *resolve* a promise with another promise
- A Promise is *unresolved* if it is not *resolved*

Relation States vs Fates:

- A Promise is not *fulfilled* if the callback returned a *pending* Promise (see chaining in next slide)
- A *fulfilled* Promise is *resolved*. A *rejected* Promise is *resolved*
- An *unresolved* Promise is *pending*

https://stackoverflow.com/questions/35398365/js-promises-fulfill-vs-resolve

https://github.com/domenic/promises-unwrapping/blob/master/docs/states-and-fates.md

Resolved but not Settled?

- As soon as the callbacks registered using then() / then().catch() returns, the Promise is *resolved*. We can *resolve* a promise with another promise. Can next then() method be called already?

```
function c1(response) { // callback 1
        let p4 = response.json(); return p4; } // returns promise 4
function c2(profile) {displayUserProfile(profile);} // callback 2
let p1 = fetch("/api/user/profile"); // promise 1, task 1
let p2 = p1.then(c1); // promise 2, task 2
let p3 = p2.then(c2); // promise 3, task 3
```

- When fetch() ends its asynchronous job task 1 (promise 1 is resolved), the output of task 1 can be sent as input to c1 (promise 1 is fulfilled)
- If p2 gets fulfilled, c2 is invoked, and task 3 begins. Anyway, when c1 returns, it returns Promise p4, which can still be rejected, maybe p4 will never provide a value, and c2 cannot be invoked yet. Promise p2 is resolved to p4, but p2 cannot settle until p4 settles.

Definition of Resolved

- “resolved” Promise means: the Promise has become associated with, or “locked onto”, another Promise or a non-Promise value
- In some cases, we don’t know yet whether p will be fulfilled or rejected, but the callback has no control anymore over that
- Promise p is “resolved” in the sense that its fate now depends entirely on what happens to something else (the Promise it returned)

Error Handling

- We already discussed that the second callback is not usually provided
- Idiomatic approach similar to try/catch/finally:
- getJSON("/api/user/profile").then(displayUserProfile).catch(handleProfileError).finally(cleanUp);
- If getJSON() ends correctly, then() is invoked; if an Exception is raised (in getJSON() or in then()), the catch() is called. After then() end correctly or catch() ends, finally() is run
- One more example (recoverFromStageTwoError() returns a Promise):

```
startAsyncOperation()
        .then(doStageTwo)
        .catch(recoverFromStageTwoError)
        .then(doStageThree)
        .then(doStageFour)
        .catch(logStageThreeAndFourErrors);
```

Promises in Parallel

- You can create an array of Promises

```
promises = urls.map(url => fetch(url).then(r => r.text()));
```

- Then you can execute all of them in parallel:

```
Promise.all(promises)
        .then(bodies => { /* do something with the array of strings */ })
        .catch(e => console.error(e));
```

- The Promise returned by Promise.all() rejects as soon as any of the input Promises is rejected

Making Promises from a Promise

- Creating a Promise that encapsulate another Promise:

```
function getJSON(url) {
        return fetch(url).then(response => response.json());
}
```

- When the internal Promise fulfills, the promise returned by getJSON() fulfills as well
- Error handling:
  - Checking response.ok and the Content-Type header?
  - No, in this case it is easier: we just allow the json() method to reject the Promise it returned with a SyntaxError if the response body cannot be parsed as JSON

Making Promises from synchronous value

- Creating a Promise based on synchronous computation:
  - Compute your value
  - Use the static methods Promise.resolve() and Promise.reject()

```
Promise.resolve('Success').then(function(value) {
  console.log(value); // "Success"
});
```

- No real asynchronicity
- The Promise will be settled as soon as the computation is done
  - They will fulfill or reject **after** the current synchronous chunk of code has finished running
  - console.log(value) outputs “Success” after the current chunk of code ends

Making Promises from scratch

- Creating a Promise to execute code asynchronously:
  - function longFunction(resolve, reject) { … }
  - new Promise(longFunction)
  - The function can call resolve / reject whenever it wants
- It is possible to implement Promise-based APIs out of code using asynchronous callbacks and events

Example of making Promises (from page 364)

```
const http = require("http");
function getJSON(url) { // Create and return a new Promise
    return new Promise((resolve, reject) => {
        // Start an HTTP GET request for the specified URL
        request = http.get(url, response => { // called when response starts
            if (response.statusCode !== 200) {
                reject(new Error(`HTTP status ${response.statusCode}`));
            } else if (response.headers["content-type"] !== "application/json") {
                reject(new Error("Invalid content-type"));
            } else { // GET was fine. Register events to read the body of the response
                let body = "";
                response.setEncoding("utf-8");
                response.on("data", chunk => { body += chunk; });
                response.on("end", () => {
```

Callback

Calling the *reject* of the Promise

Event-oriented programming

Example of making Promises (from page 364)

```
            // When the response body is complete, try to parse it
            try {
                let parsed = JSON.parse(body);
                resolve(parsed); // If it parsed successfully, resolve the Promise
            } catch(e) {
                // If parsing failed, reject the Promise
                reject(e);
            }});
        }});
        // Reject immediately if http.get fails
        request.on("error", error => {
            reject(error);
        });
    });
}
```

Agenda

- Exceptions
- Timers, callbacks and events
- Promises
- Async/await
- Fetch

Async/await

- An `async` function is a function that implicitly returns a promise and that can, in its body, `await` other promises in a way that looks synchronous
- The `await` keyword takes a Promise and turns it back into a return value or a thrown exception (if you are into an `async` function)
- Here are three functions called x, y and z that return promises:

```
async function x() {return "one";}
let y = async function () {return "two";}
let z = async () => "three"
```

- Inside one of the promises, I can `await`:

```
async function do_it() {
  console.log("message is "+await y());
}
```

Benefits of async/await

- Useful to forget about Promises and their complexity
- Given a Promise object p, the expression `await p` waits until p settles
  - If p fulfills, then the value of `await p` is the fulfillment value of p
  - If p is rejected, then the `await p` expression throws the rejection value of p
- When inside an `async` function there is an `await promise1`, your program stops until promise1 settles
  - Or actually, the function is put to sleep and Javascript interpreter can do something else
  - *Any code that uses* `await` *is itself asynchronous*

Idiomatic `await`

- It is placed before the invocation of a function that returns a Promise:
- `let response = await fetch("/api/user/profile");`
- `let profile = await response.json();`
- It is uncommon to bind the Promise to an identifier

- What if I am into a non-async function, or in the top level?
  - It is a Promise …
- `getHighScore().then(displayHighScore).catch(console.error);`

Awaiting multiple promises

- Let us imagine we have an async function:

```
async function getJSON(url) {
        let response = await fetch(url);
        let body = await response.json();
        return body;
}
```

- This is very ”sequential”:

```
let value1 = await getJSON(url1);
let value2 = await getJSON(url2);
```

- This is executed in parallel (thus probably faster):

```
let [value1, value2] = await Promise.all([getJSON(url1), getJSON(url2)]);
```

- Promise.all was discussed in slide 32

Asynchronous loops

- Promises do not work for sequences of asynchronous events
- We cannot use regular `async/await` statements
- Solution: `for/await`

```
const fs = require("fs");


async function parseFile(filename) {
        let stream = fs.createReadStream(filename, { encoding: "utf-8"});
        for await (let chunk of stream) {
                parseChunk(chunk); // Assume parseChunk() is defined elsewhere
        }
}
```

For/await loops

- It is Promise-based:
  - The asynchronous iterator produces a Promise
  - The for/await loop waits for that Promise to fulfill
  - The fulfillment value is assigned to the loop variable
  - The body of the loop is executed
  - Another Promise from the iterator is created and the loop repeats

One more example

```
const urls = [url1, url2, url3];
const promises = urls.map(url => fetch(url));

for await (const response of promises) {
    handle(response);
}
```


```
for(const promise of promises) {
    response = await promise;
    handle(response);
}
```

Agenda

- Exceptions
- Timers, callbacks and events
- Promises
- Async/await
- Fetch

Performing HTTP/HTTPS requests

- Different methods can be used to perform HTTP requests
  - Old approach: XMLHttpRequest. Let's forget about it
- Let us align on the "best" method we currently have: `fetch`
- `fetch()` defines a Promise-based API for making HTTP and HTTPS requests:
  - It accepts a URL and the kind of request as parameters
  - It returns a Promise that fulfils to the `response` (it was possible to contact the server) or it rejects if the HTTP connection could not be done
  - The `response` is a promise that resolves to a `body` (if the request went fine, e.g.: response code 200) containing the data

Fetch basic examples

- Fetch with then()

```
fetch("/api/users/current") // Make an HTTP (or HTTPS) GET request
        .then(response => response.json()) // Parse its body as a JSON object
        .then(currentUser => { // Then process that parsed object
                displayUserInfo(currentUser);
});
```

- Fetch with `async/await`

```
async function isServiceReady() {
        let response = await fetch("/api/service/status");
        let body = await response.text();
        return body === "ready";
}
```

Fetch: better GET example

- Make an HTTPS GET request
- When we get a response, first check it if for a success code (200 to 299) and return a Promise for the body
- Or throw an error
- When the response.text() Promise resolves let us print the body
- Or if anything went wrong, just log the error. If the user's browser is offline, fetch() itself will reject. If the server returns a bad response then we throw an error above

```
async function let_s_fetch() {
  fetch("https://www.cs.aau.dk")
    .then(response => {
      if (response.ok) {
        return response.text();
      } else { throw new Error(
        `Unexpected response status ${response.status}` );
    }})
    .then(body => { console.log("result is " + body); })
    .catch(error => {
      console.log("Error while fetching data:", error);
    });
}
```

Setting request parameters and headers

```
async function search(term) {
    let authHeaders = new Headers();
    authHeaders.set("Authorization",
        `Basic ${btoa(`${username}:${password}`)}`);
    let url = new URL("/api/search");
    url.searchParams.set("q", term);
    let response = await fetch(url);
    if (!response.ok) throw new
        Error(response.statusText);
    let resultsArray = await response.json();
    return resultsArray;
}
```

Setting an auth header

/api/search?q=${term}

Parsing the body

- What can I do with request (= the result of a fulfilled `fetch` Promise)?
- Two most common way of parsing the result of the GET:
  - `response.json().then( … )` to get the result as a JSON object
  - `response.text().then( … )` to get the result as text
- Other useful methods:
  - `response.formData().then(…)` to parse the result as a FormData object (“multipart/ form-data” format). Common to send data to a server, but not for the response
  - Streaming! See next slide

Fetch Streaming API

- Do not get a Promise out of the response
  - If you access the data in any way (.json(), .text(), etc), you cannot re-access it
- The response has method getReader() to get a stream reader object:

```
let reader = response.body.getReader();
while(true) { // Loop until we exit below
    let {done, value} = await reader.read();
    // Verify value is not null. Process the “value”. Parse the data. Check for errors
if (done) { // If this is the last chunk,
        break; // exit the loop
    }
}
```

Fetch: a POST

```
  let user = { name: 'Michele', surname: 'Albano' };

  let response = await fetch('/article/fetch/post/user', {
    method: 'POST',
    headers:
      {'Content-Type': 'application/json;charset=utf-8'},
    body: JSON.stringify(user)
  });

  let body = await response.json();
  // do something with the response
}
```

- The Content-Type can be other things, such as FormData or blobs

Security: CORS

- Web browsers generally disallow fetch() to servers different from the one the main HTML document comes from
  - Exceptions: images and scripts
- Cross-Origin Resource Sharing (CORS) aims to safe cross-origin requests
- The browser automatically adds an "`Origin`" header to the request
- The server has to explicitly answer with a "`Access-Control-Allow-Origin`" header, or the interaction is cancelled
  - Meaning, the Promise returned by fetch() is rejected
- The "`Origin`" header cannot be overridden via the headers property

Aborting a Fetch request

- Need to create an `AbortController` PRIOR TO the `fetch()`
- Pass the `signal` property of the `AbortController` as the `signal` property in the `options` of the `fetch()`
- Call the `abort()` method of the controller to abort the request
- Let's not get a Promise out of the response
  - If you access the data in any way (.json(), .text(), etc), you cannot re-access it

```
let controller = new AbortController();
options.signal = controller.signal;
setTimeout(() => { controller.abort(); }, 10000); // 10 seconds before aborting
fetch(url, options).then( …
```

Exercises

1. [fetch] If you did not complete exercise 6c from lecture 5, do it now that you have more knowledge about the use of fetch().
2. [exceptions] A web page using exceptions to make validation easier
3. [promises] Make a Promise for a timer-based delay. Then, use async/await
4. [fetch] fetch() a page and catch() any issue you can have

   Supplementary:

5. Parallel fetch(); sequential fetch()