howest
hogeschool

Web Technology: Asynchronous Programming

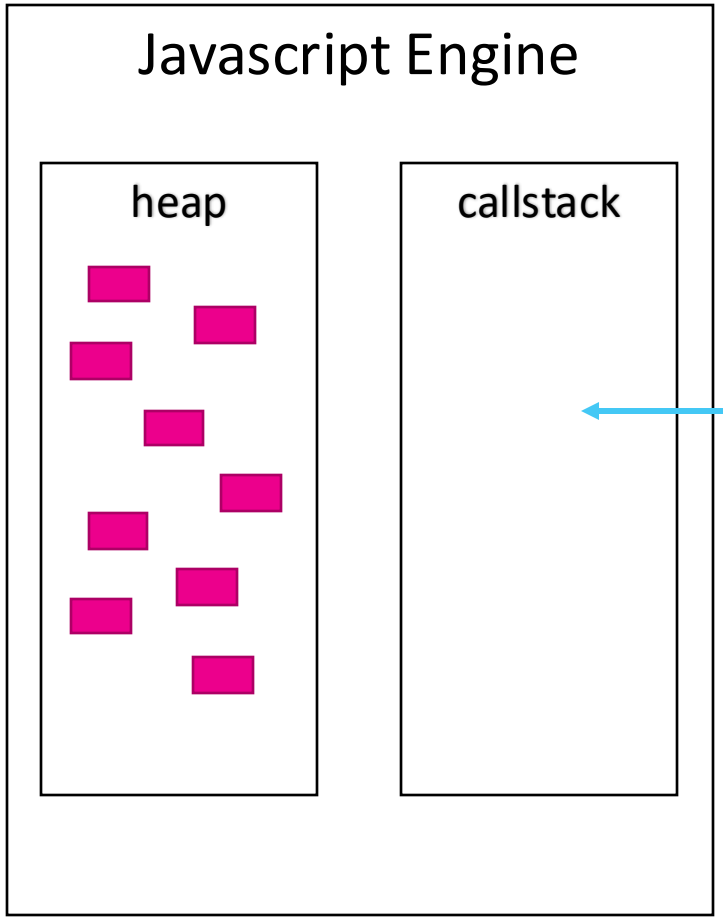
Dieter Mourisse

howest
hogeschool

The event loop

https://www.youtube.com/watch?v=8aGhZQkoFbQ&t=13s

Call Stack



The call stack is a stack data structure that stores information about the active functions (functions that haven't returned yet) of a computer program.

When a function is called, it is pushed onto the call stack, when it returns, it is popped of the call stack.

main()

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

generateRandomPrimes()

main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

generateRandomPrimes()

main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

- random()
- generateRandomPrimes()
- main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

Math.random()

random()

generateRandomPrimes()

main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

random()

generateRandomPrimes()

main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

- Math.floor()
- random()
- generateRandomPrimes()
- main()

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

- random()
- generateRandomPrimes()
- main()

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

generateRandomPrimes()

main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

isPrime()

generateRandomPrimes()

main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

- Math.sqrt()
- isPrime()
- generateRandomPrimes()
- main()

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

isPrime()

generateRandomPrimes()

main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

Math.sqrt()

isPrime()

generateRandomPrimes()

main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

isPrime()

generateRandomPrimes()

main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

generateRandomPrimes()

main()

howest hogeschool

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

generateRandomPrimes()

main()

howest hogeschool

main()

```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes(1)
```

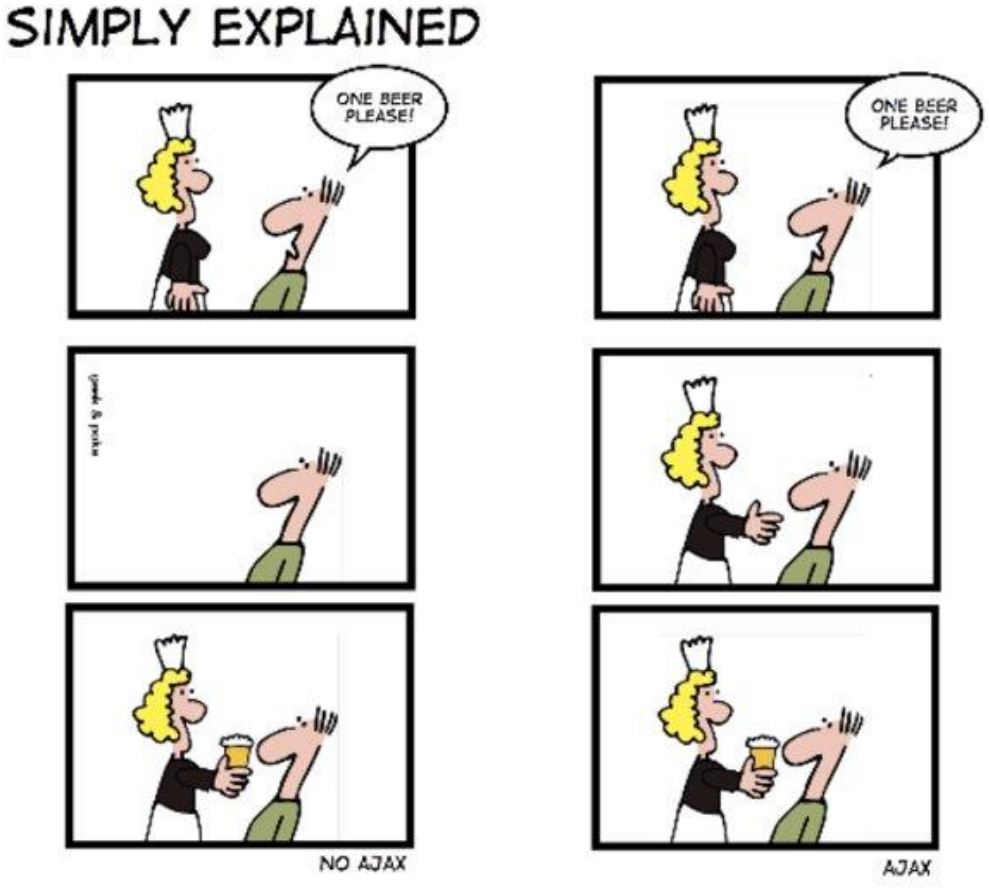
```
function isPrime(n) {
    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (n % i === 0) {
            return false;
        }
    }
    return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generateRandomPrimes(quota) {
    const primes = [];
    while (primes.length < quota) {
        const candidate = random(MAX_PRIME);
        if (isPrime(candidate)) {
            primes.push(candidate);
        }
    }
    return primes;
}
generateRandomPrimes()
```

Blocking functions

- JavaScript is a single threaded programming language
- This means there is only one call stack and the JavaScript **engine** can do only one thing at a single point in time.
- If a function takes a long time to execute, you cannot interact with the web browser during the function's executing because the page hangs (does not get repainted, does not react immediately to click events, ...).
- A function that takes a long time to complete is called a blocking function.

What is asynchronous programming

Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result.



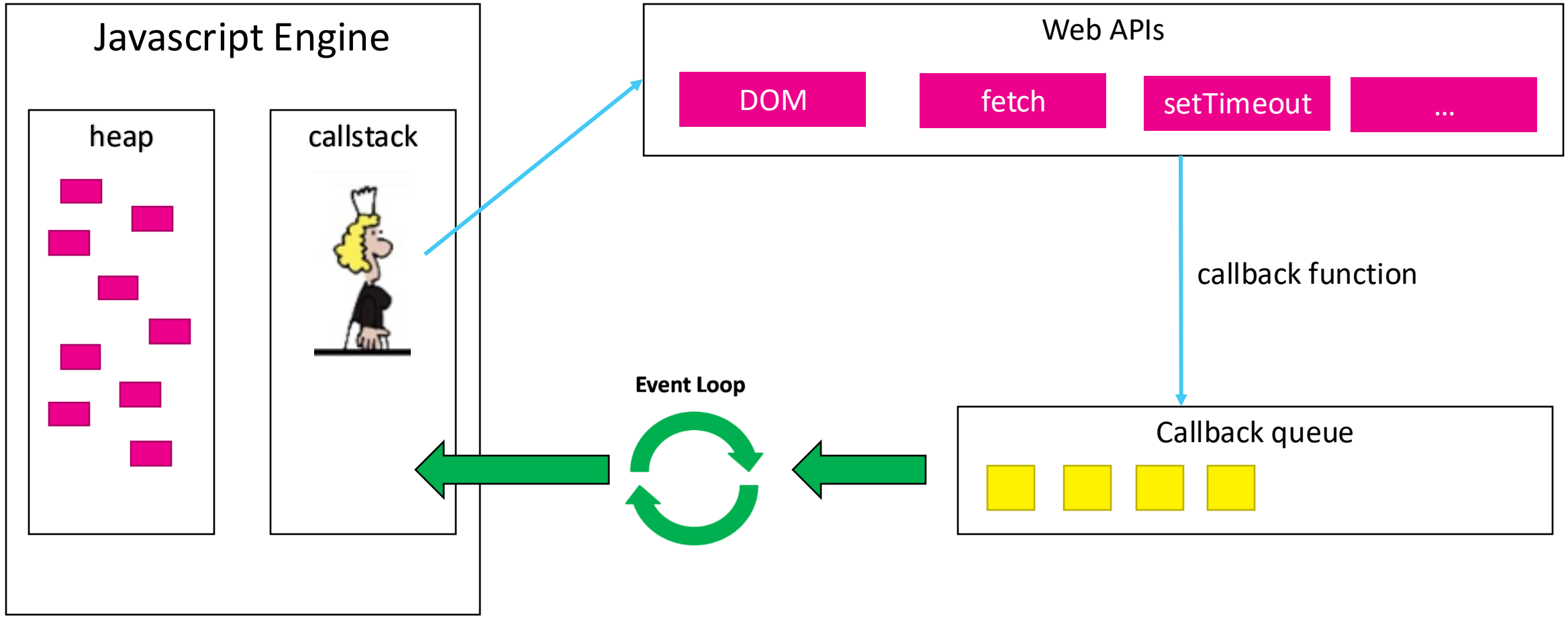
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing

Some asynchronous functions

fetch()	Making HTTP requests
getUserMedia()	Accessing a user's camera or microphone
showOpenFilePicker()	Asking a user to select files
setTimeout()	Wait to execute some code

How does the JavaScript engine handle these?

Call Stack


Javascript Engine
heap
callstack
Web APIs
DOM
fetch
setTimeout
...
callback function
Event Loop
Callback queue

Event loop

When the call stack is empty, the event loop will move the first function of the callback queue onto the call stack.

http://latentflip.com/loupe

howest hogeschool

howest
hogeschool

Callback functions

Callback function

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

```
function processUserInput(callback) {
    const name = prompt("Please enter your name: ")
    callback(name);
}

function greeting(name) {
    console.log(`Hello ${name}`);
}

processUserInput(greeting);
```

Callback function

howest hogeschool

Callback function

```
setTimeout(callbackfunction, 5000);
```

Wait 5000 ms before adding *callbackfunction* to the callback queue

```
document.querySelector("button").addEventListener("click", callbackfunction);
```

When a click event occurs on the button, add *callbackfunction* to the callback queue

Callback function

```
import fs from 'fs';

try {
    fs.readFile('data.txt', handlefile);
} catch (ex) {
    console.log(ex.message);
}

function handlefile(err, contents) {
    if (err) console.log(err);
    else console.log(contents.toString());
}
```

Callback function that will be executed as soon as full file has been read.

howest hogeschool

XMLHttpRequest (let's go ancient)

XMLHttpRequest (XHR) objects are used to interact with servers. You can retrieve data from a URL without having to do a full page refresh. This enables a Web page to update just part of a page without disrupting what the user is doing.

XMLHttpRequest is used heavily in AJAX programming.

*AJAX stands for **A**synchronous **J**avaScript **A**nd **X**ML.*

The two major features of AJAX allow you to do the following:
- *Make requests to the server without reloading the page*
- *Receive and work with data from the server*

Predecessor of fetch

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

```
const ENDPOINT = 'https://jsonplaceholder.typicode.com/users’

const xhr = new XMLHttpRequest();
xhr.onreadystatechange = handleResponse;

xhr.open('GET', ENDPOINT);
xhr.send();

function handleResponse() {
    if (xhr.readyState === XMLHttpRequest.DONE) {
        if (xhr.status === 200) {
            let data = JSON.parse(xhr.responseText);
            console.log(data);
        }
    }
}
```

Set callback function

Set request parameters

Do the request

Check if request is finished

Check if response code is 200

Get response data

howest hogeschool

Problems with callback functions

- Hard to remember which parameters the callback function should have and what order they are in.
- Hard to reuse them, logic is nested deep and it isn’t easy to do something different with the data. Possible solution for this is passing an extra callback function, but this leads to **callback hell**.
- Error handling is tricky. Something can go wrong inside the callback function or where the callback function was added. These are two completely different places in the code.

Problems with callback functions

- Don’t compose easily for multiple levels of callbacks
- Lead to code that is hard to extend
- Have no conistency between order of parameters
- Have no consistent way to handle errors

howest
hogeschool

Promises

Promises

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

Promise state	
Pending	Initial state, neither fullfilled nor rejected. The asynchronous function isn't completed.
Fullfilled	Meaning that the aynchronous function was completed succesfully
Rejected	Meaning that the asynchronous function failed
Settled	Either fullfilled or rejected
Resolved	Colloquially equivalent to fulfilled, but this isn't entirely correct

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Promise.prototype.then

*The **then()** method **immediately** returns a Promise. It takes up to two arguments: callback functions for the fulfilled and rejected cases of the Promise.*

Syntax

```
then(onFulfilled)
then(onFulfilled, onRejected)

then(
  (value) => { /* fulfillment handler */ },
  (reason) => { /* rejection handler */ },
);
```

Parameters

`onFulfilled` Optional

A `Function` asynchronously called if the `Promise` is fulfilled. This function has one argument, the *fulfillment value*. If it is not a function, it is internally replaced with an *identity* function (`(x) => x`) which simply passes the fulfillment value forward.

`onRejected` Optional

A `Function` asynchronously called if the `Promise` is rejected. This function has one argument, the *rejection reason*. If it is not a function, it is internally replaced with a *thrower* function (`(x) => { throw x; }`) which throws the rejection reason it received.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then

.then

fs.promises.readFile returns a Promise for the asynchronous readFile function

```
function succeededFunction(contents) {
    console.log(contents.toString());
}

function errorFunction(err) {
    console.log(err);
}
```

Create the promise →

```
const promise = fs.promises.readFile('data.txt');
```

Add calbacks →

```
promise.then(succeededFunction, errorFunction);
```

fullfilled ↑ rejected ↑

howest hogeschool

.then

```
function succeededFunction(contents) {
    console.log(contents.toString());
}

function errorFunction(err) {
    console.log(err);
}

fs.promises.readFile('data.txt’)
    .then(succeededFunction, errorFunction);
```

Promise.prototype.catch

*The **catch()** method returns a Promise and deals with rejected cases only. It behaves the same as calling Promise.prototype.then(undefined, onRejected)*

Syntax

```
p.catch(onRejected)

p.catch(function(reason) {
  // rejection
})
```

Parameters

`onRejected`

A `Function` called when the `Promise` is rejected. This function has one argument:

`reason`

The rejection reason.

The Promise returned by `catch()` is rejected if `onRejected` throws an error or returns a Promise which is itself rejected; otherwise, it is fulfilled.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then

.catch

```
function succeededFunction(contents) {
    console.log(contents.toString());
}

function errorFunction(err) {
    console.log(err);
}

fs.promises.readFile('data.txt’)
    .then(succeededFunction)
    .catch(errorFunction);
```

catch

```
fs.promises.readFile('data.txt’)
    .then(contents => console.log(contents.toString()))
    .catch(err => console.log(err));
```

howest
hogeschool

Chaining promises

Subsequent asynchronous calls

```
fs.readFile('filedata.txt', function(err, contents) {
    if (err) console.log(err);
    else {
        const filename = contents.toString();
        fs.readFile(filename, function(err, contents) {
            if (err) console.log(err);
            else console.log(contents.toString());
        });
    }
})
```

This is called the callback pyramid of doom

howest hogeschool

Chaining .then

.then returns a new Promise immediately. This new promise is always pending when returned. The behaviour of the returned promise depends on the handler's execution result.

The handler function	The promise returned by then
Returns a value	Gets fullfilled with the returned value as it result.
Returns nothing	Gets fullfilled with undefined.
Throws an error	Gets rejected with the thrown error as its value.
Returns an already fullfilled promise	Gets fullfilled with that promise's value as its value.
Returns an already rejected promise	Gets rejected with that promise's value as its value.
Returns another pending promise	Will fullfill or reject depending on the fullfillment/rejection of that promise. Its's value will be the same as the resolved value of that promise.

Chaining .then

```
fs.promises.readFile('filedata.txt')
   .then(contents => {
      const filename = contents.toString();
      return fs.promises.readFile(filename);
   }, errorFunction)
   .then(contents => console.log(contents.toString()), errorFunction);
```

Returns a promise that will settle when the promise returned by the callback has settled.

howest hogeschool

Chaining .catch

When onRejected is left empty, it is internally replaced with a thrower function which simple rethrows the rejection reason it received.

```
function reject(x) {
   throw x;
}
```

Syntax

```
then(onFulfilled)
then(onFulfilled, onRejected)

then(
  (value) => { /* fulfillment handler */ },
  (reason) => { /* rejection handler */ },
);
```

Parameters

`onFulfilled` Optional

A `Function` asynchronously called if the `Promise` is fulfilled. This function has one argument, the *fulfillment value*. If it is not a function, it is internally replaced with an *identity* function (`(x) => x`) which simply passes the fulfillment value forward.

`onRejected` Optional

A `Function` asynchronously called if the `Promise` is rejected. This function has one argument, the *rejection reason*. If it is not a function, it is internally replaced with a *thrower* function (`(x) => { throw x; }`) which throws the rejection reason it received.

howest hogeschool

Chaining .catch

```
fs.promises.readFile('filedata.txt')
   .then(contents => {
      const filename = contents.toString();
      return fs.promises.readFile(filename);
   })
   .then(contents => console.log(contents.toString()))
   .catch(err => console.log(err));
```

Returns a promise that will settle when the promise returned by the callback has settled.

howest hogeschool

Chaining .catch

```
fs.promises.readFile('filedata.txt')
    .then(contents => fs.promises.readFile(contents.toString()))
    .then(contents => console.log(contents.toString()))
    .catch(err => console.log(err));
```

howest
hogeschool

Writing your own promises

Writing your own promises

Promise.reject()
*The **Promise.reject()** method returns a Promise object that is rejected with a given reason.*

Promise.resolve()
*The **Promise.resolve()** method "resolves" a given value to a Promise.*

Promise() constructor
*The **Promise() constructor** takes an executor function as parameter. This is a function with two parameters. The first is a resolutionFunc and the second a rejectionFunc.*

Creating your own promise

```
const computePrimeNumbersAsync = function(amount) {
   if (amount < 0) return Promise.reject("Quota needs to be positive");
   if (amount == 0) return Promise.resolve([])

   function executor(resolveFunc, rejectFunc) {
      setTimeout(() => resolveFunc(generateRandomPrimes(amount)), 1000)
   }

   return new Promise(executor);
}
```

Immediately settle

Promise is settled when resolveFunc/rejectFunc is called in executor function passed to Promise constructor.

```
computePrimeNumbersAsync(amount)
   .then(numbers => output.textContent = `Finished generation ${numbers.length} primes!` );
```


howest hogeschool

Delay promise

```
function delay(ms) {
    return new Promise((resolve, reject) => setTimeout(resolve, ms));
}

delay(3000).then(() => console.log("Runs after 3 seconds"))
```

howest
hogeschool

Composition

Promise.all

*The **Promise.all()** method takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when all of the input's promises fulfill (including when an empty iterable is passed), with an array of the fulfillment values. It rejects when any of the input's promises rejects, with this first rejection reason.*

```
const BASEURL = 'https://jsonplaceholder.typicode.com';
const ENDPOINTS = ['posts', 'comments', 'users'];
Promise.all(
  ENDPOINTS.map(endpoint => {
    return fetch(`${BASEURL}/${endpoint}`).then(resp => resp.json());
  })
  )
  .then(([posts, comments, users]) => {
    console.log(matchUsers(users, posts, comments));
  });
```

Array of promises

Will be executed when all promises in the array are fulfilled,

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

Promise.allSettled

*The **Promise.allSettled()** method takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when all of the input's promises settle (including when an empty iterable is passed), with an array of objects that describe the outcome of each promise.*

Promise.allSettled will never be rejected. *Promise.all* will reject as soon as one of the promises rejected.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled

Promise.race

The **Promise.race()** method takes an iterable of promises as input and returns a single Promise. This returned promise settles with the eventual state of the first promise that settles.

```
function timeout(ms) {
   return new Promise((resolve, reject) =>
      setTimeout(() => reject(new Error("Request timed out")), ms));
}

Promise.race([fetch(`${BASEURL}/posts`), timeout(3000)])
   .then(res => res.json())
   .then(data => console.log(data))
   .catch(err => console.log(err))
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race

Promise.any

*The **Promise.any()** method takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when any of the input's promises fulfills, with this first fulfillment value. It rejects when all of the input's promises reject (including when an empty iterable is passed), with an AggregateError containing an array of rejection reasons.*

Promise.any will take the first succeeded promise and only reject when all promises rejected.

Promise.race will take the first one, even if that one was rejected.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/any