howest
hogeschool

Web Technology: Async / Await

Dieter Mourisse

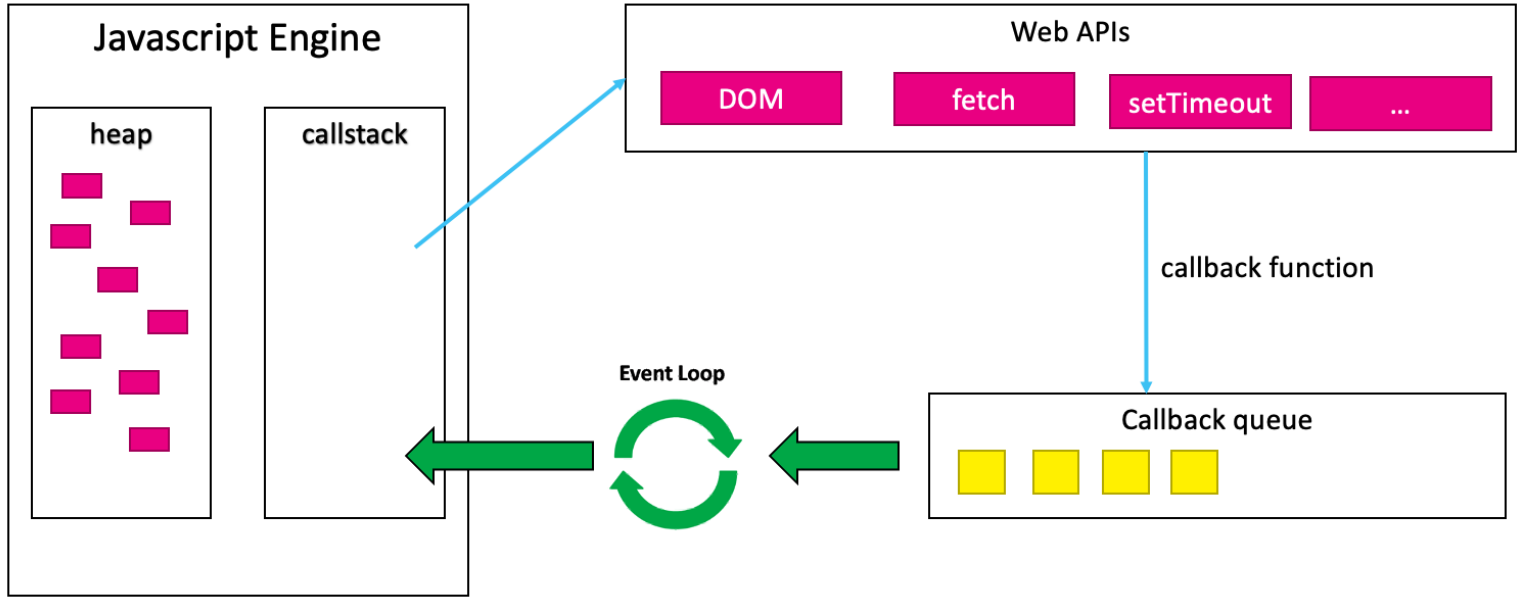
howest
hogeschool

Recap

Asynchronous

A lot of JavaScript functions are run asynchronous. This means that you don't immediately receive the result of the function call.

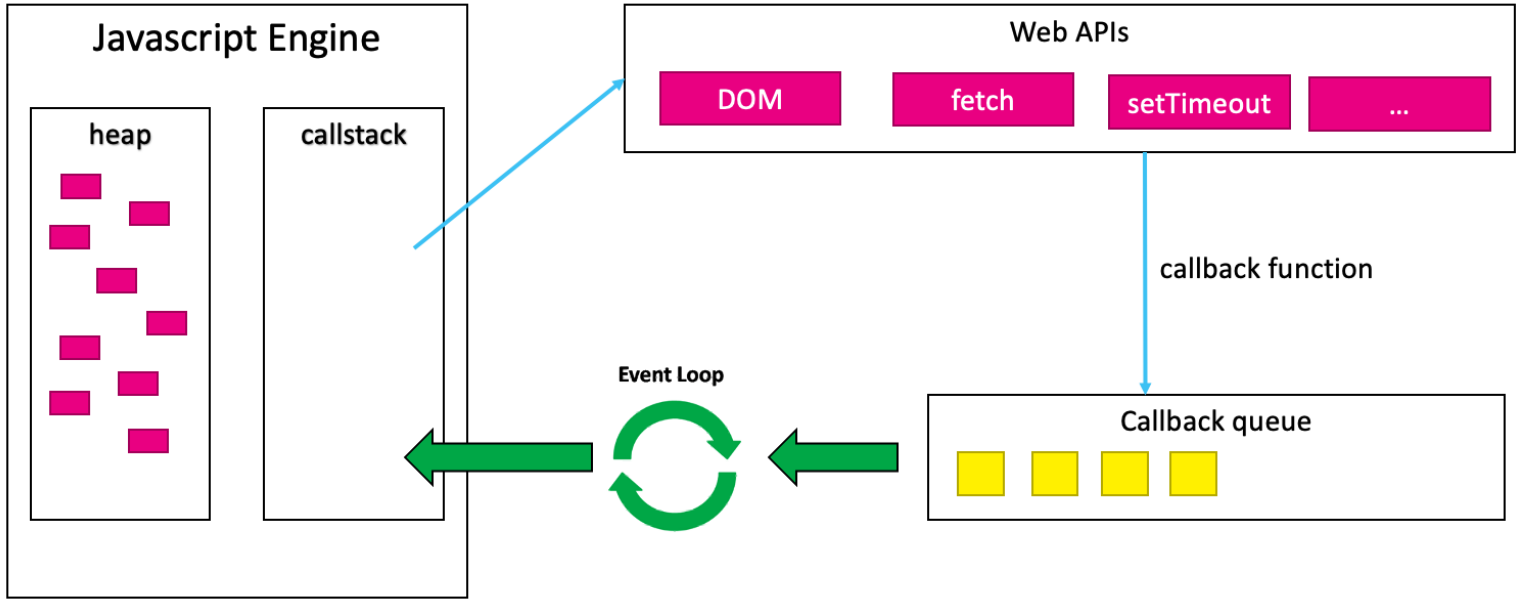
JavaScript will not wait for the result, but proceed with executing the remaining of the script after the asynchronous call has been initiated.



Callback function

When doing an asynchronous call, you provide a callback function. When the asynchronous function is done, this callback function will be called with the result of the asynchronous function.

The JavaScript engine will then execute the callback function as soon as it has no other work (the call stack is empty). The thing responsible for this is the event loop.



Callback function

When doing an asynchronous call, you provide a callback function. When the asynchronous function is **done**, this callback function will be called with the result of the asynchronous function.

The JavaScript engine will then execute the callback function as soon as it has no other work (the call stack is empty). The thing responsible for this is the event loop.

```
setTimeout(callbackFunction, 5000);
```

Asynchronous function that is “done” after 5ms

```
document.querySelector("main").addEventListener("click", callbackFunction);
```

Asynchronous function that is “done” when the user clicks the main element.

howest hogeschool

Promises

Some asynchronous functions work with promises. Those **immediately** return a promise object that contains the current state of the asynchronous call. When the asynchronous call hasn't been completed yet, the promise will be *pending*. When it has been completed the promise will be *fullfilled* and when an error occurred it will be *rejected*.

```
const promiseObject = fetch('http://localhost:3000/movies');
```

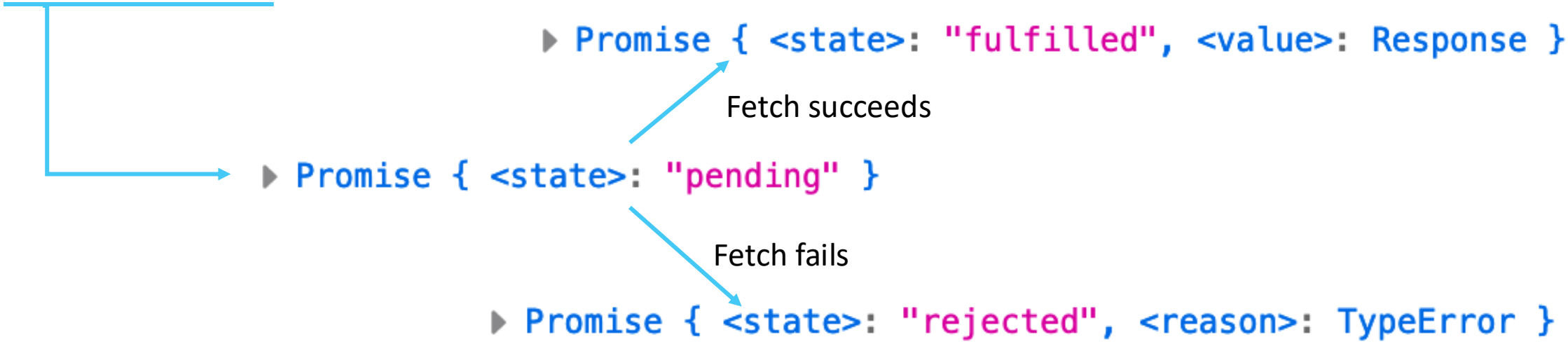
Asynchronous function that is "done" when the http request is finished.

Does **not** contain the result of the fetch!
Is a promise that will initially be pending and once the fetch has completed, it will either have the state *fullfilled* or *rejected*

Promises

Some asynchronous functions work with promises. Those **immediately** return a promise object that contains the current state of the asynchronous call. When the asynchronous call hasn't been completed yet, the promise will be *pending*. When it has been completed the promise will be *fullfilled* and when an error occurred it will be *rejected*.

```
const promiseObject = fetch('http://localhost:3000/movies');
```



Promises

Each promise object has a *then* function. This function can be given two parameters. Both are callbackfunctions. The first one will be executed when the state of the promise becomes* fullfilled, the second when the state of the promise becomes* rejected.

```
promiseObject.then(callbackFunctionWhenSucceeds, callbackFunctionWhenFails)
```

Callback function should have one argument, the fullfillment value.

Callback function should have one argument, the reason for rejection.

* when the promise is settled before calling then, the callback function will still be executed

Promises

Each promise object has a *then* function. This function can be given two parameters. Both are callbackfunctions. The first one will be executed when the state of the promise becomes* fullfilled, the second when the state of the promise becomes rejected.

```
promiseObject.then(callbackFunctionWhenSucceeds, callbackFunctionWhenFails)
```

The second parameter can be ommitted. In that case the callbackFunctionWhenFails defaults to a function that simply rethrows the error.

```
((x) => { throw x; })
```

* when the promise is settled before calling then, the callback function will still be executed

Promises

The *then* function **immediately** returns a new promise. When the promise settles and with what result depends on the return value of the callback function that was called.

All rules can be found here:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then#return_value

```
const promiseObject = fetch('http://localhost:3000/movies');
const newpromise = promiseObject.then(callbackFunctionWhenSucceeds, callbackFunctionWhenFails)
```

Does **not** contain the result of the fetch!
Is a promise that will initially be pending and once the fetch has completed, **and one of the callback functions has "completed"** will get the status fullfilled or rejected.

Promises

The *then* function **immediately** returns a new promise. When the promise settles and with what result depends on the return value of the callback function that was called.

All rules can be found here:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then#return_value

```
const promiseObject = fetch('http://localhost:3000/movies');
const newpromise = promiseObject.then(res => res.json(), callbackFunctionWhenFails)
```

The **json()** method of the Response interface takes a Response stream and reads it to completion. It returns a promise which resolves with the result of parsing the body text as JSON.

Promises

Since the *then* function returns a new promise, they can be chained.

```
const promiseObject = fetch('http://localhost:3000/movies’);
                                .then(res => res.json(), callbackFunctionWhenFails)
```

Does **not** contain the result of the fetch!
Is a promise that will initially be pending and once the fetch has completed, **and one of the callback functions has “completed”** will get the status fullfilled or rejected.

Promises

There is also a *catch* function available on promises, but this is just syntactic sugar for the following.

```
fetch('http://localhost:3000/movies')
   .then(res => res.json())
   .catch(callbackFunctionWhenFails);
```

same as

```
fetch('http://localhost:3000/movies')
   .then(res => res.json())
   .then(x => x, callbackFunctionWhenFails);
```

same as

```
fetch('http://localhost:3000/movies')
   .then(res => res.json(), x => {throw x})
   .then(x => x, callbackFunctionWhenFails);
```

howest
hogeschool

Async await

Async

An async function is a function declared with the async keyword, and the await keyword is permitted within it.

The return value of an async function is **always** a promise that will be fullfilled with the value returned by the async function, or rejected with an exception thrown from or uncaught within the async function.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Async function

```
async function myFunction() {
   return 5;
}

const result = myFunction();

const decreased = result - 1;
```

Promise that resolves to 5

NaN

result: ▸ Promise { <state>: "pending" } ⇒ ▸ Promise { <state>: "fulfilled", <value>: 5 }

Async function

Async functions always return a promise. If the return value of an async function is not explicitly a promise, it will be implicitly wrapped in a promise that resolves to the return value.

```
async function myFunction() {
   return 5;
}
```

=

```
function myFunction() {
    return Promise.resolve(5);
}
```

Await

The await operator is used to wait for a Promise and get its fulfillment value. It can only be used inside an async function or at the top level of a module.

```
const result = await expression;
```

A promise

When the promise is fulfilled, its value will be put in *result*. When it is rejected, an error will be thrown.

Await

```
async function fetchEndpoint(endpoint) {
    const responseData = await fetch(`https://jsonplaceholder.typicode.com${endpoint}`);

    const myVariable = 5;
    myVariable += 1;

    return responseData;
}

console.log(fetchEndpoint("/users"));
```

→ Promise { <pending> }

Await

```
async function fetchEndpoint(endpoint) {
    const responseData = await fetch(`https://jsonplaceholder.typicode.com${endpoint}`);

    let myVariable = 5;
    myVariable += 1;

    return responseData;
}
```

```
function fetchEndpoint(endpoint) {
    const promiseResult = fetch(`https://jsonplaceholder.typicode.com${endpoint}`)
        .then(responseData => {
            let myVariable = 5;
            myVariable += 1;
            return responseData;
        });
    return promiseResult;
}
```

Await

```
async function fetchEndpoint(endpoint) {
   const responseData = await fetch(`https://jsonplaceholder.typicode.com${endpoint}`);
   const jsonResult = await responseData.json();
   let myVariable = 5;
   myVariable += 1;
   return jsonResult;
}
```

```
function fetchEndpoint(endpoint) {
   const promiseResult = fetch(`https://jsonplaceholder.typicode.com${endpoint}`)
      .then(responseData => {
         return responseData.json().then(jsonResult => {
            const myVariable = 5;
            myVariable += 1;
            return jsonResult;
         })
      });
   return promiseResult;
}
```

Await with try/catch

```
async function fetchEndpoint(endpoint) {
   let jsonResult = undefined;
   try {
      const responseData = await fetch(`https://jsonplaceholder.typicode.com${endpoint}`);
      const jsonResult = await responseData.json();
   } catch {
      console.log("There was an error, jsonResult will be undefined");
   }
   return jsonResult;
}
```

This function will return a promise that always fullfills (succeeds). When the call to fetch fails, the catch block will be executed, the function will not throw an error and responseData will stay *undefined*. This means that promise returned by fetchEndpoint will fullfill with value *undefined*.

Await with try/catch

```
async function fetchEndpoint(endpoint) {
    const responseData = await fetch(`https://jsonplaceholder.typicode.com${endpoint}`);
    const jsonResult = await responseData.json();
    return jsonResult;
}

fetchEndpoint("/users")
    .then(users => console.log(users))
    .catch(err => console.log(err));
```

The *fetchEndpoint* function will return a promise that fullfills (succeeds) when the async function doesn't throw an error. When the fetch fails and throws an exception, the *fetchEndpoint* function will throw the error and the promise returned by it will be rejected with as value the error that was thrown.

howest
hogeschool

Arrow functions

Async arrow functions

Arrow functions can be async by prefixing the expression with the async keyword.

```
async param => expression
async (param1, param2, ...paramN) => {
  statements
}
```

```
async function anonymous(param) {
  return expression;
}

async function anonymous(param1, param2, ...paramN) {
  statements
}
```


howest hogeschool

Loops

An advantage of async/await is that you can easily use them with loops. Without async/await this is not straightforward and you will often need recursion.

```
async function fetchEndpoints(endpoints) {
    const results = []
    for (const endpoint of endpoints) {
        const result = await fetch(endpoint).then(res => res.json());
        results.push(result);
    }
    return results;
}
```

Be carefull with *foreach*. *Foreach* does not wait for each promise to resolve.

```
async function fetchEndpoints(endpoints) {
    const results = []
    endpoints.forEach(async endpoint => {
        const response = await fetch(endpoint).then(res => res.json());
        results.push(response);
    });
    return results;
}

fetchEndpoints(endpoints).then(results => console.log(results));
```

Promise will immediately fulfill

empty array

howest hogeschool

Fetch 1 by 1

```
async function fetchEndpoints(endpoints) {
    results = []
    for (const endpoint of endpoints) {
        const result = await fetch(endpoint).then(res => res.json());
        results.push(result);
    }
    return results;
}
```

Fetch in paralel

Order is not fixed!

```
async function fetchEndpoints(endpoints) {
    const results = []
    await Promise.all(endpoints.map(async endpoint => {
        const response = await fetch(endpoint).then(res => res.json());
        results.push(response);
    }));
    return results;
}
```

Filter with async await

Function to filter endpoints whose response contains data?

```
async function fetchEndpoints(enpoints) {
   return endpoints.filter(async endpoint => {
      const jsonArray = await fetch(endpoint).then(res => res.json());
      return jsonArry.length !== 0;
   })
}
```

This won't work, filter expects a function with a boolean return value. Since we pass an async function to it, the return value will be a promise.

howest hogeschool

howest
hogeschool

Top level await

Top level await

You can use the await keyword on its own (outside of an async function) at the top level of a module. This means that modules with child modules that use await will wait for the child modules to execute before they themselves run, all while not blocking other child modules from loading.

datafetcher.js

```
const vaccinations = await fetch("http://localhost:3000/vaccinations")
    .then(resp => resp.json());

const population = await fetch("http://localhost:3000/population")
   .then(resp => resp.json());

const centra = await fetch("http://localhost:3000/centra")
    .then(resp => resp.json());
```

script.js

```
  const datafetcher = await import('./modules/datafetcher.js');
```

← Will wait until datafetcher.js is completely loaded

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await#top_level_await