



# Hoe werkt JS? 🚀

Over de module en herhaling

## Agenda

1. Hoe ziet een browser er uit 😐?
2. Asynchrone JS-concepten en werking
3. JS asynchroon schrijven
4. Demo over volgorde van zaken

# <> Frontend

Development omgeving

## Work in progress (bestandsstructuur)

Vanuit de map /src.

### HTML structuur

/src/\*.html

↳ /src/index.html

We hebben meestal maar een HTML-bestand nodig. Het kan ook zijn dat we het HTML-bestand meteen in de /dist map zetten omdat we er toch niks aan willen veranderen.

### Styling

/src/style

↳ /src/style/settings/

↳ /src/style/tools/

↳ /src/style/generic/

↳ /src/style/elements/

↳ /src/style/objects/

↳ /src/style/components/

↳ /src/style/utilities/

↳ /src/style/screen.scss

Voor ITCSS maken we alle mappen aan die we kunnen gebruiken in ons project. We gaan deze bestanden bundelen in een .scss file in de root van de map.

### Scripting

/src/script

↳ /src/script/lib/

↳ /src/script/app.js

Het belangrijkste onderdeel van onze site is meestal het script gedeelte.

We maken een onderscheid tussen onderdelen die we als library gebruiken (lib) en onze file die we gebruiken om alles aan te spreken en te gebruiken in onze website.

### Testing

/src/testing

↳ /src/test/

Testing doen we ook in een aparte map. Meestal heeft een test-library nogal wat files nodig.

We kunnen makkelijk uit de test map omhoog navigeren.

## Local development (met Parcel)

Vanuit de map /src.



### Samenvoegen van:

- CSS files
- JS files

We gaan alle SASS samenvoegen tot een SASS-bestand. We doen hetzelfde voor de JS; we maken er een bestand app.js van.



### Development omgeving:

- Live reload
- Tools voor bij het developen

We gaan er ook voor zorgen dat we makkelijk aan de slag kunnen met de bestanden uit de /src map. Hier focussen we op snelheid en efficiëntie in de development periode.



### Assets verwerken

- Verkleinen
- Samenvoegen
- ...

Alles wat naar de client gaat willen we optimaliseren. Horen assets altijd bij elkaar? Die kunnen we samenvoegen. Kan iets nog gecomprimeerd worden? Dan doen we dat.



### De community

Er zijn altijd bijzondere tools die je kan gebruiken afhankelijk van je project. Zoek even via de package manager of je iets kan gebruiken voor datgene waar je mee bezig bent.

## Packing up (met Parcel)

Naar de map /dist.



### Omzetten van SASS naar CSS

Een browser kan *niks* met SASS, maar wel met CSS. Daarom zetten we alle SASS die we in een file gezet hebben om naar *leesbare* CSS.



### Uglify van:

- JS
- CSS
- HTML

We maken alles zo klein mogelijk om alles te transporteren naar de browser.



### Sourcemap

- Waar stond dat?
- CSS & JS

Als we alles samengevoegd hebben is het soms moeilijk om nog te weten wat de originele code was en waar iets staat. Sourcemap helpen om terug te linken naar de originele werkbestanden.



## Backend

Transport  
Per soort 1x.

# Browser

Client side

## Rendering

Hoe wordt onze website weergegeven?

### HTML parsing

De HTML wordt door de browser omgezet naar 'tokens'. Dat wordt een token-tree en dan wordt dat de DOM.

### CSS parsing

Van de tekst in het CSS bestand wordt een CSSOM gemaakt. Dit zijn nodes die stukjes style bevatten.

### Render tree

De lagen in de browser worden bepaald volgens de DOM en de CSSOM. Ook wordt gekeken naar de tekst en worden de lijnen uitgezet van de elementen.

### Layout

Terugkerend proces dat alle stijlen toepast, eerst van de child-elementen tot de parent. Past altijd aan als de DOM update. Waar komt de (nieuwe) node?

### Painting

Neemt de info van waar alles staat en gaat dit weergeven (als bitmap). Alle kleuren, volgens positie, shadows, transparantie, z-index, etc.. Kortom: GPU werk.

## JS engine

Hoe wordt ons script (asynchroon) uitgevoerd?

### Javascript Runtime

Kan maar een iets tegelijk.

Heap  
Geheugen plaatsen voor sommige JS-objecten.

### Stack

Houdt bij welke functie we uitvoeren. Waar zijn we nu in de code?

### Web API's

Dit zijn externe zaken die je nodig hebt in een browser; het aanspreken van de DOM, timeOuts en timers, etc...

## Event Loop

Is de stack leeg? Dan neem ik het eerste uit de event loop en zet ik dat in de stack.

### Callback Queue

Hier staan alle onderdelen die een Web API afgehandeld heeft. We kunnen dit niet zomaar in de stack zetten, dus zetten we het in een wachtrij. Dit is vaak een callback.

# De basiswerking van een browser

Hoe code uitgevoerd wordt.

# Browser: (call) stack



```
function multiply(a, b) {  
    return a * b;  
}  
  
function square(n) {  
    return multiply(n, n);  
}  
  
function printSquare(n) {  
    let squared = square(n);  
    console.log(squared);  
}  
  
printSquare(4);
```

Stack

# Browser: (call) stack

```
→ function multiply(a, b) {  
    return a * b;  
}  
  
function square(n) {  
    return multiply(n, n);  
}  
  
function printSquare(n) {  
    let squared = square(n);  
    console.log(squared);  
}  
  
printSquare(4);
```

Stack

main()

# Browser: (call) stack

```
function multiply(a, b) {  
    return a * b;  
}  
  
function square(n) {  
    return multiply(n, n);  
}  
  
function printSquare(n) {  
    let squared = square(n);  
    console.log(squared);  
}  
→ printSquare(4);
```

Stack

printSquare(4)

main()

# Browser: (call) stack

```
function multiply(a, b) {  
    return a * b;  
}  
  
function square(n) {  
    return multiply(n, n);  
}  
  
→ function printSquare(n) {  
    let squared = square(n);  
    console.log(squared);  
}  
  
printSquare(4);
```

Stack

square(n)

printSquare(4)

main()

# Browser: (call) stack

```
function multiply(a, b) {  
    return a * b;  
}  
  
→ function square(n) {  
    return multiply(n, n);  
}  
  
function printSquare(n) {  
    let squared = square(n);  
    console.log(squared);  
}  
  
printSquare(4);
```

Stack

multiply(n,n)

square(n)

printSquare(4)

main()

# Browser: (call) stack

```
→ function multiply(a, b) {  
    return a * b;  
}  
  
function square(n) {  
    return multiply(n, n);  
}  
  
function printSquare(n) {  
    let squared = square(n);  
    console.log(squared);  
}  
printSquare(4);
```

Stack

multiply(n,n)

square(n)

printSquare(4)

main()

# Browser: (call) stack

```
function multiply(a, b) {  
    return a * b;  
}  
  
→ function square(n) {  
    return multiply(n, n);  
}  
  
function printSquare(n) {  
    let squared = square(n);  
    console.log(squared);  
}  
  
printSquare(4);
```

Stack

square(n)

printSquare(4)

main()

# Browser: (call) stack

```
function multiply(a, b) {  
    return a * b;  
}  
  
function square(n) {  
    return multiply(n, n);  
}  
  
→ function printSquare(n) {  
    let squared = square(n);  
    console.log(squared);  
}  
  
printSquare(4);
```

Stack

console.log(squared)

printSquare(4)

main()

# Browser: (call) stack

```
function multiply(a, b) {  
    return a * b;  
}  
  
function square(n) {  
    return multiply(n, n);  
}  
  
function printSquare(n) {  
    let squared = square(n);  
    console.log(squared);  
} →  
printSquare(4);
```

Stack

printSquare(4)

main()

# Call stack

Kan slechts een iets tegelijk doen.  
Houdt ‘de plaats’ in de code bij.  
Wat in de stack komt, wordt meteen uitgevoerd.  
Je krijgt de stack te zien bij sommige fouten:

```
✖ ▼Uncaught ReferenceError: fail is not defined      VM303:2
    at multiply (<anonymous>:2:3)
    at square (<anonymous>:7:10)
    at printSquare (<anonymous>:11:17)
    at <anonymous>:15:1

multiply  @ VM303:2
square    @ VM303:7
printSquare @ VM303:11
(anonymous) @ VM303:15
```



# Call stack

Kan slechts een iets tegelijk doen.

Houdt ‘de plaats’ in de code bij.

Wat in de stack komt, wordt meteen uitgevoerd.

Je krijgt de stack te zien bij sommige fouten:

Wat als:

- Je ergens op moet wachten?
- iets nog niet geladen is?



# Wat als...

---

Er zaken zijn die lang kunnen duren (of zelfs nooit klaar zijn...).  
Hoe ziet dat er uit; hoe gaat dat in de browser?

# Browser: (call) stack

```
function getData(u) {  
    let data = fetchAPISync(u);  
    console.log(data);  
}  
  
getData('https://mct.be/api/');
```

Stack

main()

# Browser: (call) stack

```
→ function getData(u) {  
  let data = fetchAPISync(u);  
  console.log(data);  
}  
  
getData('https://mct.be/api/');
```

Stack

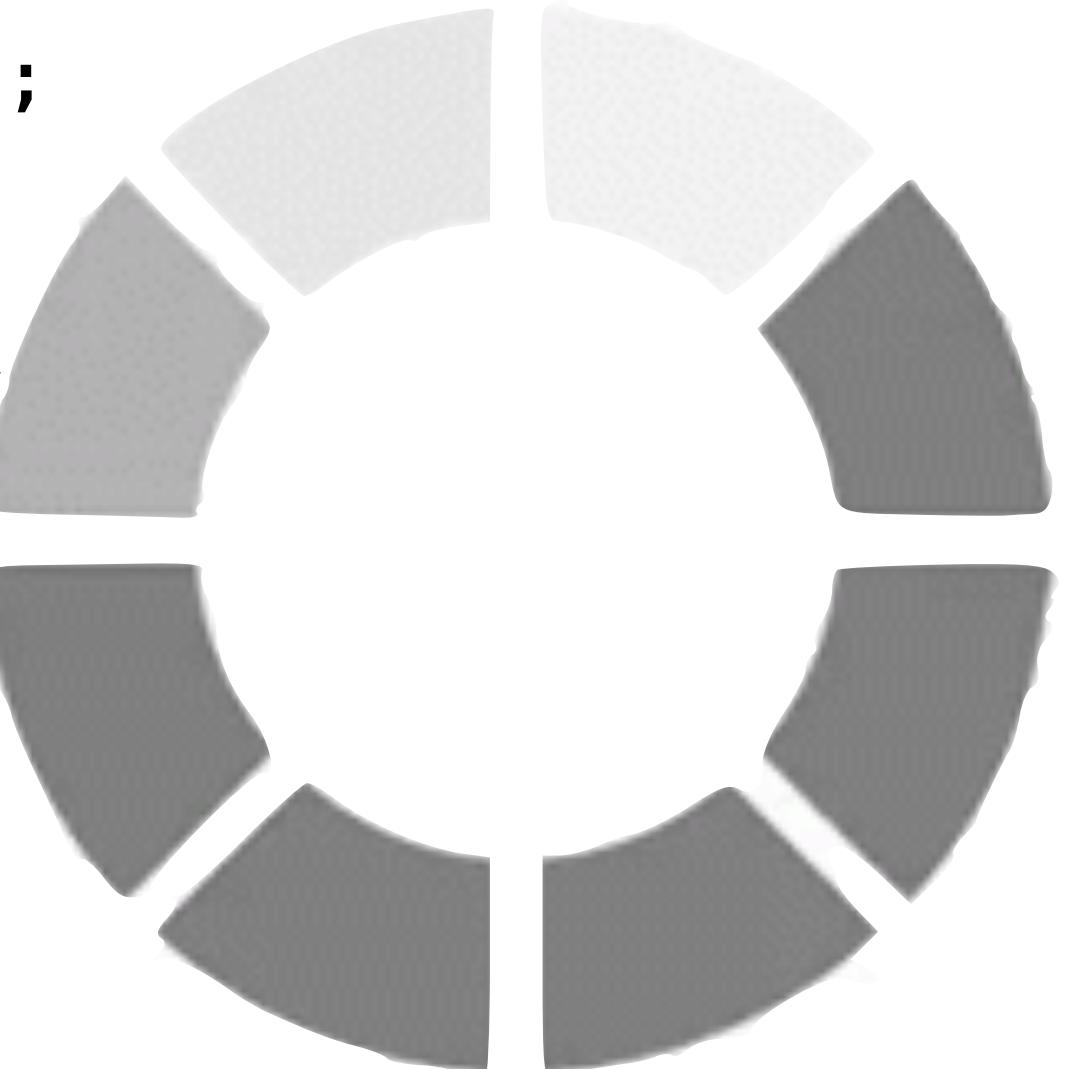
getData(u)

main()

# Browser: (call) stack

```
function getData(u) {  
    let data = fetchAPISync(u);  
    console.log(data);  
}
```

→ `getData('https://mct.be/api/')`



Stack

getData(u)

main()

# Asynchrone JS concepten en werking

Snelheid en orde.

# Asynchrone JS

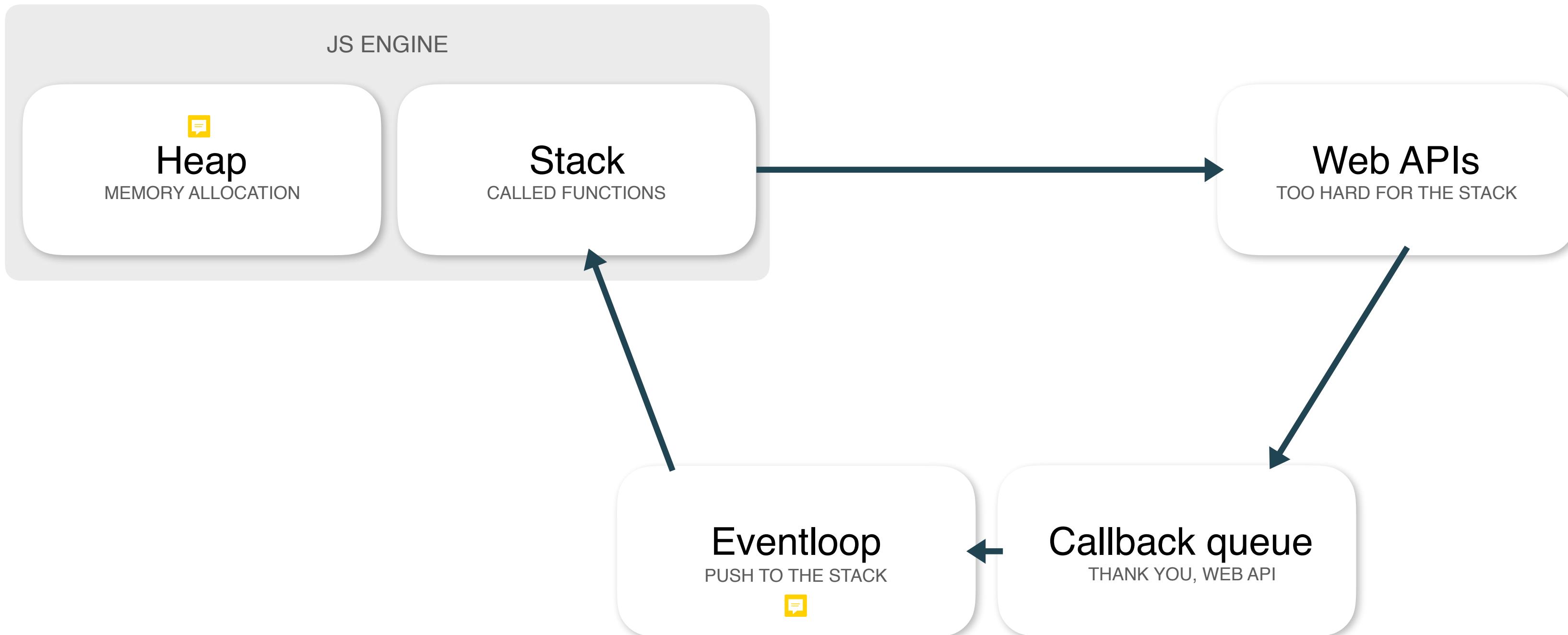
Een combinatie van de Stack, met externe hulpfunctionaliteiten.

Gemaakt om alles zo snel mogelijk te laten uitvoeren.

Hoe ziet de browser er dan uit:



# JS in de browser



# Een demo met dit in gedachten:

---

Basis werking

Volgorde van uitvoeren.



Opmerkingen:

Alles is een queue / lijst

Wordt systematisch geleegd (oudste gaat eerst weg)



# JS is:

---

Single Threaded 

Alle code wordt altijd op één proces gerund.

Blocking: als iets aan het runnen is, is het wachten totdat dit klaar is.

Je maakt het non-blocking door alles event-driven te maken.

Kleine code onderdelen die telkens na elkaar uitgevoerd kunnen worden.

Synchroon en asynchroon... 

Maar... wat als je een heel trage callback functie hebt?

→ SOON Dan krijg je problemen... JS is niet geschikt voor zware  taken. Wel voor snelle acties en veel kleine events die verwerkt moeten worden.

# JS Asynchroon schrijven

Call me back.

# Het schrijven van asynchrone JS

---

We bekijken verschillende manieren om JS te schrijven.

*There are memes to ease the pain.*

De principes zijn telkens hetzelfde.

Er zijn wel verschillende nieuwe manieren waarop dit netter kan geschreven worden (gelukkig 😊).

Maak *kleine callbacks*:

Denk eraan dat alles terug op de stack komt.

De *single-thread* moet alles (uiteindelijk) afhandelen.

# Asynchrone JS: callbacks

Duidelijk hoe je een functie-object als volgende uitvoert.

Snelle manier; de basis voor de browser.

Nooit een return schrijven! 

Kan verwarringen veroorzaken.

Al iets oudere code.

demo: <https://codepen.io/MLoth/pen/wvaGvNV>



# Asynchrone JS: promises

---

Syntactic sugaring

Vlotte (nieuwe) manier van schrijven.

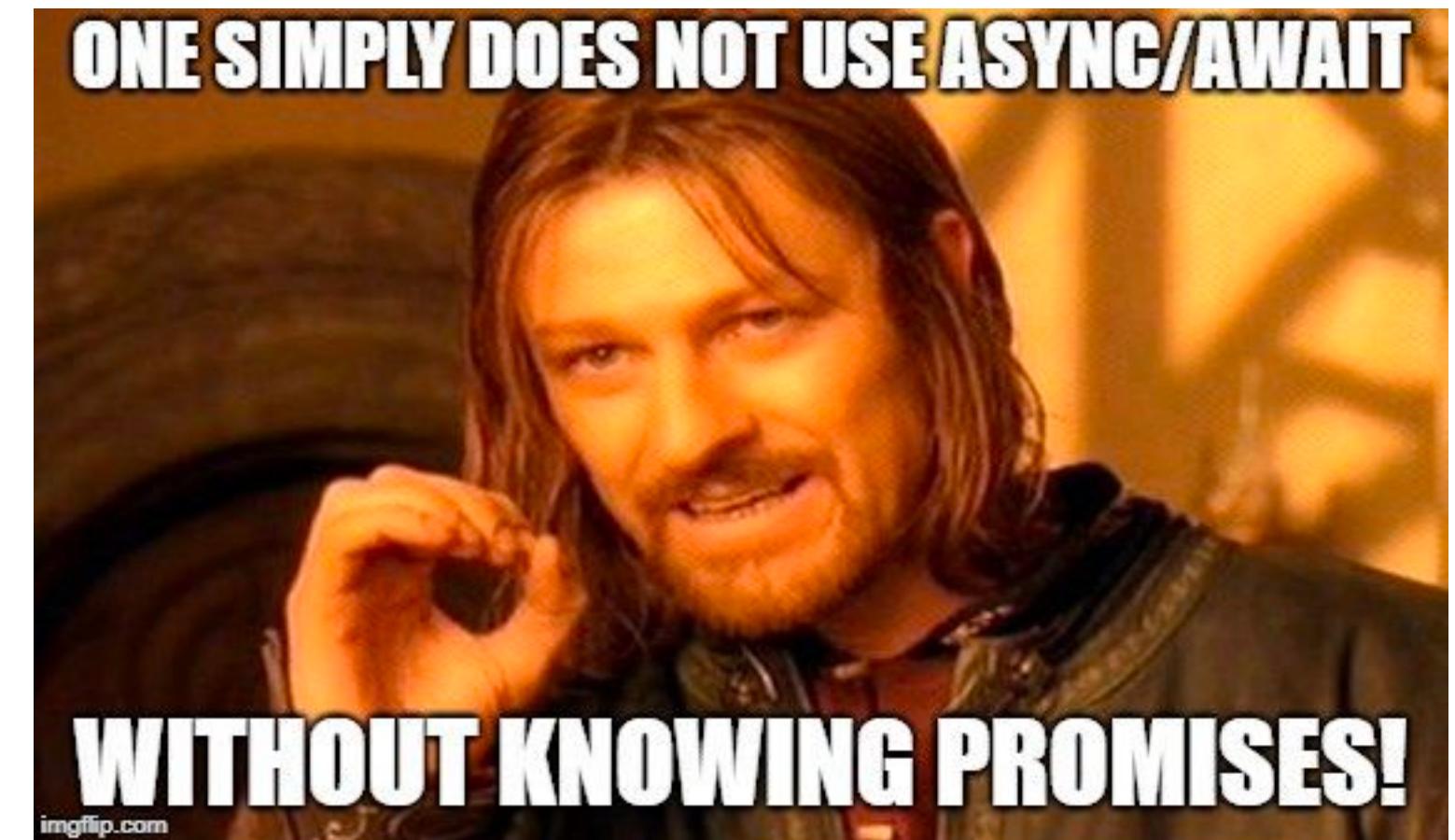
Andere mogelijkheden: bv.: resolve all.

Promise kan maar 1 iets hebben: ok of error.

Hiermee behoud je een logische volgorde en structuur.

Zie ook fetch-API.

Hier gaan we wel return schrijven.



demo: <https://codepen.io/MLoth/pen/mdJPjNa>

# Asynchrone JS: `async/await`

---

Nog betere volgorde beheersing.

Beter om te debuggen.

Leest als synchrone (opeenvolgende) code.

Vertrekt ook vanuit promises.

Hier gaan we wél return schrijven.

demo: <https://codepen.io/MLoth/pen/gOprjVE>



**CALLBACKS**

---

**PROMISES**

---

**ASYNC/AWAIT**

# Asynchrone code: Wat kiezen?

---

Callbacks

Promises

Soms betere fout-afhandeling.

Custom promises maken om te kunnen gebruiken.

**De basis om async / await toe te kunnen passen!**

Async / await

Bijna altijd eenvoudiger en leesbaarder.

Try catch voor fouten soms niet zo galant.

# Asynchrone JS: timing



setInterval -> om een tal ms code uitvoeren.

setTimeout -> na x ms code uitvoeren.

Timing in JS is niet exact.

Deze functies returnen een uniek ID om de timers eventueel te stoppen, met:

clearInterval

clearTimeout

demo: <https://codepen.io/MLoth/pen/xxGVaKP>

# Asynchrone JS: side-effects & duidelijkheid over vars

---

```
for (var i = 0; i < 2; i++) {           > 0
    console.log( i );
}                                         > 1
```

# Asynchrone JS: side-effects & duidelijkheid over vars

---

```
for (var i = 0; i < 2; i++) {  
    setTimeout(  
        () => console.log(i)  
    );  
}
```

> 2  
> 2

# Asynchrone JS: side-effects & duidelijkheid over vars

---

💡

```
for (let i = 0; i < 2; i++) {
  setTimeout(
    () => console.log(i)
  );
}
```

> 0  
> 1



# Conclusie



Voor een let wordt er een lokale scope gemaakt om deze variabele bij te houden.

Bij een var zou je met dezelfde variabele blijven werken.

setTimeout kan vreemde outputs geven.

Let op met asynchrone code!

# JS engines & browsers

De basics

# Een JS engine is:

---

Een virtuele machine (zie soms vm: in de stack)

- Neemt JS-code en zet het om naar machine taal

- Geschreven in C++

- Kan ook runnen server-side -> Node.js

Twee belangrijkste:

- V8:** Chrome, Opera

- WebKit:** Safari

- Gecko:** Firefox

# Er zijn nog andere engines:

---

**V8** – open source, developed by Google, written in C++

**Rhino** – managed by the Mozilla Foundation, open source, developed entirely in Java

**SpiderMonkey** – the first JavaScript engine, which back in the days powered Netscape Navigator, and today powers Firefox

**JavaScriptCore** – open source, marketed as Nitro and developed by Apple for Safari

**Chakra (JScript9)** – Internet Explorer

**Chakra (JavaScript)** – Microsoft Edge

**JerryScript** – is a lightweight engine for the Internet of Things.

# V8: Enorm krachtig en geoptimaliseerd!

---

Gaat altijd:

Eerst interpreteren

Dan uitvoeren

Gebaseerd op Chromium, gaat IE nu ook verder op werken!

Meer info: <https://medium.freecodecamp.org/javascript-essentials-why-you-should-know-how-the-engine-works-c2cc0d321553>

# BOM in de browser (staat los van JS engine)

---

Browser object model

Bevat: navigator, history, screen, location en document wat dan weer onderdelen van window zijn.

Is geen standaard voor, elke browser maakt hier zelf iets van.

Vergelijk met DOM qua aanspreken, etc.

= Browser API's.

1. Wat houdt single threaded in, bij JS? 
2. Wat is de event-loop en de callback queue?   
Wat is hun onderlinge relatie?
3. Hoe kan je de volgorde in JS behouden? 
4. Wat is een promise? Wat is het verschil met callbacks?
5. Hoe voer ik iets om de seconde uit in JS?
6. Wat zijn de browser API's?
7. Is JS synchroon of asynchroon? 
8. Wat als ik een heel 'zware' callback functie heb?
9. Hoe maak je een promise?
10. Hoe maak je JS non-blocking? 
11. Wat is BOM? 
12. Wat is async / await? 

## Warme oproep

1. Zijn er nog thema's die je graag wil zien deze module?



**howest**  
hogeschool