



Webdevelopment II

Hoofdstuk 08

Asynchroon programmeren in JS

Inhoud

- Synchronous JavaScript vs Asynchronous JavaScript
- Ajax
- Callbacks
- Consuming Promises en de Fetch API
- Promise.all() en Promise.any()
- Promises
- async/await

Synchronous JavaScript vs Asynchronous

Synchronous JavaScript

- Tot nu toe hebben we steeds synchrone code gebruikt.
- Bij het uitvoeren van een stuk synchrone JavaScript code worden de statements, van boven naar beneden het ene na het andere uitgevoerd. Elk statement moet hierbij volledig beëindigd zijn vooraleer het volgende statement wordt uitgevoerd.

Asynchronous JavaScript

1/2

Verschillende programma's interageren echter met zaken buiten de processor. Bijvoorbeeld communicatie via het netwerk met een andere computer. Deze communicatie verloopt veel trager dan bijvoorbeeld iets uit het intern geheugen halen.

Terwijl het programma wacht op een network request kan dus eventueel reeds andere code uitgevoerd worden die het resultaat van de network request niet nodig heeft.

Dit is mogelijk in een asynchroon programma.

Asynchronous JavaScript

2/2

In een asynchroon programma kunnen er verschillende dingen tegelijkertijd gebeuren.

Wanneer je een asynchrone actie start gaat je programma gewoon door met de uitvoering.

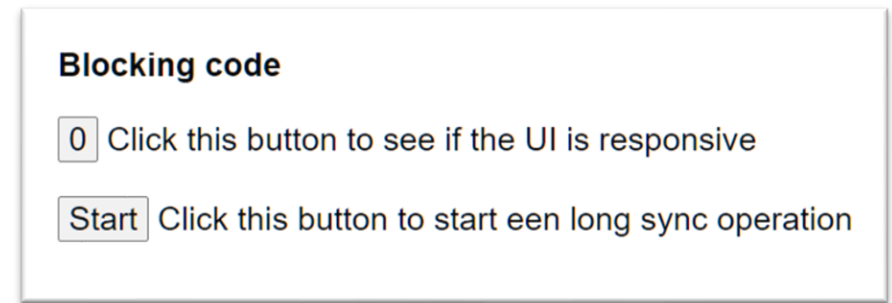
Wanneer de asynchrone actie beëindigd is wordt het programma hierover geïnformeerd en krijgt het toegang tot het resultaat van de asynchrone actie (bijv. de response van de network request).

Synchronous JavaScript is 'blocking'

- Als een script in een browser, een lang stuk synchrone code uitvoert zal de browser niet meer reageren op acties van de gebruiker. De code is dan 'blocking'.
Dit komt omdat de **JavaScript engine** synchroon werkt. Deze maakt slechts gebruik van één thread (de main thread) en werkt met een event-loop.
Voor meer info zie: [What the heck is the event loop anyway?](#)
- Naast de JavaScript engine voorziet de host environment, in ons geval de Browser, echter in een aantal API's die wel asynchroon werken. Zoals bijvoorbeeld de **XMLHttpRequest** (om http-requests uit te voeren) en zijn nieuwere broertje de **Fetch API**. Ook **setTimeout()** is bijvoorbeeld een asynchrone functie.

1-synchronous-blocking-code.html

- Open het bestand 1-synchronous-blocking-code.html met de live server.
- Controleer dat de UI (User Interface) 'responsive' is door een aantal keren op de eerste knop te klikken.
- Start een langdurend synchroon script door op de tweede knop te klikken.
- Test onmiddellijk daarna of de UI nog 'responsive' is, door opnieuw enkele keren op de eerste knop te klikken.
Pas na drie seconden zal de UI opnieuw reageren op de muiskliks. Terwijl de synchrone code uitgevoerd wordt, wordt al de rest dus gepauzeerd, ook het renderen van de webpagina.



Asynchronous JavaScript is 'not blocking'

- Om onder andere te voorkomen dat de UI 'unresponsive' wordt voorzien browsers dus asynchrone functies voor acties die lang duren.
Asynchrone code is immers 'not blocking', zodat de UI 'responsive' blijft. Wat een groot voordeel is.
- Zoals reeds vermeld wordt er bij de oproep van een asynchrone functie niet gewacht tot de functie beëindigd is, maar wordt onmiddellijk het volgende statement uitgevoerd.

Voorbeeld asynchrone operatie

- Als eerste voorbeeld van een langdurende actie die asynchroon verloopt zullen we de werking van het **XMLHttpRequest**-object bekijken.
- We zullen het **XMLHttpRequest**-object slechts kort behandelen omdat tegenwoordig (en wij zullen dit ook doen) meer en meer gebruik wordt gemaakt van de recentere **Fetch api**, zie verderop.
- Maar eerst enige historie in verband met AJAX.



AJAX

Asynchronous JavaScript And XML

AJAX: een kort historisch overzicht

- Bij de start van het World Wide Web in de 90's waren webpagina's statisch. Voor elke (kleine) wijziging aan de inhoud van een webpagina moest een volledig nieuwe webpagina opgevraagd worden.
- In 1999 implementeerde Microsoft een XMLHTTP-request in Internet Explorer, initieel voor hun Outlook web client. Deze request liet toe om **asynchroon**, in de achtergrond via JavaScript, data door te sturen en te ontvangen van een server. Andere browsers begonnen dit dan ook te implementeren. Maar deze feature was weinig bekend en werd weinig gebruikt.

AJAX: een kort historisch overzicht (vervolg)

- Het was Google die met de lancering van Google Suggest en Google maps in 2004/2005 het asynchroon laden van data onder de aandacht bracht.
- In 2005 werd de term **AJAX** (**A**synchronous **J**avaScript **A**nd **X**ML) gelanceerd door Jesse James Garret in zijn artikel '[Ajax: A New Approach to Web Applications](#)' waarin hij verwijst naar de technieken die door Google gebruikt werden in hun recente webapplicaties en de term werd door grote bedrijven als Google en Amazon overgenomen.

AJAX (spreek uit: eidzæks)

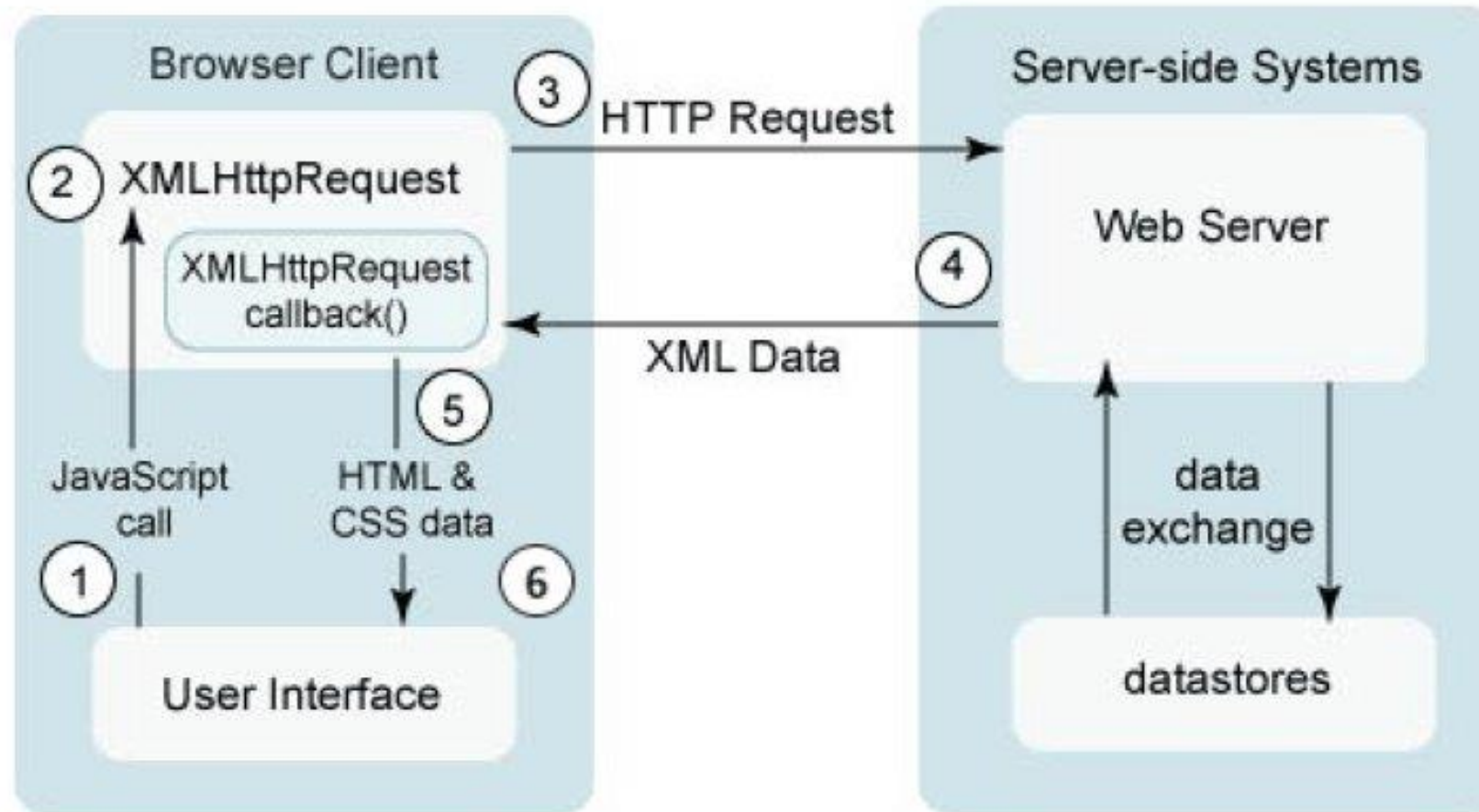
- AJAX maakt dus gebruik van het XMLHttpRequest om asynchroon data op te vragen aan de web server om vervolgens met behulp van JavaScript (DOM) de webpagina te updaten.
- De webpagina moet dus niet volledig opnieuw opgevraagd worden om nieuwe inhoud te krijgen.
- *Google Suggest* stelt bijvoorbeeld bij elke toetsaanslag een nieuwe reeks zoektermen voor zonder dat men de pagina hoeft te herladen. Zo'n pagina is te vergelijken met een gewone desktopapplicatie.

AJAX

Ajax is een manier om interactieve webapplicaties te ontwikkelen door een combinatie van de volgende technieken te gebruiken:

- **HTML** en **CSS** voor de presentatie volgens de standaarden van het W3C.
- Het **Document Object Model** voor het dynamisch tonen van informatie en voor interactie.
- **XML** voor de opslag, aanpassing en transport van gegevens. In veel gevallen wordt tegenwoordig in plaats van XML, **JSON** (JavaScript Object Notation), gewone tekst of een ander formaat gebruikt. Toen dit echter bedacht en ontwikkeld werd door Microsoft was XML dé manier om data uit te wisselen, vandaar.
- Het **XMLHttpRequest-object** voor asynchrone communicatie met de back-end server.
- **JavaScript** om alles aan elkaar te binden.

AJAX



Voorbeeld: XMLHttpRequest

- Om Ajax requests te illustreren hebben we een server (backend) nodig waaraan we over http data kunnen opvragen. We zullen daarvoor in eerste instantie onze development server gebruiken.
- In **2a-asynchronous-code-prob.html** zullen we proberen om via het **XMLHttpRequest**-object de inhoud van de file **/data/part1.txt** op te vragen en die vervolgens via DOM-manipulatie te injecteren in onze webpagina. Daar het XMLHttpRequest echter asynchroon werkt zullen we een probleem hebben. De request zal nog niet afgehandeld zijn op het moment dat we de info al proberen af te beelden.

Voorbeeld: XMLHttpRequest (XHR)

2a-asynchronous-code-prob.html

```
// 1. Creëer een nieuw XMLHttpRequest-object.
const xhr = new XMLHttpRequest();
// 2. Initialiseer het object met de open-methode
//     .open(method, URL, [async, user, password])
xhr.open('GET', '../data/part1.txt');
// 3. 'Send' de http-request.
//     Deze methode opent een connectie en verstuurt
//     de http-request naar de server.
//     xhr.send() is een asynchrone opdracht
xhr.send();

// xhr.status zal hier nul bevatten omdat de asynchrone XMLHttpRequest-opdracht
// nog niet afgerond is.
helpers.addMessage(`- xhr.status (HTTP-statuscode): ${xhr.status}`);
helpers.setText(xhr.responseText);
```

2a-asynchronous-code-prob.html (vervolg)

- Open **02a-asynchronous-code-prob.html**
- Klik op de knop **Send HTTP-request**.
Merk op dat daar de XMLHttpRequest asynchroon wordt uitgevoerd, na de 'send' , zoals reeds vermeld, onmiddellijk de opdrachten **addMessage()** en **setText()** worden uitgevoerd, terwijl de asynchrone operatie nog niet is afgerond. We krijgen dus als status 0. Zulke problemen zijn typisch voor asynchrone programma's.
- Pas door te klikken op de knop **Reprint 'xhr.status' en 'xhr.text'** krijgen we de correcte status nl. 200 en wordt de tekst uit de file geïnjecteerd in de webpagina.

Callbacks

Hoe problemen met asynchrone code oplossen?

- Om problemen met asynchrone code, zoals in het voorbeeld, op te lossen zijn er verschillende mogelijkheden:
 - de oudere manier is via **Callbacks**
 - de nieuwe manier is via **Promises**
 - en een nog nieuwere manier, die ook gebruikmaakt van promises, is **async/await**.
- We starten met de oudere manier van callbacks. Bij deze werkwijze geven we een callback-functie door en deze callback-functie(s) zal pas uitgevoerd worden bij het beëindigen van de asynchrone actie.

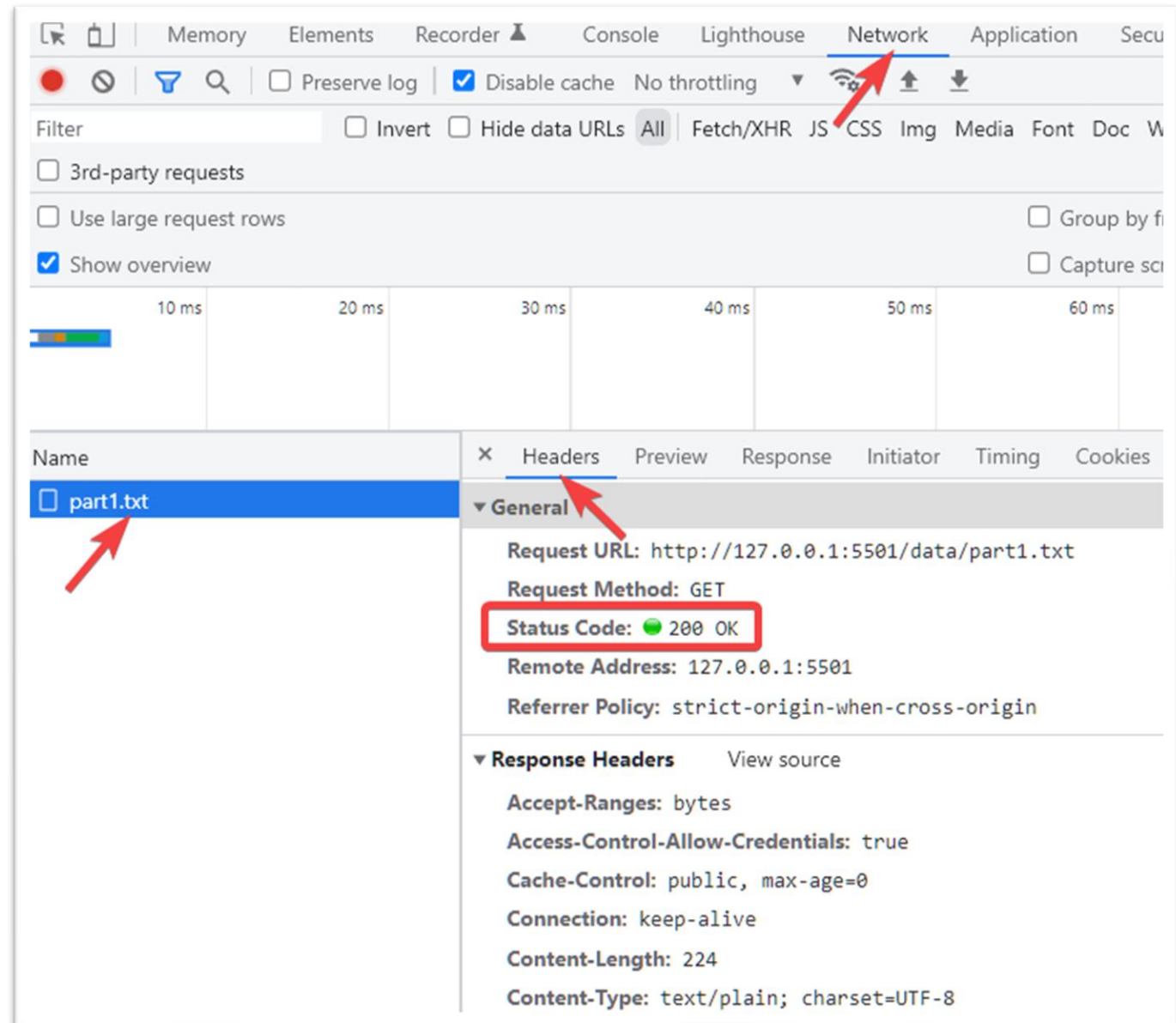
2b-asynchronous-code-callback-solution.html

In deze file werken we met een callback-functie die pas wordt uitgevoerd als de http-request volledig is afgehandeld. Hierdoor wordt de http-status nu wel correct afgebeeld als we klikken op de knop **Send HTTP-request** en wordt ook de tekst weergegeven.

```
xhr.onload = () => {  
    helpers.addMessage(` - xhr.status (HTTP-statuscode): ${xhr.status}`);  
    helpers.setText(xhr.responseText);  
};
```

Chrome developer tools > Network

- Met de Chrome developer tools kan je op de Netwerk tab info over de http-request bekijken



Callback hell of 'Pyramid of Doom'

- Het probleem met callbacks is dat als we meerdere asynchrone operaties na elkaar moeten uitvoeren de code al vlug onoverzichtelijk wordt omdat er telkens een niveau ingesprongen wordt.
- Zie **2c-asynchronous-code-callback-solution-two-parts.html** waarbij we twee asynchrone operaties uitvoeren.


2c-asynchronous-code-callback-solution-two-parts.html

```
xhr1.open('GET', '../data/part1.txt');
helpers.addMessage('- BEFORE request send 1');
xhr1.send();
helpers.addMessage('- AFTER request send 1');
xhr1.onload = () => {
    helpers.addMessage(`HTTP-statuscode part1: ${xhr1.status}`);
    xhr2.open('GET', 'data/part2.txt');
    helpers.addMessage('- BEFORE request send 2');
    xhr2.send();
    helpers.addMessage('- AFTER request send 2');
    xhr2.onload = () => {
        helpers.addMessage(`HTTP-statuscode part2: ${xhr2.status}`);
        helpers.setText(xhr1.responseText + xhr2.responseText);
    };
};
```

Callback hell of 'Pyramid of Doom'

- Bij elke bijkomende asynchrone operatie groeit de code dus naar rechts. We krijgen dus al vlug wat men soms noemt een 'Pyramid of Doom' of 'callback hell', zeker als we ook nog error-code toevoegen. Zie [Pyramid of Doom op JavaScript.info](https://javascript.info/callbacks#pyramid-of-doom).

```
loadScript('1.js', function(error, script) {  
  if (error) {  
    handleError(error);  
  } else {  
    // ...  
    loadScript('2.js', function(error, script) {  
      if (error) {  
        handleError(error);  
      } else {  
        // ...  
        loadScript('3.js', function(error, script) {  
          if (error) {  
            handleError(error);  
          } else {  
            // ...  
          }  
        });  
      }  
    });  
  }  
});
```



Bron: <https://javascript.info/callbacks#pyramid-of-doom>

Nog een probleem met Callbacks

Een ander probleem met callbacks doet zich voor als twee parallelle asynchrone functies gelijktijdig worden opgestart. Wanneer kan een derde functie gestart worden nadat beide parallel opgestarte functies afgehandeld zijn?

Dit is het geval bij het laatste voorbeeld (dia 29). We kunnen in principe het ophalen van part1 en part2 parallel starten en we hoeven enkel te wachten om de delen samen te voegen.

Merk op dat als we het ophalen van part1 en part2 parallel opstarten we niet weten (omdat we asynchroon werken) welke van de twee requests eerst zal afgerond worden.

Callback problemen oplossen via promises

- Om problemen zoals vermeld op de vorige dia en het probleem van de callback hell op te lossen gaan we gebruikmaken van promises.
- Het idee achter promises is al oud (jaren '70), maar promises zijn pas opgenomen in ECMAScript 2015 (ES6).
- De meeste programmeertalen kennen het concept ondertussen, hoewel ze vaak 'futures' genoemd worden (C++, Java, ...)

BRONNEN: [Futures and promises – Wikipedia](#), [ECMAScript 2015 \(ES6\) | Can I use... Support tables for HTML5, CSS3, etc](#)

Consuming promises en de Fetch API

Promise

- Het idee achter promises is de toestand en het resultaat van een asynchrone operatie bij te houden in een promise-object.
- Dit promise-object zal natuurlijk niet onmiddellijk het resultaat bevatten, maar houdt de toestand van de asynchrone operatie bij.
- De volgende functie kan gestart worden, zonder rekening te houden met de toestand van de promise.
- Om het toekomstige resultaat van de asynchrone functie te kunnen gebruiken zal je een callback moeten koppelen aan de promise met behulp van de **then** method. Deze callback zal uitgevoerd worden van zodra de asynchrone functie succesvol is afgehandeld. In de callback zal je dan het resultaat van de asynchrone functie kunnen gebruiken.

Fetch API

The Fetch API is vergelijkbaar met de werking van XHR maar is promise based.

- De **fetch()**-methode heeft één verplicht argument, nl. de URL van de 'resource' die je wil ophalen.
- De **fetch()**-methode retourneert een Promise (een belofte) die zal omgezet worden ('resolves') in een Response-object. Vervolgens zijn er een aantal mogelijkheden om het Response-object verder te verwerken:
 - [Response.text\(\)](#): retourneert een Promise die de body als text (UTF-8) teruggeeft.
 - [Response.json\(\)](#): retourneert een Promise die de body teruggeeft geparset naar JSON

3a-fetch-response-promise-long.html

```
const fetchPromise = fetch('data/part1.txt');
console.log(fetchPromise);

fetchPromise.then((response) => {
  console.log('response.status:', response.status);
});

console.log('Started request...')
```

De `fetch()`-aanroep retourneert een promise, die 'resolves' in een `Response`-object. De tekst uit de file (de body van de request) is op dat moment echter nog niet beschikbaar (het is een 'readable stream'). Om deze tekst te bekomen zullen we nog de asynchrone aanroep `response.text()` moeten uitvoeren (zie volgende dia). Die opnieuw een promise zal retourneren dewelke we dan opnieuw via een `.then` moeten verwerken.

3b-fetch-text-promise.html

```
const fetchPromise = fetch('./data/part1.txt');

fetchPromise.then((response) => {
  const textPromise = response.text();
  textPromise.then((text)=> {
    console.log(text);
  });
});

console.log('Started request...');
```

Merk op dat we in de 'then' terug een 'then' hebben. We hebben dus hetzelfde probleem als bij de call backs. Als we dus verschillende asynchrone operaties na elkaar gaan uitvoeren gaan we opnieuw verschillende geneste niveaus in onze code hebben. We kunnen dit nu echter oplossen door gebruik te maken van 'promise chaining'.

3c-fetch-text-promise-chaining.html

```
const fetchPromise = fetch('./data/part1.txt');

fetchPromise
  .then((response) => {
    return response.text();
  })
  .then((text) => {
    console.log(text);
  });

console.log('Started request...');
```

In plaats van de tweede 'then' op te roepen in de eerste 'then' retourneren we de promise `response.text()` en roepen we de tweede 'then' aan op die return-waarde.

Opmerking chaining van `.then`'s is altijd mogelijk want een `.then` retourneert altijd een promise. Ook als de return-waarde geen promise is. In dit geval wordt de return-waarde automatisch 'gewrapt' in een promise.

Vershil tussen promise en callback

- Op het eerste zicht is er weinig verschil met de callback. Je gaat pas een functie aanroepen (callback) van zodra de asynchrone functie is afgehandeld (**then** method van de promise), maar door 'promise chaining' te gebruiken is de code leesbaarder (meer sequentieel) in het geval van meerdere asynchrone acties en we zullen ook het probleem, waarbij twee parallelle asynchrone functies gelijktijdig worden opgestart en we een derde functie moeten opstarten nadat beide parallel opgestarte functies afgehandeld zijn, kunnen oplossen. Zie verderop **Promises.all**.

Een JSON-file inlezen

- We zouden voor het inlezen van een json-file `Response.text()` kunnen gebruikmaken, omdat een json-file een gewone tekst-file is, maar dan zouden we nog de tekst moeten parsen naar een object met `JSON.parse()`. Daarom dat we `Response.json()` zullen gebruiken want deze methode leest niet alleen de tekst in, maar zet die ook onmiddellijk om naar een JavaScript-object.

3d-fetch-json.html

```
const fetchPromise = fetch('./data/data1.json');

fetchPromise
  .then((response) => {
    return response.json();
  })
  .then((jsonResponse) => { // waarbij 'jsonResponse' een object is
                           // dat de 'parsed json' bevat.
    console.log(jsonResponse);
  });

console.log('Started request...');
```

4e-fetch-json.html

```
// zonder gebruik te maken van de hulpvariable fetchPromise bekomen we
// onderstaande code:
fetch('./data/part1.json')
  .then((response) => {
    return response.json();
  })
  .then((jsonResponse) => {
    console.log(jsonResponse);
  });

console.log('Started request...');
```

Handling errors

- Tijdens het werken met de Fetch-api kan er om verschillende redenen een fout optreden: het netwerk is down, de url bevat een fout, ...
- Om fouten af te handelen hebben promises naast een **then**-methode ook een **catch**-methode, waaraan je zoals bij de **then** een handler function doorgeeft. Deze **catch handler** zal dan uitgevoerd worden als de asynchrone operatie *faalt*.
- Wanneer je de catch toevoegt op het einde van de 'promise chain' zal deze uitgevoerd worden wanneer één van de asynchrone operaties faalt en hebben we dus één plaats waar we alle errors kunnen afhandelen.

Handling http errors

- In tegenstelling tot wat je misschien zou denken faalt de fetch-methode niet als er een http-error optreedt. Bijgevolg moeten we zelf de http status code checken en een error gooien als de response niet OK is. De uiteindelijke code vind je op de volgende slide.

3f-fetch-json-FINAL.html

```
fetch('./data/data1.json')
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((json) => {
    console.log('name:', json.name);
    console.log('age:', json.age);
  })
  .catch((error) => alert(error));

console.log('Started request...');
```

Promise.all(), Promise.any()

Promise.all()

- Een van de problemen met callbacks was: een functie pas uitvoeren nadat verschillende andere (parallelle) callbacks uitgevoerd waren.
- Het probleem is als volgt, we hebben een aantal (ongereleerde) promises die we tesamen willen starten, en we willen een andere functie uitvoeren als ze ALLEMAAL afgerond zijn.
- **Promise.all** is een statische methode die net dat bereikt: je geeft er een array van promises aan mee en je krijgt één promise terug.
- De promise wordt geresolved met een array van values als ze allemaal afgerond zijn. En dit in dezelfde volgorde als in de array van promises die je doorgegeven hebt.
- Als er één faalt, dan faalt de volledige Promise.all (en hij faalt al zodra de eerste faalt 'fail fast').
- Open voorbeeld **4-promises.all.html**

4-promises.all.html

```
const boektekst = document.getElementById('boektekst');

const promiseArray = [
  fetch('../data/part1.txt').then((response) => response.text()),
  fetch('../data/part2.txt').then((response) => response.text()),
  fetch('../data/part3.txt').then((response) => response.text())];

Promise.all(promiseArray).then((array) => {
  boektekst.insertAdjacentText('beforeend', array[0]);
  boektekst.insertAdjacentText('beforeend', array[1]);
  boektekst.insertAdjacentText('beforeend', array[2]);
});
```

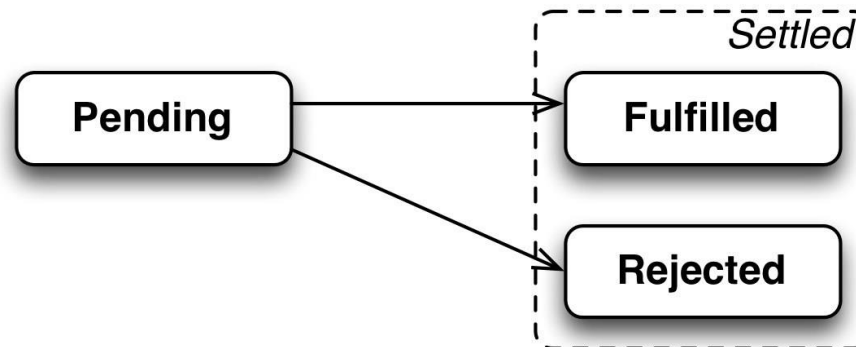
Promise.any

- Een gelijkaardige functie aan **Promise.all** is **Promise.any**
- Als één van de promises in de array 'resolves' is, dan is **Promise.any** ook 'resolved'.
- De **Promise.any** faalt enkel als alle promises falen.

Promises

Meer info over Promises

- Promises kunnen zich in 3 toestanden bevinden:
 - Pending: Het resultaat van de promise is nog niet bepaald aangezien de asynchrone operatie die verantwoordelijk is voor het resultaat, nog niet afgerond is.
 - Fulfilled: De asynchrone operatie is afgerond en de promise heeft een waarde.
 - Rejected: De asynchrone operatie is niet goed afgerond waardoor de promise niet kan gerealiseerd worden. In deze status heeft de promise een reden die aangeeft waarom de operatie faalde.



Promises

- Wanneer een promise (belofte) 'pending' is, kan die dus overgaan naar de toestand 'fulfilled' of 'rejected'.
- Eens een promise 'fulfilled' of 'rejected' is, kan ze niet meer overgaan naar een andere status. De value of de reden waarom de operatie faalde, zal niet meer veranderen.
- Opmerking: als een promise overgegaan is naar de 'fulfilled' of 'rejected' toestand zegt men ook dikwijls dat de promise 'settled' is. Dit betekent dus dat de 'promise' niet langer in de 'pending' toestand is.

Promises

Zelf een promise maken:

```
const p = new Promise((resolve, reject) => {  
    // Voer een (asynchrone) taak uit en vervolgens ...  
  
    /* als de (asynchrone) taak gelukt is voer je de  
       methode resolve() uit */  
    resolve(resolveWaarde);  
    /* als de (asynchrone) taak faalt voer je de  
       methode reject() uit */  
    reject(rejectWaarde);  
});  
  
p.then((resolveWaarde) => {  
    /* Doe iets met het resultaat.  
       Hier komt de actie die moet uitgevoerd worden als het resultaat  
       beschikbaar is. */  
}).catch((rejectWaarde) => {  
    /* Error :(  
       Hier komt de actie die moet uitgevoerd worden als de taak faalde */  
})
```

p bevat dus geen gewone waarde, maar een object waarop je methodes (**then** en **catch**) kan aanroepen om iets te doen met een toekomstig resultaat;

Promises

- De promise wordt DIRECT gecreëerd, maar de promise weet nog niet wat het resultaat gaat zijn.
- Je stelt in wat er moet gebeuren bij het (toekomstig) resultaat, door een callback functie mee te geven aan de then method.
- Deze zal aangeroepen worden op het moment dat de (asynchrone) taak “correct” afgewerkt is.
- De then callback doorgeven doet op zich niets, en is niet blocking.
- De code die moet uitgevoerd worden op het moment dat de asynchrone code “niet correct” afgewerkt is, plaats je in de catch.

Promises

- Promises zijn intern eigenlijk kleine state machines, die een zekere levensloop hebben.
- Als ze aangemaakt worden, starten ze altijd in de pending state, de asynchrone code is aan het lopen, de then en catch functies worden niet gestart maar onthouden.
- Eens de asynchrone code afgewerkt is, komt de promise ofwel in de fulfilled state of in de rejected state, al naargelang resolve() of reject() aangeroepen was. En dan zullen de then en de catch methods aangeroepen worden (in principe kunnen er ook meerdere then of catch functies zijn en dan worden ze één voor één aangeroepen. Maar dit is zelden het geval)

Voorbeeld/ oefening

Open **5a-create-promise.html**

In welke volgorde zullen de
console.log's uitgevoerd worden?

```
const p = new Promise((resolve, reject) => {
  console.log('In promise');
  setTimeout(function () {
    console.log('In setTimeout');
    resolve(100);
  }, 2000);
});

console.log(p);
console.log('After promise');

const p2 = p.then((resolvewaarde) => {
  console.log(`The resultvalue is
${resolvewaarde}`);
  return resolvewaarde * 2;
});

p2.then((waarde) => {
  p.then((resolvewaarde) =>
    console.log(`The resultvalue is
${resolvewaarde}`)
  );
  console.log('Endvalue: ', waarde);
});

console.log('End script');
console.log('-----');
```

Async/await

Waarom async/await?

- Als we promises gebruiken dan schrijven we onze asynchrone code in verschillende `.then's`. Zou het niet handig zijn als we async code zouden kunnen schrijven zoals synchronous code? Dit is nu juist de bedoeling van `async/await`.
- **async functions** en het **await keyword** zijn toegevoegd aan ECMAScript in versie ECMAScript 2017.

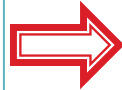
Async/await

- Async/await geeft ons een eenvoudigere manier om te werken met asynchrone, promise-based code.
- **await** plaats je voor de aanroep van een functie die een promise retourneert. Hierdoor zal de code gepauzeerd worden totdat de promise 'settled' is, op welk punt de 'fulfilled value' van de promise zal behandeld worden als de 'return value', ofwel zal de 'rejected value' gegooid (throw new) worden.
- Hierdoor kan je asynchrone code schrijven die eruitziet als synchrone code. We kunnen bijvoorbeeld de code uit **3e-fetch-json.html** als volgt schrijven met **async/wait**:

3e-fetch-json.html

(zonder async/await)

```
fetch('../data/data1.json')
  .then((response) => {
    return response.json();
  })
  .then((jsonResponse) => {
    console.log(jsonResponse);
  });
```



6a-async-await-module.html

(met async/await)

```
const response=await fetch('../data/data1.json');
const jsonResponse = await response.json()
console.log(jsonResponse);
```


async/await

① De functie zal hier pauzeren en wachten tot de `fetch()`-promise 'fulfilled' is. De `response`-variabele zal hier geen promise bevatten maar een gewoon response-object, net alsof `fetch()` een synchrone functie was.

② De functie zal hier opnieuw pauzeren en wachten tot de `response.json()`-promise 'fulfilled' is. De `jsonResponse`-variabele zal terug geen promise bevatten maar een gewoon JavaScript-object (de 'parsed' json), net alsof `response.json()` een synchrone functie was.

```
① const response = await fetch('./data/data1.json');  
② const jsonResponse = await response.json();  
   console.log(jsonResponse);  
}
```

async

- Door **async** aan het begin van een functie toe te voegen, wordt het een asynchrone functie. Binnen die async-functie kan je dan het **await**-sleutelwoord gebruiken.
- **await** werkt enkel in async functions in gewone javascriptcode en het kan gebruikt worden in modules (top level). Een module kan je dus eigenlijk beschouwen als een grote async function.

6b-async-await-function.html

await kan in gewone javascriptcode enkel gebruikt worden in async functions

```
<script>
  'use strict';
  async function getData1() {
    const response = await fetch('../data/data1.json');
    const jsonResponse = await response.json();
    console.log(jsonResponse);
  }
  getData1();
</script>
```

6a-async-await-module.html

Await in top level module

```
<script type="module">
  const response=await fetch('../data/data1.json');
  const jsonResponse = await response.json()
  console.log(jsonResponse);
</script>
```

Error handling

- Code met async/await lijkt goed op synchrone code en we kunnen zelfs een gewone try...catch gebruiken voor de foutafhandeling:

6c-async-await-FINAL.html

```
try {  
    const response = await fetch('./data/data1.json');  
    if (!response.ok) {  
        throw new Error(`HTTP error: ${response.status}`);  
    }  
    const json = await response.json();  
    console.log(json);  
} catch (error) {  
    alert(error);  
}
```

Async/await

- Methodes in classes en arrow functies zijn ook functies zodat je ook deze async kunt maken.
- Een async function retourneert altijd een promise.
- Je kan await gebruiken bij elke functie die een promise retourneert, inclusief web API functions.
- Merk op dat als je await gebruikt de asynchrone operaties opeenvolgend worden uitgevoerd, zoals in een 'promise chain'.
- Daar await werkt bij elke functie die een promise retourneert kan je await ook gebruiken in combinatie met **promise.all()** of **promise.any()**.

Voorbeeld JokeAPI

- Surf naar <https://sv443.net/jokeapi/v2/>
- Creëer een url om een random joke op te halen.
- Open de map **07-joke** en implementeer de functie `getData()`.
- Telkens je klikt op de knop 'Get joke' moet er een nieuwe random joke verschijnen.

