Inleiding Node.js

Samenvatting (versie 1.0)

https://jessym.com

Inhoudsopgave

2 Ontwikkelomgeving

2.1 Installatie Node

Om Node te installeren, kun je het installatiebestand downloaden via de officiele website https://nodejs.org.

Als je dit bestand vervolgens hebt geopend, en de installatieprocedure hebt doorlopen, kun je controleren of alles is gelukt door de *command line* te openen: ofwel **PowerShell** (Windows gebruikers), ofwel **Terminal** (MacOS gebruikers).

Als je zonder foutmelding de volgende twee commando's kunt uitvoeren, en de geinstalleerde versies te zien krijgt (bijvoorbeeld: `v10.9.0` en `6.2.0`), dan weet je dat de installatie is gelukt.

```
node --version
npm --version
```

2.2 Werken met de Command Line

De command line is een soort omgeving waar je de mappen van je computer kunt navigeren en programma'tjes kunt uitvoeren En de meeste command lines, waaronder **Powershell** voor Windows en **Terminal** voor MacOS, hebben een groot aantal ingebouwde commands waar je gebruik van kunt maken.

De volgende commands kun je gebruiken om de verschillende mappen op je computer te navigeren.

- `pwd` - Dit is een afkorting voor "Print Working directory", dus als je deze command uitvoert zul je zien in welke map je je momenteel bevindt.
- `ls` - Dit is een afkorting voor "list" of "listing", en met dit command kun je een overzicht krijgen van alle bestanden en mappen die zich *binnen* de huidige map bevinden.
- `cd` - Dit is een afkorting voor "Change Directory", en dit is waarschijnlijk het belangrijkste command voor navigatie, omdat je deze gebruikt om daadwerkelijk naar een andere map te gaan binnen de command line te gaan.

In de meeste gevallen beland je in je *home folder* als je de command line opent; dit is een map als `C:\Users\Jessy` (Windows) of `/Users/Jessy` (MacOS). Als je vanuit hier naar de map `C:\Users\Jessy\Pictures\Vakantie2016` wilt navigeren, dan zijn daar 2 manieren voor. Ofwel je voert de volgende 2 commando's stap-voor-stap uit:

```
cd Pictures
cd Vakantie2016
```

Ofwel je voert het volgende commando in 1 keer uit:

```
cd Pictures/Vakantie2016
```

Waarbij deze laatste manier vaak een stuk sneller is. Je kunt jezelf trouwens een berg typfouten besparen, door zoveel mogelijk gebruik te maken van de **Tab** toets: als je de eerste paar letters van de map typt, gevolgd door **Tab**, dan zal de command line automatisch de rest van de mapnaam voor je uitschrijven (en automatisch 'quotes' toevoegen als de mapnaam een spatie bevat).

Daarnaast is het mogelijk om met `cd` gebruik maken van een aantal belangrijke symbolen:

- `cd ..` - De dubbele puntjes staan symbool voor de *parent folder*, dus met dit voorbeeld kun je navigeren van `C:\Users\Jessy` naar `C:\Users`.
- `cd .` - Een enkel puntje staat symbool voor de *huidige map*, dus als je dit voorbeeld uitvoert, verandert er niets.
- `cd ~` - De tilde (`~`, te vinden onder de 'Esc' toets) staat symbool voor de *home folder*. Dus waar je je ook bevindt met de command line, als je dit voorbeeld gebruikt kom je altijd terug in je home folder (in mijn geval `C:\Users\Jessy`).

Verder kun je je scherm "schoonvegen" en ruimte vrijmaken met **Ctrl-L**.

2.3 Installatie Visual Studio Code

Het is beter als je voor deze cursus de laatste versie van Google Chrome gebruikt (minstens versie 74). Je kunt vervolgens Visual Studio Code downloaden via https://code.visualstudio.com.

Het is verder aan te raden om zowel op Windows als MacOS in te stellen dat je alle bestandsextensies kunt zien, aangezien we met een aantal verschillende soorten bestanden zullen werken (`.js` en `.txt`).

2.4 Werken met Visual Studio Code

Het is slim om een aparte map te maken voor deze cursus, bijvoorbeeld *Inleiding Node* op je bureaublad.

Als je deze map vervolgens *sleept* naar Visual Studio Code, of opent via het menu, dan kun je aan het begin van elke les een nieuwe map voor die les aanmaken, rechtstreeks vanuit Visual Studio Code.

3 Scripts

3.1 Allereerste Script

Het is gebruikelijk binnen Node om in elk project een scriptbestand genaamd `index.js` aan te maken.

Aangezien dit een ordinair JavaScript bestand is, kun je binnen deze `index.js` het grootste deel van de JavaScript gebruiken, die je gewend bent te gebruiken in een browseromgeving. Bijvoorbeeld:

```
// index.js
const x = 5;
const y = 6;
console.log(x + y);
```

Een fundamenteel verschil bij Node.js, echter, is dat je helemaal geen browser nodig hebt om de JavaScript in deze `index.js` uit te voeren; je hoeft enkel Node geinstalleerd hebben.

Als dit het geval is, dan kun je dit scriptbestand uitvoeren door de command line te openen, PowerShell of Terminal, en door te *navigeren* naar de map waar deze `index.js` zich bevindt. Hiervoor moet je het commando `cd` gebruiken, afkorting voor: "change directory". Bijvoorbeeld:

```
cd Desktop
cd 'Inleiding Node'
cd 'Allereerste Script'
```

De makkelijkste manier om deze commands uit te voeren, is door de eerste paar letters van de map te typen, gevolgd door de **Tab** toets: de command line zal dan automatisch de rest van de mapnaam voor je uitschrijven (en automatisch 'quotes' toevoegen als de mapnaam een spatie bevat).

Als je denkt dat je uiteindelijk in de juiste map bent beland, dan kun je dit controleren door te kijken of je je `index.js` scriptbestand kunt zien, als je het `ls` commando uitvoert.

```
ls
```

Als je hier `index.js` ziet staan, dan zit je in de juiste map, en dan kun je met het volgende Node commando je script daadwerkelijk *uitvoeren*.

```
node index.js
```

3.2 Verschillen Tussen Node en De Browser

Bij een website of frontend applicatie is je JavaScript altijd gekoppeld aan de HTML pagina, via een `<script>` tag in de HTML header.

Aangezien je Node scripts gewoon uitvoert via de command line, staan deze scripts compleet los van de browser. Dit betekent dat een aantal browserspecifieke JavaScript functies en properties niet beschikbaar zijn in Node.

Dit zijn voorbeelden van ingebouwde JavaScript functies die je ***niet*** kunt gebruiken binnen Node (maar wel in de browser):

```
alert('This is a pop-up');
confirm('Do you agree with this yes/no question?');
prompt('What is your name?');
document.getElementById('main-paragraph');
document.getElementsByTagName('div');
```

Dit zijn voorbeelden van ingebouwde JavaScript functies die je ***wel*** kunt gebruiken binnen Node:

```
setTimeout(() => console.log('Called once after 2 seconds'), 2000);
setInterval(() => console.log('Called every 2 seconds'), 2000);
```

3.3 Program Arguments

Bij het uitvoeren van je Node script is het mogelijk om 1 of meerdere *program arguments* mee te geven. Dit doe je door elk van deze program arguments, gescheiden door spaties, achter de naam van je script (`index.js`) te zetten in de command line. Bijvoorbeeld:

```
node index.js apple banana coconut
```

Binnen een Node omgeving heb je namelijk altijd toegang tot de ingebouwde `process.argv` array. Elk element uit deze array is een **string**, waarbij:

1. het **eerste** element de locatie van je Node bestand aangeeft, bijvoorbeeld `/usr/bin/node"` of `"C:\nodejs\node.exe"`
2. het **tweede** element de locatie van je scriptbestand aangeeft, bijvoorbeeld `/home/jessy/index.js"` of `"C:\Users\Jessy\index.js"`
3. het **derde** element je **eerste** program argument aangeeft
4. het **vierde** element je **tweede** program argument aangeeft
5. het **vijfde** element je **derde** program argument aangeeft
6. enzovoort...

Dus het uitvoeren van de onderstaande `index.js`:

```
// index.js
console.log(process.argv);
```

Geeft het volgende resultaat, als je 3 program arguments meegeeft:

```
node index.js apple banana coconut
```

```
> [
>   "/usr/bin/node",
>   "/home/jessy/index.js",
>   "apple",
>   "banana",
>   "coconut"
> ]
```

Omdat het **derde** element in de `process.argv` array altijd gelijk staat aan je **eerste** program argument, is het gebruikelijk om een variable aan te maken zoals `const fruit = process.argv[2];`, als je van plan bent om maar 1 program argument te gebruiken in je script.

3.4 OPDRACHT: Rekenmachine

-

3.5 Bestanden Lezen (Callbacks)

Node heeft een aantal ingebouwde *modules* voor verschillende functionaliteiten, waar je gebruik van kunt maken in je scripts.

Een belangrijk voorbeeld daarvan is de `fs` module (afkorting voor "filesysteem"). Deze module kan dan ook gebruikt worden voor alles gerelateerd aan het bestandssysteem, voornamelijk: (lokale) bestanden lezen en opslaan.
Als je bijvoorbeeld, in dezelfde map als `index.js`, een tweede bestandje hebt genaamd, `message.txt`:

```
// message.txt
abcdefghi
```

Dan kun je als volgt de `fs` module importeren, en dit lokale bestand genaamd `message.txt` lezen.

```
// index.js
const fs = require('fs');

fs.readFile('./message.txt', 'utf8', (err, content) => {
  if (err) {
    console.log(err);
    process.exit(1);
  }
  console.log(content);
});
```

Hierbij gebruik je de `fs.readFile` method, en geef je een drietal arguments mee:

1. (string) De relatieve locatie van het bestand dat je wilt inlezen, bijvoorbeeld: `./message.txt`
2. (string) De naam van de character encoding die je wenst te gebruiken (in 99.9% van de gevallen `utf8`)
3. (functie) Een zogenaamde *callback* functie die Node voor je zal aanroepen, zodra het bestandje is gelezen; Node zal hierbij 2 arguments meegeven:
   (a) een **err** object dat altijd `undefined` is (tenzij er een fout is opgetreden)
   (b) een **content** string bestaande uit de tekst in je externe bestand (tenzij er een fout is opgetreden)

Aangezien het lezen van een bestand lang *zou kunnen* duren, laat Node je een functie meegeven (callback), die wordt aangeroepen zodra het bestand is gelezen.

Een sterk alternatief voor de callback, is de zogenaamde *Promise*.

3.6 Bestanden Lezen (Promises)

Stel je hebt een potentieel langdurige operatie binnen Node, zoals het inlezen van een bestandje. Dan zijn er binnen JavaScript meestal 2 manieren om daarmee om te gaan:

1. Je kunt een *callback* meegeven
2. Je kunt zorgen dat je een *Promise* terugkrijgt

Een Promise is een bepaald type JavaScript object, dat je kunt gebruiken om te wachten totdat het resultaat van de (potentieel) langdurige operatie beschikbaar is. Vandaar ook de naam *Promise*, het is een soort *belofte* dat dit resultaat eventueel beschikbaar zal zijn.

In Node is het mogelijk om een aangepaste versie van de `fs.readFile` method te gebruiken, die geen callback als 3e argument accepteert, maar in plaats daarvan een Promise teruggeeft; hiervoor moet je wel gebruik maken van de `const fsp = fs.promises;` submodule van `fs`:

```
// index.js
const fs = require('fs');
const fsp = fs.promises;

const promise = fsp.readFile('./message.txt', 'utf8');
```

Als je eenmaal beschikt over zo'n Promise object, dan is het meestal de bedoeling dat je gebruik maakt van de ingebouwde `.then` method.

Net zoals bij de meeste functies en methods binnen JavaScript, is er ook bij deze `.then` method sprake van **arguments** en een **return value**:

- Als **eerste argument** geef je een functie mee, die aangeeft wat er moet gebeuren met het resultaat van de langdurige operatie (als er ***geen*** fout optreedt)
- Als **tweede argument** kun je (optioneel) ook een functie meegeven, die aangeeft wat er moet gebeuren met een opgetreden fout (`err`) object
- Als **return value** krijg je een 100% *nieuwe* Promise terug

In het onderstaande voorbeeld wordt ofwel de tekst "Succes!" ofwel de tekst "Error!" gelogd naar de console, afhankelijk of het bestand `message.txt` wel of niet kon worden ingelezen.

```
// index.js
const fs = require('fs');
const fsp = fs.promises;

function successFunction(content) {
  console.log('Success!');
}

function errorFunction(err) {
  console.log('Error!');
  process.exit(1);
}

const promise = fsp.readFile('./message.txt', 'utf8');
promise.then(successFunction, errorFunction);
```

En aangezien de return value van de `.then` method ook weer een Promise is, is het mogelijk om daar ook een nieuwe variable voor aan te maken.

```
// index.js

...

const promise = fsp.readFile('./message.txt', 'utf8');
const secondPromise = promise.then(successFunction, errorFunction);
```

Deze `secondPromise` zal pas "klaar" zijn als:

- de originele `promise` "klaar" is
- de `successFunction` van `.then` met success is afgerond

En omdat deze `secondPromise` ook weer een object van het type Promise is, kunnen we ook hier de `.then` method weer op aanroepen.

```
// index.js

...

const promise = fsp.readFile('./message.txt', 'utf8');
const secondPromise = promise.then(successFunction, errorFunction);
const thirdPromise = secondPromise.then(..., ...);
```

Dit wordt ook wel *Promise-chaining* genoemd. En meestal plaats je alle `.then`s gewoon onder elkaar zodat je niet telkens een aparte variable hoeft aan te maken.

```
// index.js

...

fsp.readFile('./message.txt', 'utf8')
  .then(successFunction, errorFunction)
  .then(someOtherSuccessFunction, someOtherErrorFunction)
  .then(yetAnotherSuccessFunction yetAnotherErrorFunction);
```

Het belangrijkste om te begrijpen, is dat `.then` altijd een gloednieuwe Promise teruggeeft, die pas "klaar" is als de oorspronkelijke Promise (waar `.then` op wordt aangeroepen) "klaar" is, en als de succesfunctie voor die `.then` zelf ook met succes is afgerond.

3.7 Callbacks vs Promises

Stel dat je een langdurige operatie uitvoert, waarbij je een callback meegeeft. En stel dat je *binnen* die callback ook weer een langdurige operatie uitvoert, waarbij je ook weer een callback meegeeft. En stel dat je binnen *die* callback *ook* weer een langdurige operatie uitvoert, waarbij je ook weer een callback meegeeft (enzovoort). Bijvoorbeeld:

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

fs.readFile('file-a.txt', 'utf8', (contentA, err) => {
  fs.readFile('file-b.txt', 'utf8', (contentB, err) => {
    fs.readFile('file-c.txt', 'utf8', (contentC, err) => {
      fs.readFile('file-d.txt', 'utf8', (contentD, err) => {

        console.log('All 4 files read!');

      });
    });
  });
});
```

Deze situatie, waarbij je vele lagen diep callback-binnen-callback zet, en waarbij je code steeds verder naar rechts begint te lopen, is een voorbeeld van JavaScript Callback Hell.

Soms ontkom je hier niet aan, en moet je callbacks wel binnen elkaar zetten. Maar er zijn een aantal specifieke situaties waarbij je dit eleganter aan kunt pakken, met behulp van Promises.

Promises bieden namelijk een manier om het resultaat van de vorige operatie *door te geven* naar de volgende Promise.

Vraag: Stel dat `file-a.txt` een simpel bestandje is met de tekst `Hello A`. Welke 2 regels worden er dan in dit voorbeeld gelogd naar de console?

```
const fs = require('fs');
const fsp = fs.promises;

fsp.readFile('file-a.txt', 'utf8')
  .then((content) => {
    console.log(content);
    return 44;
  })
  .then((previousResult) => {
    console.log(previousResult);
  });
```

Het antwoord is:

```
> Hello A
> 44
```

En de reden daarvoor, is dat de *succes-functie* die je aan `.then` meegeeft, *bepaalt* wat voor type Promise je terugkrijgt van die `.then`.

Als je succes-functie namelijk eindigt met `return 44;`, dan zal `.then` je een Promise/belofte teruggeven die (uiteindelijk) de value `44` bevat.

Hiervoor is het dus wel essentieel dat je succes-functie een return-value heeft. Welke 2 regels worden er in het volgende voorbeeld gelogd naar de console?

```
fsp.readFile('file-a.txt', 'utf8')
  .then((content) => {
    console.log(content);
  })
  .then((previousResult) => {
    console.log(previousResult);
  });
```

Het antwoord is:

```
> Hello A
> undefined
```

Dus op die manier kun je het resultaat van de vorige Promise doorgeven aan de volgende, en op die manier is het in *sommige* gevallen mogelijk om te ontsnappen aan Callback Hell. We kunnen het voorbeeld van eerder omschrijven naar Promises, maar als je hier niet oplet, beland je in exact dezelfde situatie als toen:

```
fsp.readFile('file-a.txt', 'utf8');
  .then((contentA) => {
    fsp.readFile('file-b.txt', 'utf8')
      .then((contentB) => {
        fsp.readFile('file-c.txt', 'utf8')
          .then((contentC) => {
            fsp.readFile('file-d.txt', 'utf8')
              .then((contentD) => {

                console.log('All 4 files read!');

              });
          });
      });
  });
```

Merk op dat geen enkele van deze 4 succes-functies op dit moment een return value heeft. Als we daar wel gebruik van maken, kunnen we deze 4 `.then`s onder elkaar zetten:

```
fsp.readFile('file-a.txt', 'utf8');
  .then((contentA) => {
    const promiseB = fsp.readFile('file-b.txt', 'utf8');
    return promiseB;
  })
  .then((contentB) => {
    const promiseC = fsp.readFile('file-c.txt', 'utf8');
    return promiseC;
  })
  .then((contentC) => {
    const promiseD = fsp.readFile('file-d.txt', 'utf8');
    return promiseD;
  });
  .then((contentD) => {
    console.log('All 4 files read!');
  });
```

Dus als de succes-functie (meestal een fat-arrow functie) die je aan `.then` meegeeft een return value heeft, dan bepaalt die return value wat voor type promise je van `.then` terugkrijgt.

Als die return value gewoon een simpele value zoals het getal 44 is, dan krijg je van `.then` een nieuwe gloednieuwe Promise terug, bestaande uit het getal

44. Maar als die return value in je succes-functie al een Promise *is* (bijvoorbeeld het resultaat van: `fsp.readFile('file-b.txt', 'utf8')`), dan krijg je van `.then` exact die Promise terug.

En op deze manier kun je dus in sommige gevallen voorkomen dat je code zo ver naar rechts loopt.

3.8 OPDRACHT: Verdubbelaar (Callbacks)

-

3.9 OPDRACHT: Verdubbelaar (Promises)

-

4 Modules

4.1 Allereerste Module

Voor een grote Node applicatie waar meerdere mensen aan werken is het niet handig om alle honderdduizenden regels code in 1 enkele `index.js` te plaatsen.

Om zo'n grote applicatie uit te splisen over verschillende bestanden en mappen, kun je binnen Node gebruik maken van het *CommonJS Module System*: binnen dit systeem kun je van elk JavaScript bestand een aparte module maken, die je kunt importeren vanuit `index.js` of vanuit een andere module.

Als je zo'n module wilt maken die je kunt importeren, dan moet je wel zelf eerst binnen die nieuwe module aangeven wat je wilt *exporteren*: dit doe je aan de hand van `module.exports`. Als je bijvoorbeeld een bestandje genaamd `animals.js` hebt, dan kun je als volgt een array exporteren vanuit dat bestand.

```
// animals.js
const animals = ['Cat', 'Dog', 'Zebra'];
module.exports = animals;
```

Om deze array vervolgens te importeren vanuit een JavaScript bestand in dezelfde map, gebruik je het woordje `require`.

```
// index.js
const animals = require('./animals.js');
animals.forEach((animal) => console.log(animal));
```

Als je een *lokale* module zoals `animals.js` importeert, moet je niet vergeten om de relatieve locatie van deze module mee te geven: om die reden gebruik je niet `require('animals.js')` maar `require('./animals.js')`.

Je bent trouwens niet verplicht om een array te exporteren; je kunt elke JavaScript value exporteren die je maar wilt. Omdat je wel beperkt bent tot het exporteren van 1 value, wordt vaak gebruik gemaakt van een object-export, waar je zoveel properties aan toe kunt voegen als je wilt. Bijvoorbeeld:

```
// animals.js
module.exports = {
  animals: ['Cat', 'Dog', 'Zebra'],
  sayHello: () => {
    return 'Hello';
  },
  magicNumber: 44
};
```

4.2 Wat is NPM?

Je kunt wel alles zelf programmeren, maar Node heeft ook een gigantisch ecosysteem van externe modules, die je in de meeste gevallen gratis kunt gebruiken binnen je eigen scripts.

De afkorting NPM staat voor *Node Package Manager*, en dit is een programma'tje dat je als het goed is tegelijkertijd met Node hebt geinstalleerd. Om te controleren, kun je het volgende commando uitvoeren via de command line:

```
npm --version
```

waarbij je als het goed is de geinstalleerde versie van NPM te zien krijgt (bijvoorbeeld `6.2.0`).

Dit programma'tje kan dan ook gebruikt worden om al deze externe modules uit de *NPM Registry* te installeren, welke je kunt browsen via de officiële website https://www.npmjs.com.

4.3 NPM Modules

Om externe modules uit de NPM registry te installeren, maak je gebruik van het command line programma'tje `npm`.

Als je via de command line naar de map van je project bent genavigeerd met het `cd` commando, kun je vervolgens je Node project *initialiseren* met het

volgende commando:

```
npm init
```

Hierbij zal NPM je een aantal vragen stellen over je project (bijvoorbeeld: naam, beschrijving, versie, etc.), maar je kunt meestal gewoon op Enter blijven drukken totdat `npm init` klaar is. Als dit het geval is, zul je in je projectmap een nieuw bestand genaamd `package.json` vinden:

```
// package.json
{
  "name": "example-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Het is belangrijk om deze `package.json` te hebben, omdat deze gebruikt wordt als je via NPM een externe module wilt installeren.

Om bijvoorbeeld de externe `stupid-joke` module te installeren, gebruik je het volgende commando:

```
npm install stupid-joke
```

Als je dit doet, zullen er 3 dingen gebeuren.

1. NPM zal in de registry op zoek gaan naar een module waarvan de naam exact gelijk is aan `stupid-joke`.

2. NPM zal je `package.json` updaten met een nieuwe `dependencies` property, waarin je de namen en versies van alle externe modules kunt zien waar dit Node project *afhankelijk* van is, bijvoorbeeld:

```
// package.json
{

  ...,

  "dependencies": {
    "stupid-joke": "^1.0.1"
  }
}
```

3. NPM zal de daadwerkelijke code van deze module naar een nieuwe map genaamd `node_modules` downloaden.

Aangezien alle dependency informatie in `package.json` ligt opgeslagen, kun je te allen tijde deze `node_modules` map verwijderen. Als je vervolgens gewoon `npm install` uitvoert (zonder de naam van de module), dan zal NPM je `package.json` uitlezen, en alles wat ontbreekt gewoon opnieuw downloaden.

En zodra je externe module geinstalleerd is, kun je deze in je script bestanden als volgt importeren:

```
// index.js
const stupidJoke = require('stupid-joke');
```

Hierbij is het belangrijk dat je in dit geval de naam van de module **niet** laat beginnen met `./`, omdat dit geen *locale* module is, maar een *externe* NPM module.

Een ander woord voor module is trouwens *package* of *library*.

5 HTTP

5.1 Wat is HTTP?

-

5.2 HTTP Requests met Node

De makkelijkste manier om vanuit Node HTTP requests te maken, is aan de hand van de externe Axios module. Het is ook mogelijk om hier enkel de ingebouwde modules `http` en `https` voor te gebruiken, maar Axios maakt alles net iets eenvoudiger (en stelt je ook in staat om Promises te gebruiken).

Na het initialiseren van je Node project met `npm init` kun je Axios installeren door het volgende commando uit te voeren.

```
npm install axios
```

De eenvoudigste manier om Axios te gebruiken, is door deze module te importeren en de ingebouwde `.get` method aan te roepen, waarbij je de volledige URL opgeeft waar naartoe je een HTTP request wilt maken. Bijvoorbeeld:

```
// index.js
const axios = require('axios');

axios.get('https://www.google.com');
```

Nu zal het lijken alsof er vrij weinig te gebeurt als je het bovenstaande voorbeeld uitvoert, en dat komt omdat de return value van `axios.get` een Promise is. Het (uiteindelijke) resultaat van die Promise zal een HTTP Response object zijn, en om daar gebruik van te maken, kun je `.then` aanroepen op de Promise.

```
// index.js
const axios = require('axios');

axios.get('https://www.google.com')
  .then((response) => {
    console.log(response);
  });
```

Een aantal belangrijke properties van dat response object zijn:

- `response.status`: Een getal dat de HTTP response status code aangeeft
- `response.headers`: Een JavaScript object dat alle HTTP response headers bevat
- `response.data`: De daadwerkelijke HTTP response body, bijvoorbeeld een string bestaande uit de HTML van de Google homepagina

5.3 Pokemon

De PokeAPI is een server waar je in een handig formaat via HTTP informatie kunt opvragen over verschillende Pokemon. Zo'n soort server wordt vaak een *API* genoemd, vandaar de naam PokeAPI.

Als je bijvoorbeeld een HTTP request stuurt naar de URL https://pokeapi.co/api/v2/pokemon/1 (je kunt dit uittesten in je browser), dan krijg je de volgende HTTP response body terug.

```
{
  "id": 1,
  "name": "bulbasaur",
  "height" 7,
  "weight" 69,
  ...,
}
```

Plus nog een berg andere properties die ik in het bovenstaande voorbeeld niet heb opgenomen. En omdat het formaat van deze response zo sterk overeenkomt met de object die je binnen JavaScript gewend bent, wordt dit bestandsformaat ook wel *JSON* genoemd (afkorting voor: JavaScript Object Notation).

Tot slot zouden we de externe Axios module kunnen gebruiken om HTTP requests naar de PokeAPI te maken, en het resultaat hiervan bijvoorbeeld loggen naar de console.

```
// index.js
const axios = require('axios');

axios.get('https://pokeapi.co/api/v2/pokemon/1')
  .then((response) => {
    const pokemon = response.data;
    console.log(pokemon.name + ' weighs ' + pokemon.weight + 'kg');
  });
```

6 Servers

6.1 Wat is een Server?

-

6.2 Allereerste Server

Om een Node server applicatie op te zetten, zou je gebruik kunnen maken van de ingebouwde `http` module. Maar het een veel betere en populairdere optie, is om gebruik te maken van de extere *Express* module.

Je kunt deze module op de gebruikelijke manier installeren in je Node project.

```
npm install express
```

Vervolgens kun je binnen je Node app gebruik maken van deze module om een eenvoudige HTTP server op te zetten.

```
// index.js
const express = require('express');

const app = express();

app.listen(3000, () => console.log('Server started on port 3000'));
```

Het getal `3000` in het bovenstaande voorbeeld geeft aan op welke *poort* deze lokale server moet worden gestart. Elke server stelt zichzelf namelijk open voor netwerkverkeer, door te luisteren op een bepaalde *netwerkpoort* tussen 0 en 65535, waarbij het binnen Express gebruikelijk is om poort 3000 te gebruiken.

Als je de bovenstaande Node applicatie start met `node index.js`, dan zul je merken dat het Node proces in de command line deze keer niet meteen afsluit, maar "bezig" blijft.

De reden hiervoor, is dat je Node applicatie is begonnen met luisteren op poort 3000 voor inkomende HTTP requests, en hier pas mee zal stoppen als je *zelf* je server afsluit; dit kun je doen door 1 of 2 keer **Ctrl-C** in te typen in de command line.

Je kunt de server vervolgens gewoon opnieuw starten door `node index.js` opnieuw uit te voeren.

En als je lokale Node/Express server eenmaal gestart is, kun je `http://localhost:3000` in je browser bezoeken om ermee te verbinden, waar je waarschijnlijk de volgende foutmelding te zien krijgt.

```
Cannot GET /
```

Maar dat betekent dat je server in ieder geval online is.

6.3 URLs en Route Parameters

Om vanuit een Express server daadwerkelijk met een HTTP response te reageren als iemand een HTTP request naar een bepaalde URL stuurt, moet je een zogenaamde *request handler* voor die URL opzetten. Dit doe je aan de hand van de `app.get` method waarbij je als 1e argument de URL opgeeft, en als 2e argument

de daadwerkelijke request handler: een (fat-arrow) functie die Express voor je aanroept met een `req` (request) en een `res` (response) object.

Het is mogelijk om meerdere request handlers voor verschillende URLs op te zetten.

```
// index.js
const express = require('express');

const app = express();

// http://localhost:3000
app.get('/', (req, res) => {
  res.send('Welcome to the base URL');
});

// http://localhost:3000/maps
app.get('/maps', (req, res) => {
  res.send('Welcome to the /maps URL');
});

app.listen(3000, () => console.log('Server started on port 3000'));
```

Hierbij roep je de ingouwde `.send` method van het response object, om daadwerkelijk een HTTP response terug te sturen naar de client.

Daarnaast is het ook mogelijk om gebruik te maken van zogenaamde *route parameters* binnen je URLs. Hiermee is het mogelijk om 1 enkele request handler te registreren voor alle onderstaande *routes*:

```
/wiki/Deer
/wiki/Tiger
/wiki/Elephant
/wiki/Panda
```

Binnen Express kun je namelijk voor het laatste deel van deze routes een dynamische parameter maken, waarbij je een dubbele punt (`:`) gebruikt, gevolgd door de naam van je parameter.

```
app.get('/wiki/:topic', (req, res) => {
  res.send('Wikipedia page');
});
```

Nu zullen alle bovenstaande routes als HTTP response de tekst `Wikipedia page`

te zien krijgen. Maar omdat de client altijd een HTTP request naar specifieke URL zoals `http://localhost:3000/wiki/Tiger` zal maken, is het mogelijk om de daadwerkelijke value van die `:topic` parameter te verkrijgen uit het `req` (request) object.

```
app.get('/wiki/:topic', (req, res) => {
  const topic = req.params.topic;
  res.send('Wikipedia page about ' + topic);
});
```

Hierbij is `req.params` een ordinair JavaScript object, waarvan de property keys gelijk zijn aan je router parameter names (bijv. `topic` in `'/wiki/:topic'`), en waarvan de property values gelijk zijn aan de daadwerkelijke values uit de HTTP request URL die hiermee overeenkomen (bijv. `Tiger` in `http://localhost:3000/wiki/Tiger`).

6.4 HTML Serven

Het is mogelijk om binnen Express ook echte HTML terug te sturen met `res.send`. Dus in plaats van `res.send('Hello World')` kun je bijvoorbeeld ook de volgende response terugsturen vanuit je request handler.

```
app.get('/', (req, res) => {
  const html = '<!DOCTYPE html><html><h1>Hello!</h1></html>';
  res.send(html);
});
```

Dit werkt als je `http://localhost:3000` bezoekt in de browser, omdat Express automatisch een `Content-Type` header met de value `text/html` meestuurt met de HTTP response.

Mocht je dit niet willen, en mocht je echt rauwe tekst terug willen sturen die niet als HTML moet worden weergegeven, dan kun je ook zelf de gewenste Content-Type header voor de HTTP response opgeven, met behulp van `res.set` method.

```
app.get('/', (req, res) => {
  const html = '<!DOCTYPE html><html><h1>Hello!</h1></html>';
  res.set('Content-Type', 'text/plain');
  res.send(html);
});
```

Maar de standaard value van de `Content-Type` header is dus `text/html`, als je aan de `res.send` method een JavaScript string meegeeft.

6.5 Pug Templates

Het is mogelijk om elke inkomende HTTP request voor een bepaalde URL te beantwoorden met dezelfde HTML pagina, maar het komt vaak voor dat je de response pagina *dynamisch* moet opbouwen.

Denk bijvoorbeeld aan een soort dashboard pagina waarbij je in de hoek "Welkom terug, Jessy" te zien krijgt (dus een persoonlijke HTML pagina, op basis van de ingelogde gebruiker).

Het is wel mogelijk om zelf zo'n HTML pagina op te bouwen, door letterlijk stukjes HTML en dynamische parameters aan elkaar te plakken:

```
const name = 'Jessy';
const html = '<!DOCTYPE html><html><h1>Hi ' + name + '</h1></html>';
```

Maar omdat dit vrij snel onoverzichtelijk wordt en moeilijk om bij te houden, bestaan er zogenaamde HTML *template engines* zoals Pug (vroeger: Jade).

Pug stelt je in staat om in een map binnen je Node project (standaard: "views"), zogenaamde HTML/Pug templates te definieren. En zo'n template is eigenlijk gewoon een ordinaire HTML pagina die een aantal *placeholders* bevat, die later kunnen worden ingevuld. Bijvoorbeeld (in het geval van Pug templates):

```
// views/index.pug
html
    body
        h1 Hello #{name}
        p Welcome back to our website
```

Dit is uiteraard niet de standaard HTML syntax, maar een Pug-specifieke variant van HTML, dat in principe neerkomt op de volgende HTML:

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Hello #{name}</h1>
    <p>Welcome back to our website</p>
  </body>
</html>
```

Wat belangrijk is, is dat Express met behulp van Pug in staat is om de `#{name}` placeholder in deze template te vervangen door een specifieke value. Vervolgens kan Express de resulterende HTML pagina als HTTP response body terug sturen naar de client.

Hiervoor moet je Pug wel eerst installeren.

```
npm install pug
```

Als je Pug vervolgens registreert als "view engine" binnen Express, kun je gebruik maken van de speciale `res.render` functie (in plaats van `res.send`), om je Pug template om te zetten naar HTML en terug te sturen als HTTP response body.

```
// index.js
const express = require('express');

const app = express();
app.set('view engine', 'pug');

app.get('/', (req, res) => {
  res.render('index', { name: 'Jessy' });
});

app.listen(3000, () => console.log('Server started on port 3000'));
```

Hierbij roep je `res.render` meestal aan met 2 arguments:

1. Het **eerste** argument is de bestandsnaam van je Pug template binnen de "views" map; dus `'index'` komt overeen met `./views/index.pug`.
2. Het **tweede** argument is een ordinair JavaScript object, bestaande uit alle properties die je wilt vervangen binnen je Pug template. Aangezien onze template een `#{name}` placeholder heeft, kunnen we hier een object met een `name` property meegeven, welke gebruikt zal worden om die placeholder te vervangen.

6.6 Static Assets

Een *dynamische* website is een website of (Node) server applicatie die gebruik maken van server-technologieën zoals template engines, om op dynamische wijze de HTML pagina voor de HTTP response op te bouwen.

Hiertegenover staat de *statische* website die alleen bestaat uit HTML, CSS en eventueel JavaScript; je kunt bijvoorbeeld denken aan een restaurant website met

een paar foto's en een telefoonnummer om te bellen. Dit houdt in dat voor een **statische** website, de HTML (en CSS en JavaScript) die je in je HTTP response body terugstuurt, in principe voor elke client gelijk is.

Omdat aan zo'n statische website, in principe, geen server-technologie zoals template engines te pas komt, zul je bij zo'n website over het algemeen **niet** boven in de hoek een tekst zoals "Welkom terug, Jessy" zien staan.

Stel dat je als Frontend developer een volledige statische website hebt ontwikkeld in een map genaamd `public` binnen je Node project:

```
./public/index.html
./public/style.css
./public/script.js
```

Dan kun je deze *static assets* als volgt *serven* met Express:

```
// index.js
const express = require('express');

const app = express();

app.use(express.static('public'));

app.listen(3000, () => console.log('Server started on port 3000'));
```

De return value van `express.static('public')` is een zogenaamd *middleware* object dat aangeeft dat er statische website assets te vinden zijn in de `public` map.

Vervolgens moet je je Express app op de hoogte stellen van deze middleware, door deze als eerste argument mee te geven aan de `app.use` method (niet te verwarren met `app.get` voor standaard request handlers).

En als je de bovenstaande server opstart met `node index.js`, dan zul je gepresenteerd worden met je `index.html` pagina als je `http://localhost:3000` bezoekt. De reden om op deze manier je statische website aan te bieden (te *serven*), is omdat dit de manier is waarop websites werken. Je kunt wel via Windows Explorer of MacOS Finder de `index.html` pagina van je website rechtstreeks openen in je browser, maar elke échte website is beschikbaar via `http` op een echt *domein*: bijvoorbeeld `www.google.com` in het geval van Google, of `localhost` in het geval van een lokale server.

6.7 OPDRACHT: Vluchtprijzen

-

7 Fastfood API (Eindproject)

Hierbij wat extra uitleg van de ingebouwde `.find` method van JavaScript arrays. Stel je hebt een array bestaande uit de leeftijden van een willekeurige personen.

```
const ages = [14, 8, 19, 44, 10, 32];
```

Met het blote oog zie je direct dat de eerste *volwassene* uit deze array 19 jaar oud is. Maar hoe kom je hierachter met behulp van JavaScript?

De makkelijkste manier, is met behulp van de ingbouwde `.find` method voor arrays:

```
const firstAdultAge = ages.find(testFunction);
```

Je roept dus `.find` aan, waarbij je een bepaalde "testfunctie" mee moet geven. Als return value zul je van `ages.find` vervolgens het eerste element uit de array terugkrijgen, dat voldoet aan deze "testfunctie".

Als je de eerste volwassene wilt vinden in een array met leeftijden, dan ziet je testfunctie er als volgt uit.

```
function testFunction(age) {
  if (age >= 18) {
    return true;
  }
  return false;
}
```

Of, in fat-arrow notatie:

```
const ages = [14, 8, 19, 44, 10, 32];

const firstAdultAge = ages.find((age) => {
  if (age >= 18) {
    return true;
  }
  return false;
});

console.log(firstAdultAge); // 19
```

Om het juiste element te vinden in deze `ages` array, zal JavaScript namelijk het volgende proberen:

- JavaScript roept de je testfunctie uit, waarbij het eerste element uit de array (`14`) als argument wordt meegegeven.
- Als de return value van je testfunctie gelijk is aan `true` voor dit specifieke element (`14`), dan gaat `ages.find` er vanuit dat je element is gevonden, en zal `ages.find` dit element als return value teruggeven.
- Als de return value van je testfunctie gelijk is aan `false` , dan zal `ages.find` het nog een keer proberen door het volgende element uit de array (`8`) aan je testfunctie als argument mee te geven.
- Dit blijft doorgaan totdat
  1. ofwel een element is gevonden dat wél voldoet aan de testfunctie
  2. ofwel het einde van de array is bereikt; in dit geval geeft `ages.find` gewoon `undefined` terug

Dus op die manier is het mogelijk om met de ingebouwde `.find` method een JavaScript array te doorzoeken voor een element dat voldoet aan een bepaalde testfunctie. En uiteraard kan de logica uit de testfunctie zo complex of eenvoudig zijn als je zelf wilt.

Om een tweede voorbeeld te geven: stel dat je een JavaScript array hebt, bestaande uit “person” objects:

```
const people = [
  { name: 'Tim', city: 'Apeldoorn' },
  { name: 'Harry', city: 'Vlissingen' },
  { name: 'Bart', city: 'Maastricht' },
  { name: 'Marcel', city: 'Breda' }
];
```

Dan kun je als volgt de *woonplaats* loggen van de eerste persoon met een voornaam van 4 letters.

```
const firstPersonWithFourLetterName = people.find((person) => {
  if (person.name.length === 4) {
    return true;
  }
  return false;
});

console.log(firstPersonWithFourLetterName.city); // Maastricht
```