



**NEW MEDIA &
COMMUNICATION
TECHNOLOGY**

BACKEND DEVELOPMENT



3 NMCT

Johan Vannieuwenhuyse
Kevin De Rudder

Inhoud

1	Het web (en javascript) is real-time geworden	5
2	Introductie en kenmerken	6
2.1	Cursus inhoud	6
2.2	De belangrijkste Node.js kenmerken:.....	6
2.3	Nodejs en zijn toepassingen	7
2.4	Historiek in het kort:	8
3	Installatie, testen en debuggen van node.js	10
3.1	Installatie:	10
3.2	Command Line Interface (CLI)	11
3.3	File(*.js) execution:	13
3.4	Enkele basis node objecten vooraf (global, process, Buffer)	14
3.4.1	Global	14
3.4.2	process voorziet readable/writable datastreams	14
3.4.3	Console	15
3.4.4	timers	15
3.4.5	Buffer	15
3.4.6	require()	16
3.5	Een node project opzetten (package.json).....	16
3.6	Debuggen in node.js	17
3.7	JSLint en JSHint:	20
3.7.1	Hinting installeren.....	21
3.7.2	Hinting via een taskmanager	22
4	Alternatieve installatie: IISNode (ter info).....	23
4.1	IISNode als handler	23
4.1.1	Waarom IISNode	23

4.1.2	IISNode installeren.....	23
4.2	Node IIS als een Azure App service	24
5	Visual Studio of PHPstorm/Webstorm	26
5.1	Visual studio (NTVS).....	26
5.2	PHPStorm / WebStorm.....	29
6	Asynchroon programmeren met de event loop	31
6.1	Callback om Asynchroon (=event-driven) te programmeren	31
6.2	De event loop beheert de asynchrone taken	35
6.3	Callback Hell = Pyramid of Doom = the Boomerang effect	38
6.4	Benoemen van callback functies.....	39
6.5	Promises om asynchroon te programmeren.....	39
6.5.1	Wat is een promise?	40
6.5.2	Hoe een Promise aanmaken?	40
6.5.3	Een 'then' maakt nesten van callbacks overbodig	42
6.6	Async/await om asynchroon te programmeren :.....	43
7	Over Javascript functies, modules, constructors en classes.	44
7.1	Javascript (ECMA) en node.....	44
7.2	Functie scope !== block scope	46
7.3	Closures.....	47
7.4	Zelfuitvoerende functies (IIFE)	49
7.5	Argumenten bij closures en zelfuitvoerende functies	50
7.6	Module patroon	51
7.7	ES5 Constructor patroon	53
7.8	ES6 class als het nieuwe ctor-patroon.....	55
7.9	Welk patroon kiezen: module- of ctor-patroon?.....	57
7.10	Extensiemethoden via het prototype	57

7.11	Overerving en compositie	59
7.11.1	Module pattern: Compositie met Object.assign()	59
7.11.2	Constructor pattern : Overerving van het prototype met Object.create()	60
7.11.3	ES6 class overerving.	60
7.12	JSON als transport middel	61

1 Het web (en javascript) is real-time geworden

Het web verandert. Van kijken naar beelden en videos evolueert het naar interactie en dan nog liefst in *real time*. Toepassingen zoals chatting, gaming, social media updates en samenwerken, werden een groot deel van de toepassingen. Bovendien moet de samenwerking niet alleen mogelijk zijn met een klein aantal gebruikers maar ook met honderden. Additioneel wordt met IoT (Internet of Things) voorspeld dat in 2020 rond de 50 miljard devices op internet geconnecteerd zullen zijn. Een groot deel ervan zal *real time data* produceren.

Javascript, die vroeger alleen een rol speelde in het interactief maken van website, groeide hierbij tot de taal die naast de cliënt nu ook zijn toepassingen kent op de server en IoT devices. Getuigen hiervan zijn het groeiend aantal en snel veranderende javascript libraries en API's (ook in devices). De vernieuwde javascript technologieën worden daarbij ondersteund door gedreven communities en een reeks javascript wizards:



Realtime toepassingen betekent real time communicatie tussen cliënt en server. Eén van de protocollen die lange tijd aangewend wordt voor deze real time communicatie is http; en dit omdat http overal ondersteund en begrepen wordt. Nochtans, http werd niet gemaakt voor real time communicatie. Door de manier waarop klassieke http-servers ontworpen zijn, is bij http voor iedere request/response cycle een thread nodig die de connectie aanvaardt, afhandelt en terugstuurt. Iedere thread die gestart wordt, heeft een zekere overhead, die in de context van realtime en massive scalability te zwaar en niet langer aanvaardbaar is.

De servers hadden een aanpassing nodig om met eenzelfde thread meerder connecties af te handelen. Dit resulteert in een betere performantie door het teniet doen van de overhead van threads. We zien dat ondertussen al heel wat http-servers dit model overgenomen hebben. Zo hebben we onder andere *nginx* en *node.js*. In deze cursus focussen we op node.js daar dit niet enkel een event-driven webserver (= de officiële term) is, maar ook een platform dat van de grond af heropgebouwd werd om alles in een **asynchrone** alias event-driven manier af te werken.

2 Introductie en kenmerken



2.1 Cursus inhoud

Node.js wordt in veel omgevingen en op veel manieren gebruikt. In het bijhorend document worden volgende elementen behandeld:

1. asynchroon programmeren met Node.js in object oriented EcmaScript 6,
2. het verkennen van de mogelijkheden van de node API library
3. het passend gebruiken van een NoSQL database (MongoDB)
4. het gebruiken van node frameworks met extra middleware en sockets om realtime toepassingen te bouwen in de cloud(=toe te passen in het project)

2.2 De belangrijkste Node.js kenmerken:

- Een javascript interpreter die ook buiten de browsers draait en daardoor geschikt is voor het ontwikkelen van backends of embedded toepassingen. Javascript werd gekozen als taal voor node.js omwille van zijn mogelijkheid om asynchroon te werken. Er werd niet gekozen voor een framework, omdat je anders eerst dit framework onder de knie moet krijgen.
- De interpreter maakt gebruik van V8 (Chrome engine), een javascript machine op basis van C++ en een reeks async libraries. Herinner dat elke browser zijn eigen javascript runtime bezit: Spider Monkey voor Firefox, Nitro voor Safari (komt van Squirrel Fish en JavascriptCore), V8 voor Opera (komt van Carakan) en natuurlijk V8 voor Google Chrome. Noot 1: in mei 2015 voorzag Microsoft met Windows 10 de mogelijkheid om node te runnen op de Chakra javascript engine (<https://github.com/Microsoft/node>)



Noot 2 :De ontwikkeling van V8 gaat continu verder en gebeurt in 3 stadia: shipped (aanwezig features) -> staged (features die bijna af zijn) -> In progress (in ontwikkeling). Hoor je de term "harmony" dan verwijst men naar de staged features.

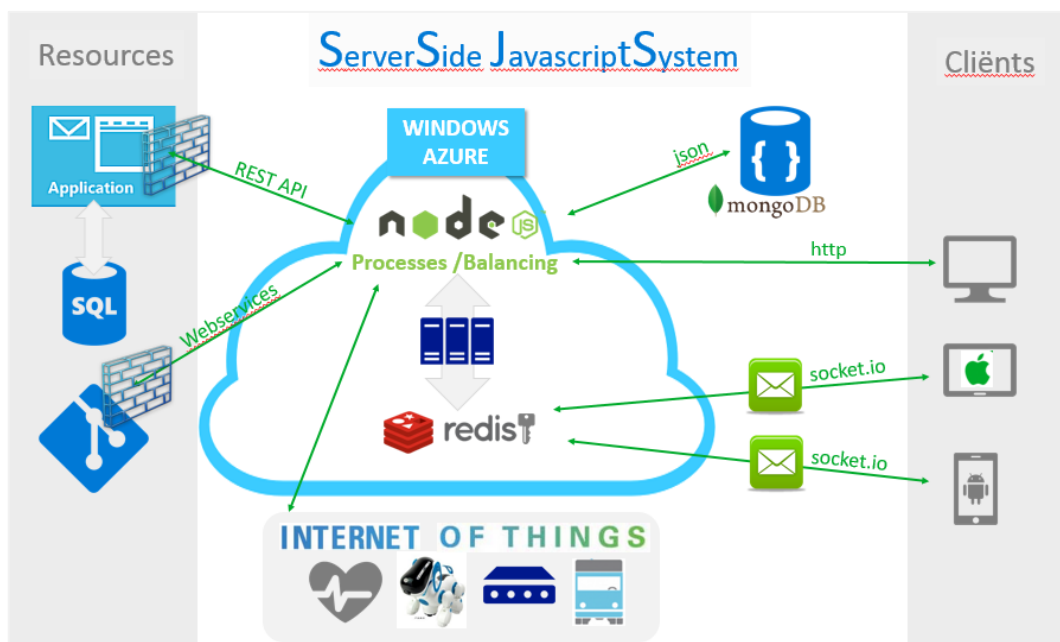
- Een javascript V8 engine verwerkt wel Javascript maar laat op zich geen visualisatie van data noch I/O van data toe. Node voorziet daarom naast een javascript interpreter ook een hosting environment voor javascript. Node kan zo naast het javascript environment van een browser ook runnen in de javascript environment van een SoC (System on Chip zoals Raspberry, Intel Edison) of het javascript environment van een embedded device. Node breidt hierbij de native javascript omgeving uit met scherm visualisering en I/O handling vanuit een command line interface (CLI).
- Het command line tool (CLI).
Het CLI-tool is eveneens geschreven in javascript. Dit betekent dat het ook op andere

platformen (bvb. windows server) kan gerund worden. De CLI voorziet o.a. de volgende mogelijkheden:

- een http server starten (trager dan tcp server, gebruikt de browser),
 - een tcp server te starten (sneller dan http server, moeilijker bij firewalls),
 - zelf aangemaakte modules te installeren,
 - gebruik maken van third party modules uit de community.
- Node is open source en wordt ondersteund door een sterke community.
 - Beschikt over een **non blocking file system**. Non blocking betekent dat node.js elke aangeboden file of opdracht event-driven opneemt en verwerkt. Een andere gebruikte term voor event-drive is asynchroon. Het **asynchrone/event driven** karakter wordt bereikt door:
 - Het gebruik van callback functies.
In een niet event driven omgeving wordt een programma lineair verwerkt. Zo wordt een database call afgewerkt, waarna het programma verder gaat. Event driven betekent verschillende taken opstarten en deze blijven monitoren tot wanneer een taak vervolledigd is en intussen het lopende programma verder afwerken.
 - Het simultaan verwerken van meerdere taken of requests:
Node.js realiseert dat met een eventloop (zie verder) in een uitermate optimaal geheugengebruik. Ideaal om bijvoorbeeld honderd(en) users simultaan te behandelen.

2.3 Nodejs en zijn toepassingen

- Dankzij nodejs kunnen applicaties nu gebruik maken van dezelfde taal bij de cliënt als bij de server (Javascript). Men spreekt soms over SSJS = Server Side Javascript Systems.. Een andere term voor het gebruik van javascript in client en server is "isomorphic javascript".



Figuur 1: Een volledige applicatie met enkel javascript.

- Door de sterkte op I/O gebied is node.js ideaal voor het maken van **SCALABLE REAL-TIME APPLICATIONS** met **MEERDERE PARALLELE CLIËNTS - die geen CPU intensieve taken moeten verwerken**:
 - social applications;
 - multiuser games;
 - business collaboration areas;
 - news, weather, of financial applications die veelvuldige updates vragen;
- Deze sterkte op I/O gebied wordt technisch ondersteund door:
 - sockets voor realtime communicatie,
 - media servers en proxies voor custom netwerk services,
 - JSON communicatie (webservices en NoSQL databases)
 - schaalbaarheid (scaling) op multi-core servers,
 - side by side running met bvb. Rails/ASP.NET,
 - herbruikbaarheid van dezelfde javascript code op de server (als middle end) als op de cliënt (bvb: validatie regels).
- Gebruik Node.js niet (!) voor CPU intensieve taken zoals video transcoding.
Gebruik Node.js niet(!) voor toepassingen met zwaar wiskundige berekeningen.
Gebruik node ook niet (!) voor het bouwen van zwaar data driven applicaties, die veel relationele relaties verwachten (= gebruik daarvoor Rails, ASP.NET)
- Node.js kan ook runnen onder een IIS handler op windows. Men spreekt dan over "IIS-Node".
De native module Node.exe runt zo ook als handler op windows azure. De handler wordt op een klassieke wijze geconfigureerd in de web.config:


```
<configuration>
  <system.webServer>
    <handlers>
      <add name="iisnode" path="server.js" verb="*" modules="iisnode" />
    </handlers>
  </system.webServer>
```
- Node wordt o.a gebruikt door: Netflix, Groupon, SAP, LinkedIn, Wallmart, PayPal, Uber, Yammer van Microsoft...
 - <https://nodejs.org/en/foundation/case-studies/> ,
<https://insights.stackoverflow.com/survey/2016>
 - <https://www.quora.com/What-companies-are-using-Node-js-in-production-in-Texas>
 - Groeiend aantal javascript repositories op github: <http://github.info/>

2.4 Historiek in het kort:

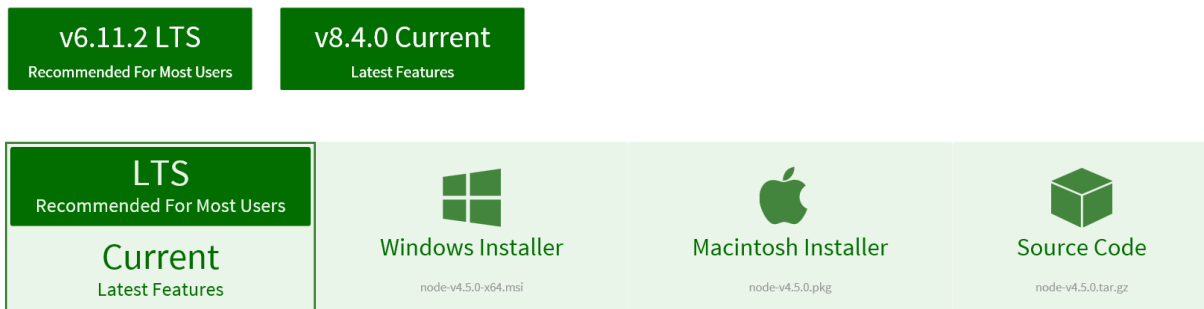
- 2008: chrome lanceert in september zijn V8 engine, met als specifieke eigenschap een precompilatie van Javascript in C++ (naast de interpreter).

- 2009: Ryan Dahl start met node.js en presenteert in november het non blocking mechanisme op de EuropeanJSConf.
- 2011: Een nieuw belangrijk onderdeel: een package manager voor nodejs (npm door Isaac Schlueter).
- 2015: De node.js foundation wordt opgericht vanuit twee communities en resulteert in node versie 4.0. Een stijgend aantal nodejs frameworks naast Express. Om de QoS (Quality Of Service) naar firma's te verhogen wordt NodeSource TM (= The Enterprise Node Company) opgericht en N|Support geactiveerd op 3 niveau's (developer, standard, enterprise).
- 2016: In mei werd node versie 6 beschikbaar met LTS support (Long Term Support), wat een extra accent legt op stabiliteit en veiligheid voor firma's die node als key technologie gebruiken.
- 2017: Node is nu 97% ES6 (Ecma Script 6) compatibel (ref: <https://kangax.github.io/compat-table/es6/>)
Tevens wordt naast npm een eerste versie van een vernieuwde package manager gereleased: Yarn. Dit omwille van security en snelheidsredenen.

3 Installatie, testen en debuggen van node.js

3.1 Installatie:

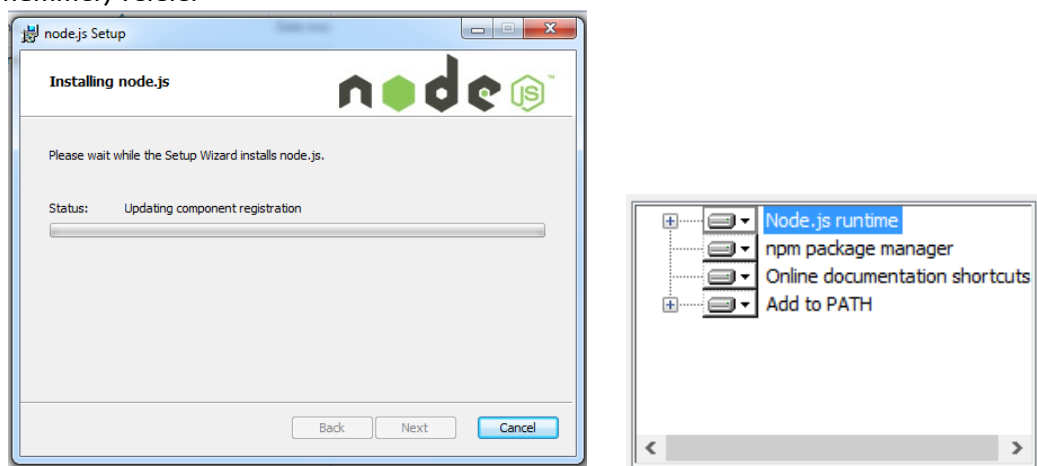
Raadpleeg de officiële site voor installatie (<https://nodejs.org/en/>). Zowel een windows als mac installer package zijn rechtstreeks beschikbaar op de site van node.js. Je kan hierbij kiezen tussen een Long Time Support versie (= 30 maanden) of de laatste versie (current) met alle mogelijke nieuwe features:



Download en installeer node.js op windows:

Je kan gebruik maken van manuele of automatische installatie. De laatste *.msi en automatische installatie vind je hier: <http://nodejs.org/dist/latest/>

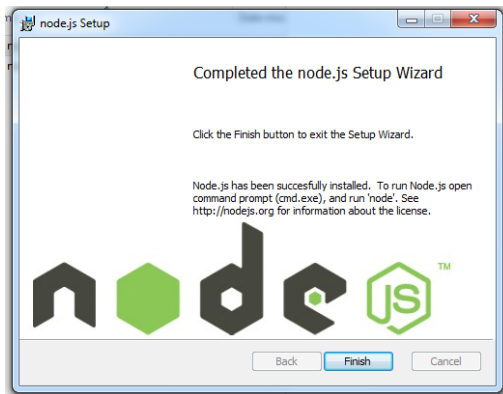
Versies die eindigen op een even nummer worden als stabiel aanzien, versies die eindigen op een oneven nummer worden als onstabiel aanzien. Installeer de current en stable (even nummer) versie.



Default installatie map is "nodejs" onder Program Files waarbij zowel een runtime versie, een package manager en documentatie geïnstalleerd worden. Node wordt eveneens toegevoegd aan je PATH.

Downlad en installeer op OSX:

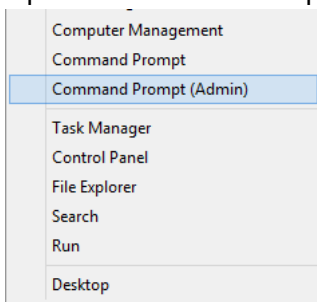
Installeer de package manager Homebrew vanop <http://brew.sh>:
\$ brew install node



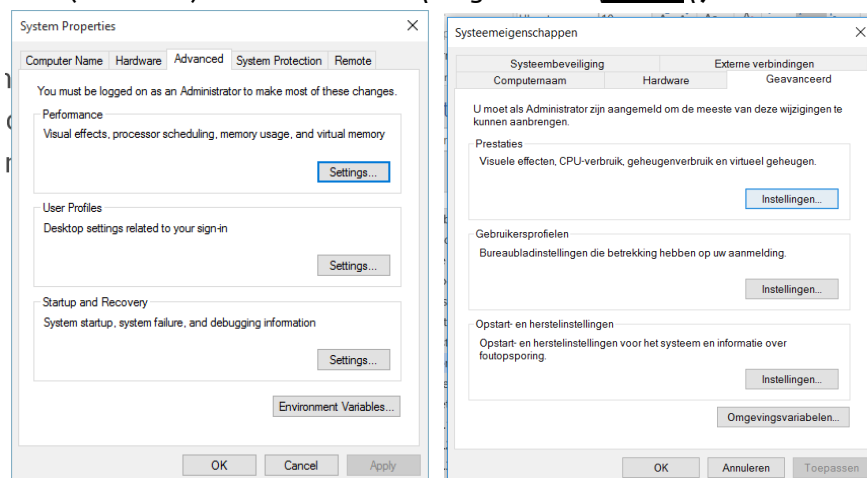
Noot: Open source programma's kunnen het moeilijk maken om downwards compatible te blijven. Een tool dat hierbij kan helpen om vlot te schakelen tussen verschillende node versies is de node version switcher. Deze switcher kan globaal geïnstalleerd worden met het commando "npm install n -g". Overschakelen gaat met één command: "n <version>". (meer info: <https://www.npmjs.org/package/n>)

3.2 Command Line Interface (CLI)

1. Open de command Prompt als administrator



2. Ga in de command omgeving naar de default node.exe locatie of voeg node toe aan je PATH toe, mocht dit niet gebeurt zijn tijdens de installatie (Configuratiescherm>>Systeem en beveiliging>>Systeem>> Geavanceerde systeeminstellingen >> **Omgevingsvariabelen** >> Path (onderaan) uitbreiden met C:\Program Files\nodejs\)



- Geïnstalleerde versie opvragen: `node -v` of `node --version` (let op de spaties).

Nog andere command line opties zijn beschikbaar en kun je op een analoge manier opvragen (of je kan gaan kijken in de documentatie van nodejs.org):

`node -h` of `node - -help`

- Activeer met het command "node" de node applicatie (of maak gebruik van de interactief node consoles in een IDE). Hierdoor kom je in de REPL console terecht. REPL staat voor **Read-Eval-Print-Loop**. De manual van REPL met alle commando's vind je op <http://nodejs.org/approci/repl.html>

```
C:\Program Files (x86)\nodejs>node
>
```

Noot: Door het toevoegen van command flags krijg je extra informatie

Staged V8 features kan je runnen met `>node --harmony`

Node help toont beschikbare options `>node --help`

Test een aantal lijn commando's (javascript) in de REPL-console. Het command "console.log" is een wrapper rond `process.stdout()`:

```
> console.log ("Hello World");
> function add(a,b) {return (a+b)}
//ES6: let add=(a,b)=> a+b
> add(23,10)
> process          toont lopende process, running environment
> process.env      toont de user environment zoals USERNAME,
                   env vaak aangevuld met bv.PORT via een config.js
> process.versions  versies van alle aanwezige processen
> _                underscore haalt laatste command met resultaat terug .helpop

>.help              opent een basic help, let op het punt vóór de instructie,
                   en toont alle dot instructies
```

```
> .help
.break      Sometimes you get stuck, this gets you out
.clear      Alias for .break
.editor      Enter editor mode
.exit        Exit the repl
.help        Print this help message
.load        Load JS from a file into the REPL session
.save        Save all evaluated commands in this REPL session to a file
```

```
> TAB TAB zorgt voor autocomplete (of toont de beschikbare objecten, methoden)
> require("./helloworld.js")    (haalt een module op)
> var global.test = "een global beschikbaar tekst"
   global is de top omgeving, namespace => globale variabelen vermijden!
> Buffer          een inherent aanwezig object voor I/O doeleinden.
                   een geheugen (buiten de V8) wordt hiervoor toegekend
                   en stokeert binaire (!) data
> console.time('aTimerLabel')   start een timer (handig...)
> console.timeEnd('aTimerLabel') stopt de timer en toont het
                                resultaat
```

```
> node --harmony --use-strict aFile*.js { awesome : true }    extra ES6 mogelijkheden
```

Met CTRL break (CTRL+C) kan je een foutief commando ongedaan maken.

Met process.exit() of CTRL+D verlaat je de console en kom je terug in de command.prompt.

3.3 File(*.js) execution:

Externe javascript files (*.js) kunnen buiten de REPL console runnen met het **"node"** command. Je gaat van commandline execution naar file execution:

Schrijf een hello-world.js en test met het node command: node hello-world.js. Natuurlijk hou je hierbij rekening waar node geïnstalleerd staat of waar de file geïnstalleerd staat. Het oproepen van de file hoeft niet case sensitive te gebeuren. Het suffix ".js" is optioneel en de opgeroepen filename is niet(!) case sensitive.

```
C:\Program Files (x86)\nodejs>node hello-world.js
hello
world
C:\Program Files (x86)\nodejs>
```

Er kunnen extra argumenten meegegeven worden voor de file uitvoering door gebruik te maken van het process object en zijn eigenschap argv.

Process.argv om argumenten mee te geven

"process.argv" is een array met twee standaard argumenten van node: de absolute file name van node.exe en de absolute file naam van de uitvoerde javascript file.

Je kan meer dan twee argumenten meegeven. Je voegt nog extra argumenten toe als string, gescheiden via een spatie: node <fileName> <extra arg1> <extra arg2> ///.
bvb.: "node HelloWorld.js Mister" zorgt ervoor dat process.argv[2] de waarde "Mister" bevat.

Oefening 3.3: Menu in de console

Opgave: Raadpleeg de documentatie en test dit uit.

Toon ook eens alle argumenten van argv in de console met een for of forEach lus. Gebruik eens een ES6 lus. (<http://es6-features.org> , <https://kangax.github.io/compat-table/es6/>)

Opgave: Deze eenvoudige techniek met argv wordt soms gebruikt om vanuit de console een klein menu aan te bieden, dat je informeert hoe een taak (zoals bvb. een installatie, een help) op te roepen. Hierbij wordt soms, hoewel onnodig, gebruik gemaakt van externe modules zoals require("minimist") of require("prompt") die extra mogelijkheden aanbieden om console

argument uit te lezen.

```
Johans MENU
How to use:
--help      show this help file
--name <NAME> say welcome to <NAME>
```

3.4 Enkele basis node objecten vooraf (global, process, Buffer)

3.4.1 Global

Ref: <https://nodejs.org/api/globals.html>

“global” is de global overkoepelende namespace van node.

Node beschikt over een aantal globals, die je kan opvragen in REPL met TAB TAB. Een deel zijn herkenbaar vanuit Javascript (Array, timers, constructor en WeakMap uit ES6...). Maar andere zijn typisch voor Node zoals process, Buffer, zlib.

3.4.2 process voorziet readable/writable datastreams

Ref: <https://nodejs.org/api/process.html>

Het meest globale object is “process” en vervangt het traditionele “window” top object van javascript aan cliënt kant.

Het process object voorziet data streams (!) voor het behandelen van input, output en errors:

- process.stdin is een “readable stream” waarbij data geaccepteerd wordt van de user terminal. Dit process bevindt zich bij opstart van een applicatie in standby mode tot gegevens ingevoerd worden.
Via een callback kunnen met stdin de ingevoerde gegevens opgehaald worden en via stdout getoond worden:

```
process.stdin.on("data", function (data) { //stdout gebruiken });
```

Noot: Evengoed kan je met `var repl = require("repl");` de repl module opnemen (of een andere zoals prompt) in een javascript file om prompts op te vragen.

- process.stdout is een “writable stream” en voorziet output mogelijkheden voor een programma via de write methode:
`process.stdout.write(data, [encoding], [callback])`
`process.stdout.write("\033c");` //zorgt voor een break
`console.log()` schrijft via process.stdout als een wrapper rond process.stdout
- process.stderr is eveneens een “writable stream” gelijkaardig aan stdout:
`process.stderr.write(data, [encoding], [callback])`

Het process object laat meer toe dan enkel file execution met argumenten.

- Er zijn bvb. methoden die toelaten om van werk directory te veranderen: (`process.chdir('otherMap')`) of om de current directory op te vragen (`process.cwd()`)...
- Environment eigenschappen kunnen opgevraagd worden via het `process.env` object. De benamingen van de eigenschappen spreken voor zichzelf. Een aantal kunnen onmiddellijk opgevraagd worden zoals `process.env.PATH`. Andere variabelen blijven undefined tot ze werkelijk gebruikt worden. Een typische toepassing is het configureren van de execution mode (productie of ontwikkeling) via de variabelen `process.env.DEVELOPMENT`, `process.env.PRODUCTION`, `process.env.TEST`.
Om default waarden aan te brengen gebruik je ||

```
var env = process.env.NODE_ENV || 'development';
```

Meer info over het process.object: <http://nodejs.org/api/process.html>

3.4.3 Console

Ref: <https://nodejs.org/api/console.html>

"console" is het console object aan server kant. Deze console buffert standaard het output resultaat en zorgt voor output via `process.stdout.write()`

"console.error" en "console.warn" printen hun errors via `process.stderr`.

bvb.: `console.error("enkel een foutsimulatie");`

"console.trace()" maakt gebruik van `process.stderr` om een volledige stacktrace uit te schrijven

3.4.4 timers

De klassieke javascript timers zijn globaal voorzien in node: `setTimeout()` en `setInterval()`.

Informatie over deze timers is te vinden op <http://nodejs.org/api/timers.html>. Node maakt uitvoerig gebruik van deze timers om verschillende taken op te starten in een specifieke volgorde. Dit komt verder nog uitgebreid aan bod bij het opbouwen van een asynchrone workflow.

3.4.5 Buffer

Ref: <https://nodejs.org/api/buffer.html>

Buffer verzorgt I/O van binare data en gebruikt daarvoor toegekend geheugen, buiten het V8 werkingsgeheugen. Gezien Buffer binair stokeert is encoderen/decoderen noodzakelijk om de data te lezen. Het gebruik van `toString()` als decodering is hier handig. Of je gebruikt de `StringDecoder`, die gegarandeert de volledige String op te nemen, deze volledig decodeert en pas op einde teruggeeft. (Ref: https://nodejs.org/api/string_decoder.html).

Buffers kunnen op verschillende manieren aangemaakt worden. Let erop hoe GEEN instantie aangemaakt wordt omdat Buffer een inherent aanwezig object is in Node.

- `Buffer.allocate(number)` // vult de buffer niet
- `Buffer.allocUnsafe(number)` // vult de buffer met bogus data uit het geheugen
- `const buffer = Buffer.from("een willekeurige string")`

Buffers zijn vooral van belang bij de behandeling van streams.

3.4.6 require()

```
require("./myModule.js");
```

Het command `require()` laat toe andere node modules of eigen javascript modules op te halen, voor zover die voorzien werden van "exports". De techniek komt van het CommonJS modulesysteem. De `require` gebeurt sequentieel in een aantal stappen.

1. Het absolute path van de file wordt gecontroleerd en opgehaald(resolve)
2. De module wordt ingeladen
3. Wrap de module in zijn eigen private scope (belangrijk! No globals!)
4. Check de aangeboden code op errors
5. Cache het resultaat zodat het geen twee keer geladen wordt (belangrijk! Memory)

Noot: Node voorziet de mogelijkheid om zelf addons te maken. Hiervoor worden C++ objecten gecompileerd in een binaire `addon.node` file op basis van het `node-gyp` tool. Ook deze addons worden met `require` opgehaald. (Ref: <https://nodejs.org/api/addons.html>)

3.5 Een node project opzetten (package.json)

Een **package.json** file is een **VALID JSON** file en bepaalt binnen een folder de basis parameters van een node project. De file bevat meta-elementen (name, version....) , de startfile (main) maar ook de versie en naam van andere afhankelijke pakketten. De file wordt gebruikt om een bestaand project lokaal te installeren met "npm install" door het ophalen van de vermelde modules (met de juiste versie!).

Node IDE's maken de `package.json` file wel aan, maar je kan deze file ook manueel aanmaken met een texttool als file bestand of je doet het via de console met ➤ **npm init** :

```
D:\>mkdir myNodeApp
D:\>cd myNodeApp
D:\myNodeApp>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (myNodeApp)
```

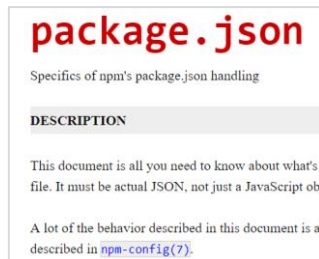
```
{
  "name": "01.node-intro",
  "version": "0.0.0",
  "description": "Inleiding op Node",
```



```
"main": "app.js",           //het entry point van de applicatie
"scripts": {                //laat toe instructie te runnen via > npm run . ..
  start: "init.js",         //frekwent gebruikt
  test: "doTest()"
},
"author": {
  "name": "Johan.Vannieuwenhuyse@howest.be"
}
}
```

De mogelijkheden van npm en package.json komen verder nog bod, maar toch al enkele basis mogelijkheden. De package.json file kan worden beheerd in npm (node package manager):

> npm install	zoekt package.json en installeert de app indien gevonden
> npm install --save aModule	installeert de module lokaal en neemt ze op in package.json
> npm install --save -g aModule	installeert de module globaal, neemt ze op in package.json
> npm run start	start dit script in de "scripts" tag van de package.json.
> npm help json	uitleg over de package.json tags:



3.6 Debuggen in node.js

1. Met `console.log()` en `console.trace()`:
Met `console.log()` kan rudimentair de inhoud van variabelen uitgevraagd worden. Dit blijft handig om de flow van een applicatie op te volgen met regelmatige console boodschappen. Er kunnen placeholders gebruikt worden, in combinatie met een specifiek karakter om bijvoorbeeld een *JSON object* (%j), een *getal* (%d) of een *string* (%s) weer te geven.
`console.log("Weergave van zowel het json object %j als het getal %d", oDieren, 0xab);`
Naast het `console.log()` command is ook het `console.trace()` command interessant voor debugging doeleinden. De volledige stack trace wordt uitgeprint door dit command.
2. Met de built-in text debugger van V8 en node (niet gebruiksvriendelijk):
 - a. Start een debug sessie vanuit de CLI met `node inspect myFile.js`

```
D:\BE-201617\Temp>node inspect HelloWorld
< Debugger listening on ws://127.0.0.1:9229/72096028-5e9b-4fcc-807
< For help see https://nodejs.org/en/docs/inspector
Break on start in HelloWorld.js:1
> 1 (function (exports, require, module, __filename, __dirname) {
  2 const inspector = require('inspector');
  3
  debug>
```

- b. De console toont de debug mode aan met een "debug>" prompt. Beschikbare commands vraag je op met debug>help

```
debug> help
Commands: run <r>, cont <c>, next <n>, step <s>, out <o>, backtrace <bt>, setBreakpoint <sb>, clearBreakpoint <cb>, watch, unwatch, watchers, repl, restart, kill, list, scripts, breakOnException, breakpoints, version
```

De debugger onderbreekt de uitvoering vanaf de eerste lijn. De volgende lijn oproepen kan met debug>next. Een breekpunt, te bereiken met debug>cont, voeg je toe met "debugger" in de source code.

Zo kan een watch list opgebouwd worden voor bijvoorbeeld de variabele iets. Let op: de variabele wordt als string opgeroepen:

```
debug > watch ('iets').
```

3. Met een debugging tool zoals "node-inspector".

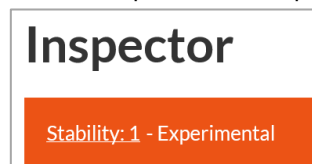
Dit tool zorgt ervoor dat je een node file kan runnen en debuggen binnen de chrome dev tools.

- a. Oorspronkelijke diende de Inspector te worden geïntalleerd als een globale module met het command:

```
npm install -g node-inspector
```

Sedert node 6.3+ voorziet node zelf een inherente module die je kan oproepen met

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



Ref: <https://nodejs.org/dist/latest-v8.x/docs/api/inspector.html>

Een handleiding voor de inherente inspector vind je op :

https://medium.com/@paul_irish/debugging-node-js-nightlies-with-chrome-devtools-7c4a1b95ae27

- b. Start de debug sessie in CLI met
 > node --inspect myFile.js
 of > **node -inspect-brk myFile.js** om bij het eerste breekpunt te stoppen (en vergeet -- niet in te voeren).
- c. Open chrome met **chrome://inspect**
 Gebruik verder de chrome devtools.en zijn node dev tools
 Om de chrome dev tools te verkennen kan je terecht op
 <http://discover-devtools.codeschool.com/>
- d. kill node om een debug sessie te herstarten

4. Maak gebruik van een IDE die debugging voorziet.

Gebruik jouw favoriete editor:

TextMate (http://macromates.com/):	Alleen voor MAC OS X
---	----------------------

Sublime Text (http://www.sublimetext.com/) met als opvolger Atom (https://atom.io)	Onbeperkte evaluatie periode – voor MAC en Windows. Atom beschikt over een rijke community met specifieke node addons zoals https://atom.io/packages/node-debugger
Coda (http://panic.com/coda/):	Supporteert ontwikkeling met iPad. FTP browser.
Aptana Studio (http://aptana.com/)	Volwaardige IDE
Enide Eclipse  (http://www.nodeclipse.org/enide/)	De eclipse versie voor node noemt Enide-Eclipse
Notepad ++ (http://notepad-plus-plus.org/):	Windows only text editor
Atom	
<u>WebStorm IDE</u> (http://www.jetbrains.com/webstorm/) (http://plugins.jetbrains.com/plugin/6098?pr=phpStorm)	Volwaardige en rijke IDE met node js debugging. PHP storm voorziet een plugin
<u>Visual Studio 2017</u> (met Node Tools for Visual studio)	Volwaardige en rijke IDE. NTVS dienen te worden geïnstalleerd bij versies lager of gelijk aan VS2015.
<u>Visual Studio Code</u> (https://code.visualstudio.com/)	Open source IDE met intellisense, debugging en Git built-in.

5. Nog meer plugins voor gemakkelijker debuggen en ontwikkelen:

a. Auto restart na error:

Bij een programmeer fout kan het nodig zijn het lopende process manueel te killen en daarna terug op te starten. Een aantal tools kunnen dit voor jou doen en versnellen zo de ontwikkeltijd.

Deze tools kunnen geïnstalleerd worden vanuit npm en worden gerund via een node console commando: > node-dev theToolName.js

forever (<http://npmjs.org/forever>)

node-dev (https://npmjs.org/package/node-dev)
nodemon (https://npmjs.org/package/nodemon)
supervisor (https://npmjs.org/package/supervisor)

- b. Sommige IDE's verwachten sowieso het gebruik van een plugin voor Node.js. bvb. Sublime Text3 voorziet ,na het installeren van Package Control, een nodejs plugin (info: <https://www.exratione.com/2014/01/setting-up-sublime-text-3-for-javascript-development/>).

3.7 JSLint en JSHint:

JSLint en JSHint zijn beide een code kwaliteitstool voor javascript dat kijkt naar syntax fouten, stijl conventies en structurele problemen. Er bestaat ook een ESLint, die de community versie van jsLint is.

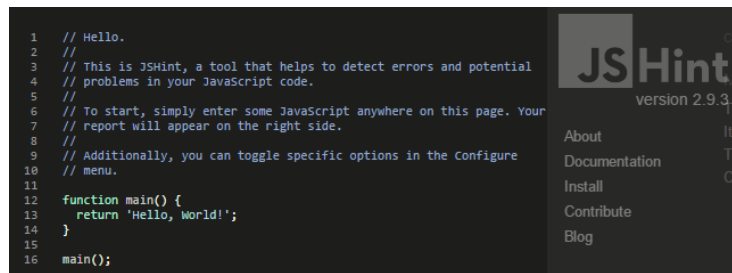
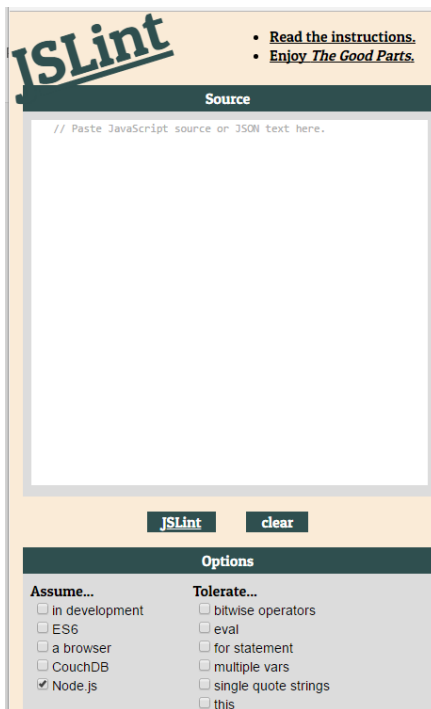
JSLint kijkt naar de regels zoals gedefinieerd in <http://javascript.crockford.com/code.html>. Deze regels zijn vergaand (= veel warnings). Als reactie erop ontstond **JSHint**, die minder streng is. Je hoeft immers niet alle opmerkingen te volgen van beide tools, soms is het bvb. verantwoord om een variabele lager in de code te plaatsen of om == te gebruiken in plaats van ===. De resultaten van beide moet je correct interpreteren. Vaak krijgt **JSHint** de voorkeur.

Info: Beide tools laten toe om online te testen:

- JSLint: <http://www.JSHint.com>
- JSHint: <http://www.JSLint.com>

Online

Online: testen doe je op de <http://www.jslint.com/> //vink nodejs of ES6 aan



3.7.1 Hinting installeren

Via npm en dus ook via de scripts file van de package.json file kunnen we een hint test uitvoeren. Vooraf installeer je jsHint globaal:

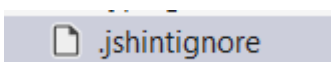
```
>npm install -g jshint
>jshint Helloworld.js
```

JSHint genereert onmiddellijk fouten, waarvan sommige onmiddellijk herkenbaar en aan te passen zijn (missing semicolon). Andere fouten binnen Node kunnen gegenereerd worden door niet gekende variabelen zoals "require", "before", "after", net zoals alle ES6 operatoren ongekend zijn. Dit is zeker het geval indien het volledig project gelint wordt.

```
col 26, Missing semicolon.
col 22, 'arrow function syntax (=>)' is only available in ES6 (use 'esversion: 6').
col 1, 'const' is available in ES6 (use 'esversion: 6') or Mozilla JS extensions (use moz).
```

Via de package.json kunnen deze fouten (die geen fouten maar warnings zijn voor bvb. ES6) geneutraliseerd worden in een configuratie tag:

```
"jshintConfig": {
  "esversion": 6,
  "moz": true,
  "node": true,
  "predef": [ "MY_GLOBAL" ]
}
```



Files die je niet met JsHint wenst te controleren komen optioneel in een **.jshintignore** file in de root. Een voorbeeld van de inhoud van deze file:

```
node_modules
assets
```

coverage

3.7.2 Hinting via een taskmanager

Een hinten kan via een taskmanager zoals gulp uitgevoerd worden.

Installeer gulp en nodige dependencies:

```
> npm install --save-dev gulp  
> npm install --save-dev jshint gulp-jshint //beide installeren  
> npm install --save-dev gulp-util  
> npm install --save-dev jshint-stylish //indien een reporter nodig vanuit gulp
```

Maak een gulpfile.js aan volgens de referentie: <https://www.npmjs.com/package/gulp-jshint>
Maak gebruik van een .jshintignore voor files die niet moeten gehint worden.

Roep de aangemaakte task van gulpfile op via de command console (>gulp hint) of via een package.json "scripts" tag (npm run testGulp)

Voorbeeld: "scripts": { "testGulp": "gulp " }

4 Alternatieve installatie: IISNode (ter info)

4.1 IISNode als handler

4.1.1 Waarom IISNode

Ref: <https://github.com/tjanczuk/iisnode>

Node.js kan als javascript server ook op IIS geïnstalleerd worden. Voordeel zou het gemak kunnen zijn om alles op één en eenzelfde server (IIS) te beheren.

Om dit te realiseren wordt Node.js geïntegreerd in IIS onder de vorm van een standaard IIS handler. Hiervoor maakt IIS gebruik van de zo genoemde "IISNode" module. De IISNode module laat toe om verschillende node.exe processen voor een applicatie op te starten. Ook hier is geen infrastructuur nodig voor het starten of stoppen van processen en runt IISNode elk proces/elke taak single threaded op één CPU kern. Door het opstarten van meerdere processen per applicatie wordt aan load balancing gedaan.

Hoe werkt de handler? Door IISNode als een HTTPHandler – geschreven in javascript- te implementeren worden *aspx requests naar ASP.NET gestuurd, terwijl de node requests bij node.exe terecht komen*. De IISNode handler werkt m.a.w. moeiteloos samen met andere content types, zoals bijvoorbeeld ASP maar ook PHP.

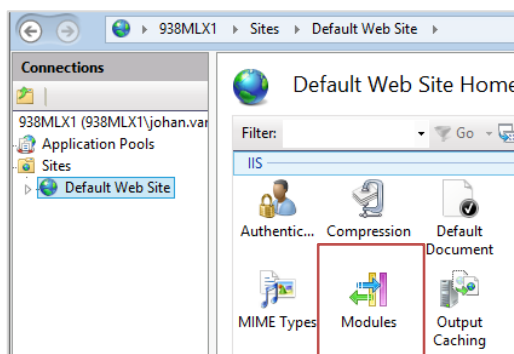
IISNode wordt niet alleen gebruikt omdat men in een productie omgeving al beschikt over IIS maar ook wordt het gedaan om vanuit Visual Studio te kunnen ontwikkelen met node.js als handler. Hoewel, voor dit laatste bestaat een alternatief dat verder aan bod komt.

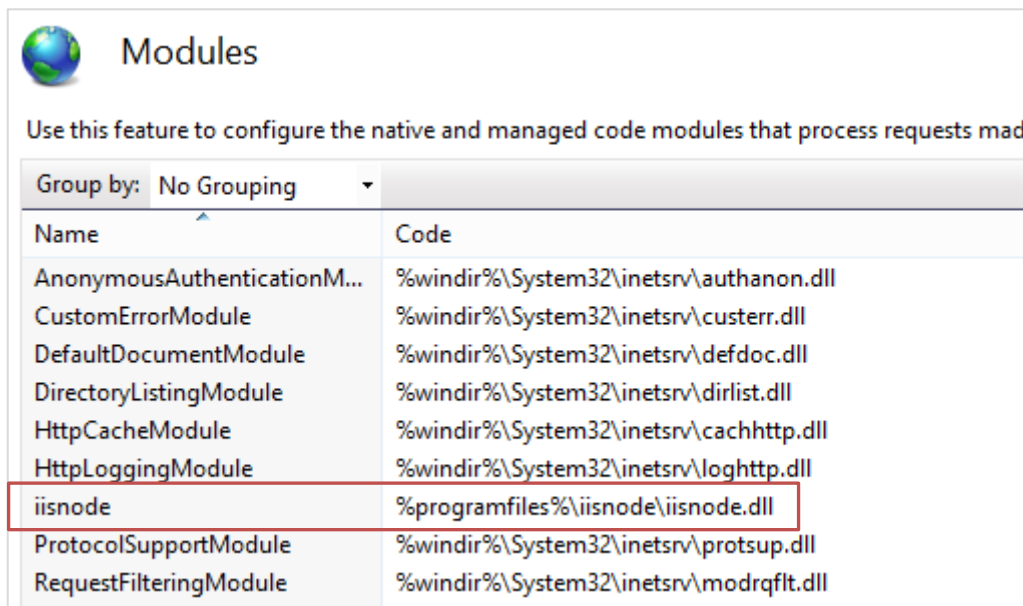
4.1.2 IISNode installeren

Het installeren van IISNode verloopt minder vlot dan het installeren van node.js. Een beschrijving van de installatie vind je terug op:

<http://www.hanselman.com/blog/InstallingAndRunningNodejsApplicationsWithinIISOnWindowsAreYouMad.aspx> of op <https://github.com/tjanczuk/iisnode>

Na installatie is de module iisnode zichtbaar op de IIS server. Meer bepaald, in het module overzicht van IIS :



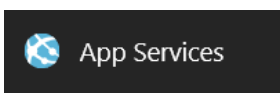


Use this feature to configure the native and managed code modules that process requests made to the web site.

Name	Code
AnonymousAuthenticationModule	%windir%\System32\inetsrv\authanon.dll
CustomErrorModule	%windir%\System32\inetsrv\custerr.dll
DefaultDocumentModule	%windir%\System32\inetsrv\defdoc.dll
DirectoryListingModule	%windir%\System32\inetsrv\dirlist.dll
HttpCacheModule	%windir%\System32\inetsrv\cachhttp.dll
HttpLoggingModule	%windir%\System32\inetsrv\loghttp.dll
iisnode	%programfiles%\iisnode\iisnode.dll
ProtocolSupportModule	%windir%\System32\inetsrv\protsup.dll
RequestFilteringModule	%windir%\System32\inetsrv\modrqflt.dll

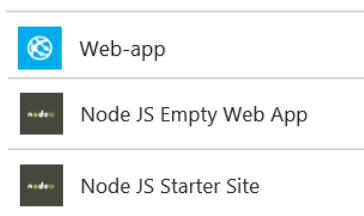
4.2 Node IIS als een Azure App service

Ref: <https://docs.microsoft.com/en-us/azure/app-service-web/app-service-web-get-started-nodejs>
<https://docs.microsoft.com/en-us/nodejs/azure/?view=azure-node-2.0.0>



"**Azure App Service** uses **iisnode** to run Node.js apps. The Azure CLI and the Kudu engine (Git deployment) work together to give you a streamlined experience when you develop and deploy Node.js apps from the command line"

Node kan zowel in **IaaS** (Infrastructure as a service met enkel Azure Hardware) als **PaaS** (Platform as a Service met Azure hardware en Azure OS). De App Services laten toe zowel Web applicaties, API applicaties als Mobiele applicaties (push notificaties) te bouwen. Er is een template voor een Node WebApp maar evengoed kan gestart worden vanuit een standard web-app, waarin de node applicatie gedeployed wordt.



Zoals gebruikelijk hoort bij een App Service een App Service Plan, dat bepaalt hoe krachtig (hoeveel CPU, aantal GB, aantal cores, aantal RAM) de service hoort te zijn. Het plan kan gebruikt

worden voor meerdere App Services zodat scaling eenvoudiger wordt door upscaling (=toevoegen van CPU-power) of outscaling (=meer machine instances).

Combineer de de IIS Azure service met git deployment en je krijgt een deftige ontwikkelomgeving met debug mogelijkheden in de cloud. Om de Azure App service (vb. webapp) aan te maken kan gewerkt worden met het publish command vanuit de Azure SDK in Visual Studio (= **SDK centric** deployment) of er kan gewerkt worden met de Azure Command Line Interface om te pushen naar een eerder aangemaakte Azure webapp via de portal (= **Azure centric** deployment) .

Via de CLI kan je diverse node templates/frameworks ophalen (vb. vanuit de yeoman generator) om nog sneller een lokale app te hebben. Je maakt deze app lokaal git controlled (>git init) en je pusht naar Azure. Dit kan evengoed vanuit de project templates in Visual Studio. Dit komt verder nog aan bod.

5 Visual Studio of PHPstorm/Webstorm

5.1 Visual studio (NTVS)

Ref: <https://www.visualstudio.com/vs/node-js/>

Node.js ontwikkelen kan complex worden, zeker als je er een volledige webapplicatie mee wil bouwen. In **nov 2013** zorgde Microsoft voor **NTVS: Node Tools for Visual Studio**. Debuggen kan zowel lokaal als remote en een interactieve REPL console is voorzien.. in combinatie met WebEssentials krijg je een node ontwikkel platform.

1. De NTVS zijn inherent geïmplementeerd in VS2017. Wie gebruik maakt van een eerdere versie dient NTVS en WebEssentials te installeren via Menu >> Tools >> Extensions and Updates:



Web Essentials 2015

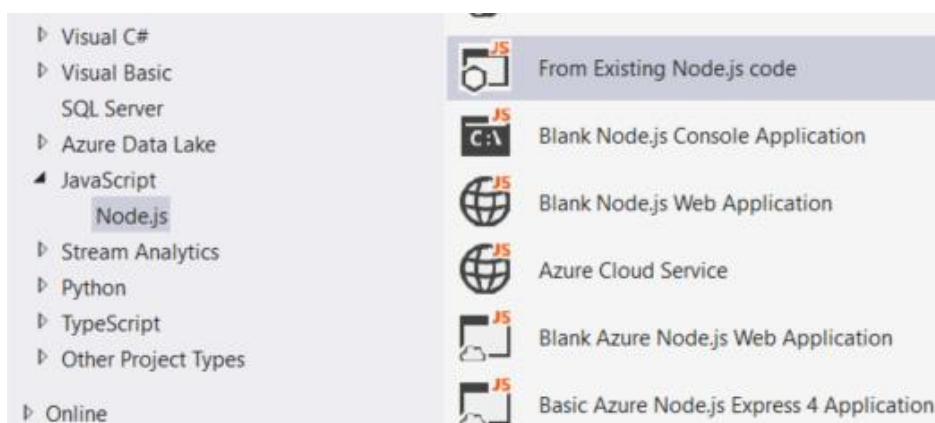
Adds many useful features to Visual Studio for web developers. Requires Visual Studio 2015 RTM



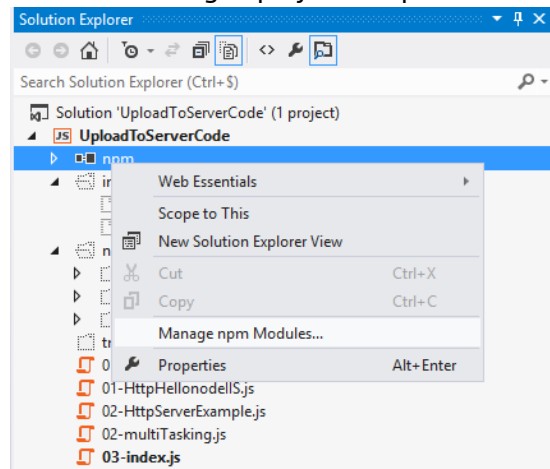
Node.js Tools

Provides support for editing and debugging Node.js programs.

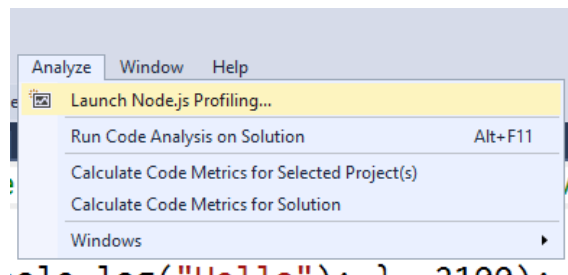
2. Aanmaken van een node project:
 - a. Er kunnen verschillende types van nodejs projecten aangemaakt worden vanuit new project >> javascript:
 - i. een nodejs project, waarbij ofwel een nieuwe nodejs applicatie gemaakt wordt of waarbij een bestaande nodejs app kan geopend worden.
Dus: bij het doorsturen van een node app, stuur je de nodemodules niet mee, maar laat je ze genereren via package.json!
 - ii. een express project
 - iii. een Azure applicatie



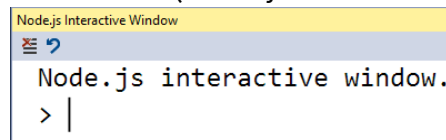
- b. Installatie van modules kan nog altijd via het npm command (npm install) of kan via de visuele manager: project >> npm >> Manage npm modules



- c. Nodejs profiling kan via het Analyze menu of via een profiler onder het Debug menu.



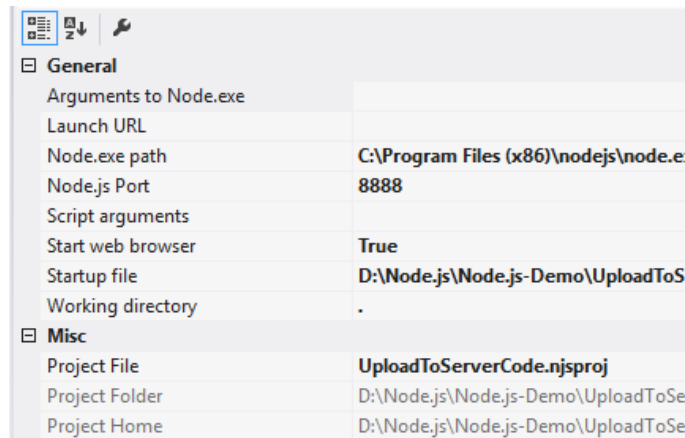
- d. View >> Other windows >> node js interactive window opent een interactieve console (gelijkaardig aan REPL-console werking). Kan ook met CTRLK , N. Een rood vierkant in de toolbar van de console laat toe een running command te onderbreken (bvb. bij een foutief ingegeven command).



Via de Tools >> NuGet Package Manager >> PM console bereik je een console, gelijkwaardig aan de opdrachtprompt:

```
PM> node 360-TCPserver.js
luisteren naar poort1337
PM>
```

3. Maak een nieuwe "Blank Node.js" Web application aan in een map naar keuze. Gebruik jouw eerder aangemaakte nodejs als test in het project.
 - a. Kopieer de *.js file.
 - b. Zet deze file via de properties (rechtermuis) als "Set as Node.js startup file".
 - c. Kijk eens naar de properties van het project. Je ziet er niet alleen de node executable, maar ook de startup file en het poort nummer. Je kan een webbrowser automatisch starten.

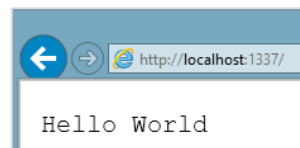
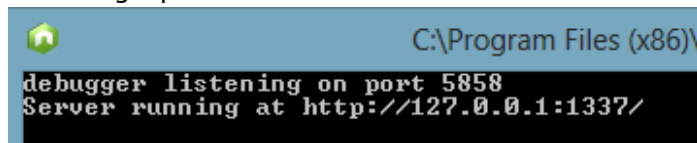


d. Run en debug de toepassing.

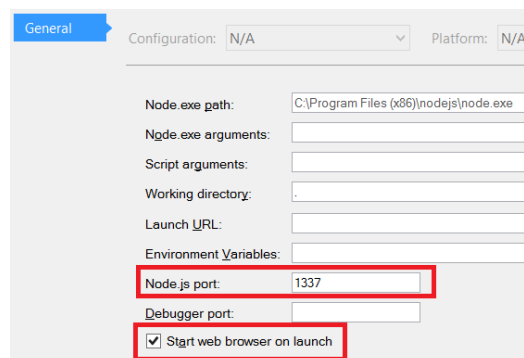
- i. De console opent automatisch maar sluit ook automatisch na voltooide taken. Je kan de console live houden met setTimeout:

```
setTimeout(function () {
    process.exit();
}, 15000)
```

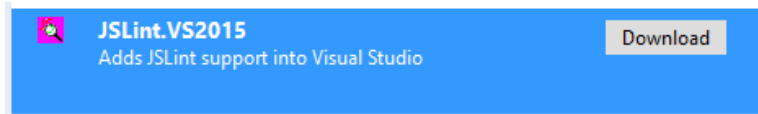
- ii. Bij een webbrowser toepassing wordt de console en eventueel de browser geopend.



- iii. Bij een browser toepassing kan je het poortnummer aanpassen en de browser al dan niet automatisch starten via de project properties.



Noot: Visual Studio voorziet een extensie voor JSLint met documentatie in [GitHub wiki](#)

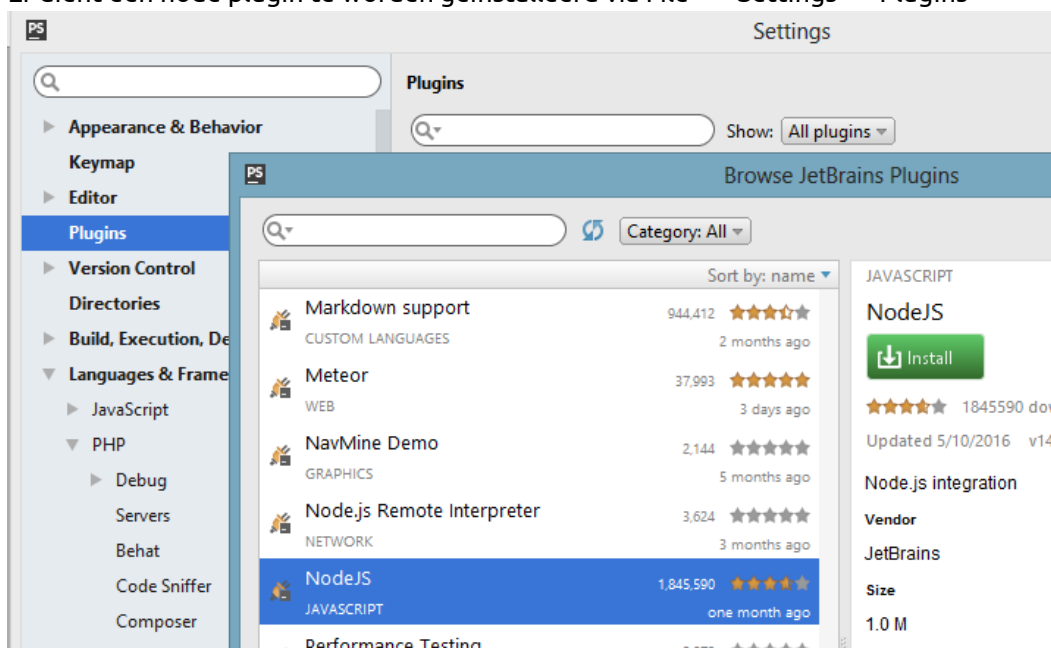


5.2 PHPStorm / WebStorm

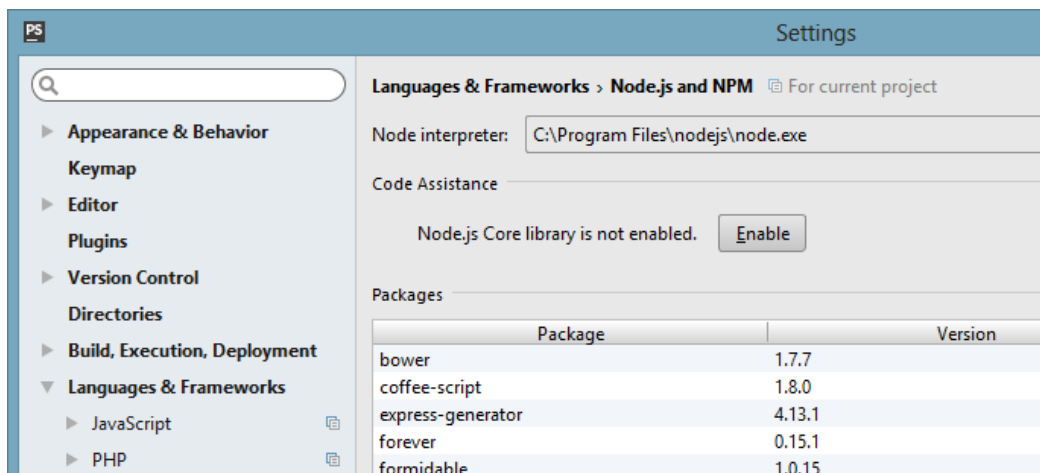
Ref PHP Storm: <https://www.jetbrains.com/help/phpstorm/running-and-debugging-node-js.html>

Ref Webstorm: <https://www.jetbrains.com/help/webstorm/running-and-debugging-node-js.html>

1. Er dient een node plugin te worden geïnstalleerd via File >> Settings >> Plugins

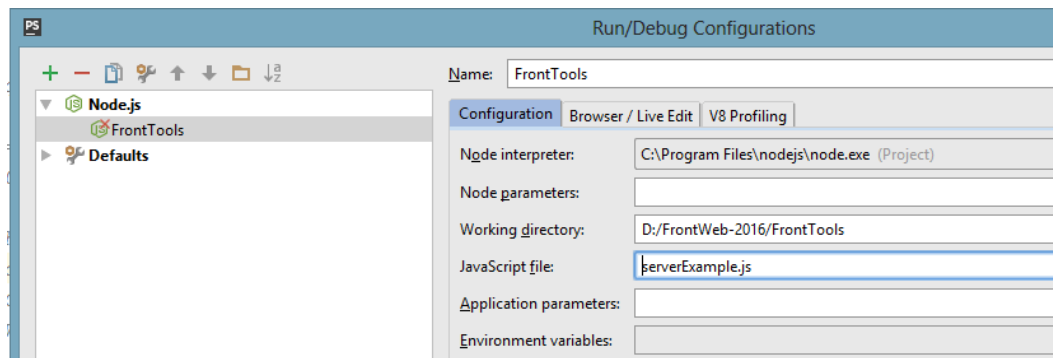


2. Interpreter instellen:
Settings >> Preferences >> Languages & Frameworks >> Node.js
controleer indien de interpreter ingevuld is en "ENABLE" de node core.



Na enablen is de usage scope opvraagbaar (= map waarin node en node libraries runnen).

3. Configuratie op basis van de interpreter (rechts boven) >>> run en debug Startfile instellen:



6 Asynchroon programmeren met de event loop

6.1 Callback om Asynchroon (=event-driven) te programmeren

Het asynchroon of event-driven programmeermodel

Bij multi-threading worden verschillende threads (= een process waarbij geheugen gedeeld wordt) gebruikt om processen van verschillende gebruikers te beheren. Een thread kan wachten op het voltooiën van een I/O zoals data van een disk, het ontvangen van database gegevens en gedurende deze tijd de wordt CPU vrijgeven aan een andere thread. Dit is een typische blocking/blokkerende methodiek. Een thread wordt simpelweg tijdelijk opgehouden tot zijn actie voltooid is. Voor de programmeur kan het moeilijk worden hierbij controle te houden over de synchronisatie van verschillende processen over verschillende threads. Het kan moeilijk worden om te weten welk proces op een specifiek moment op welke thread wordt uitgevoerd. Het gebeurt wel. Apache maakt bijvoorbeeld per request een thread aan.

Javascript (en dus ook Node.js) is **single threaded**. Wat betekent dat javascript maar één ding simultaan kan afhandelen. Geen overhead meer van multithreaded processes. Wel wordt de illusie gewekt dat meerdere zaken simultaan gebeuren door het toepassen van **event-driven programmeermodel** in combinatie met een **event-loop**.

Bij event-driven programmeren wordt de volgorde van de proces uitvoer bepaald door events. Een event wordt afgehandeld door een event handler, die gebruik maakt van een event callback functie. Nodejs gebruikt hiervoor zijn "libuv" library, geschoold op het voorbeeld van de Ruby Event Machine.

Bij event-driven programmeren wordt eerst de callbackback functie gedefinieerd, die beschrijft wat moet gebeuren en pas opgeroepen wordt als een voorgaand proces beëindigd is. Deze callback kan zowel via een benoemde als anonieme functie. De callback functie wordt meegegeven als argument van een uit te voeren actie. Een return value, te vinden in klassiek blocking programmeren, bestaat hier niet (!) en is vervangen door de callback functie, die in de achtergrond uitgevoerd wordt. Men spreekt van event-driven programmeren maar noemt het ook vaak asynchroon programmeren. Het is één van de basistechnieken van node.js. In node worden I/O operaties asynchroon uitgevoerd. I/O operaties blokkeren zo de single threaded werking van javascript niet. Voer in node de I/O operaties asynchroon uit met een callbackfunctie en de uitvoer ervan gebeurt (zonder zorgen van blokkeren) in de achtergrond.

```
//01. Synchrone uitwerking met een return -----
//definieer de functie
task(args) {
    return do_task_with(args);
};

//gebruik vd functie
task(args);

//02. async. uitwerking met anonieme callback, die return vervangt -----
var task = function (args, callback) {
    do_task_with(args);
```

```

        callback(err,taskresult );
    }

    task(args, function(err,data) { //result van callback uitvoering });

    //02. ES6 async. uitwerking met anonieme callback, die return vervangt -----
    const task = (args, callback) => {
        do_task_with(args);
        callback(err,taskresult);
    }

    task(args, (err, data) => { //result van callback uitvoering });

```

De ES6 arrow functie heeft als voordeel om binnen de callbackfunctie het keyword this eenvoudiger te interpreteren. In ES5 callbackfuncties wordt met this het top "Window" object teruggegeven of krijg je "undefined" naargelang de browser. Vaak wordt dit probleem opgelost door het aanmaken van een **self** variabele, buiten de callback, die **this** bevat. Het oorspronkelijke callback object teruggeven kan nu in ES6 wel door de pijlnotatie te gebruiken. Men spreekt over het "**lexical binding**" principe van arrow functies. Gebruik daarom in Node zoveel mogelijk de arrow functie of pijlnotatie. M.a.w. vervang function(){ } door ()=>{ }

```

    //03. async. uitwerking met benoemde callback voor het resultaat -----
    const task = (args, callback)=> {
        do_task_with(args);
        callback(err, taskresult);
    }

    const taskFinished = (err, data) => {
        //resultweergave
    }

    task(args, taskFinished);

```

Node gebruikt de error-first syntax voor callbacks:

Het resultaat van de callback functie kan succesvol zijn, maar ook evengoed een error.

Bij asynchrone werking wacht het hoofdprogramma niet op het resultaat van de callback functie. De callback functie wordt in de achtergrond uitgevoerd, terwijl het hoofdprogramma verder gaat. Ideaal dus om in de callback een input/output actie onder te brengen die wat tijd kan vragen. Denk bijvoorbeeld aan een antwoord van een database query of het resultaat van complexe calculate(), die niet onmiddellijk verwerkt moet worden in jouw programma. Evengoed kan je gedurende de wachttijd een al een tweede database call starten.

Het lezen van de code kan complexer worden, zeker wanneer ook error controle nodig is (= altijd dus). Volgende afspraken worden gemaakt:

1. Callback last:
Typisch wordt volgens (een niet geschreven) conventie het laatste opgeroepen argument de callbackfunctie. Andere argumenten nodig voor de verwerking gaan de callback arg vooraf.

2. Error first:

Bij de callbackfunctie is het eerste argument typisch de error waarde. Men kan spreken over een "ERROR FIRST" syntax. Soms wordt de term Continuation-Passing Style (CPS) gebruikt om aan te duiden dat een functie een callback veroorzaakt, die verder zorgt voor de afhandeling (continuation) van het programma.

SYNCHRONE werking	ASYNCHRONE werking met callback
<pre>let data = getData(); console.log("Synchroon: " + data); //data kan een error zijn</pre>	<pre>getData((err,data)=> { console.log("Asynchroon: " + data); })</pre>
<p>Een langdurige en synchrone processData() blokkeert het hoofdprogramma (single threaded) :</p> <pre>function processData(increment) { if (err) { console.log("An error occurred."); return err; } var data = getData(); data += increment; return data; }</pre> <p>console.log(processData(2));</p>	<p>Een callback functie wordt opgeroepen. Van zodra de callback beëindigd is wordt een resultaat of een fout teruggegeven. Intussen is het hoofdprogramma niet geblokkeerd (non-blocking)</p> <pre>processData = (increment, callback) => { getData((err, data)=> { if (err) { console.log("An error occurred."); callback(err, null); } data += increment; callback(null, data); }); }</pre> <pre>processData(2, function (err, returnValue) { //deze code wordt pas async uitgevoerd na het beëindigen van getData() console.log(err returnValue); });</pre>

Betere leesbaarheid met ES6 (= ES2015)

Ecma Script 6 (ES6) maakte er werk van om de lastig te lezen callback structuur te vereenvoudigen. De pijlnotatie (= arrow functies) werden ingevoerd. Om ook het functie scope probleem zoals hoisting aan te pakken werden let en const ingevoerd.

1. Gebruik const (een constante) voor het aanmaken van een functie definitie, zo heb je een controle dat een tweede functie nooit dezelfde naam krijgt. Vergeet ook niet strict te werken ("use strict"; //bovenaan de file plaatsen). Strict werken wordt verwacht door let en const.
const ES6_sum = (a,b) => a + b;

- Gebruik `let` in een `for` lus. Dit maakt een nieuwe block scope aan voor je `for` lus. Met een `var` is dit niet het geval en krijg je een functie scope.
`for (let element in elements){ }`
- Gebruik de pijlnotatie bij callbackfuncties van eventhandler. Naast de verkorte schrijfwijze heeft de pijlnotatie meerwaarde dat binnen de functie het keyword "this" verwijst naar het oproepende object, het event target dus. (en niet naar het window object zoals in ES5).

Een voorbeeld voor het optellen van twee getallen (error control niet uitgewerkt):

ES5:	
synchron	<pre>var sum = function (a, b) { return (a + b); } console.log(sum(2, 3));</pre>
asynchron	<pre>var aSum = function (a, b, callback) { callback(null, a + b); } aSum(2, 3, function (error, result) { if (error) { } console.log(result); })</pre>
ES6:	
synchron	<pre>const ES6_sum = (a,b) => a + b; // is een functie en resulteert in console.log(ES6_sum(3, 4));</pre>
asynchron	<pre>const ES6_aSum = (a, b, callback) => callback(null, a + b); ES6_aSum(3, 4, (error, result) => { if (error) { } console.log(result); }); //setTimeout(()=>process.exit(), 25000); //IDE niet direct afsluiten</pre>

Oefening 6.1: Async tegenover sync

We wensen een synchrone functie (load) te herschrijven op een asynchrone manier. De load functie verplaatst een willekeurige lijst van `userIds[]` naar een tweede array `users[]`. De load functie voor één id duurt één seconde.

We willen de code omzetten van een synchrone werking naar een asynchrone werking. Om de volledige duurtijd van het process op te meten, maken we gebruik van `console.time()`

Bij de synchrone werking zal de duurtijd minstens gelijk zijn aan het aantal `userIds` * 1 seconde. De vertraging simuleren we met de `delay` parameter in `sleep()`. De synchrone werking ziet er als volgt uit en is herkenbaar aan de returns en while structuren. Let tevens op de volgorde van de functies, die wordt opgelegd door het bloc scoping van `const`:

```
let users = []; //op te laden target Array
let usersIds = ["P1", "P2", "P3", "P4", "ERROR", "P6", "P7", "P8", "P9"]; //source
let delay = 1000;

const sleep = (time) => {
  var start = new Date().getTime();
  while (new Date().getTime() - start < time) {
    //just wait in a synchronous while
  }
}
```

```
const loadSync = (element) => {
  sleep(delay);
  return "element " + element + " loaded";
};

const loadArraySynchroon = (array, elements) => {
  //ES6 for/of lus -> value return
  for (let element of elements) {
    array[element] = loadSync(element);
    console.log(array[element]); // informatie wanneer ingeladen
  }
};

console.time("Synchrone Laadtijd");//monitoren van synchrone doorlooptijd
loadArraySynchroon(users, usersIds);
console.timeEnd("Synchrone Laadtijd");
```

Opgave: herschrijf bovenstaande code asynchroon met callbacks in ES6:

Bij de asynchrone werking kunnen alle loads onafhankelijk van elkaar gebeuren. Het inladen van de userIds zal veel sneller voltooid zijn. De delay blijft gesimuleerd, niet door een synchrone while, wel door de asynchrone setTimeout() !!! De functies load en loadArray behouden hun argumenten maar worden beiden uitgebreid met een extra argument: de callback functie (cb) die de returndata als zijn arg zal meebrengen.

```
const loadAsync = (element, delay, callback) => { . . . }

const loadArrayAsync = (arrayTarget, elements, callback) =>{ }
```

Alle elementen worden op een asynchrone manier ingeladen met loadAsync. Tel de elementen in loadArrayAsync om te weten wanneer alle elementen ingeladen zijn. Pas daarna kan de callback gebeuren met het finaal resultaat. Vergeet ook niet dat de console.logout met de finale doorlooptijd OOK asynchroon moet verwerkt worden. In een asynchrone toepassing moeten ALLE functies asynchroon aangebracht worden.

Noot: Waarschijnlijk gebruikte je een for lus (is ok). Maak voor het asynchroon inladen van alle elementen nu eens gebruik van forEach(callback[, thisArg]), precies omdat **forEach** een callback functie oplegt en automatisch een index en value argument voorziet.

6.2 De event loop beheert de asynchrone taken

Hoe werkt dit asynchroon model nu? Hoe zorgt Node.js ervoor dat I/O taken op een asynchrone manier uitgevoerd?

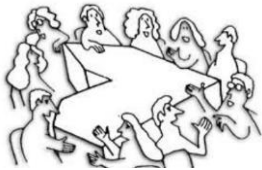
Het asynchroon programmeer model maakt gebruik van callback functies en wordt ondersteund door de event-loop. De event-loop start automatisch (!) op, wanneer je node start en behandelt het volledig beheer van de pool van events en hun callback functies. De event-loop voert simultaan twee zaken uit:

- De event loop haalt een event af van de event queue en *roept voor het event de bijhorende callback **handler** op*. De handler (of callback) wordt asynchroon uitgevoerd in de achtergrond en op het einde wordt het resultaat terug aan de applicatie bezorgd.
- De event loop houdt bij welk event juist gebeurde en *bouwt een event **queue** op*, die de volgorde van uit te voeren taken bijhoudt.

De event-loop verloopt single threaded en zorgt dat de callback taak gedurende zijn uitvoer ook nooit onderbroken wordt. Mocht je de event loop stoppen (sleep()) dan stopt ook je volledige applicatie. De voordelen van een event-loop zijn triviaal: synchronisatie software is niet langer nodig, concurrent processen verlopen non-blocking, efficiënt geheugen gebruik omdat er minder moet geschakeld worden tussen geheugen plaatsen, scaling wordt eenvoudiger. Natuurlijk kan node in de achtergrond zijn eigen threads en afzonderlijk processen gebruiken, maar tussenkomst van de ontwikkelaar is onnodig. De ontwikkelaar kan zich concentreren op de applicatie software eerder dan op synchronisatie software.

Node.js voorziet voor deze manier van werken verschillende non blocking libraries voor I/O acties zoals database queries, file reading, netwerk access. Het is duidelijk dat door de non blocking voordelen een groot aantal taken simultaan kan beheerd worden zoals bijvoorbeeld meer cliënt connecties of meerdere cliënt operaties.

Van synchroon (blocking) naar
asynchroon en non-blocking:



```
var result = db.query("select..");  
// use result
```

wordt asynchroon met een callback functie:

```
db.query("select..", function (result) {  
  // use result  
});
```

Tijds winst in de queue bij non-
blocking:



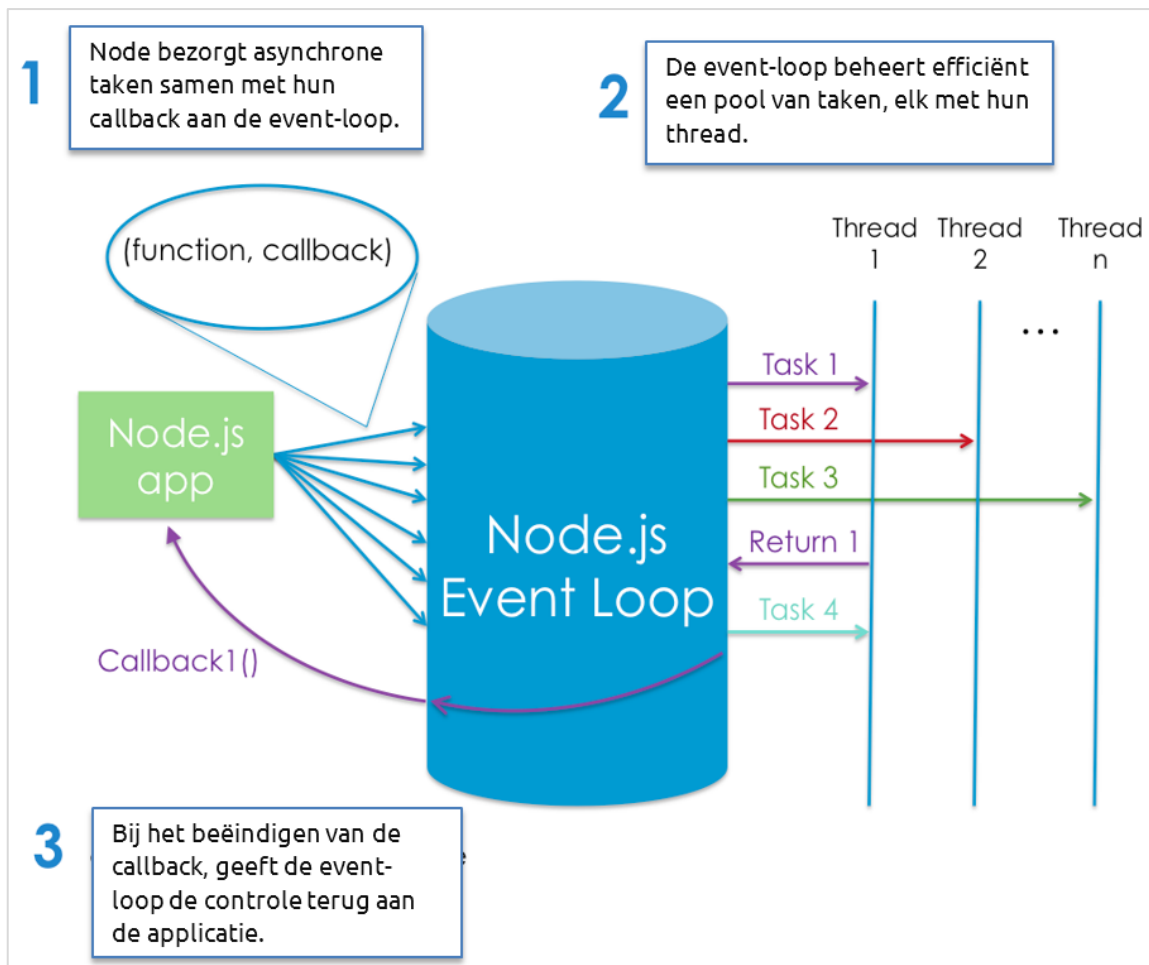
```
var callback = function(err, contents) {  
  console.log(contents);  
}  
fs.readFile('/etc/hosts', callback);  
fs.readFile('/etc/inetcf', callback);
```

blocking



non-blocking





Figuur 2: De Node.js Event-Loop lifecycle

Noot: Wisselwerking tussen V8-callstack en de eventqueue

Javascript maakt gebruik van de V8 callstack om de volgorde van functies bij te houden. De callstack (first in, last out) van V8 wordt met de functies gevuld. Er is maar één V8 callstack, want Javascript is single threaded. De stack wordt eerst (onderaan) gevuld met de hoofdfunctie. Erboven komen de opgeroepen functies. De laatst toegevoegde functies worden eerst uitgevoerd tot de volledige hoofdfunctie afgewerkt is.

Node beschikt echter over timers (of asynchrone functies), die slechts kortstondig op de stack komen. Deze timers (of async functies) voeren immers niets uit maar plaatsen hun callback functie in de eventqueue. De eventloop zorgt voor de uitvoer van de queue zonder dat de V8 callstack gebruikt wordt. Als de callstack leeg is en pas als het resultaat van de queue er is, wordt het naar de V8 callstack gebracht, die het resultaat onmiddellijk verwerkt.

Maar! Time consuming callbacks vertragen drastisch het output resultaat.

Node.js en javascript zijn gebouwd op single-threaded event loops. Iedere keer dat node.js een nieuwe eventloop start, spreekt men over een "tick". Deze tick bevat een queue van events (met bijhorende callback functies, die hun eigen gang gaan). Bij elke loop worden de events uit de queue opgenomen. Dit betekent ook wanneer een fired callback functie heel veel tijd vraagt, dit kan leiden tot een aanzienlijke vertraging van de pending events in deze queue. **Zware CPU-acties of extreem lang durende callbackfuncties kunnen hierbij resulteren in het sterk vertragen van de**

applicatie. Node verwacht daarom een snelle return van request, lukt dit niet dan moet toch aan het opsplitsen in processen gedacht worden of aan het gebruik van webworkers (beschikbaar via een module webworker-threads)

Volledigheidshalve kunnen we ook vermelden dat deze asynchrone techniek niet nieuw is maar reeds gebruikt werd door Ruby, Perl en Python. Wel is het zo dat Node.js het eerste platform is dat vanuit niets geschreven werd met dit in het achterhoofd.

6.3 Callback Hell = Pyramid of Doom = the Boomerang effect

Doordat de verwerktijd onbekend is voor een asynchrone transactie, krijgen we een probleem wanneer twee of meerdere taken na elkaar moeten gebeuren. In code betekent dat twee callbacks in elkaar gaan nesten. De geneste callback zal zo pas starten na de beëindiging van zijn parent callback.

Het wordt duidelijk dat een asynchroon programmeer model opgebouwd wordt door een keten van **geneste callback functies**. Dit kan het geheel zeer onleesbaar maken. Het kan lastig worden om je weg te vinden in een reeks callback functies. Bovendien is het niet altijd even duidelijk welke callback eerst zijn taak zal beëindigen. Men spreekt over de callback hell (ook pyramid of doom of boomerang effect genoemd).

Een filereader voorbeeld met drie geneste callback functies illustreert de driehoekige pyramide stijl:

```
const fs = require("fs");
let fileName = __dirname + '/MyTextFile.txt'; //absolute path vd file map

fs.exists(fileName, function (exists) {
  //callback 1: bestaat de file (enkel voor de demo -> gebruik stats)
  if (exists) {
    fs.stat(fileName, function (error, stats) {
      //callback 2: haal statistische data vd file op (is het een file?)
      if (error) { throw error };
      if (stats.isFile()) {
        fs.readFile(fileName, null, function (error, data) {
          //callback 3: lees binair indien stats een file aanduidt
          if (error) { throw error };
          console.log(data.toString('utf-8'));
        });
      }
    });
  } else {
    console.log(fileName, "bestaat niet.");
  }
});
```

Hoe kan je het geheel dan leesbaar houden?

Er wordt aangeraden om maximaal een tweetal callback functies te nesten voor leesbaarheid.

Is meer nesting nodig dan kunnen een aantal zaken gebruikt worden om de leesbaarheid te verhogen. Jaarlijks komen nieuwe technieken bij. We vermelden hier om de callback te vermijden methodieken zoals:

- het benoemen van callback functies

- het gebruik van Promises
- het gebruik van async/await
- het gebruik van distributed events

6.4 Benoemen van callback functies

Door elke callbackfuncties te benoemen wordt het geheel overzichtelijker. Benoemde functies zorgen tijdens het debuggen voor een duidelijker stack trace (`console.trace()`):

```
fs.access(fileName, cbAccess)

//callback 1
function cbAccess(access) {
  if (access) {
    fs.stat(fileName, cbStat)
  }
}
```

Oefening 6.3.1: Benoemde callbackfuncties

Verder uitbouwen van de benoemde callback functies op het fileReader voorbeeld, *inclusief error handling*.

6.5 Promises om asynchroon te programmeren

Asynchroon schrijven vraagt voor veel ontwikkelaars een aanpassing, zodat er continu gezocht wordt om dit eenvoudiger te maken. Denk hierbij niet alleen aan callbacks die goed aflopen, maar ook aan de behandeling van fouten in de callback functie. Promises bieden een oplossing aan die de Doom pyramide verhindert.

Het zelf aanmaken van promises en deze consumeren is geen noodzaak voor elk programma of elke I/O toepassing. Zeker niet wanneer het om een beperkt aantal async methodes gaat en alles overzichtelijk blijft zonder promises. Wel maken meer en meer libraries gebruik van promises, omdat de syntax eenvoudiger wordt (lagere instapdrempel).

Oorspronkelijk diende men een addon library te installeren om gebruik te maken van Promises in node. Elke library met promises kan wel verschillende syntaxen hanteren, zodat het lezen van de library API vaak noodzakelijk was. Voorbeelden van veel gebruikte promise libraries zijn: **Q.js** (<https://www.npmjs.com/package/q>) of **RSVP.js** (<https://github.com/ttildeio/rsvp.js>) of een deel van RSVP (= polyfill te installeren via npm install **es6-promise**).

Sedert Javascript native support heeft voor Promises (Mozilla >> MDN >> Javascript >> builtin Objects), zien we een groeiend aantal applicaties waar Promises gebruikt worden eerder dan callback functies. Nieuwe features in Node en ook sommige Node ontwikklers kiezen er bovendien voor om de beide methoden (callbacks & Promises) te combineren in één functie.

Noot: De techniek van Promises is niet alleen bruikbaar op de server, maar zorgt evengoed voor een betere leesbaarheid van asynchrone functies bij de browsers. (jQuery gebruikt promises sedert v1.5)

Ref:

https://developer.mozilla.org/nl/docs/Web/JavaScript/Reference/Global_Objects/Promise

<http://www.html5rocks.com/en/tutorials/es6/promises/>

6.5.1 Wat is een promise?

Een promise is een object: een “belofte” dat een resultaat volgt. Vergelijk het met een nog niet geleverde bestelbon. De levering (al dan niet foutief) volgt wel. De levertijd (die je niet onder controle hebt) beïnvloedt het uiteindelijke resultaat niet.

Promises bevinden zich in 4 mogelijke stati:

- Een succesvol of positief resultaat, waarbij data kan teruggegeven worden aan de aanroepende functie. (promise is fulfilled)
- Een failure of negatief resultaat, waarbij een error optreedt en de foutmelding wordt teruggegeven. (promise is rejected)
- De promise is in wachtstatus, voordat de callback wordt uitgevoerd. (promise is pending)
- De promise is volledig afgewerkt (promise is settled / done)

Dit betekent dat een promise enkel kan slagen of falen bij het uitvoeren van zijn asynchrone taak. Bijkomend (en verschillend ten opzichte van event listeners) kan een promise maar één keer slagen of één keer falen. De promise behoudt ook zijn status. Eénmaal resolved behoudt de promise deze status. We krijgen een immutable promise = status kan niet meer gewijzigd worden door third parties.

6.5.2 Hoe een Promise aanmaken?

Bij promises wordt geen callback **teruggegeven maar wel een Promise instantie** als wrapper voor een resolve en reject methode. Een Promise kan zo bekeken worden als een callbackmanager, waar de constructor als arguments twee callbackfuncties bevat: een resolve en een reject.

Een promise instantie wordt als volgt aangemaakt:

```
var promise = new Promise(function(resolve, reject) {
  //indien succesvol : resolve(aValue) op het einde
  //indien error: reject(anError)
})
```

Wees ervan bewust dat de resolve en reject asynchroon gebeuren. Dit betekent zoveel als: het resolve resultaat hoeft er niet onmiddellijk te zijn. Er wordt gewacht op het resultaat. Ook de reject wordt asynchroon opgeroepen. Een synchrone reject genereert onmiddellijk fouten.

resolve() en reject() zijn ook beschikbaar als statische methodes van Promise:
Promise.resolve(aValue)

Het consumeren van de promises steunt op de twee callbacks. In tegenstelling tot error-first syntax bij callbackhandlers komt nu eerst het succes (succes-first, error-last). Bovenstaande promise kan als volgt geconsumeerd worden:


```
promise.then(
  //success
  function () {console.log ("done!")},
  //failure
  function(error) { console.log (error.message) }
);
```

Een object dat gebruik maakt van promises (lees: dat een promise kan teruggeven) wordt ook “thenable” genoemd. Soms noemt men een dergelijk object ook een “deferred” object. Een “then” kan in code aangebracht om na een succesvolle of falende actie over te gaan naar een volgende actie van de promise. Dit laat toe om een keten te maken van promises. Verschillende asynchrone taken kunnen zo na elkaar uitgevoerd worden: succesvol of met een error. Hoe promises hiervoor gebruikt worden komt verder aanbod bij de “process flow van asynchrone taken”.

Een Promise voorbeeld:

De eerder aangemaakte sum met callbacks ziet er met Promises als volgt uit:

```
//definiëren van een Promise return
const pr_sum = function (a, b) {
  return new Promise((resolve, reject) => {
    //if (err) {reject(err); }
    if ( typeof(a) !== 'number') {reject('Dit is geen getal'); }
    resolve(a + b);
  });
};

//de Promise consumeren
pr_sum(12, 4)
  .then(
    result => { console.log('enkel Promise resultaat: ', result); },
    error => { console.log('enkel Promise error: ', error ) } ) ;
```

Promises en callbacks gecombineerd:

```
const PrCb_sum = function (a, b, cb = () => { }) {
  return new Promise((resolve, reject) => {
    //if (err) {reject(err); cb(err,null); }
    if ( typeof(a) !== 'number') {
      reject('Dit is geen getal');
      cb ('Dit is geen getal', null);
    }
    resolve(a + b);
    cb(null, a + b);
  });
};

PrCb_sum(16, 4)
  .then(
    result => { console.log('Promise resultaat: ', result); },
    error => { console.log('Promise fout: ', error ) } ) ;

PrCb_sum(20, 4, (error, result) => {
  console.log("Callback resultaat: ", result||error);
```

```
});
```

6.5.3 Een 'then' maakt nesten van callbacks overbodig

Een promise zelf retournt geen data maar kunnen we bekijken als *een wrapper of functie die het resultaat van een asynchrone taak retournt*. Een then resulteert in een success of een error. Maar Promesses laten toe om meerdere thens na elkaar te plaatsen. Het na elkaar plaatsen van 'thens' komt overeen met het nesten van callbacks, zodat de pyramide van Doom verhinderd wordt.

Een fulfilled promise kan gechained worden met een volgende promise. Dit wordt aangegeven met het keyword "then". Deze then retournt ook zelf een "promise object". Een `return` binnen de then methode zorgt dat op het resultaat gewacht wordt, alvorens een volgende then uit te voeren. Het resultaat kan opvangen in een argument van de then functie:

```
doPromiseTask1()
  .then(function () { return doTask2(); }) //retournt waarde naar cbResult2
  .then(function (cbResult2) { return doTask3(cbResult2); })
  .then(function () {console.log ("done!");}, new Error("taak faalde"))
```

Een rejected promise kan gechained worden met een catch. Ook deze catch retournt een promise, maar kan verkort geschreven worden als catch(reject) in plaats van then(undefined, reject). De returned promise kan je opnieuw in een variabele onderbrengen om eventueel verder in je applicatie nog extra thens toe te voegen:

```
const myPromiseInstance =
  new Promise(function (resolve, reject) {
    fs.readFile('tex.txt', function (err, text) {
      if (err)
        reject(err);
      else
        resolve(text.toString());
    })
  })
  .then (// de resolve
    function (response) {console.log("Success!\n", response);},
    //de reject
    function (error) {console.error("Failed!\n", error);}
  )
  .then (resolve('einde promise')); //resolve geeft de waarde terug
  .catch (reject(Error('foutje !'))); //verkorte then schrijfwijze
```

Oefening 6.3: Basis gebruik van Promises

Herschrijf de eerder gemaakte oefening, waar je een target array invult, met Promises. Voeg aan de class of de module een methode (loadArrayPromise) toe die een Promise retournt. Zorg hierbij wel voor een random inlaadtijd van elk element.

- De promise kent slechts 2 toestanden: ofwel success ofwel een reject.
 - Bij success zorgt een then sequence voor het stringifyen naar JSON van de users Array (= kan handig zijn om een service API aan te maken).
 - Bij reject wordt onmiddellijk een error teruggegeven.

- Een catch vangt alle mogelijke fouten in de then constructie op zonder de applicatie af te sluiten (test door te stringifyen op een niet bestaand iets).

De Promise runnen gebeurt als volgt. Bemerkt de eenvoudige schrijfwijze:

```
Loader.loadArrayPromise(users , usersIds)
```

```
.then(function () {
    console.log(users);
    return users;
})
.then(function (arr) {
    console.log(JSON.stringify(arr));
})
.catch(function (error) {
    console.error("Failed!\n", error);
});
```

Het resultaat:

```
ERROR laadtijd: 1.662ms
P6 laadtijd: 30.987ms
P7 laadtijd: 89.974ms
P3 laadtijd: 170.705ms
P4 laadtijd: 207.026ms
P8 laadtijd: 521.071ms
P9 laadtijd: 602.448ms
P2 laadtijd: 619.384ms
P1 laadtijd: 693.354ms
laadtijd met Promise: 694.532ms
```

Optionele oefening: Combineer callback en promises.

6.6 Async/await om asynchroon te programmeren :

Ref: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await>

De zoektocht naar leesbaarder async programma's blijft maar doorgaan. Met de async/await schrijft men als het ware synchroon, maar wordt de code toch asynchroon uitgevoerd. De await instructie kan enkel binnen een async gemerkte functie, voorzien van een try/catch:

```
async function asySum(a,b) {
    try {
        await console.log("async/await resultaat: ",a + b);
    } catch (err) {
        console.error(err);
    }
}

asySum(24, 4);
```

7 Over Javascript functies, modules, constructors en classes.

Werken met javascript en node betekent werken met callbackfuncties. Het zijn speciale objecten die daarom ook aan variabelen kunnen toegekend worden. Wat javascript functies méér kunnen dan pure objecten, is dat ze ook opgeroepen kunnen worden voor uitvoer (invoke a function). Het wordt duidelijk dat veel functies kunnen genest worden wat aandacht vraagt voor de scope van functies.

De functies zorgen voor een overstap naar herbruikbare code. We willen niet immers tweemaal dezelfde code schrijven, laat staan tweemaal opladen in eenzelfde app. Functies zorgen zo voor de overstap naar herbruikbaarheid via verschillende mogelijkheden gestructureerd in een framework of pattern. Enkele mogelijkheden zijn:

- closures and Immediate Invoked Function Expressions (IIFE)
- constructor pattern
- **module pattern**
- **ES6 classes**

Omdat node stilaan de 100% compatibiliteit met ES6 bereikt, zien we een stijgend gebruik van het module pattern en ES6 classes. Ook in dit document gaan we die richting uit. Eerst een opsomming van belangrijkste of meest gebruikte Javascript elementen.

7.1 Javascript (ECMA) en node

Javascript versies

- ES5: de javascript standaard (ECMA-262) sedert 2009
ECMA staat voor "European Computer Manufacturers Association" en bestaat sedert 1961 voor het publiceren, aanbevelen en standaardiseren van informatie en communicatie systemen. Javascript ontstond in 1995 met Brendan Eich.
- ES2015 is de nieuwste officiële javascript versie sedert juni 2015. Ook **ES6** genaamd.
info: <https://babeljs.io/docs/learn-es2015/>
 - o Nieuwe syntax (= minder lijnen code maken het overzichtelijker tot eenvoudiger)
 - o Nieuwe mogelijkheden om javascript code te structureren (nettere code, onderhoudbaarder)
 - o Nieuwe features: generators, iterators, promises, api's , collections, classes en object extensions.(native implementaties maken het sneller)
- ES2016 (ES7) is nog work in progress.

Bij node runt javascript op de server. Dit resulteert in een veel grotere uniformiteit van javascript versies. Reeds in versie 4 voorzag node een gedeeltelijke compatibiliteit met ECMA Script 5 en 6. In mei 2016 werd versie 6 van node voor 93% ES6 compatibel met interessante eigenschappen zoals: default, REST parameters, destructuring, class en super keywords

Een comptabiliteit kaart is te vinden op github: <https://kangax.github.io/compat-table/es6/>

Meest gebruikte ES5/ES6 EcmaScript eigenschappen/methodes in een node applicatie:

Onderwerp	Eigenschappen/methodes	Meer info
Array	<p>Array.isArray(array) Array.from(arrayLike, cb, args)</p> <p><i>prototype</i></p> <p>join(), concat(). pop(), push() shift(), unshift() sort(), reverse() slice(), splice()</p> <p>array.forEach(callback)</p> <p>array.some(callback, args)) array.find(callback) array.indexOf() array.filter(callback) array.map(callback)</p>	<p>Controleren of we een array hebben. (ES6) Transformeert array like object volgens cb(ES6)</p> <p>string converties laatste item: verwijderen, toevoegen eerste item:verwijderen, toevoegen sortering deel kopiëren, deel verwijderen</p> <p>Voert de callback op elk element uit</p> <p>returnt true indien gevonden returnt de gevonde waarde (ES6) index based opzoeken filteren volgens callback Transformeert array naar nieuwe array volgens cb</p> <p>Meer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array</p>
Date	Date.now()	Ophalen van de huidige tijd in de vorm van <code>new Date().getTime()</code>
Math	random()	Meer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math
String	<p>substr(), substring(): length indexOf() split() string.trim() string.trimRight()</p>	<p>delen vd string: lengte opzoeken converteren van string naar array witruimte verwijderen (ES6)</p> <p>Meer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String</p>
JSON	<p>JSON.stringify(obj [, replacer [, space]])</p> <p>JSON.parse(string)</p>	<p>Maakt JSON string aan.(serializeert javascript objects naar een string)</p> <p>Returnt het object (deserialiseren); Is veiliger(!) en performanter dan het eerder gebruikte "eval"</p>
Timers	setInterval(), setTimeout(), requestAnimationFrame()	
Object	<p>Object.create(proto[, props]) Object.defineProperty(obj, prop, descriptor)</p>	<p>Nieuw object aanmaken vanuit een ander (proto) Object eigenschappen aanmaken waarbij descriptor writable, enumerable, configurable beheert.</p>

Get/Set	<pre>Object.preventExtensions(obj) Object.hasOwnProperty("aProp") var obj = { get iets(){ return "aValue" }, set iets(v){ "initialize" } }</pre>	<p>Laat niet toe om eigenschappen toe te voegen. Controleren of de eigenschap bestaat.</p> <p>getter en setter syntax</p>
vars	<pre>{ const iets = "read only vasteWaarde"; // a geeft hier een ref.error let a = 10; }</pre>	<p>Het command 'let' laat expliciet "block scoping" toe. De variabele is enkel gekend binnen het block { } en is <i>pas gekend op de lijnen na zijn declaratie</i>. De garbage collector kuist sneller op.(ES6).</p>

Enkele voorbeelden:

```
let foundValue = arrPersons.find(person => person === "Johan");
let foundIndex = arrPersons.findIndex(person => person === "Johan");
let persons = persons.map(x => x.toUpperCase());
```

7.2 Functie scope !== block scope

Kenmerkend voor de functies van javascript, is dat *pas bij het oproepen* van een functie de functie scope aangemaakt wordt! Hierbij blijven de variabelen (met var) binnen de functies verborgen voor de buitenwereld ("encapsulation"), terwijl de variabelen in zijn parent scope toegankelijk blijven voor de functie. OO-talen werken daarentegen met met een block scope, waardoor binnen een functie de variabele eerst moet gedefinieerd worden, vooraleer ze kan gebruikt worden.

Binnen een functie scope voert javascript de declaratie (niet de initialisatie!) van de variabele eerst uit, waar deze declaratie ook maar staat. De variabele kan zelfs eerst vermeld worden in code en pas daarna worden gedeclareerd. Javascript plaatst als het ware de declaratie (= de var) op de eerste lijn van zijn functie scope. Men noemt dit **hoisting**. Dit is geldig voor vars en functies, waarbij functies voorrang hebben. Maar: controlestructuren (if, for ...) maken een blockscope aan maar geen functie scope!

```
//ES5
var checkMyScope = function (a, b) {

  counter = 0;

  var counter ; //hoisted = declaratie komt eerst (zonder initialisatie)
  console.log("counter returnt hier ", ++counter); //1

  console.log("en hier is de waarde van init: ", init); //undefined
  var init=10;

}
```

ES6 voorziet blockscoping met `let` en `const` zoals eerder vermeld:

```
const letMyScope = function (a, b) {

  //counter = 0; //error counter is not defined gezien geen hoisting gebeurt
  let counter;
  console.log("let counter retournt hier ", ++counter); //counter NaN

  console.log("en hier is de waarde van let init: ", init); //init is not defined
  let init = 10;

}
```

Nochtans kan het gebruik van functiescope wel nuttig zijn. Een functiescope laat immers toe afgebakende scopes te bouwen binnen zijn accolades. Dit wordt handig gebruikt door closures en zelf-uitvoerende functies. Deze "speciale functie's" hebben als bedoeling zo weinig mogelijk de global scope van een applicatie te vervuilen. Variabelen horen zoveel mogelijk thuis binnen de functies. Collision met andere gelijknamige variabelen op global niveau wordt zo verhinderd. Dit laatste kan natuurlijk ook seder ES6 met `let` binnen een functie of `for` lus:

```
var x = 1;

for (let x = 0; x < 5; x++) {
  console.log("block scope variabele ", x) //wordt 1 ... 4
}

console.log("functie scope variabele ", x) //blijft 1
```

Sommige variabelen binnen een functie moeten wel aanspreekbaar blijven van buitenuit zonder daarom globaal te zijn op applicatie niveau. Andere variabelen blijven gewoon encapsulated (local), en dit kunnen ook functies zijn. Deze lokale variabelen worden verwijderd als de functie afgewerkt is. Globale variabelen blijven bestaan tot je de pagina of applicatie sluit.

7.3 Closures

Closures zijn functies die naast hun eigen variabelen ook nog variabelen ontvangen van een omvattende functie. Plaatsen we een inner functie in een outer functie, dan zorgt de outer functie ervoor dat de status van de variabelen ook bewaard blijft in de inner functie. De closure functie (= de inner functie) bewaart zo de status van zijn variabelen binnen zijn eigen functie scope en heeft toegang tot alle variabelen van zijn omvattende functie.

Een voorbeeld:

```
const processData = function (x) {
  let init = 2, key = 10;
  //closure:
  function secretCalc(y) {
    console.log(x + 2 * y + (++init));
  }
  secretCalc(key);
};
processData(2); // 25 met enkel x als argument
processData(2);
```

De functie `inner()` wordt een closure genoemd. Dit wordt gebruikt om geen enkele variabele een globale scope te moeten geven, waardoor meer geheugen werkruimte beschikbaar blijft. In bovenstaand voorbeeld is de variabele `y` onbeschikbaar buiten `processData`. Deze variabele(n) bruikbaar maken van buitenaf is nu juist de bedoeling van een closure. Er wordt een `return` van de closure functie toegevoegd om dit te realiseren:

```
const processData = function (x) {
  let init = 2;
  //closure:
  function secretCalc(y) {
    console.log(x + 2 * y + (++init));
  }
  return secretCalc; // GEEN ARG, want dit zorgt voor uitvoer.
};

processData(2)(10); //functie invoken = nieuwe scope maken met 10 voor y
```

Anders geschreven:

```
const doeIets = processData(2) ;
console.log(doeIets); //returnt "Function" == secretCalc
doeIets(10); //argument y nu bereikbaar van buitenuit => 25
doeIets(10); //wordt 26 (scope werd immers bewaard)
```

Wat testen toont nu wel dat `processdata(2)(10)` telkens een nieuwe scope aanmaakt en zo altijd 25 teruggeeft.

De laatste schrijfwijze waarbij we `processData` niet meer oproepen, zorgt dat de scope (de waarden van de variabelen) behouden blijft. De garbage collector ruimt de waarden niet op (= sluit de closure niet af) dankzij de aanwezigheid van de `return`.

Een "echte" closure kunnen we nu nog beter definiëren:

Een closure is een functie, die ingesloten is in een outer scope; Hierdoor heeft hij toegang tot alle private variabelen van de outer functie. De closure bewaart zijn eigen scope en laat toe die te bewerken van buiten zijn bovenliggende outer scope dankzij een toegevoegde return. Een closure behoudt de status van zijn variabelen en zijn functie, ook als is deze functie al gereturd.

Dit vindt zijn toepassingen in een teller, het bijhouden van game scores of het aantal levens voor elke gamer (geen singleton dus) enz...

Noot: Hoe je de closure beschikbaar stelt aan de buitenwereld, kan op verschillende manieren. In het voorbeeld zorgt een return hiervoor. Dit is de meest gebruikte methode. Evengoed kan een globale variabele zorgen voor de beschikbaarheid.

```
let fn; //gn const gebruiken, const vraagt een initialisatie

const processData4 = function (x) {
  let init = 2;
  function secretCalc(y) {
    console.log(x + 2 * y + (++init));
  }
  fn = secretCalc; // functie als globale var intialiseren
};
```


Na (!) het runnen van processData4(2) is "fn" niet langer undefined an kan "fn" uitgevoerd worden: fn(10).

7.4 Zelfuitvoerende functies (IIFE)

IIFE staat "Immediate Invoked Function Expression" ook vaak uitgesproken als "iffy".

Een functie kan zichzelf oproepen en uitvoeren. Dit doen we door onze functie tussen ronde haken te plaatsen gevolgd door ():

```
(function (args) { . . . })() //ES5
((args) => { . . . })() //ES6
```

Door de anonieme zelfuitvoering maakt deze functie zijn eigen scope aan. Plaatsen we de IIFE in een variabele en combineren we dit met closure vorming dan behoudt een IIFE zijn scope of: de variabelen van een IIFE behouden hun waarden. Dit brengt ons heel dicht bij een object (of een module) die zijn scope bewaart.

Om deze zelfuitvoerende functie gemakkelijk op te roepen wordt ze in een variabele geplaatst. ES6 laat toe het iets korter te schrijven.

```
const processData5 = ( (x, y) => {
  //declaraties en initialisaties
  let init = 2
  const secretCalc= (x, y) => console.log("IIFE result2: ", x + 2 * y +
(++init)) ;
  //publiek maken met return
  return {
    secretCalc: secretCalc } ;
    // secretCalc ; // verkorte notatie in ES6 indien dezelfde naam
  })();
```

secretCalc wordt nu (als variabele met een functie) beschikbaar via processData5:
processData5.secretCalc(2, 10);

Het kan nog korter wanneer we gebruik maken van een javascript object als return object. In ES6 hoeft je de eigenschapsnaam van een javascript object niet te vermelden als die dezelfde is als de naam van teruggegeven functie.

```
const processData7 = ((x, y) => {
  let init = 2;
  return {
    secretCalc(x, y) { console.log("IIFE result3: ", x + 2 * y + (++init)) }
    //zelfde als: secretCalc : secretCalc(x,y) { }
  };
})();
```

Closures en zelf uitvoerende functies worden heel veel gebruikt binnen node.js. Reden hiervoor is te zoeken in de doorgave van de callback functie als argument. Deze callback moet als closure functie zijn scope onthouden en fungeert zo net als de eerder vermelde inner functie. De callbackfunctie behoudt hierdoor de status van zijn variabelen.

7.5 Argumenten bij closures en zelfuitvoerende functies

Zowel bij closures als bij zelfuitvoerende functies kan het nodig zijn om parameters of functies publiek te maken en andere privaat te houden. De parameters meegegeven aan een IIFE kunnen objecten zijn. Ook voor de return kiest men meestal voor een javascript object. Hierbij een voorbeeld van parameter doorgave bij een zelfuitvoerende functie in een namespace "Game". Het is de bedoeling de score bij te houden via een IIFE. Bemerkt hoe meerdere variabelen kunnen geretund worden door deze return als een javascript object terug te geven.

```
const Game = {}; //literal object
Game.player = "Johan"; //hier

const score = (function (Game) {
  let counter = 0;
  //var player = Game.player;

  let inner = function (bonus) {
    bonus = bonus === undefined? 0 : bonus
    return ({
      player: Game.player,
      points: ++counter + bonus
    })
  }
  return inner;
})(Game); //entry point voor het argument

//Bij elke score() worden de "points" verhoogd (eventueel met extra bonus).
console.log(score());
console.log(score());
let myGame = score(2);
console.log(`De punten van $ { myGame.player } : ${ myGame.points }`);
```

Oefening 6.3.5: asynchrone for lus

Wat is het resultaat van de volgende for lus. Besef dat javascript geen scope aanmaakt voor een block gezien javascript een scope aanmaakt binnen zijn functies.

```
for (var i = 0; i < 5; i++) {
  setTimeout( () => {
    console.log(i);
  }, 20);
}
```

1. Verklaar het resultaat, dat je op het eerste zicht niet verwacht.
2. Pas de code aan en zorg voor een resultaat 0 1 2 3 4. Blijf wel asynchroon werken (= met setTimeout()).
TIP: een aantal oplossingen zijn mogelijk. Kan je er twee aanmaken? Ofwel gebruik je een IIFE om de scope te bewaren ofwel gebruik je blockscoping van ES6.

7.6 Module patroon

Het is maar een kleine stap om van IIFE's over te gaan naar het module patroon. Een node applicatie wordt immers opgebouwd met verschillende modules om het overzicht en "separation of concerns" te bewaren. Zowel bestaande modules als eigen gemaakte modules maken gebruik van het module patroon).

Het module patroon wordt vooral gebruikt om een interne status te verbergen (= gelijkaardig aan **information hiding** in OOP) en toch een interface naar de buitenwereld aan te bieden. Het algemeen patroon van een module is meestal een reeks hidden variabelen, die gebruikt worden door toegankelijke methodes. Deze toegankelijke methodes worden in het module patroon beschikbaar gemaakt door "**de return**". We herkennen daarin de closure vorming.

1.1.1.1 Module patroon in ES5

Het typische module patroon ziet er in ES5 als volgt uit:

```
var MyModule;

MyModule = function () {
    //1. Private variabelen
    var something = "Hier is";
    var another = [1, 2, 3];
    var startTime = startTime? startTime: new Date().getTime();

    //2. Inner functies met een scope in MyModule
    function doSomething(x) { console.log(something + " " + x); }
    function doAnother() { console.log(another.join(" ! ")); }
    function duration() {
        return (new Date().getTime() - startTime);
    }

    return {
        //3. Publieke elementen via een javascript object { }
        doSomething: doSomething,
        doAnother: doAnother,
        duration: duration
    };
};

var myModule = MyModule(); //nieuwe scope bij iedere oproep
myModule.doSomething("iets."); //Hier is iets.
console.log("De doorlooptijd bedraagt", myModule.duration());
```

Pas bij het oproepen van de module instantie worden de scopes aangemaakt. Door telkens opnieuw oproepen van de module kunnen **meerdere instanties** aangemaakt worden elk met hun **eigen scope**.

Wenst men echter een singleton dan kan de module opgebouwd worden als een zelf uitvoerende functie. Door de éénmalige zelf oproep wordt de scope bewaard. Omdat het om een singleton gaat, start men meestal (geen verplichting) de modulenaam met een kleine letter

```
var myModule = (function () { return { } })();
```

1.1.1.2 Module patroon in ES6

In ES6 wordt de schrijfwijze voor modules weinig veranderd. Bovenstaand module patroon als singleton:

```
"use strict"
```

```
let myModule = (function () {
  //1. Private variabelen
  const something = "Hier is";
  const another = [1, 2, 3];
  let startTime
  startTime = startTime ? startTime : new Date().getTime();

  //2. Inner functies met een scope in MyModule
  const doSomething = (x) => { console.log(something + " " + x); }
  //const doAnother = () => { console.log(another.join(" ! ")); }
  const getDuration = (cb) => {
    setTimeout(() => cb(null, new Date().getTime() - startTime),1000);
  }
  return {
    //3. Publieke elementen via een javascript object met mogelijks korte notaties
    doSomething,
    doAnother() { console.log(another.join(" ! ")); },
    duration: getDuration
  };
})();

//var foo = MyModule(); //onnodig -> singleton
myModule.doSomething("iets."); //Hier is iets.
myModule.duration((error, result) => { console.log(`De doorlooptijd bedraagt
${result}`); })
```

1.1.1.3 Een modulesysteem: CommonJS

De verwerking en aanmaak van de bovenstaande voorbeelden gebeuren in dezelfde file. Onlogisch, we willen juist de modules in een afzonderlijke file plaatsen voor hergebruik. Node gebruikt hiervoor een bestaand modulesysteem: **CommonJS** (CJS).

Een modulesysteem zorgt er bovendien voor dat elke module binnen zijn **eigen context** runt en zo de global scope niet vervuilt! Modules werken op die manier in hun eigen blackbox. Het CJS systeem steunt op de keywords exports en require, waarbij exporten en ophalen van een module uit een andere file als volgt gebeurt:

CJS voorziet "exports" voor export:

```
//module aangemaakt in file "myModule.js"
let moduleName = (function () { })()
module.exports = moduleName;
```

CJS importeren/oproepen in een andere file gebeurt met "require":

```
let myModuleName = require("./myModule.js");
```

ES6 streeft naar een ander modulesysteem (Ecma Script Modulesystem - ESM), dat steunt op de keywords export (zonder s!) en import (ook zonder s). ESM is default nog niet in Node aanwezig is. Voorlopig gebruiken we CJS voor redenen van compatibiliteit. Modulesystemen en de mogelijkheden van CJS komen nog verder aan bod.

Oefening 7.6: Loader volgens het module patroon

Maak van de asynchrone oefening, waarbij users opgehaald worden uit een Array, **een module met de naam "Loader"**. Als resultaat runt de module asynchroon via zijn instructie:

```
Loader.loadArrayAsync(users , usersIds, (err, arr, duration) => {...
```

Maak gebruik van `module.exports = Loader;` om de module beschikbaar te maken in een andere file, waar je ze ophaalt met `const Loader = require("./Loader.js")`. Het puntje bij de relatieve adressering is verplicht!

Kies voor ES6 syntax maar met het commonJS module systeem.

7.7 ES5 Constructor patroon

Naast het module patroon, dat zoekt naar het imiteren van objecten, kan ook het constructor patroon gebruikt worden. Javascript ES5 gebruikt dit constructor patroon omdat het niet beschikt over classes. ES5 benadert de techniek door het gebruik van prototype, aangevuld met een constructor (ctor) functie.

- Object prototype: Aan het **prototype** kunnen methoden en eigenschappen worden toegekend. Deze prototype functionaliteiten worden nadien beschikbaar voor **elk object afgeleid** van dit oorspronkelijke (inheritance). Soms noemt men de eigenschappen en methoden die beschikbaar komen via het prototype de "**Standaard properties en standaard methodes**". De term "standaard" weerspiegelt duidelijk de default aanwezigheid van deze functionaliteiten bij elke afgeleid object. Merk op: standaard properties en methodes zijn *niet enumerable*, zodat ze niet worden opgesomd in een for lus! Wel kunnen ze zondermeer dynamisch uitgebreid of *aangepast worden in runtime*. Iets wat in OOP zeker niet kan. Het Object prototype van objectX vraag je op met `ObjectX.prototype` `//Object{}`
- Function prototype : Dit is het resultaat van de **constructor functie**. Eigenschappen en methodes aangemaakt via de constructor functie zijn typisch voor een afgeleide instantie en zijn *wel enumerable*. Het function prototype van objectX vraag je op met `ObjectX.__proto__` `//function(){}` Vraag je het function prototype op van een instantie objectX. `__proto__` dan krijg je ook `Object()`.
- Prototype chain: Dit definieert het samenspel (de volgorde) waarin het function prototype en het object prototype opgeroepen worden.
 - Bij aanmaken van een object:

Alles van het prototype object wordt beschikbaar voor de ctor functie en voor alle afgeleide objecten. Elk object steunt op het object prototype van zijn super object (overervings principe). Een keten die eindigt bij `Object{}` waar alles van afgeleid wordt.
 - Bij opvragen van een object eigenschap/methode.

Eerst wordt naar de instance eigenschappen gezocht via het function prototype. Bestaat deze niet dat wordt in een fallback bij het object prototype naar deze property gezocht, en verder in de object prototype van zijn superobject.

Bovenstaand modulepatroon omgebouwd naar het constructor patroon ziet er als volgt uit. De module benaming start (best) met een hoofdletter, want de afgeleide instanties van de module starten met een kleine letter.

```
const MyModule = function (name) {
  //private variabelen en methodes:
  const author = "Johan";
  const self = this;
  const double = function (nمبر){ return (nمبر * 2); } ;

  //publieke eigenschappen met initialisatie:
  this.name = name;

  //publieke methoden met handlers in het object prototype
  this.sender = (msg) => { return this.onSender(this, msg) };

  //pseudo class eigenschappen:
  MyModule.subject = "Het constructor patroon";

  //pseudo class methoden:
  MyModule.parse= (myModule) => {
    try {
      return JSON.stringify(myModule);
    } catch (x) {
      throw new TypeError("Parse error ");
    }
  }
}

MyModule.prototype = {
  //1. object properties:
  self: this,

  // pré ES5 notatie
  //startTime: this.startTime? this.startTime: new Date().getTime(),
  // idem met ES5 getter en setter notatie

  _startTime: "", //member
  get startTime() {return this._startTime? this._startTime: new Date().getTime();},
  set startTime(value) { this._startTime = value; },

  // gebruik van getters en setters maakt VALIDATIE mogelijkheden in de setter
  _createdIn: "",
  get createdIn() { return this._createdIn; },
  set createdIn(value) {
    if(isNaN(value) || value > new Date().getFullYear())
    {
      throw "This is not a valid date";
    }
    this.createdIn= value;
  }

  //2. object methods (sync of async) - verkorte functie notatie :
```

```
durationSync: function () { return (new Date().getTime() - this.startTime); },
saySomething: function (msg, cb) {
  if (x === "ERROR") {
    cb("ERROR", null);
  } else {
    cb(null, this.something + " " + x);
  }
},
onSender: function (obj, msg) {
  return (obj.name + " has a message " + msg);
}
}

//uitvoer:-----
var myModule = new MyModule("NodeJS");
myModule.startTime = new Date().getTime();
myModule.saySomething(" This is an async instance method calling. ", function (err,
info) {
  console.log(info);
});
myModule.sender("Welkom", function (err, result) { console.log(result); })
```

Optionele Oefening 7.7: Loader volgens ES5 constructor patroon

De asynchrone oefening, waarbij users opgehaald worden, bouwen we om naar een constructor patroon, zoals ES5 dit doet.

Als resultaat runt de module asynchroon via het command:

```
var loader = new Loader(users, usersIds);
loader.loadArrayAsync(users ,usersIds, (err, arr) => { ..
```

7.8 ES6 class als het nieuwe ctor-patroon.

ES6 maakt een class structuur beschikbaar. Let wel dat in de achtergrond alles nog naar het object prototype en het function prototype vertaald wordt! Je kan het bekijken als een andere syntax op het constructor patroon. Opnieuw "**syntax sugar**" inclusief voordelen van een verkorte ES6 functie notatie.

Onze voorkeur gaat naar classes. Een class Person ziet bijvoorbeeld als volgt uit.

```
//0.imports (require() voor node)
//0.member variables (ook private functies) met module scope (let, const)

//private functie oproepbaar van in de class
const _privateDoTask=(task, cb) =>{
  if (task === "ERROR") {
    cb( new Error(), null)
  } else {
    . . .
    setTimeout(function () { cb(null, taskResult); }, 1000);
  }
}
```

```

class Person {
  //geen private variabelen mogelijk op hoogste niveau (onbestaand)

  //1. Function prototype: vervangt de ES5 constructor functie
  //ook default waarden kunnen worden toegekend
  constructor(firstName, lastName, age=20) {
    //11.private members met const of let

    //12.instance members (publiek voor de class door "this")
    //a. properties met initialisatie
    this.firstName = firstName;
    this.lastName = lastName;
    this.time = new Date();

    //b. events (met handlers in object prototype)
    this.sender = (msg) => { return this.onSender(this, msg) };

    //c. variabelen nodig in het object prototype (get/set)
    this._age = age //niet this.age
  }

  //2. Object prototype:
  //21.getters en setters gebouwd als functies

  get age() { return this._age; }
  set age(value) {
    //validation
    if (value >= 110 || value < 0) {console.log( value +" is not a valid age.")}
    this._age = value;
  }

  //22. methodes (wel nog altijd op het toegankelijke object prototype)
  toString() { return ("Mijn voornaam is " + this.firstName); }
  onSender(evt, myMessage) { console.log("log:", myMessage) }

  //23. static method met het keyword static
  static getSchool() { return "NMCT in HOWEST"; }
}

//3. static (class) properties komen buiten de class beschrijving
Person.school = "Howest"

```

Het ES6 module systeem met *keywoorden* "export" en "import" is default nog niet geïmplementeerd in nodejs. We blijven voor het importeren van objecten aangewezen op CommonJS met "module.exports" en "require".

Noot: Om toch ES6 module systeem te gebruiken kan je wel gebruik maken van een transpiler zoals babel (> npm install babel) met zijn bijhorende presets (> npm install babel-preset-es2015) voor de configuratie file (.babelrc).

Oefening 7.8: Loader in een ES6 class onderbrengen

De asynchrone oefening, waarbij users opgehaald worden, bouwen we om naar een ES6 class.. Enkel de methode "loadArrayAsync()" is hierbij public. Hete ophalen van één enkel element is een private methode.

Als resultaat runt de module asynchroon via het command:

```
var loader = new Loader(users, usersIds);
loader.loadArrayAsync(users, usersIds, (err, arr) => { . . . })
```

7.9 Welk patroon kiezen: module- of ctor-patroon?

"this" keyword bij constructor patroon kan veranderen naargelang de caller:

Bij objecten in het constructor pattern wordt de status van een *interne* eigenschap doorgegeven en bewaard door het keyword "this". De status van this kan veranderen naargelang de caller. Dit kan moeilijker worden wanneer veel met callback functies gewerkt wordt en bvb. de status vóór/na callback moet behouden blijven. ES6 heeft hier wel een aantal oplossingen.

Private methoden in een constructor pattern kunnen onhandig worden en veel code vragen:

De "this" eigenschappen die via de constructor aangeboden worden zijn public. ENKEL In de constructor kunnen private variabelen of private methoden rechtstreeks aangebracht worden. Dit kan zorgen voor complexe constructies om private methodes verborgen te houden. In het prototype zijn de methodes public.

Bij closures wordt de status van interne eigenschappen bewaard door de functional scope en blijven aangebrachte variabelen sowieso niet toegankelijk, tenzij ze returnt worden. Dit is iets veiliger naar het gebruik van private variabelen in een javascript omgeving waar alles gemakkelijk publiek wordt.

Extensies (=extra features) en overerving maken op een module pattern kan complexer zijn.

Een constructor object is eenvoudig om uit te breiden en over te erven dankzij het object prototype en de prototype chain. Het prototype kan gewoon aangevuld worden (in runtime). Uitbreidingen in closures van modules daarentegen kunnen meer code vragen tot het herschrijven of toevoegen van een nieuwe functie.

Performantie verschillen verwaarloosbaar

Naar performantie bestaat er een te verwaarlozen verschil. Een eenvoudige object instantie kan iets sneller zijn dan zijn evenwaardig module patroon. Reden is te zoeken in de langere aanwezigheid van een object in het geheugen. *Het prototype wordt maar éénmalig in het geheugen geplaatst en blijft herbruikbaar.* Een module is dan weer memory efficiënter, omdat de garbage collector er sneller zijn werk doet.

7.10 Extensiemethoden via het prototype

Zowel het module patroon als het constructor patroon (een ES6 class dus) kan gebruikt worden om een bestaand javascript object te voorzien van extra publieke methoden. Men spreekt over "extensie" methoden. In javascript betekent dat het object.prototype uitbreiden.

Om dit te realiseren voeg je gewoon de nieuwe methode of eigenschap toe aan het prototype van het object. Zo kan je een eigen String.encrypt() maken om een string te encrypteren. In volgend voorbeeld wordt een IIFE aangemaakt, die slechts 1 functie resultaat op een asynchrone manier teruggeeft. Deze extensie methode show64() returnt de String in zijn base64 code.

```
String.prototype.show64 = (function () {
```

```
let show = function (cb) {
  console.log("this :", this) //gn arrow functie (this wordt anders {})
  let buffer = Buffer.from(this, 'base64'); //default unicode
  cb(null, buffer.reduce((a, b) => a.toString() + b.toString()));
}
return show;
})();

//run
"Ik ben een extensie op String".show64(function (error, result) {
  console.log(result);
});
```

Oefening 7.10: object String verder aanvullen (extensiemethoden)

Het native String object willen we uitbreiden met een methode "leadingZero()" (die zelfuitvoerend is). Deze methode zorgt ervoor dat bij het aanbieden van een string 1 tot 9, deze omgezet wordt naar 01 tot 09. Er komt een 0 voor elk getal kleiner dan 10.

1. Maak de extensie methode in een afzonderlijke module en importeer ze voor gebruik:
String.prototype.leadingZero = (function () { })(0);
2. Tips:
 - a. Van een tekst zoals P1 kan je met Array.from(eenTekst) een array maken. Dit laat toe om elk karakter van eenTekst op te halen met een for lus,
 - b. Met array.map(function (value, index, array) { }) kan je elk karakter van array aanpassen waar nodig. Zo kan je controleren of het een cijfer 0 tot 9 is. Een alleenstaand cijfer 0 tot 9 wordt teruggegeven met een extra 0.
 - c. De aangepast array terugomzetten naar een string doe je met array.reduce();
3. Test de aangemaakte extensie op onze Loader module, waarbij de userIds P1 tot P9 omgezet worden naar P01 tot P09 met bijvoorbeeld "P01.leadingZero()".

Het resultaat van het laden wordt:

```
ERROR
P01 is loaded
P02 is loaded
P03 is loaded
P04 is loaded
P06 is loaded
P07 is loaded
P08 is loaded
P09 is loaded
P10 is loaded
Doorlooptijd van de Loader module: 1001
```

7.11 Overerving en compositie

Overerving (inheritance) en compositie hebben beide als doel om functionaliteiten (constructor, properties, methodes, events) van een andere module of van een andere class over te nemen. Men spreekt over de **base class**, die gemeenschappelijke functionaliteiten bevat. Deze base class kan vergaand **geabstraheerd** worden, zodat zijn functionaliteiten in veel afgeleiden bruikbaar zijn. Abstractie in de meest uitgewerkte vorm zorgt ervoor dat deze base class uiteindelijk *nooit geïntanceerd* wordt, maar zijn afgeleiden wel.

- compositie: de afgeleide neemt eigenschappen van de base class over of neemt zelfs een volledige instantie van de base class over. De afgeleide "HAS" functionaliteiten van de base class.
- overerving: de afgeleide reageert volledig als de base class. De afgeleide "IS" een base class.

Beide worden toegepast in Node. Voor beide bestaan verschillende patronen, gaande van mixins, over factories tot decorators. Het prototype en overerving ervan (Professor.prototype = Person.prototype) speelt telkens de grootste rol samen met een paar steeds terugkerende methodes:

- methodes op het base object zoals : `baseObject.call(derivedObject, arg1, arg2 ...)`, `baseObject.apply(derivedObject, [argArray])`
- methodes op Object =(het base Javascript object): `Object.getPrototypeOf()`, `Object.assign()`, `Object.create(baseObject.prototype)`

Voor het gebruik ervan verwijzen we naar referenties zoals https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object. In Node worden vaak één van de volgende technieken gebruikt bij modules, constructor functies of classes.

7.11.1 Module pattern: Compositie met Object.assign()

```
let cat =(function() {
  return {
    skill: "lenig en snel"
  }
})();

let man = (function() {
  return {
    car: "sportwagen",
    topspeed: 360
  }
})();

//compositie:
let batman = (function () {
  //compObject bevat de compositie van cat en man
  let compObject = {};
  Object.assign(compObject, cat);
  Object.assign(compObject, man);

  //batman `speak` methode, waarbij "this" beide source objecten kan zijn
  compObject.speak = function () {
    console.log("Batman rijdt in een " + this.car + " en is " + this.skill);
  }
})();
```

```
};

return compObject;
})();
```

7.11.2 Constructor pattern : Overerving van het prototype met Object.create()

De volgende methodiek is vooral toegepast in ES5 applicaties. In het constructor pattern moet/kan zowel de constructorfunctie (**Function.prototype** met enumerable "own" functionaliteiten) en het **Object prototype** (niet enumerable "object" functionaliteiten).

- Opvragen van own properties (this): `Object.getOwnPropertyNames(myInstance)`
- overereven Function.prototype:


```
BaseObject.call(DerivedObject, arg1, arg2 ...), //bepaalde args overnemen
BaseObject.apply(DerivedObject, [argArray])
Object.assign(DerivedObject, BaseObject) //alles van ctor overnemen
DerivedObject.prototype = new BaseObject()
BaseObject.prototype.isPrototypeOf(new DerivedObject()) //true
Object.prototype.isPrototypeOf(new BaseObject()) //true prototype chain
```
- overerven Object.prototype:


```
DerivedObject.prototype = Object.create(BaseObject.prototype).
```

7.11.3 ES6 class overereving.

Ook hier zorgt ES6 voor syntax sugar, die wel zeer gewaardeerd wordt door OO developers.

```
class DerivedClass extends Baseclass { }
```

Vergeet wel het keyword `super()` niet in de constructor functie. Het is verplicht!

```
class Student extends Person {

  constructor(firstName, lastName, age , schoolyear) { {
    super(firstName,lastName, age) //MUST op eerste lijn
    this.age = 24 ; // base property overschrijven (overriding)
    this.schoolyear =new Date(); //Student instance property
  }

  toString() {
    super.toString(); // een base method kiezen en overschrijven
    // super() als alternatief: indien dezelfde functienaam
    return `De voornaam van deze student is ${this.firstName}` );
  }

}
```

7.12 JSON als transport middel

Vooral JSON wordt gebruikt (ipv XML) bij transport van data binnen een node applicatie. Verwar een javascript literal object (`var obj = { }`) niet met een JSON object . Een JSON object is onderworpen aan **specifieke voorwaarden**. Een korte herhaling van JSON en zijn voorwaarden met enkele aandachtspunten:

Een generisch JSON object ziet er als volgt uit, waarbij de key als string tussen dubbele (!) quotes aangeboden wordt: `{ "key1": value1, "key1": value1, ... "keyN": valueN }`

Maak gebruik van een validator om JSON te evalueren: <http://jsonlint.com/>

JSON ondersteunt volgende data types als values:

- **Number:**
Enkel base-10 getallen worden geaccepteerd. Een expliciete conversie van hex of octale getallen moet buiten de notatie gebeuren.
`{ "hexgetal": 0xFF } //invalid -> maar return geen error`
- **String:**
Er wordt verwacht dat strings tussen double quotes komen.
`{ 'string': 'iets' } //invalid`
Alle string karakters zijn valid, maar sommige verwachten wel een escape (met backslash), want het blijft javascript. Voorbeeld: single quote ' , double quote " , backslash \ , alle speciale chars zoals \t \n
- **Boolean:**
Enkel true en false (kleine letters) worden geaccepteerd.
`{ "boolean": 1 } //is geen Boolean`
- **Array:**
Wordt weergegeven met vierkante haakjes (niet `new Array()`) en kunnen genest worden. Associatieve arrays resulteren bij `stringify` in een lege array. Indexed array worden wel gestringified. Je kan daarom best gebruik maken van `array.push()` om deze aan te maken.
`{ "arr1": [100, true, ["string1", "string2"]] }`
Er wordt binnen node veel gebruik gemaakt van zo genoemde JSON arrays. Hierbij verwijst met naar Array's die JSON objecten bevatten. Ze kunnen bvb.gebruikt worden bij sockets om validatie errors van server naar cliënt te sturen.

```
var arrErrors = [
  { "field": "userId", "errMsg": "userId is verplicht" },
  { "field": "msg", "errMsg": "Minstens 10 karakters invullen" }
];
```


De cliënt kan de ontvangen datastring verwerken met de `JSON.parse()` methode:
`JSON.parse(arrErrors) returnt {Array}`
`JSON.parse(arrErrors[0]).errMsg returnt JSON eigenschap`
- **Object :**
Kunnen net zoals arrays genest worden. Ook JSON objects kunnen genest worden:
`var person = { "person1": { "adres": { "stad": "Kortrijk" } } }`
- **null:**
JSON ondersteunt niet undefined, maar wel null (net zoals een lege string, lege Array, ..)

Voor niet gesupporteerde datatypes zoals Date, RegExp, Math ...moet een conversie gebeuren naar één van bovenstaande gesupporteerde datatypes. Vaak wordt gewoon `toString()` toegepast.

Het Date object bevat echter wel de method `toJSON`: `new Date().toJSON()`-die je kunt gebruiken om een universeel interpreteerbare string met een datum te genereren onder de vorm van 2015-10-01T12:22:45.547Z

Je kan **geen commentaar** zomaar toevoegen in een JSON file. Dit kan niet met `//` of `/* */` ook al is het javascript. Wil je toch commentaar toevoegen, maak dan bijvoorbeeld een comment data element aan:

```
{
  "comment": "testfile.js met hier commentaar",
  "user": {  },
  "user": {  }, ...
}
```

Een JSON file heeft geen length eigenschap zoals Array! Itereren over een JSON file doe je met `for(key in data) {}`, `for(key of data) {}` of met een `foreach(key in data)`

`JSON.stringify(obj [, replacer [, space]]` laat toe een javascript object te serializeren:

```
//javascript object: {ID: 100, Name: 'Johan', City : "Brugge"}
var person = { ID: 100 , Name : "Johan", City : 'Brugge' }
//
//JSON object: {"ID": 100, "Name": "Johan", "City" : "Brugge"}
var personStringified = JSON.stringify(person);
```

De optionele argumenten `replacer` en `space` bieden extra mogelijkheden aan.

Met de replacer kan het stringification proces beïnvloed of gefilterd worden. Zo kan je bvb. enkel strings stringifyen. De `replacer` zelf is een functie met twee argumenten : een key en een value (`JSON.stringify(myObj, myFilter)`). Als eerste key wordt het object zelf aangebracht en daarna al zijn properties.

Met het space argument kan white space geformateerd worden. Een getal duidt op het aantal gebruikte lege spaties, een string op de te gebruiken string als spatie. (meer info op MDN)

`JSON.parse(string [, reviver]-)` maakt een javascript object van een JSON string.

```
JSON.parse(personStringified)
JSON.parse('{"ID":100, "City":"Brugge"}') //string=> enkele quotes
```

`JSON.parse` wordt gebruikt als alternatief op het vroegere `eval()`. De `eval()` instructie is een slecht performerende en bovendien onveilig, omdat alle mogelijke betekenissen geëvalueerd worden. **JSON.parse** evalueert enkel valid JSON "strings". Het is **een synchrone methode** en wordt daarom ook best voorzien van een `try/catch/finally` om mogelijke fouten op te vangen. Het `reviver` argument is op zijn beurt een functie met twee argumenten (key/value). Tijdens het parsen kan de key waarde (= een eigenschap) verwerkt worden naar een nieuwe waarde. Vooral een `reviver` is interessant wanneer een API onbruikbare karakters aflevert en je wil ze verwijderen of vervangen tijdens et opvragen.

```
var result;
```

```
try {
    //JSON === string=> enkele quotes
    result = JSON.parse( '{"ID":100, "City":"Brugge"}' , reviver)
}
catch (error) {
    console.log("invalid json", error) ;
}
finally {
    console.log("done:", result) ;
}

function reviver(key, value) {
    if (key === "City") {
        return "Kortrijk";
    } else {
        return value; //niet vergeten
    }
}
```

De reviver wordt in praktijk gebruikt om foutief ontvangen API waarden aan te passen.

Optionele oefening 7.11

Maak van de users array manueel een JSON array (met een fout op ID6):

```
var usersJSONArray = [{ "ID" : "P1" } , { "ID" : "P2" } , { "ID" : "P3" } , { "ID" :  
"P4" } , { "ID" : "ERROR" } , { "ID" : "D6" }];
```

Pas de oefening aan om nu via parsing de IDs op te halen. De fout bij ID6 wordt in runtime (met een reviver) omgezet naar P6.