



NEW MEDIA &
COMMUNICATION
TECHNOLOGY

BACKEND DEVELOPMENT



Contents

1 Het web (en javascript) wordt real-time	5
2 Introductie en kenmerken.....	6
3 Installatie, testen en debuggen van node.js	9
3.1 Installatie:	9
3.2 Command Line Interface (CLI)	10
3.3 File(*.js) execution:.....	11
3.4 Het global process object	12
3.4.1 global.....	12
3.4.2 process voorziet readable/writable datastreams.....	12
3.4.3 console.....	13
3.4.4 timers	13
3.5 Debuggen in node.js	13
4 Alternatieve installatie: IISNode.....	17
4.1 IISNode als handler.....	17
4.2 IISNode installeren	17
4.3 Configuratie van de IIS node server	18
5 Node.js in Visual Studio met NTVS.....	19
6 Asynchroon programmeren met de event loop	22
6.1 Asynchroon (=event-driven) programmeren	22
6.2 De event loop beheert de tasks	24
6.3 Javascript functies maken hun functiescope aan.....	27
6.3.1 Functie scope !== block scope.....	27
6.3.2 Closures	27
6.3.3 Zelfuitvoerende functies.....	29
6.3.4 Doorgeven van argumenten bij closures en zelfuitvoerende functies	29

6.3.5	Module patroon.....	30
6.3.6	Constructor patroon.....	31
6.3.7	Constructor functies of closures?	32
6.3.8	Objecten uitbreiden	33
6.4	Events en eventemitters.....	34
6.5	Javascript en node.....	37
7	Het module systeem	41
7.1	Node Packaged Modules (npm)	41
7.2	Het CommonJS module systeem.....	44
7.3	Modules en het Module pattern	47
8	Process beheer in een asynchroon programmeer model.	49
8.1	Single threaded javascript	49
8.2	Async error handling	51
8.2.1	Try/catch vervangen door domain errors.....	51
8.2.2	Error argument in callback functies.....	52
8.3	Callback Hell = Pyramid of Doom = the Boomerang effect	53
8.3.1	Benoemen van callback functies	54
8.3.2	Distributed events	54
9	Flow control en scheduled processen	57
9.1	Sheduling node processen	57
9.2	Externe processen beheren	59
9.3	De volgorde van callback taken beïnvloeden(flow control)	61
9.3.1	Generisch flow control systeem	61
9.3.2	Async.js.....	63
9.3.3	Het gebruik van promises	65
10	De API quick tour.....	69



11 Files en streams	70
11.1 File system.....	70
11.2 Lezen en schrijven van streaming data	73

1 Het web (en javascript) wordt real-time

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, worden een groot deel van de toepassingen tegen 2017. 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, groeit hierbij tot de taal die naast de cliënt ook zijn toepassingen vindt op de server en IoT devices. Getuigen hiervan zijn het aantal javascript libraries, API's (ook in devices), vernieuwde technologien en een gedreven community met javascript wizards:



Realtime toepassingen betekent real time communicatie tussen cliënt en server. Eén van de protocollen die vandaag aangewend wordt voor deze real time communicatie is http; en dit omdat http oorspronkelijk 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 beter 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 andere 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 event-driven** manier af te werken.

2 Introductie en kenmerken



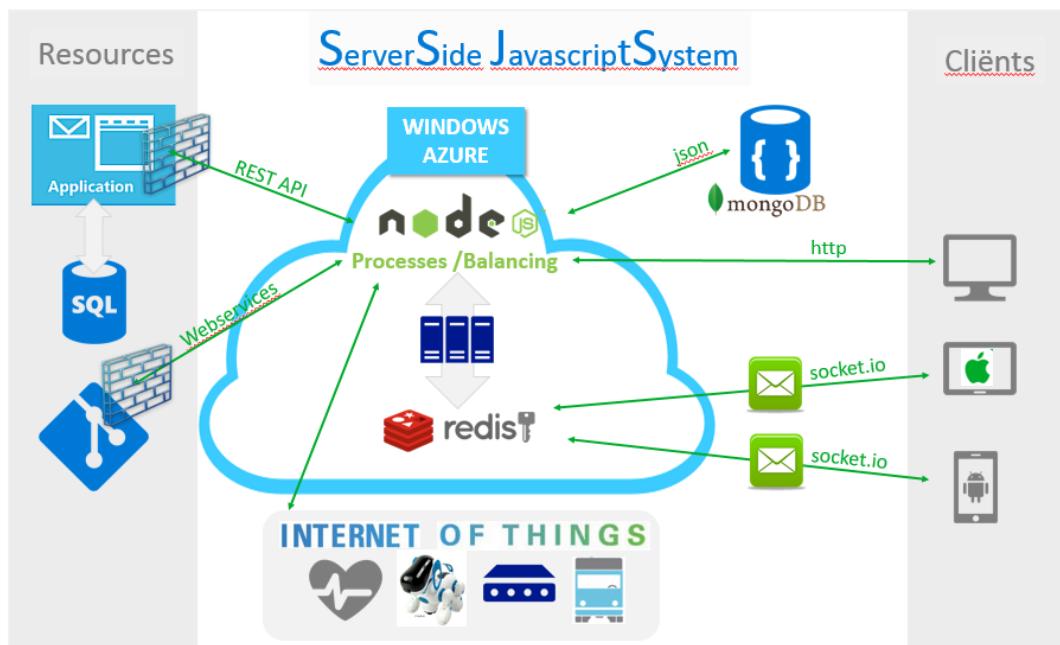
Een opsomming van de belangrijkste Node.js kenmerken:

- Ontstaan in 2009 onder impuls van Ryan Dahl met een presentatie op de Europeese JSConf;
- Een interpreter voor javascript die buiten de browsers draait en daardoor geschikt is voor het ontwikkelen van backend. Javascript werd gekozen als taal voor node.js omdat 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 virtual machine. 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: in mei 2015 voorzag Microsoft met Windows 10 de mogelijkheid om node te runnen op de Chakra javascript engine (<https://github.com/Microsoft/node>)



- 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 en voorziet scherm visualisering (via de console) en I/O handling vanuit een command line tool. Deze I/O handling wordt bijvoorbeeld gebruikt voor request/response processing. Een eventlistener luisters hierbij naar een specifieke poort.
- Werkt vanuit een command line tool (CLI). Via de command line kan je:
 - een http server starten (trager dan tcp server, gebruikt een browser),
 - een tcp server starten (sneller dan http server, moeilijker bij firewalls),
 - zelf aangemaakte modules installeren,
 - gebruik maken van third party modules uit de community.
- Node is open source. Het meest globale object is “process” en vervangt het traditionele “window” top object van javascript aan cliënt kant.
- Beschikt over een non blocking file system, gebaseerd op een asynchroon javascript model, voorzien van getters, setters, JSON, XMLHttpRequest...
- Non blocking betekent dat node.js volledig event-driven is:

- Maakt gebruik van callback functies.
In een niet event driven omgeving wordt een programma liniair verwerkt. Zo wordt een database call afgewerkt, waarna het programma verder gaat. Event driven betekent asynchroon blijven monitoren tot wanneer een taak vervolledigd is en intussen het lopende programma verder afwerken.
- Meerdere requests kunnen simultaan verwerkt worden. En dat met een veel optimaler gebruik van geheugen. Ideaal om bijvoorbeeld een game te ondersteunen met honderd(en) players.
- Applicaties kunnen nu gebruik maken van de zelfde taal bij de cliënt als bij de server (Javascript). Men spreekt soms over SSJS = Server Side Javascript Systems.



Figuur 1: Node.js behandelt meerdere cliënt requests ASYNC op een SINGLE thread.

- Door de sterke op I/O gebied is node.js ideaal voor het maken van **SCALABLE REAL-TIME APPLICATIONS** met **MEERDERE PARALLELLE CLIËNTS** - die geen CPU intensieve taken moeten verwerken:
 - social applications;
 - multiuser games;
 - business collaboration areas;
 - news, weather, or financial update applications;
 - updates ontvangen van social network apps.
- Deze sterke op I/O gebied wordt technisch ondersteund door:
 - sockets voor realtime communicatie,
 - media servers en proxies voor custom netwerk services,
 - JSON webservices,

- cliënt geörieenteerde web UI's,
- 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 ook niet (!) voor het bouwen van zwaar data driven applicaties, die veel relationele reaties verwachten (= 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, Walmart , PayPal, Yammer van Microsoft...
ref: <https://github.com/joyent/node/wiki/projects,-Applications,-and-companies-using-node>.
- Referenties nodig:
 - <http://radar.oreilly.com/2011/06/node-javascript-success.html>
 - javascript repositories op github: <http://githut.info/>

3 Installatie, testen en debuggen van node.js

3.1 Installatie:

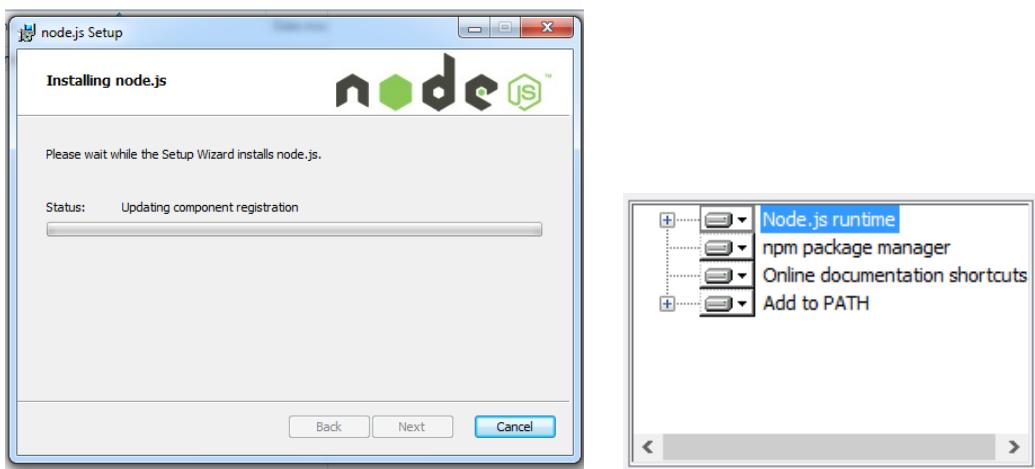
Raadpleeg de officiële site voor installatie (<http://nodejs.org/download/>). Zowel een windows als mac installer package zijn rechtstreeks beschikbaar op de site van node.js



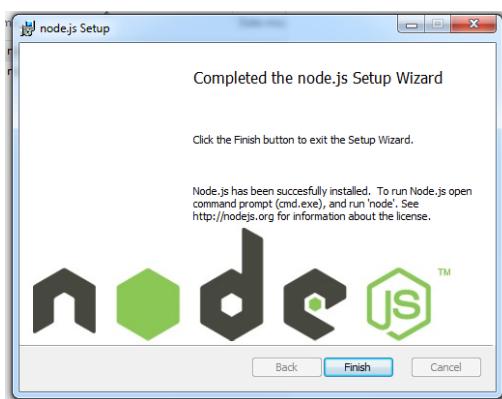
Download en installeer node.js

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.



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.

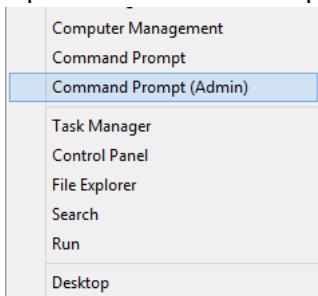


Noot: Open source programma's kunnen het moeilijk maken om downwards compatible te blijven.

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 -g n". (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)..
C:\Program Files\nodeis
3. Activeer met het command "node" de node applicatie. 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

4. 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)}
>add(23,10)
>process // toont lopende process
```

>.help opent een basic help, let op het punt vóór de instructie
>TAB-key toont de variabelen

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

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.

Noot: Het CLI tool zelf is eveneens geschreven in javascript. Dit betekent dat het ook op andere platformen (bvb. windows 8, windows server 2012 ...) kan gerund worden.

3.3 File(*.js) execution:

Externe javascript files (*.js) kunnen buiten de REPL console runnen met het “**node**” command. Je gaat van command line 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.

Oefening

“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 uitgevoerde javascript file. Je herkent hier de conventies van C, C++ en vele scripttalen in.

Je kan extra argumenten meegeven via de CLI door ze achteraan toe te voegen aan het command.

bvb.: “node HelloWorld.js Mister” zorgt dat process.argv[2] de waarde “Mister” bevat.

Opgave: Raadpleeg de documentatie en test dit uit.

Toon ook eens alle argumenten van argv in de console met een for of forEach lus.

Deze eenvoudige techniek met argv wordt vaak gebruikt om vanuit de console een klein menu aan te bieden, dat je informeert hoe een taak (verzameling van taken) op te roepen. Hierbij wordt soms , hoewel onnodig , gebruik gemaakt van externe modules zoals require(“minimist”) die extra mogelijkheden aanbiedt om een console argument aan te brengen.

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

3.4 Het global process object

Node beschikt over een aantal globals:

3.4.1 global

"global" is de global overkoepelende namespace van node.

3.4.2 process voorziet readable/writable datastreams

"process" voorziet interactie op het hoogste niveau als wrapper rond een uitvoerend process.(te vergelijken met window object).

Naast de twee belangrijke events voor proces beheer (on('exit') en on('uncaughtException')) voorziet het process object 3 data streams voor het behandelen van input, output en hun 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. Stdin kan ook uit deze pauze toestand gehaald worden met `process.stdin.resume();`
Dit kan toelaten de input van de console op te vragen:
`console.log("Wat is je naam?");
process.stdin.resume();`
Via een callback kunnen met stdout de ingevoerde gegevens opgehaald worden.
`process.stdin.on("data", function (data) { //stdout gebruiken });`
- **process.stdout** is een "writable stream" en voorziet output mogelijkheden voor een programma via de write methode:
`process.stdout.write(data, [encoding], [callback])`
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.HOME...`

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

3.4.3 console

“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. Dit komt verder nog uitgebreid aan bod.

3.5 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 / node (niet gebruiksvriendelijk):

Met het “debug” command in de console (“`node debug Hello.js`” “`node --debug Hello.js`”) kan een command line debugger gestart worden voor een javascript file. De console toont de debug mode aan met een “`debug>` prompt. Het debuggen gebeurt op poort 5858.

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.

(<http://docs.strongloop.com/display/DOC/Debugging+with+Node+Inspector>)

a. Installeer de module inspector globaal met het command:

`npm install -g node-inspector`

De nodige files voor installatie worden online opgehaald en als resultaat wordt de boomstructuur van de geïnstalleerde module getoond.

- b. Start inspector als command met "node-inspector":
Op poort 8080 start een debugger.

```
Node Inspector v0.7.4
Visit http://127.0.0.1:8080/debug?port=5858 to start debugging.
```

- c. In een tweede command window dient een node programma te worden gestart met een break instructie (= break op de eerste lijn).

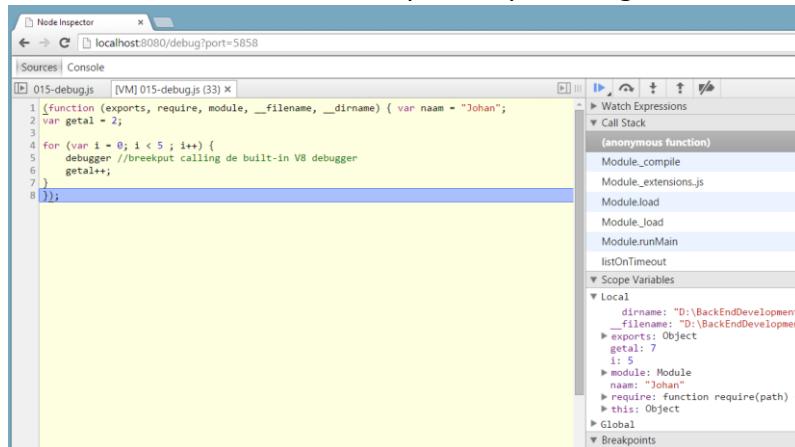
"node --debug-brk HelloWorld.js"

```
C:\Program Files (<x86>)\nodejs>node --debug-brk 02-HttpServerExample.js
debugger listening on port 5858
```

- d. Surf in de Chrome(!)browser naar de applicatie link (en zijn poort indien een http server toepassing) of voor de console app naar de link :

<http://localhost:8080/debug?port=5858>.

Vanaf nu kom je terecht in de gekende browser development tools van Chrome. Dit werkt enkel in Chrome omdat inspector op webkit gebaseerd is.



Om deze tools te verkennen kan je terecht op <http://discover-devtools.codeschool.com/>

- 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/):	Onbeperkte evaluatie periode – voor MAC en Windows.
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

<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 strom voorziet een plugin
<u>Visual Studio NTVS</u>	Volwaardige en rijke IDE als addon op Visual studio.

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 ontwikkelijd.

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/>).

c. JSLint en JSHint:

Voor het oplossen van bugs kan je niet alleen beroep doen op google, bing, Visual Studio help of de node community, maar ook op JSLint

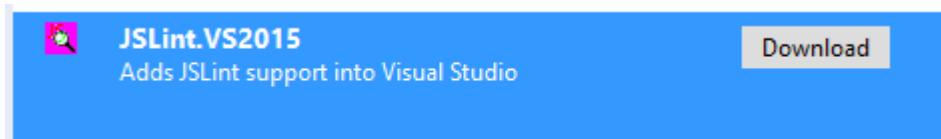
Ter herinnering: JSLint is een code kwaliteitstool voor javascript dat kijkt naar syntax fouten, stijl conventies en structurele problemen (zoals gedefinieerd in <http://javascript.crockford.com/code.html>). Je hoeft niet alle opmerkingen te volgen, soms is bvb. verantwoord om een variabele lager in de code te plaatsen of om == te gebruiken in plaats van ===

Je kan zowel online of offline linten

- i. Online: testen op <http://www.jslint.com/> //vink nodejs aan



Visual Studio voorziet een extensie met documentatie in [GitHub wiki](#)



- ii. Offline:

```
npm install -g jshint//of esHint = community variant
jshint helloworld.js
```

jsHint is afgeleid van jsLint maar biedt meer configuratie opties waardoor je meer flexibiliteit hebt over de – iets minder strenge – fout rapportering.

4 Alternatieve installatie: IISNode

4.1 IISNode als handler

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.

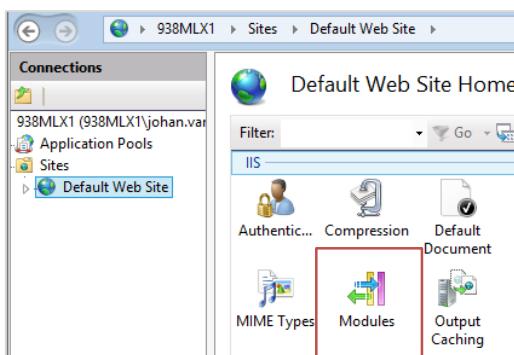
4.2 IISNode installeren

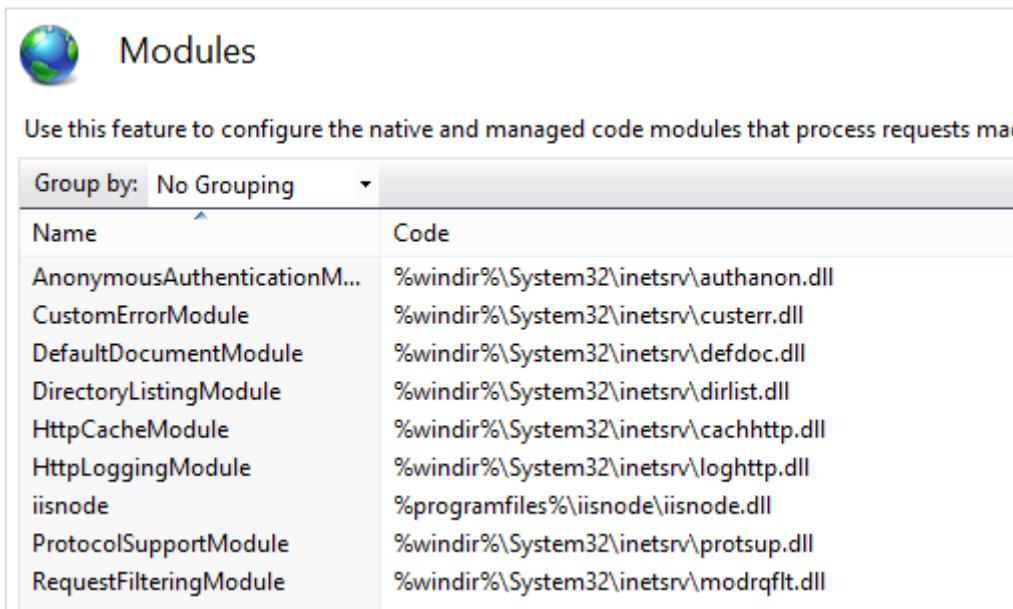
Het installeren van IISNode verloopt minder vlot dan het installeren van node.js. Een beschrijving van de installatie vind je terug op:

<http://www.hanselman.com/blog/InstallingAndRunningNodejsApplicationsWithinIISOnWindowsAreYouMad.aspx>

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

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





The screenshot shows the 'Modules' section of the IIS Manager. A table lists various modules with their corresponding code paths:

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

4.3 Configuratie van de IIS node server

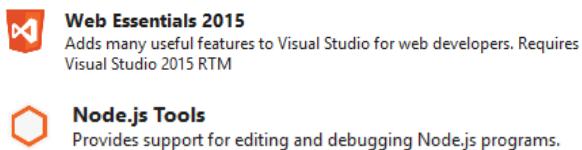
Wanneer de http server runt via iisnode worden poortnummer en domein bepaald vanuit IIS. De node server neemt deze waarden over via zijn process environment: process.env

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, world! [helloworld sample]');
}).listen(process.env.PORT);
```

5 Node.js in Visual Studio met NTVS

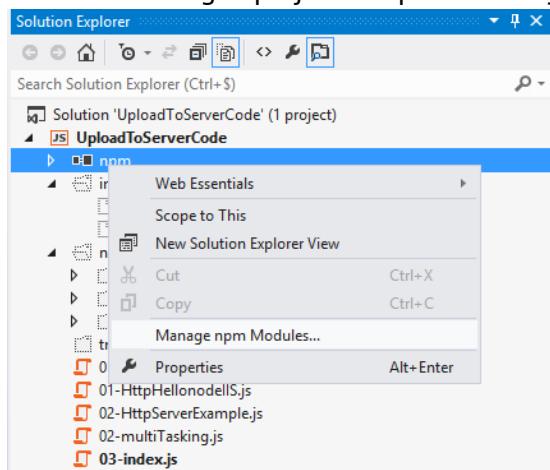
Node.js ontwikkelen kan complex worden, zeker als je er een volledige website mee wil bouwen. Dan komt de vraag naar ontwikkeltools. Combineren met IISNode als handler was een eerste stap naar meer overzichtelijkheid, maar in **nov 2013** zorgde Microsoft voor **NTVS: Node Tools for Visual Studio**. Debuggen kan zowel lokaal als remote en een interactieve REPL console is voorzien.. in combinatie met WebEssentials krijg je een ideaal noded ontwikkel platform.

1. Installeer NTVS en WebEssentials via Menu >> Tools >> Extensions and Updates:

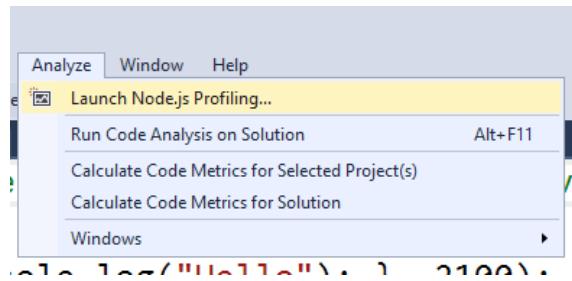


2. Aanmaken van een node project:

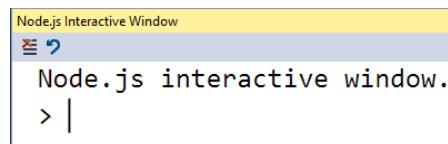
- a. Er kunnen verschillende types van nodejs projecten aangemaakt worden vanuit new project >> javascript:
 - i. een nodejs project, waarbij ofwel een nieuwe nodejs applicatie gemaakt wordt of waarbij een bestaande nodejs app kan geopend worden.
 - ii. een express project
 - iii. een Azure applicatie
- b. Installatie van modules kan nog altijd via het npm command (npm install) of kan via de visuele manager: project >> npm >> Manage npm modules



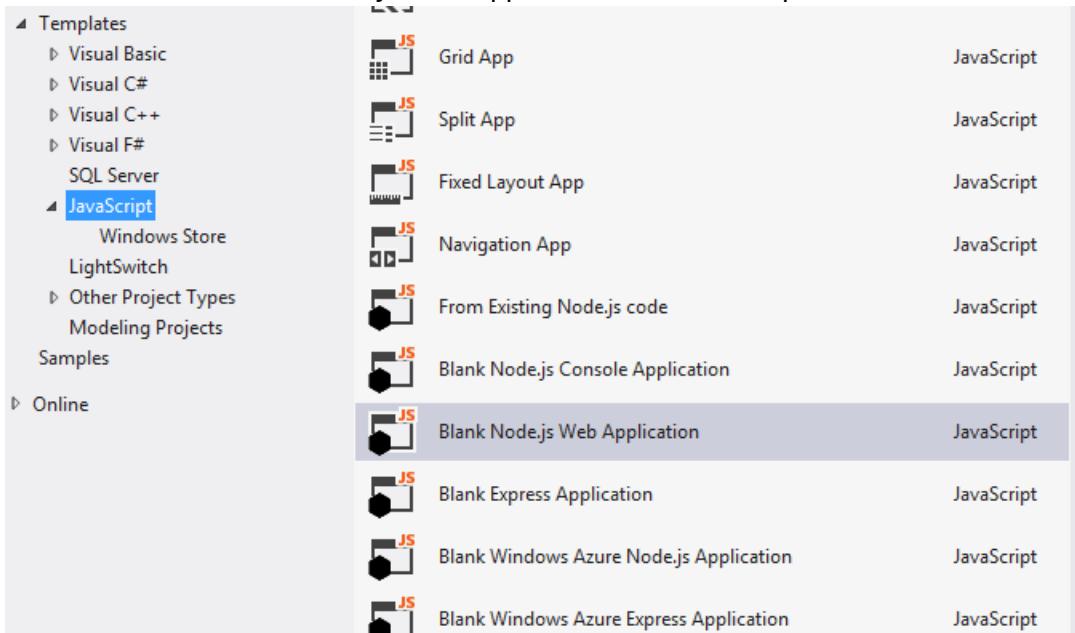
- c. Nodejs profiling kan via het Analyze menu of via een profiler onder het Debug menu.



- d. View >> Other windows >>node js interactive window opent een interactieve console.

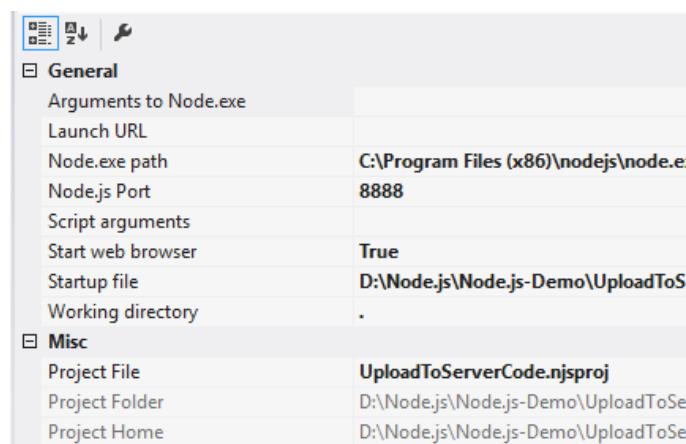


3. Maak een nieuwe "Blank Node.js" Web application aan in een map naar keuze.



4. Test uit door een eerder aan gemaakte nodejs file onder te brengen in het project.

- Kopieer de *.js file.
- Zet deze file via de properties (rechtermuis) als "Set as Node.js startup file".
- 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.

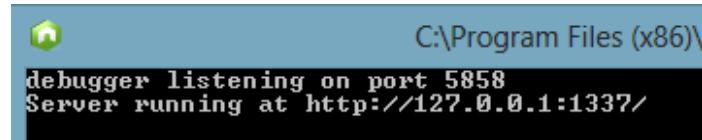


d. Run en debug de toepassing.

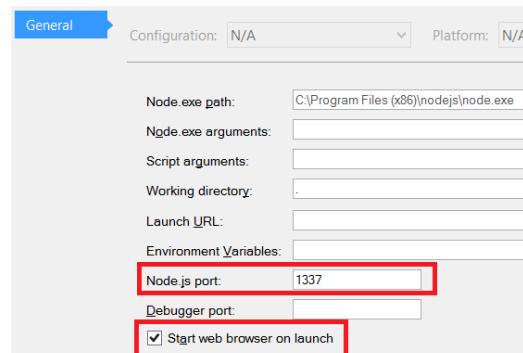
- i. De console opent automatisch maar sluit ook automatisch na voltooide taken. Je kan de console live houden met setTimeout:

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

- ii. Bij een webbrowser toepassing wordt de console en 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.



6 Asynchroon programmeren met de event loop

6.1 Asynchroon (=event-driven) programmeren

Het 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 voltooien van een I/O of op het ontvangen van database gegevens en gedurende deze tijd de CPU vrijgeven aan een andere thread. Dit is een typische blocking/blokkerende methodiek. Een thread wordt simpel weg tijdelijk opgehouden tot zijn actie voltooid is. Voor de programmeur kan het moeilijk worden hierbij controle te houden over de synchronisatie van verschillende processen. Het kan moeilijk worden om te weten welk proces op een specifiek moment op welke thread wordt uitgevoerd.

Javascript (en dus ook Node.js) is **single threaded**. Wat betekent dat javascript maar één ding simultaan kan afhandelen. Wel kan de illusie gewekt worden 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.

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.

```
//async. uitwerking met anonieme callback
task( args, function(args) {
    do_cbtask_with(args);
});

//async. uitwerking met benoemde callback
var task_finished = function(args) {
    do_cbtask_with(args);
}

task( args, task_finished);
```

Een voorbeeld met error-first syntax:

Bij asynchrone werking wacht het hoofdprogramma niet op het resultaat van een I/O, een antwoord van een database query of het resultaat van complexe calculate().

SYNCHRONE werking	ASYNCHRONE werking
<pre>var data = getData(); console.log("Synchroon: " + data);</pre> <p>Een langdurige en synchrone processData() blokkeert het volledige programma (single threaded):</p> <pre>function processData(increment) { if (err) { console.log("An error occurred."); return err; } var data = calculate(); data += increment; return data; } console.log(processData(2));</pre>	<pre>getData(function (data) { console.log("Asynchroon: " + data); })</pre> <p>Een callback functie wordt opgeroepen van zodra processData() beëindigd is. Calculate werkt async waarbij zijn callback functie, na voltooiing, het resultaat doorgeeft als argument.</p> <pre>function processData(increment,callback) { calculate(function (err, data) { if (err) { console.log("An error occurred."); callback(err, null); } data += increment; callback(null, data); }); } processData(2, function (err, returnValue) { //deze code wordt pas async uitgevoerd na het beëindigen van calculate console.log(returnValue); });</pre>

Door het asynchroon programmeren wordt alles meer functie driven, waarbij het aantal argumenten stijgt met de callbackfunctie.

Typisch wordt volgens (een niet geschreven) conventie het laatste opgeroepen functie-argument de callbackfunctie.

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.

Oefening:

We wensen een synchrone functie (load) te herschrijven op een asynchrone manier. De load functie plaatst een willekeurige lijst van userIds in een tweede array. De load functie voor één id duurt één seconde en willen we omzetten van een synchrone werking naar een asynchrone werking.

Bij de synchrone werking zal de duurtijd minstens gelijk zijn aan het aantal userIds * 1 seconde. De vertraging simuleren we met de delay parameter. De synchrone werking ziet er als volgt uit en is herkenbaar aan de returns en while structuren:

```
var delay = 1000;

function loadSync(element, delay) {
    var start = new Date().getTime();
    while (new Date().getTime() - start < delay) {
        //just wait
    }
    return "element " + element + " loaded";
}

//monitoren van synchrone doorlooptijd
function loadArraySynchroon(array, elements) {
    var start = new Date().getTime();
    for (element in elements) {
        array[element] = loadSync(element, delay);
        console.log(array[element]); //informatie wanneer ingeladen
    }
    return (new Date().getTime() - start) + "\n";
}
```

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 door eenzelfde 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.

```
function loadAsync (element, delay , cb) { . . . }

function loadArrayAsync(arrayA , elements, cb) { }
```

Na het inladen van alle elementen op een asynchrone manier wordt cb van loadArrayAsync opgeroepen. Tel de elementen om dit op het juiste ogenblik te doen. Let op: de waarde i van een for lus wordt synchroon bepaald, waardoor de waarde niet gekend is in een asynchrone functie binnen de for lus.

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 argument voorziet (dat wel asynchroon behouden wordt).

6.2 De event loop beheert de tasks

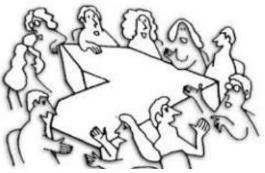
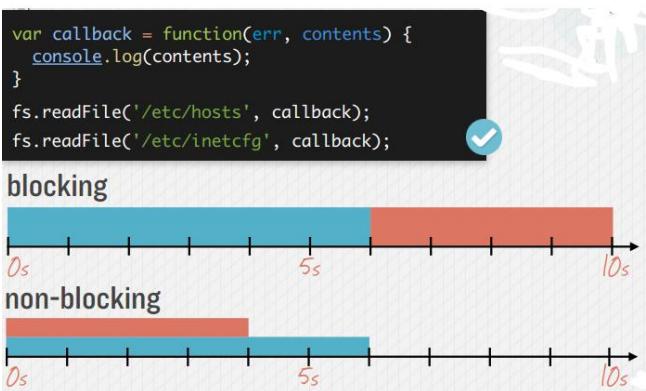
Node.js zorgt dat I/O taken op een asynchrone manier uitgevoerd worden door events.

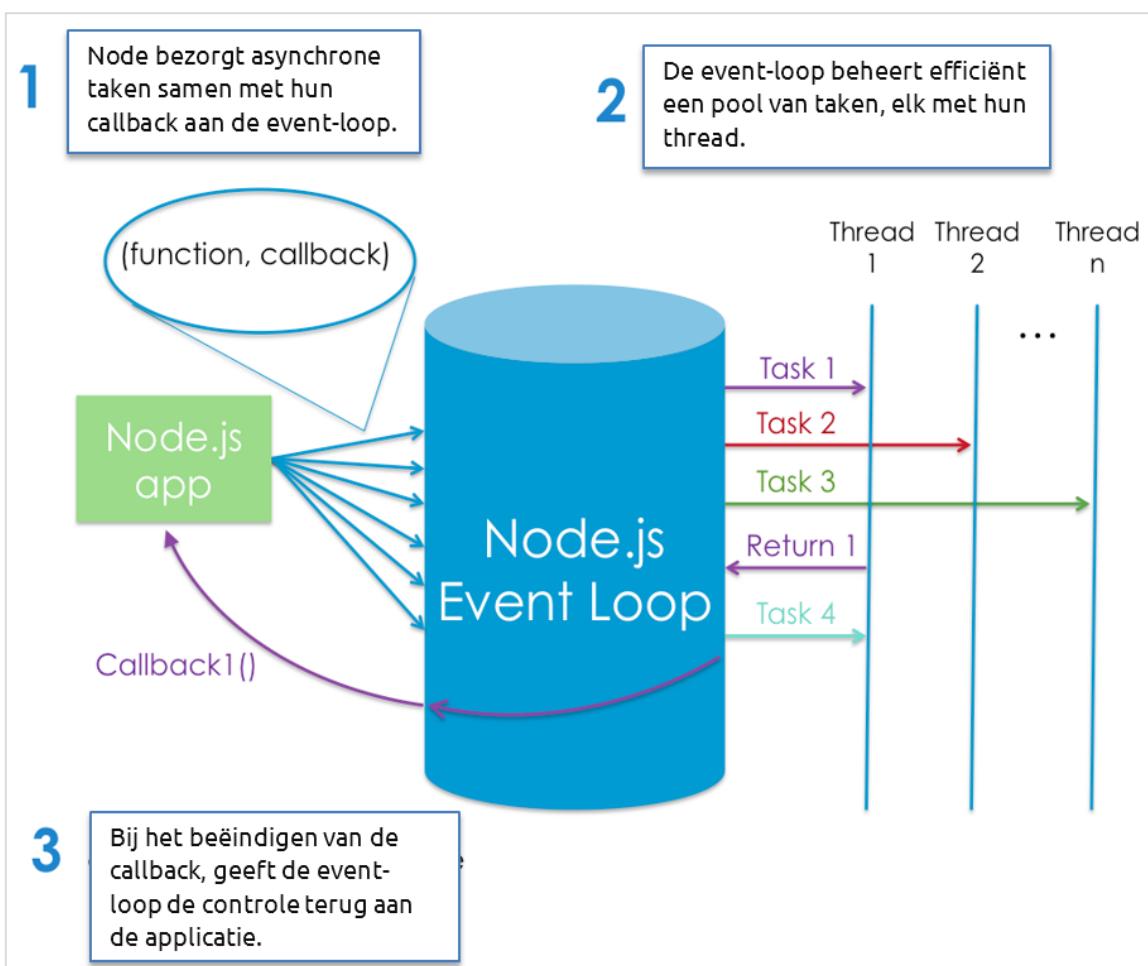
Het asynchroon programmeer model maakt gebruik van callback functies en wordt ondersteund door de event-loop. De event-loop start automatisch op, wanneer je node start en behandelt het volledig beheer van de pool van events en hun callback functies. De event-loop voert simultaan twee zaken uit:

- De event loop *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, efficient 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.

<p>Van synchroon (blocking) naar asynchroon en non-blocking:</p> 	<pre><code>var result = db.query("select.."); // use result</code></pre> <p>wordt asynchroon met een callback functie:</p> <pre><code>db.query("select..", function (result) { // use result });</code></pre>
<p>Tijdswinst in de queue bij non-blocking:</p> 	<pre><code>var callback = function(err, contents) { console.log(contents); } fs.readFile('/etc/hosts', callback); fs.readFile('/etc/inetcfg', callback);</code></pre>  <p>blocking</p> <p>non-blocking</p>



Figuur 2: De Node.js Event-Loop lifecyclus

Time consuming callbacks vertragen output

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 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 Javascript functies maken hun functiescope aan.

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 en controlestructuren.variabelen.

6.3.1 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. Controlestructuren (if, for ...) maken een blockscope aan maar geen functie scope!

```
var checkMyScope = function (a, b) {  
  
    counter = 0;  
    var counter ; //hoisted = declaratie komt eerst (zonder initialisatie)  
    console.log("counter returnt hier ", ++counter);  
  
    console.log("en hier is de waarde van init: ", init);  
    var init=10;  
  
}
```

Kortom: Een functiescope laat 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 wordt zo verhinderd. Sommige variabelen 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.

6.3.2 Closures

Closures zijn functies die naast hun eigen variabelen ook nog variabelen ontvangen van een omvattende functie. Plaatsen we de closure als inner functie in een outer functie, dan zorgt de outer functie ervoor dat de status van de variabelen ook bewaard blijft in de closure. De closure 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:

```
var processData = function (x) {
    var 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 variabelen y onbeschikbaar buiten calculate. Deze variabele(n) bruikbaar maken van buitenaf is nu juist de bedoeling van een closure. Er wordt een return toegevoegd.

```
var processData = function (x) {
    var init = 2;
    //closure:
    function secretCalc(y) {
        console.log(x + 2 * y + (++init));
    }
    return secretCalc; // GEEN ARG
};

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

Anders geschreven:

```
var doeIets = processData(2) ;
console.log(doeIets); //returnt "Function" == secretCalc
doeIets(10); //argument y nu bereikbaar van buitenuit
```

Wat testen toont nu wel dat processData(2)(10) telkens een nieuwe scope aanmaakt; terwijl de laatste schrijfwijze de scope (de waarden van de variabelen) behoudt. De garbage collector ruimt de waarden niet op (= sluit de closure niet af) dank zij 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. Een closure behoudt de status van zijn variabelen en zijn functie ook als is deze functie al gereturnt.

Dit vindt zijn toepassingen in een teller, het bijhouden van game scores of het aantal levens voor elke gebruiker (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.

```
var fn;

var processData = function (x) {
```

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

6.3.3 Zelfuitvoerende functies

Ook IIFE genoemd = Immediate Invoked Function Expression.

Een functie kan zichzelf oproepen en uitvoeren. Door de anonieme zelfuitvoering maakt deze functie zijn eigen scope aan. Anders geformuleerd: zelfuitvoerende functies zorgen voor closure vorming. Een IIFE behoudt zo zijn waarden. Om deze zelfuitvoerende functie gemakkelijk op te roepen worden ze in een variabele geplaatst.

Toegepast op het voorbeeld met wat naamsveranderingen:

```
var score = (function () {
    var counter = 0;
    var inner = function (bonus) {
        bonus = bonus === undefined? 0 : bonus;
        return (++counter + bonus);
    }
    return inner;
})();

console.log(score());
console.log(score());
console.log(score(2));
```

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 fungert zo net als de eerder vermelde inner functie. De callbackfunctie behoudt hierdoor de status van zijn variabelen.

6.3.4 Doorgeven van 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". .

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

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

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

console.log(score());
console.log(score());
var myGame = score(2);
console.log(" De punten van ", myGame.player , ":" , myGame.points);.
```

Oefening:

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.

```
/* volgend voorbeeld geeft een resultaat, dat je op het eerste zicht niet
verwacht. Test uit en verklaar.
TIP: Wil je een resultaat in de vorm van 0 1 2 3 4, dan moet je setTimeout()
ombouwen naar een zelf uitvoerende closure.*/
for (var i = 0; i < 5; i++) {
    setTimeout(function () {
        console.log(i);
    }, 20);
}
```

6.3.5 Module patroon

Het is maar een kleine stap om over te gaan van closures en zelf uitvoerende functies naar het module patroon. Een node applicatie wordt immers opgebouwd met verschillende modules. 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 toegankelijk methodes worden in het module patroon beschikbaar gemaakt door "de return". We herkennen daarin de closure vorming.

Het typische module patroon ziet er 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,
        doAnother,
        duration
    };
};

var foo = MyModule(); //nieuwe scope bij iedere oproep
foo.doSomething("iets."); //Hier is iets.
console.log("De doorlooptijd bedraagt", foo.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 in een variabele gebracht worden als een zelf uitvoerende functie. Door de zelf oproep wordt de scope bewaard.
var foo = (function () { return { } })();
```

Oefening:

De asynchrone oefening, waarbij users opgehaald worden, bouwen we om naar een module met de naam "Loader". Als resultaat runt de module asynchron via het command:
Loader.loadArrayAsync(users , userIds, function (err, arr, duration) { ... }

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

6.3.6 Constructor patroon

Naast het module patroon , dat zoekt naar het maken van objecten, kan ook het constructor patroon gebruikt worden. Javascript beschikt niet over classes maar benadert de techniek door het gebruik van prototype.

Elk object doorloopt eerst zijn prototype om nadien via een constructor functie de beginwaarden te initialiseren.

Bovenstaand modulepatroon omgebouwd naar het constructor patroon ziet er als volgt uit:

```
var myModule;

myModule = function (something) {
    //private variabelen:
    var author = "Johan";
    var self = this;

    //publieke eigenschappen voor initialisatie:
    this.something = something;

    //pseudo obj (class) variabelen:
    myModule.subject = "Het constructor patroon";
}

myModule.prototype = {
```

```
//instance properties:  
self: this,  
  
startTime: this.startTime? this.startTime: new Date().getTime(),  
// idem met ES5 notatie  
/_startTime: new Date().getTime(),  
//get startTime() {return this._startTime? this._startTime:new Date().getTime(); },  
//set startTime(value) { _startTime = value; },  
  
//instance methods (sync of async):  
duration: function () {  
    return (new Date().getTime() - this.startTime);  
},  
doSomething : function doSomething(x , cb) {  
    if (x === "ERROR") {  
        cb("ERROR", null);  
    } else {  
        cb(null, this.something + " " + x);  
    }  
}  
}  
  
//uitvoer:  
-----  
var cObj = new myModule("Hier is");  
cObj.doSomething(" NodeJS", function (err, info) {  
    console.log(info);  
});  
  
console.log("De doorlooptijd bedraagt :" + cObj.duration());
```

Oefening:

De asynchrone oefening, waarbij users opgehaald worden, bouwen we nu om om naar een constructor object. Als resultaat runt de module asynchrone via het command:

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

6.3.7 Constructor functies of closures?

"this" keyword bij constructor pattern 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.

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 maken op een module pattern kan complexer zijn.

Een constructor object eenvoudig om uit te breiden. Het prototype kan gewoon aangevuld worden. Uitbreidingen van closures 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 eenvoudig object kan iets sneller zijn dan zijn evenwaardig module patroon. Reden is te zoeken in de langere aanwezigheid van een object in het geheugen. Een module is memory efficiënter.

6.3.8 Objecten uitbreiden

Zowel het module patroon als het constructor patroon kan gebruikt worden om een bestaand javascript object te voorzien van een extra methode. In javascript betekent dat het object.prototype uitbreiden. Het is daarom ook iets eenvoudiger om een constructor object uit te breiden in vergelijking met een module.

Het native String object uitbreiden met een encryptie methode kan bijvoorbeeld als volgt:

```
String.prototype.encrypt = (function () {  
  
    //Hier komt een private key "secret" , die elk vermeld karakter omzet naar een  
    ander var secret = {  
        'p': '\u0044' ,  
        '1': 'a' ,  
        '3': ':' ,  
        '5': '\u00A5' ,  
        '7': '\u00C6'  
    };  
  
    //De return bevat de encrypterende (replace) functie  
    return function (){  
        return function (){  
            }();  
    }();  
  
    console.log('Een tekst'.encrypt());
```

Oefening:

1. Breidt bovenstaande zelfuitvoerende enclosure verder uit volgens het module patroon.
Bouw de replacement functie asynchroon op.
Dit betekent dat in "myString.replace(searchValue, newValue)" de newValue een callback functie wordt:
`'Een tekst'.replace(oRegExp, function (a, b) { //return encoded })`
2. Test de encryptie uit op de Loader module, waarbij je user ids encypteert.
`var userIds = ["P1".encrypt(), "P2".encrypt(), "P3".encrypt() . . .`

6.4 Events en eventemitters

De eventloop is gebaseerd op het afvuren van events, en het beheer ervan met callback functies. Wat kan allemaal op het niveau van events binnen node?

Event methods

Voor het beheren van events en callbackfuncties zijn volgende node commands beschikbaar:

Command	Beschrijving
anObject. <u>addListener</u> ("eventType", callbackFunction) anObject. <u>on</u> ("eventType", callbackFunction)	Zowel "addListener" als het verkorte "on" worden gebruikt om een eventListener aan een event type toe te kennen. Niet "addEventListener"! Het event type wordt als een string aangeboden. Dank zij de listeners kunnen meerdere callbacks op eenzelfde event type toegevoegd worden. Elke callback functie zal hierbij worden uitgevoerd.
anObject. <u>once</u> ("event", callbackFunction)	Het event type kan slechts één keer opgeroepen worden.
anObject. <u>removeListener</u> ("eventType", callbackFunction)	Verwijder een specifiek event type.
anObject. <u>removeAllListeners</u> ("event", callbackFunction)	Verwijder alle listeners

Callbackfuncties bij events worden meestal listeners genoemd: anObject("event", listener)

EventEmitter patroon vereenvoudigt event binding

Er wordt heel veel gebruik gemaakt van events in node. Een aangemaakt object (zoals een HTTP server of TCP server) dat events kan uitsenden noemt men algemeen in javascript een "**evented object**". Node noemt een object dat events kan uitsenden een *event emitter*. De event emitters maken gebruik van callback functies (ook listeners genoemd), net zoals de standaard events, maar kunnen naargelang een status of ontvangen antwoord een andere event uitsenden en bijgevolg een andere actie (een andere callback) ondernemen.

Dit is handig wanneer meerdere acties mogelijk zijn in een callback functie. Als voorbeeld kijken we naar een http server (wordt verder behandeld) waarbij de response *het event emitter object* is en andere acties oproept naargelang de ontvangen response. Het API contract in de documentatie moet geraadpleegd worden om te achterhalen welke event types de eventemitter ondersteunt. Ook de signatuur van de argument staat in de API beschreven. Er is altijd wel een "error" event type voorzien.

```
var req = http.request(options, function (response) {  
    /* eerst mogelijke event */  
    response.on("data", function (data) {
```

```
        console.log("Hier response data verwerken", data);
    });
/* tweede mogelijke event */
response.on("end", function () {
    console.log("Hier response beëindigen");
});
});

req.end();
```

Bemerk hoe `response.on("data")` een verkorte schrijfwijze is voor `response.addListener("data")`. Voor alle duidelijkheid, vermeld ik erbij dat ook de verkorte schrijfwijze meerdere eventlisteners toelaat.

Bemerk eveneens hoe gebruik gemaakt wordt van een anonieme functie i.p.v een benoemde zoals in:

```
response.addListener("data", receiveData)
function receiveData(data) { }
```

Event Listeners en de EventEmitter constructor

Javascript beschikt niet over het "class" principe maar werkt met modules of constructor functies en prototypes om objecten en instanties te genereren. In documentatie kan eenvoudigheidshalve wel het woord class gebruikt worden. Zo vernoemen we hier bvb. de EventEmitter class, die tot de events library behoort. (info: <http://nodejs.org/api/events.html>)

```
var emitter = new (require('events')).EventEmitter();
```

Je maakt (registreert) je eigen event handlers met "emitter.on...":
bvb.: `emitter.on(eventName, function (listener){ })`.

Je vuurt het event af met "emitter.emit".
bvb.: `emitter.emit(eventName[, arg1][,arg2][...])`.

Pas op: de volgorde waarin de handlers aangemaakt worden is van groot belang. **Een event kan pas opgeroepen worden NA registratie!** De events worden bovendien afgehandeld volgens de volgorde van aanmaak. Bij het luisteren zonder dat een voorafgaande emit gebeurde, wordt een event niet uitgevoerd, maar krijg je ook geen error boodschap.

Meerdere listeners op één en eenzelfde event zijn hier geen probleem. Je kan ook een willekeurig object voorzien van zijn eigen event emitter. Andere objecten kunnen dan weer zelf event emitters worden of luisteren naar afgevuurde events om naargelang het afgevuurde event een andere actie te ondernemen.

Je kan vaak de applicatie gewoon afwerken met een reeks van callbackfuncties in plaats van events. Die callbackfuncties kunnen mooi na elkaar opgeroepen worden. Maar bij meerdere statussen of verschillende callback functies over verschillende objecten, kan het interessanter worden om event emitters te gebruiken. Een listener luistert gewoon naar een afgevuurd event en doet het nodige.

Simultaan kan het dynamisch aanmaken van event emitters voor memory leaks zorgen. Door een programmeer fout zou het kunnen dat je blijft listeners toevoegen (denk aan de continue eventloop) op hetzelfde event. Node zal wel een warning produceren, maar nog beter is het om het maximaal aantal listeners te tellen met `emitter.listenerCount(eventName)` of te beperken met: `emitter.setMaxListeners (5)`. Dit onderbreekt het programma niet, maar zal wel een warning geven:

```
debugger listening on port 5858
(node) warning: possible EventEmitter memory leak detected. 3 listeners added. Use
  emitter.setMaxListeners() to increase limit.
Trace
```

```
//ophalen van de EventEmitter prototype uit de events lib.
var emitter = new (require('events').EventEmitter)();

var name = "johan";
var counter = 0;

//eventlistener addedUser aanmaken
//-> moet VOORAF: voordat via emit opgeroepen wordt ( wel geen error)
emitter.on("addedUser", function (data, counter) {
    console.log("Je voegde een nieuwe user toe: %s (%s)", data, counter);
});
emitter.setMaxListeners(5); //safety voor verhinderen van memory leaks

//aangemaakt type event oproepen
//argumenten worden met een komma separated list toegevoegd.
emitter.emit("addedUser", name, ++counter)
```

Oefening:

Beschouw een taak waarbij data opgehaald wordt van een server. Na 500msec krijgen we een succesvol antwoord (= wordt een event getriggerd). Na 1 sec komt een fout. Dit simuleren we als volgt:

```
//ophalen van data succesvol
setTimeout(getData, 500, 'success');
//ophalen van data faalt
setTimeout(getData, 1000, 'error');
```

Je kan dit uitwerken met een callbackfunctie "getData" en een extra "receivedData" als callback..

```
setTimeOut(getData, 500, 'success', receivedData);
```

```
function getData(status, callback) {
    callback(status)
}
```

Maar even goed kan dit met een eventemitter, waardoor het extra argument (received data) overbodig wordt. Maak een eventemitter aan, die deze situatie afhandelt met een passende boodschap:

"Het ontvangen van data is afgehandeld met de status"

Oefening:

In de async oefening waar we users ophalen (Loader module), wordt waarschijnlijk wel "console.log()" gebruikt om feedback te krijgen. Om in de toekomst ook andere zaken te kunnen doen met het resultaat, verwijderen we deze console.log om te vervangen door emittor.

```
emitter.emit("addedUser", element);
```

Dit levert ons meer flexibiliteit voor elke andere module die de Loader wil verwerken.

In de file die de applicatie gebruikt doen we de verwerking (asynchroon) via:

```
Loader.emitter.on("addedUser", function (data) { ... })
```

Maak volgende extra events:

- `Loader.emitter.on("end" . . .)` om de totale doorlooptijd weer te geven.
- `Loader.emitter.on("error" . . .)` om foutberichten weer te geven.

6.5 Javascript en node

Meest gebruikte javascript methodes in een node applicatie:

Array	<code>some()</code> , <code>every()</code> : cb op elk element <code>join()</code> , <code>concat()</code> : string converties. <code>pop()</code> , <code>push()</code> , <code>shift()</code> , <code>unshift()</code> : stacks en queues <code>map()</code> : model mapping voor elke item in het array <code>indexof()</code> , <code>filter()</code> : ondervragen <code>sort()</code> , <code>reverse()</code> : sortering <code>slice()</code> : deel kopiëren <code>splice()</code> : deel verwijderen operator <code>in</code> : overlopen van de items	Meer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
Math	<code>random()</code>	Meer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math
String	<code>substr()</code> , <code>substring()</code> : delen vd string <code>length</code> : lengte <code>indexOf()</code> : opzoeken <code>split()</code> : converteren van string naar array	Meer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String
Andere	<code>setInterval()</code> , <code>setTimeout()</code> , <code>ForEach()</code>	

Node.js en ECMASCIPT 5 en 6 (ES5/ES6)

In verschillende browsers kan javascript van een verschillende versie zijn. Bij node runt javascript op de server. Dit resulteert in een veel grotere uniformiteit van javascript versies. De huidige node.js versie beschikt over een aantal bruikbare en interessante mogelijkheden van ECMA script versie 5 en 6, die daarom zonder vrees voor compatibiliteit, kunnen gebruikt worden.

Onderwerp	Eigenschappen/methodes	Meer info
Array	<code>Array.isArray(array)</code>	Controleren of we een array hebben.
<i>prototype</i>	<code>array.filter(callback)</code> <code>array.forEach(callback)</code> <code>array.map(callback)</code>	Maakt een nieuwe array volgens het filter Voert de callback op elk element uit Nieuwe array aanmaken volgens callback functie
Date	<code>Date.now()</code>	Ophalen van de huidige tijd in de vorm van <code>new Date().getTime()</code>

Object <i>Get/Set</i>	<pre>Object.create(proto[, props]) Object.defineProperty(obj, prop, desc) Object.preventExtensions(obj) Object.hasOwnProperty("aProp") var obj = { get iets(){ return "aValue" }, set iets(){ "initialize" } }</pre>	<p>Nieuw object aanmaken vanuit een ander (proto) Object eigenschappen aanmaken</p> <p>Laat niet toe om eigenschappen toe te voegen.</p> <p>getter en setter syntax</p>
JSON	<pre>JSON.stringify(obj [, replacer [, space]]) JSON.parse(string)</pre>	<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>
String <i>prototype</i>	<pre>string.trim() string.trimRight()</pre>	Witruimte verwijderen
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 enkele gekend binnen het block { } en is <i>pas gekend op de lijnen na zijn declaratie</i>. De garbage collector kuist sneller op.</p> <p>Noot: let is nog niet geïmplementeerd in node.</p>

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 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 supporteert 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`

- 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 supporteert 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) {}` 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 betekenis 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.

```
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.

Oefening

Onze `users` array is een JSON array geworden (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 omgezet naar P6.

7 Het module systeem

Node.js voorziet vanuit de command line maar een een low-level API. Het gebruik van third-party modules, naast je eigen applicatie modules, is daarom noodzakelijk om een complexere toepassing te maken, zodat je niet alles zelf moet programmeren.

7.1 Node Packaged Modules (npm)



- Node.js wordt verstuurd met een uitgebreide repository aan bruikbare modules. Deze repository is te vinden op <https://www.npmjs.org/>. Het aantal bruikbare modules groeit zeer snel (76.000 in mei 2014, 82.000 in juli 2014, 188.000 in sept 2015...). Installatie van de modules verloopt vlot via een package manager met naam **Node Packaged Modules** (npm). Deze manager maakt gebruik maakt van nuget.
- Het installatie commando voor een module is "install". Met het bijkomend argument - -save wordt de installatie opgenomen in het json package. Met –dev wordt de module niet geïnstalleerd bij productie deployment.
Enkele voorbeelden:
 - Met "**npm install jslint**" installeer je jslint
 - Met "**npm install colors--save**" installeer je een module om de kleuren en tekst formaat van de console aan te passen, en wordt dit bijgehouden in package.json
 - "**npm install --save-dev -g protractor**" zorgt er voor dat de test runner protractor niet op de productie server komt maar wel globaal op je lokaal systeem.
 - Met "**npm install express@4.1.2 --save**" installeer je het framework express van de vernoemde versie en voeg je dit toe aan de package.json.
Op hun beurt laten geïnstalleerde modules vaak extra opties/installaties toe:
`--express [options] [dir|appname]`
Voorbeeld: --express –e,-ejs voegt support toe voor een EJS view engine (<http://embeddedjs.com>), terwijl express default van jade gebruik maakt.
- npm kan in twee modi gebruikt worden: Local en Global
 - Local is de default en aanbevolen mode en verwijst naar de lokale mappen waarmee de applicatie werkt. In dat geval werkt node met de modules in een onderliggende map met als naam "*node_modules*"
 - Global verwijst naar modules die globaal beschikbaar zijn op systeem niveau. Om de globale modus te activeren tijdens installatie is een “–g” vlag nodig:
`"npm install -g moduleName"`
- Zelf modules toevoegen aan de npm repository kan met "**npm publish**" gevolgd door de modulenaam. Er verschijnt een error bij het gebruiken van een modulenaam die reeds in gebruik is.
Om te publiceren moet de package.json file in de root komen van het npm package.

- Zoeken naar modules doe je met “**npm search**”. Voorbeeld: Met “npm search pdf” krijg je een overzicht van alle modules met pdf in de beschrijving, met ernaast een korte beschrijving van de module. Meer CLI commands vind je in de documentatie: <https://docs.npmjs.com/>:
 - Bijkomende installatie commando's zijn “**npm update**”, “**npm uninstall**”, “**npm link**”, “**npm unlink**”. Linken is interessant indien je een module uit een andere applicatie wil gebruiken in je huidige applicatie.
- npm is een packaging tool, en geen server tool. npm kan bijgevolg ook cliënt script programma's bundelen. (vb: <http://requirebin.com/>)
- Modules kunnen onderlinge dependancies hebben. De snel groeiende module “express” biedt tools aan voor het verwerken van het http protocol en wordt gerefereerd in heel veel andere modules : meer dan 3500 in sept 2014, meer dan 6500 in sept 2015. Overweeg het gebruiken/installeren van een module met zeer veel dependancies. Dit kost overhead en is misschien onnodig.
De dev-dependencies zijn alleen nodig bij ontwikkelwerk (vb. runnen van een test). Dev-dependencies worden niet automatisch gedownload. Ze worden wel gedownload indien je vanuit de command prompt binnen de module map de “install” oproept.
- Een verzameling van modules kan eveneens geïnstalleerd worden via één json package met naam **package.json**. Deze file “package.json” moet een VALID JSON file zijn en wordt toegevoegd aan de root van de applicatie. Je kan deze file manueel toevoegen of je doet het via een node command: >**npm init**. Het command zorgt voor een basis guideline voor de belangrijkste features.

```
C:\Program Files (<x86>)\nodejs>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane 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: <nodejs>
```

Na init kunnen packages geïnstalleerd worden en zorgt --save ervoor dat de package.json file aangevuld wordt

> **npm install express --save**

Omgekeerd kan de package.json manueel aangepast worden en zorgt >npm install dat de modules met hun gewenste versie (!) geïnstalleerd worden.

In de package.json file zijn verschillende eigenschappen aanwezig (tussen double quotes)

- “*name*” is de unieke identifier van de module en verhindert collision met andere modules.
- “*main*” bepaalt het startpunt van de applicatie.
- “*version*”: de versie wordt aangebracht volgens de semantic versioning specificatie (<http://semver.org/>) bvb. 0.1.2-a
 - Hierbij is “0” het MAJOR versie getal en verwijst naar een reeks nieuwe features of functionaliteiten,
 - “1” het MINOR versie getal voor aangepaste features,

- “2” een PATCH versie getal voor updates, “-a”. Het is een optioneel karakter dat geen functioneel effect heeft maar bvb. voor meer documentatie gebruikt wordt.

Spendeer de nodige aandacht aan het nummeren en de compatibiliteit van versies. Node is immers open source! Met het command “npm update” worden de modules naar de laatste gewenste versie gebracht.

- “*dependencies*” en “*devDependencies*”: Afhankelijkheden van andere modules (en hun versie) zijn terug te vinden in de eigenschap “*dependencies*”. Afhankelijkheden die enkel nodig zijn tijdens ontwikkeling (bvb.: testing framework mocha) en niet in productie komen terecht in de eigenschap “*devDependencies*”. Bij de versies verwijst een tilde (~) naar patch updates (geen major of minor versies). Het gebruik van package.json garandeert dat bij deployment je applicatie de juiste versie van een afhankelijke module installeert en gebruikt. De subdependencies van een afhankelijke module worden wel niet gespecificeerd. Uitzonderlijk kan dit voor een probleem zorgen bij het deployen van je applicatie op een andere server, die de vermelde dependancies en (niet compatibele) subdependancies ophaalt. Sommige servers voorzien hiervoor een extra json file. Voorbeeld: windows azure met een npm-shrink-wrap.json file.
- In de “*scripts*” eigenschap kunnen enkel voorgedefinieerde eigenschappen gebruikt worden (op te vragen via npm help scripts). Deze voorgedefinieerde eigenschappen zorgen voor een één op één relatie met commando’s die je in node kan gebruiken. Meer info: <https://www.npmjs.org/doc/files/package.json.html>

- Een uitgewerkte package.json:

```
{  
  "name": "MainApp",  
  "main": "./lib/myStartModule.js",  
  "version": "1.0.0",  
  "description": "Een beschrijving van de module",  
  "author": {  
    "name": "Johan FromHowest",  
    "email": "JohanV@howest.be",  
    "url": "http://www.howest.be"  
  },  
  "keywords": [  
    "myMainApp",  
    "npmExample"  
  ],  
  "repository": {  
    "type": "git",  
    "url": "http://github.com/project/module.git"  
  },  
  "dependencies": {  
    "MainSubapp1": "0.3.x",  
    "MainSubapp2": ">1.2.0",  
  }  
}
```

```
        "TheGitApp": "git+http://git@github.com:project/action"
    },
    "devDependencies" : {
        "mocha" : "~1.9.1"
    },
    "engines" : {
        "node" : ">=0.10.10",
        "npm" : "1.2.x"
    },
    "scripts" : {
        "start" : "httpserver.js",
        "test" : "echo 'Geen testing voorzien.'",
        "postinstall" : "echo 'Bedankt voor de installatie.'"
    }
}
```

Om te helpen bij het aanmaken van een package.json voorziet node het "init" command. Als je dit command inbrengt in de console krijg je een guideline met de belangrijkste eigenschappen.

7.2 Het CommonJS module systeem

CommonJS is een term die sedert 2009 gebruikt wordt voor het definiëren (MIT licentie) van de opbouw van *javascript applicaties, die buiten de browser draaien*; zoals bijvoorbeeld node.js. Er wordt gewerkt aan deze specificatie door een werkgroep, die echter nog niet erkend is door de ECMA werkgroep. Basis informatie is te vinden op <http://wiki.commonjs.org/wiki/Modules/1.1>

Node baseert zich op de CommonJS specificatie om een node applicatie op te splitsen in verschillende modules. De modules zelf kunnen we onderverdelen in verschillende type modules. We kunnen 3 types onderscheiden: default aanwezige kern modules, third party npm modules, zelf aangemaakte modules.

Noot: Naast CommonJS gebruikt door Node, bestaan voor Javascript nog andere component standaarden zoals AMD(Asynchronous Module Definition – vooral gebruikt door RequireJS) en UMD(Universal Module Definition – compromis tussen CommonJS en AMD).

Javascript, dat ingeladen wordt op een webpagina, injecteert zijn variabelen in een global namespace waardoor deze variabelen voor alle scripts van de applicatie adresseerbaar worden. Bij grotere Javascript projecten kan dit problemen leveren en moet er opgelet worden om geen collision van deze variabelen te bekomen. Bij het gebruik van third party modules is dit zeker een aandachtspunt. Niet alleen om geen conflicten maar ook om geen beveiliging issues te hebben. De CommonJS module van node.js helpt hierbij en zorgt ervoor dat er geen conflicten ontstaan binnen de global namespace van Javascript. Door CommonJS krijgt elke module zijn eigen context of scope in de global namespace.

Noot: door het gebruik van een library zoals (<http://www.commonjs.org/>) en RequireJS (<http://requirejs.org/>) kan dezelfde eigenschap gebruikt worden in een raw javascript omgeving.

De modules in node worden opgehaald door hun naam of door een filePath. Eénmaal ingeladen worden de modules bruikbaar via hun publieke API.

Volgende CommonJS commando's worden gebruikt voor het integreren van de modules:

OPHALEN van functies of module	
<pre>var module = require('module_name'); var moduleFolder = require('module_Folder');</pre>	<p>Returnt het object met de API vd module en beïnvloedt de global namespace niet. Het return resultaat kan een functie zijn, een object, een Array. Een ingeladen module wordt automatisch gecached, de instructies ervan worden zo maar één keer uitgevoerd. Er kan ook een folder opgevraagd worden via require, waarbij node er eerst naar index.js , index.json of package.json zoekt.</p>
<pre>var modulePath = require.resolve('module_name');</pre>	<p>Met resolve wordt de module niet ingeladen, maar wordt het path van de ingeladen module gereturned.</p>
<pre>//kernmodule var http = require('http'); //eigen module in node_modules var myModule5; var myModule5 = require('my_module_5');</pre>	<p>Inladen van module op basis van zijn module naam Dit kan zowel voor <i>een kern module</i> (= binair beschikbare modules van node) als voor eigen aangemaakte modules die ook in de folder node_modules geplaatst worden..</p>
<pre>//relatief adres via een punt: var myModule = require('../my_modules/my_module'); var myModule2 = require('../lib/my_module_2.js'); //absoluut adres: var myModule3 = require('C:/lib/my_module_3.js'); var myModule4 = require('C:\\lib\\\\my_module_4');</pre>	<p>Inladen van een module op basis van zijn adres: Externe javascript files (*.js) staan in een één op één relatie met een uitvoerbare module. Classes, objecten, herbruikbare functies plaats je daarom in een externe file. Het adres kan zowel relatief als absoluut zijn. Het suffix ".js" moet niet/mag vermeld worden. Node zoekt default naar .js , .json en .node extensies. Is de module beschikbaar in een andere folder dan is het gebruik van een puntje of twee puntjes een must (!) voor relatieve adressering. Wanneer de module beschikbaar is in de standaard "node_modules" folder moet de folder niet vermeld worden. (= geen puntje = zoek in node_modules) Bij common practice worden de eigen modules dan ook vaak in deze folder geplaatst.</p>
<pre>var myModule = require('./myModuleDir');</pre>	<p>Bij het inladen van een folder wordt naar index.js of naar een json package gezocht. Binnen het json package wordt naar de eigenschap <i>main</i> gezocht met de naam van de opstart file.</p>

EXPORTEREN van functies of module

```
module.exports = function() {
    /* bvb. constructor fctie*/
    function doeIets(a,b) {
        return a*b;
    }
}

/* Ook een module pattern (IIFE)
kan geëxporteerd worden */
var Customer = (function() {
    var _login=function(pwd, callback) {};
    return {
        login: _login
    }());
}

module.exports = Customer;

/* of alles als een reeks van
afzonderlijke functies exporteren
*/
module.exports.doeIets =
    function doeIets(a,b, callback) {
        return a*b;
};
module.exports.login = . . .
```

Een zelf aangemaakte module wordt bruikbaar voor de andere modules binnen het project door deze module te exporteren met module.exports of kortweg exports. Het exporteren doe je op een object (= zijn constructor functie) of op een functie(= is ook een object , kan dus een module patroon zijn).

Alleen zaken die met exports aangestuurd worden zijn public beschikbaar. Alle andere variabelen, funties, objecten blijven private binnen de module.

```
// in config.js
var Config = {
    connection: 'localhost:test'
};
module.exports = Config;

// in app.js en andere modules
var config =
require('./config.js');
console.log(config.connection);
```

Modules vervuilen de scope niet, maar blijven binnen hun eigen (object) context. Wil je een variabele delen over verschillende modules, dan kan je niet anders dan deze variabele in een object te plaatsen en de module telkens opnieuw te "requiren" in elke module, waar nodig. Een typisch voorbeeld hiervan is een configuratie file.

Noot: Soms zijn meerdere keren dezelfde modules nodig over het project. Het herhalen van require op een reeds ingeladen module heeft geen performantie gevolgen. Natuurlijk kan je om code lijnen te sparen het geheel anders schrijven door modules te bundelen in een overkoepelende module met bvb.als naam common.js:

```
Common = {
    util: require('util'),
    fs: require('fs'),
    path: require('path')
};
```

```
module.exports = Common;
```

Oproepen in je applicatie app.js kan nu met één lijn: `var Common = require('./common.js');`

Oefening: gebruik van een externe module (optioneel)

Maak een console applicatie die gebruik maakt van (een) externe module(s) om een prompt met validatie aan te maken. De prompt vraagt naar een naam en een geboortedatum. Nadien wordt gevraagd een kleur te kiezen.

```
prompt: Voer je naam in: Johan
prompt: Wat is je geboorte datum?: 1990/04/01
error: Invalid input for Wat is je geboorte datum?
error: Verwacht formaat:MM/DD/YYYY
prompt: Wat is je geboorte datum?: 04/01/1990
prompt: Kies een kleur: yellow, white, blue, green, cyan: cyan
```

Bij valid ingave wordt een boodschap getoond met je leeftijd en dit in het gekozen kleur.

```
Welcome Johan
You are 24 years old.
```

7.3 Modules en het Module pattern

Het opsplitsen van een groter programma in verschillende herbruikbare modules is zeker niet ongewoon. In node wordt dit nog belangrijker om het overzicht te bewaren. Afsplitsen in herbruikbare modules kan zoals hierboven vermeld met “*require*” en “*exports*” en is een must do. Het is eveneens aangeraden om afgesplitste modules, die als zelfstandige blackboxes fungeren, af te splitsen in een eigen namespace.

Voorbeeld: De node module fs is een basis module van node en laat toe een file uit te lezen. Je kan fs bijvoorbeeld gebruiken om een eigen module aan te maken (myReader.js). Documentatie van de fs module vind je terug op <http://nodejs.org/api/fs.html> en niet op npm.

Een module “myReader.js” met een read methode kan er in zijn eenvoudigste vorm als volgt uit zien, waarbij alleen een synchrone read methode publiek bereikbaar is. Bemerk hoe node zijn data returnt via een Buffer object. Dit Buffer object verwerkt de gegevens binair, waardoor veel efficiënter data gelezen wordt in vergelijking met het lezen van strings.:

```
//myReader.js:
fs = require("fs");

var read = function (fileName) {
    //toString() zet de array Buffer met binaire data(!) om naar leesbare tekst
    return fs.readFileSync(fileName).toString();
}

module.exports.read = read;
```

Gebruiken van de module “myReader”:

```
myReader =require("myReader");
console.log(myReader.read("testFile.txt"));
```

Oefening (Herhaling) :

1. In bovenstaand voorbeeld wordt gebruik gemaakt van een synchrone read methode. Er bestaat een asynchrone tegenhanger (`fs.readFile`). Maak eerst een kleine toepassing met de module `fs` en zijn asynchrone read methode. Vergeet de error behandeling niet.
 - a. Lees met `fs.readFile(filename, [options], callback)` een willekeurige tekst uit.
 - b. Zorg dat elke nieuwe lijn in de tekst voorafgegaan wordt door het lijnnummer. Haal daartoe elke lijn van de tekst op met

```
var lines = text.split('\n');  
lines.forEach(function (line) { ..... })
```
 - c. Vul de callbackfunctie van `forEach` aan.
 - d. Toon het resultaat in de console.
2. Omdat we het lijn nummeren nog willen hergebruiken als afzonderlijke module splitsen we de code ervoor af in een afzonderlijke module.
 - a. Maak een file (`LineNumbering.js`) aan voor de gelijknamige module
 - b. Maak in de module gebruik van een constructor functie om een object `GetText` aan te maken :

```
var GetText = function () { }  
Of dit kan ook via: function GetText() { }
```

Noot: Meestal stelt met de constructor naam gelijk aan de file naam. Het oproepen van de module gebeurt wel via de filename (Hier: `LineNumbering`).
 - c. Ken een methode "reader()" toe aan `GetText` via zijn prototype.

```
GetText.prototype.reader = function (text) {  
    //werk hier het lijn nummeren uit met forEach zoals hierboven in punt 1  
    //return het resultaat  
}
```
 - d. Maak de module beschikbaar voor opvragende applicaties door zijn constructor te exporteren, waardoor ook zijn prototype geëxporteerd wordt:

```
module.exports = GetText;
```
3. Maak een applicatie (voor het ophalen van jouw file) die gebruik maakt van de module `LineNumbering.js`
 - a. Haal de module op (en vergeet het adresserende puntje niet!)

```
var LineNumbering = require('./LineNumbering');
```
 - b. Maak gebruik van `fs.readFile()` om een tekst file in te lezen. In de callbackfunctie maak je gebruik van de linenumbers instantie:

```
var lineNumbers = new LineNumbering();
```
 - c. De prototype methodes (zoals `reader()`) zijn nu bruikbaar:

```
lineNumbers.reader(text)
```
4. Noot: de module werd aangemaakt op basis van een constructor functie, maar de module kon even goed aangemaakt worden op basis van een zelfuitvoerende closure.

8 Process beheer in een asynchroon programmeer model.

8.1 Single threaded javascript

Zoals eerder vermeld zijn Javascript en dus ook Node single threaded. Asynchroon werken op basis van een callback functie verwacht bovendien een aanpassing voor vele ontwikkelaars. Deze aanpassing manifesteert zich op twee vlakken: bij het asynchroon programmeren van non-blocking I/O acties en bij het shedulen van taken in een bepaalde volgorde. Enkele voorbeelden:

1. Een blokkerende while in setInterval.

Door het gebruik van verschillende setInterval's na elkaar kan de indruk gewekt worden dat meerdere asynchrone taken gelijktijdig verlopen. De methoden setInterval en setTimeout bestaan immers globaal in node, en kunnen als timers gebruikt worden om functions teshedulen in de tijd.

```
setInterval(function () { console.log("Hello"); }, 0);
setInterval(function () { doeIetsAnders(arg) ; } , 1000);
```

Maar toch blijft dit single threaded. Ook al wordt gebruik gemaakt van een callback functie.

```
setInterval(function () {
    var teller=0;
    console.log("Ik blokkeer de andere setInterval met een endless while(true).");
    while (true) {
        console.log(teller++);
    }
},1000);

setInterval(function () {
    console.log("Komt nooit aan de beurt");
},1000);
```

Dit voorbeeld toont dat setInterval singlethreaded uitgevoerd wordt. De eerste setInterval blokkeert met while de thread en toont dat voorzichtigheid geboden is voor het uitwerken van een non-blocking I/O applicatie. Denk maar aan "while there is a keypres...". Besef dat Javascript niet beschikt over een methode om te returnen uit een niet afgewerkte asynchrone methode.

2. Volgorde waarin processen afgewerkt worden

Dat het geheel met setInterval of setTimeout wel degelijk asynchroon verloopt zien we in volgende voorbeeld doordat de synchrone boodschap "Dit kom eerst" getoond wordt, terwijl niet gewacht wordt op setInterval(). Wanneer setInterval opgeroepen wordt als event wordt het als event in de queue geplaatst. De methode setInterval houdt het programma niet op, maar het programma gaat gewoon verder.

```
setInterval(function () { console.log("Hello"); }, 0);
console.log("Dit komt eerst op de console");
```

3. Verschil synchroon gedrag / asynchroon gedrag is niet altijd even duidelijk.
Dat we met een asynchrone functie te maken hebben in node is duidelijk te zien aan de aanwezigheid van een callback functie. Maar in combinatie met cliënt code, die bovendien kan cachen, kan er verwarring ontstaan. Een asynchrone methode kan zich plots synchroon gaan gedragen.

Voorbeeld hiervan is de jQuery operator "\$". Er wordt gewacht voor uitvoering van \$ tot de DOM geladen is (= asynchroon), maar als de DOM reeds in cache bestond, start de uitvoering onmiddellijk (=synchroon), zonder te wachten op andere functies in je programma.

De methoden van node zijn wel duidelijk asynchroon of synchroon. Het is vaak te zien aan de benaming van de methode. Zo bestaat een synchrone `fs.readFileSync(aFile)` en een asynchrone `fs.readFile(file, function (err, contents) { });`

4. Loops (for, while) die voor verwarring zorgen in combinatie met de event queue.
Er is slechts één variabele i die lokaal binnen de for lus draait. De eventHandler van setTimeout wordt ook hier in de queue geplaatst en wordt pas aangesproken wanneer de thread van de incrementerende lus vrijgegeven wordt.

```
//setTimeout wachtend op de event queue waardoor de console ...
for (var i = 1; i <= 3; i++) {
    setTimeout(function () { console.log(i); }, 0);
}
```

De oplossing werd reeds eerder vermeld om met een zelf uitvoerende closures een tweede scoop aan te maken voor setTimeout.

5. Volgend voorbeeld toont eveneens dat een while loop van 1 seconde niet onderbroken wordt.

```
var start = new Date;
//setTimeout wacht 1000msec om op de event queue geplaatst te worden
setTimeout(function () {
    var end = new Date;
    console.log('Verlopen tijd:', end - start, 'ms');
}, 500);

while (new Date - start < 1000) { };
```

6. Ajax als asynchrone leerschool.

Het wordt duidelijk dat éénmaal in Javascript je een I/O asynchrone behandelt, je deze niet kan onderbreken (ook niet voor een error). Dit is één van de key features van node.js: non-blocking I/O. Wachten op een keypress van een gebruiker gebeurt niet met een blokkerende while maar wel door het gebruik van een handler en zijn callback functie. Deze methodiek werd in javascript voor het eerst toegepast bij AJAX calls.

```
var ajaxRequest = new XMLHttpRequest;
ajaxRequest.open('GET', url);
ajaxRequest.send(null);
ajaxRequest.onreadystatechange = function() {
    // ...
};
```

7. setInterval en setTimeout zijn van nature uit traag.

De asynchrone uitvoering van setInterval en setTimeout zorgen inherent voor een vertraging. De HTML specificatie definieert zelf een *minimum(!)* vertraging van 4msec. Om hieraan

tegemoet te komen (= sneller of gecontroleerde een actie aan te spreken) bestaan een aantal mogelijkheden:

- a. node.js laat toe om met het command `process.nextTick()` een functie af te vuren bij de eerstvolgende eventloop.
 - b. De recente browsers voorzien en methode `requestAnimationFrame()`, die vooral handig is om een animatie over een vastgelegde tijd te laten gebeuren. Een animatiecyclus wordt bij default afgesteld op 60Hz. (60 frames per seconde)
 - c. Gebruik van een distributed event patroon (komt verder aan bod).
8. Toch kan ook multithreaded gewerkt worden in javascript door het gebruik van een `Worker()` object, waardoor de applicatie op multiple cores draait. Node voorziet hervoor een `fork()` `process()`. (komt verder aan bod).

8.2 Async error handling

8.2.1 Try/catch vervangen door domain errors

Try/catch bestaat in javascript en is handig in een synchrone omgeving: je krijgt een foutmelding op de lijn en een stack trace te zien. Maar bij asynchrone werking is try/catch zinloos. Door de asynchrone functie in de try (die in de event loop komt), zal de catch nooit bereikt worden.

```
try {
    setTimeout(function () {
        throw new Error("Ik ben een 'uncaught' error en stop de applicatie!");
    }, 0);
}
catch (e) {
    console.error("Deze catch werkt alleen in synchrone omstandigheden:" + e);
}
```

Hoe de API er ook uit ziet, *asynchrone errors kunnen alleen behandeld worden vanuit de callback functie*. Hoe kan je dan de fout opvangen onder dat de applicatie afsluit?

De eerste versies van node hadden een eventhandler `process.on("uncaughtException", function(error, data) {})`.

Deze handler werd vanaf versie 0.8 vervangen door "domains". Binnen een domein worden errors via event emitters vrijgegeven. Door het gebruik van een domain worden alle fouten, timers, callback methoden implicit *en alleen geregistreerd binnen het domein*. Bij een fout wordt een on "error" event uitgestuurd en crasht de applicatie niet.

Om de fout weer te geven wordt naar dit distributed "error" event geluisterd.

Verschillende domeinen kunnen bovendien gescheiden naast elkaar bestaan waardoor geen collision problemen optreden. In volgend voorbeeld is de fout explicet gebeurd op een timer in domain d2. Het gebruik van domeinen kan echter nieuwe problemen meebrengen onder de vorm van memory leaks. Het gebruik ervan moet zorgvuldig overwogen worden. Meer info: info: <http://nodejs.org/api/domain.html>

Om memory leaks te verhinderen wordt aan een vernieuwde implementatie voor domeinen gewerkt. Er wordt aanbevolen om intussen vooral gebruik te maken van het first "error" argument in de callbackfuncties (en dus toch dit deel van de user applicatie af te sluiten). Is toch een

duidelijker beeld nodig in welk domein een fout zich voordeed, dan bestaat de mogelijkheid om de buggy code op een afzonderlijke worker te plaatsen met require ("cluster").

Voor de volledigheid vind je hier toch nog enkele voorbeelden van domains zoals ze momenteel nog gebruikt worden ondanks de "deprecated" warning:

Stability: 0 - Deprecated

This module is pending deprecation. Once a replacement API has been finalized, this module will be fully deprecated. Most end users should **not** have cause to use this module. Users who absolutely must have the functionality that domains provide may rely on it for the time being but should expect to have to migrate to a different solution in the future.

```
var domain = require("domain");
var d1 = domain.create();
var d2 = domain.create();

//run laat toe een functie toe te kennen aan het domein
d1.run(function () {
    //add laat toe een emitter, callback of timer toe te voegen
    d2.add(setTimeout(function () {
        throw new Error("error op de timer van domain 2");
    },0));
});

//handlers voor een abstracte foutbehandeling zonder applicatie crash
d1.on("error", function (error) {
    console.log("fout in domain1: " + error);
});
d2.on("error", function (error) {
    console.log("fout in domain2: " + error); //foutmelding door d2.add
});
```

8.2.2 Error argument in callback functies

Het "throwen van algemene exceptions" is op zich niet de beste manier om fouten te onderscheppen. Je kan dit gebruiken om gedurende het ontwikkel proces het oplossen van een (moeilijke) fout uit te stellen.

In een productie omgeving is het niet aangeraden om de volledige applicatie te stoppen en bvb. alle connecties op te geven zonder de gebruiker een nieuwe kans te geven. In deze gevallen is het aangeraden om de fout af te handelen *via het error argument in de callback functie*. Via een if error functie kan je de fout afhandelen door een boodschap te bezorgen aan de gebruiker, door een tweede request te lanceren enz... Daarom voorzien de meeste functies van node *inherent als eerste argument een "error" en als laatste argument "de callback functie"*.

Wil je toch de fout throwen zonder de applicatie af te breken, dan kan je het error argument binden aan een specifiek domain. Dit gebeurt door de callback functie toe te kennen aan het domain met de methode bind():

```
var fs = require("fs");
var d1 = require("domain").create();
```

```
//fs.readFile(filename, [options], callback)
fs.readFile("onbestaand.docx", null, d1.bind(
    function (error, data) {
        if (error) {
            console.log("Gelieve opnieuw te proberen.");
            throw (error);
        } else {
            console.log("File wordt gelezen:" + data);
        }
        //dispose verwijdert het domein en kuist alle I/O data op voor het verhinderen
        // van mem.leaks.
        d1.dispose();
    }
));

d1.on("error", function (error) {
    console.log("Er gebeurde een fout in domain d1", error);
});
```

8.3 Callback Hell = Pyramid of Doom = the Boomerang effect

Het wordt stilaan duidelijk dat een asynchroon programmeer model opgebouwd wordt door een keten van geneste callback functies. Dit kan het geheel 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.

Een filereader voorbeeld met drie geneste callback functies:

```
var fs = require("fs");
var fileName = __dirname + '/MyTextField.txt';

fs.exists(fileName, function (exists) {
    //callback 1: bestaat de file
    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);
                });
            }
        });
    }
});
```

Hoe kan je het geheel dan leesbaar houden?

Er wordt aangeraden om maximaal een tweetal callback functies te nesten voor leesbaarheid. De overige callback functies kunnen benoemd worden of er kan gebruik gemaakt worden van distributed events.

8.3.1 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.exists(fileName, cbExists)

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

Oefening:

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

8.3.2 Distributed events

Behavior driven patronen

Er zijn een verschillende event driven (= behavior driven) patronen, die vaak ook door elkaar gebruikt worden. Een status verandering bij een subject (of **event emitter**) zorgt dat één of meerdere ontvangers (**event consumer**) op de hoogte gebracht worden via een **event channel**. Het event channel kan op verschillende manieren gerealiseerd worden van point-to-point tot message oriented over een queue. Verschillende patronen zijn mogelijk.

Bij een "Command pattern" wordt behavior driven gerealiseerd door een relatief harde koppeling tussen subject en ontvanger. Maar bij grotere en zeker distributed applicaties wil je meer loosely coupled programmeren om het overzicht te bewaren en geen data te verliezen. Dit kan perfect event driven waarbij je naargelang het patroon al dan niet meer voor minder "tight coupling" kiest. Enkele voorbeelden:

- Observer patroon: waarbij het subject al zijn listeners informeert bij een status verandering. Subject en ontvanger kennen elkaar, wat nog een relatief tight coupling inhoudt.
- In het klassiek Event-model patroon (zoals in javascript, .Net) wordt het observer patroon uitgebreid, waarbij het subject meerdere status wijzigingen aanbiedt (meestal voorgeprogrammeerde zoals click, double click..). Het is de ontvanger die kiest op welk event hij wil reageren. Verschillende subjects kunnen hetzelfde event uitzenden, maar het subject hoeft zijn ontvangers niet meer te kennen (less tight coupling). Om zelf een ontvanger/receiver aan te maken kan dit in node kan dit met het "EventEmitter" package.
- Pubsub patroon (of "publish/subscribe") is een doorgedreven event model gebaseerd op het sturen van "*messages*". Ook hier kent de emitter zijn consumers niet en hoeft hij ook niet te weten of events geconsumeerd worden. Maar je stapt nog verder af van tight coupling door tussen het subject en ontvangers *een extra object* aan te brengen. Deze laatste regelt (meestal in cache) al het verkeer van subjects naar ontvangers, ook over verschillende instanties of verschillende servers. De ontvanger kan zich zo op verschillende locaties bevinden en hoeft maar te luisteren naar een "message". *In node zelf is een volwaardig tussenliggende pubsub interface niet geïmplementeerd*. Je kan hiervoor wel beroep doen op

extra packages zoals pubsub voor mongoDB of Redis. Dergelijke packages maken ook wel gebruik van het "EventEmitter" object. dit komt verder nog aan bod.

Distributed events met "EventEmitter"

In node worden distributed events geïmplementeerd met een EventEmitter package. De EventEmitter van node kan gebruikt worden om de callback hell maar ook om complexe en uitgebreide eventhandlers te vermijden.

Jouw zelf aangemaakte custom objecten erven hierbij van EventEmitter. Daarna kunnen deze objecten voorzien worden van specifieke applicatie logica, die naargelang de status een andere handler oproept. Denk aan het MVC patroon waar het model (= het object) ervoor kan zorgen dat specifieke acties ondernomen worden naargelang de situatie. Schalen van een applicatie, het toevoegen of verwijderen van extra functies wordt door deze techniek eenvoudiger. Het gebruik van distributed events wordt gebruikt bij het horizontal scalen (= extra hardware/software entities) op de cloud. Verschillende instanties op deze cloud informeren elkaar via distributed events.

Via de methode inherits() uit de node library util wordt in javascript het prototype erving mechanisme gestart. Geen class (onbestaand) maar wel een basis object (hier: EventEmitter) dient als prototype voor het nieuw object (hier: UserEmitter) :

```
UserEmitter.prototype = EventEmitter.prototype;
```

Via de call() methode kan een constructor functie van ObjectA uitgebreid worden met de constructor argumenten van een ObjectB. Zo wordt de basis EventEmitter constructor opgeroepen om ons eigen object aan te vullen. Voor het overige kunnen nu willekeurig methodes bijgevoegd worden aan het object. Zorg dat je hierbij het geërfde prototype niet opnieuw overschrijft met UserEmitter.prototype= {} maar wel aanvult met UserEmitter.prototype.say = function() {}

We beschikken nu over een custom object dat event aware geworden is.

```
var EventEmitter = require("events").EventEmitter;

//1. Object constructor
function UserEmitter() {
    //instantie eigenschappen
    var self = this;
    //constructor van EventEmitter wordt toegevoegd aan constructor van UserEmitter
    //call(thisArg[,arg1 [, arg2 ], ... ]])
    EventEmitter.call(self);
    //instantie methodes
    self.addUser = function (username, password) {
        //publish (=emit) het event.
        self.emit("userAdded", username, password)
    };
}

//2. prototype erving:object UserEmitter erft van EventEmitter om te publishen
util.inherits(UserEmitter, EventEmitter);

//prototype aanvullen (niet met prototype = { say:  })
UserEmitter.prototype.say = function (iets) { console.log(iets); }
```

Testen van het object en zijn distributed event:

```
var user = new UserEmitter();
```

```
//3. Subscribe op het event met on("  ", callback) vóór de emit gebeurt (async)
user.on("userAdded", function (userName, pwd) {
    console.log("User " + userName + " werd toegevoegd.");
});

//4. Emit event (als laatste)
user.addUser("Johan", "myPWD");
```

Oefening:

Maak van het Loader constructor object een “event-aware” object, dat zowel fouten als taken afwerkt via een event.

```
var loader = new Loader(users, usersIds);
loader.on("addedUser", function () {});
loader.on("end", function () {});
loader.on("error", function () {});
```

Oefening (optioneel- herhaling):

Maak een publiciteits Ticker, die met een interval van 5 seconden een publiciteits boodschap uitstuurt naar ingeschreven ontvangers. Maak eens gebruik van het module pattern (IIFE).

Het runnen van de Ticker gebeurt als volgt:

```
Ticker.on("publicity", function (message) {
    console.log("publicity received" , message);
});

Ticker.showTicks(interval, arr);
```

Oefening:

Gebruik een EventEmitter om in het voorbeeld met de filereader de callback hell te vermijden.
Tips om dit uit te werken:

1. Importeer (require) "EventEmitter", "util" en "fs"
2. Maak een constructor "FileReader" en zorg dat deze erft van EventEmitter;
3. Roep in FileReader de methode fs.exists op..
4. Binnen de methode exists wordt, indien de file bestaat, een event uitgestuurd naar de volgende callback : de stats methode.

```
fs.exists (this.url, function (exists) {
    if (exists) { self.emit("stats") };
});
```
5. Werk het stats event uit.

```
self.on("stats", function () { fs.stat(...)});
```
6. Roep vanuit de stats handler de derde callback "read" op als een event en lees hierbij de file uit.

```
self.on("read", function () { fs.readFile(...)});
```
7. Test het object FileReader()

```
var fileReader = new FileReader(__dirname + '/MyTextFile.txt');
```

9 Flow control en scheduled processen

9.1 Scheduling node processen

Tot nu toe werden hoofdzakelijk enkelvoudige of alleenstaande asynchrone taken behandeld. Maar het kan ook nodig zijn om meerdere asynchrone taken in een bepaalde volgorde te gaan uitvoeren.

Het **shedulen** van taken dringt zich op en kan bijvoorbeeld nodig zijn voor het uitstellen van remote data synchronisaties, voor het regelmatig opkijken na een bepaalde tijd van cached data, voor het vrijgeven van connecties of voor het tijdelijk behouden van sessions en polling processen. Herinner hierbij dat een process-intensieve callback, de event loop in grote mate kan opeisen.

De basis scheduling methoden zijn gekend vanuit browser Javascript en gebruiken we reeds:

- `setInterval/clearInterval` voor het periodiek uitvoeren van processen
- `setTimeout()/clearTimeout()` om de uitvoer van een proces uit te stellen of te plannen in de toekomst.

Periodiek proces stilleggen in de toekomst

Een interval proces stilleggen na bijvoorbeeld 5 seconden kan als volgt. Het interval proces zelf wordt hiervoor in een variabele geplaatst en gecleared:

```
var interval = setInterval(function message() {  
    console.log("timed out na 5 sec!"); },  
    1000);  
  
setTimeout(function B() {  
    clearInterval(interval); },  
    5000);
```

Een lang durende process tijdelijk onderbreken:

Een lang durend process kan de event loop onderbreken. Maar wat als dat proces toch moet uitgevoerd worden (bvb. Éénmalig) en men toch niet de andere processen wil laten wachten. Soms onderbreekt men met `setTimeout` het langdurende process voor een tijdje na een willekeurig aantal iteraties (num). Door `setTimeout` wordt het process opnieuw in de queue geplaatst.

```
function longProcess(operation, num) {  
    if (num <= 0) return  
    operation()  
    //onderbreek hier tijdelijk na bvb 10 iteraties  
    if (num % 10 === 0) {  
        setTimeout(function () {  
            longProcess(operation, --num)  
        })  
    } else {  
        //staat niet op de eventloop:  
        longProcess(operation, --num)  
    }  
}.
```

Bruikbare scheduling commands

Command	Beschrijving
<code>setTimeout(callback ,0)</code> <code>setTimeout(callback)</code>	Door setTimeout komt de callback in een scheduling queue. De 0 of niets duidt aan in javascript dat het event zo snel mogelijk moet worden uitgevoerd. Dit is onmiddellijk nadat alle lopende events uitgevoerd zijn. Dit kan handig zijn om bvb. verschillende animaties na elkaar in queue uit te voeren.
<code>process.nextTick(callback)</code>	"process" is het meest global object in node. Een tick verwijst naar de eventcyclus. Door nextTick(= de volgende event cyclus) wordt de callback direct na het procesen van de lopende events uitgevoerd. Dit is <u>sneller dan de activatie van de scheduling queue bij setTimeout</u> met een minimale delay 0.
<code>stream.on("data", function(data) { stream.end("einde boodschap"); process.nextTick(function() { /* hier uitgestelde taak */ }); });</code>	Door nextTick te gebruiken in de callback kan je een niet cruciale taak ook uitstellen tot later (= tot een volgende event cyclus). Het gebruik van nextTick is ook een oplossing wanneer te snel verschillende async taken na elkaar opgeroepen worden en je een "stack size exceeded" error krijgt.
<code>(function schedule() { setTimeout(function do_it() { my_neccesary_process(function() { console.log('process done!'); schedule(); }); }, 1000); }());</code>	<i>setInterval</i> houdt geen rekening met een vertraagde uitvoering van een proces. Dit kan resulteren in een collision tussen de verschillende opeenvolgende processen. Wil je zeker zijn dat elk proces "volledig" uitgevoerd wordt, roep dan recursief het proces op. Of gebruik een self-executing functie die zichzelf oproept, na de tijd nodig voor het proces, die je dan wel moet inschatten (bvb.1000msec).

Het gebruik van `process.nextTick` vraagt enige aandacht:

- Blijf consistent werken en zorg dat asynchrone werking niet gemengd wordt met synchrone werking. Synchrone taken worden steeds onmiddellijk uitgevoerd. Alle processen binnen eenzelfde functie uitvoeren op de nextTick is het eenvoudigste om te volgen.
- Bijkomende tip: "Return" een `process.nextTick()` om zeker te zijn dat in een callback slechts één tick uitgevoerd wordt. Door de return ben je weg uit de callback functie. Zonder de return worden in volgend voorbeeld beide `process.nextTick()` uitgevoerd bij een negatief getal:

Niet consistent gebruik van nextTick	Beter (volledig asynchroon)
<pre>function isNegative(n, callback) { if (n < 0) { process.nextTick(function () { callback(true); //asynchroon }); } //NOK : synchroon - eerst uitgevoerd callback(false); }</pre>	<pre>function isNegative(n, callback) { if (n < 0) { return process.nextTick(function () { callback(true); //asynchroon }); } //OK: asynchroon return process.nextTick(function () { callback(false); //asynchroon }); }</pre>

9.2 Externe processen beheren

Een taak die veel processing power vraagt van de CPU of een langdurige callback functie kan de volledige event-loop blokkeren. In dat geval is het aangewezen de zware taak naar een child process over te brengen. Dit child process staat onder controle van de parent, die het proces lanceerde. De event loop staat er los van. Het child process staat via een tweeweg communicatie in verbinding met de parent. De parent kan het child deels controleren, of het child zal bij voltooiien van de taak het resultaat aan de parent bezorgen. Dit process kan een extern process *buiten node* zijn (C++, unix command., externe *.exe). Na afloop wordt het resultaat aan node bezorgd.

Uitvoeren van externe commands met child_process.exec

Om een relatief zwaar, extern commando of een executable te runnen, wordt het extern proces gestart met de exec methode. In de callback functie van exec. wordt het resultaat tijdelijk *gebuffered* (en dus niet gestreamd):

Referentie: child_process.exec(command[, options], callback)

```
var child_process = require('child_process');
var exec = child_process.exec;

// exec(command, callback) zorgt voor uitvoer van een command
//vb. command 'doeIets' uit het child process
// de prefix std verwijst naar data return via een Buffer
exec('doeIets', function(err, stdout, stderr) {
  if (err) {
    console.log('child process gestopt met error code', err.code);
    return;
  }
  console.log(stdout); // output van het childprocess wordt gebufferd in stdout
});
```

Er kunnen nog extra opties meegegeven worden. Typisch worden deze extra's als een javascript object aangeboden. Klassiek komt ook hier de error als eerste en de callbackfunctie als laatste argument.

```
var options = {
  timeout: 1000,
```

```
    encoding: ascii
}
exec('doeIets', options, function(err, stdout, stderr){ } )
```

Tweeweg communicatie met het child proces via child_process.spawn

Het exec command heeft enkele nadelen: er kan niet gecommuniceerd worden met een child process en de output van het child process wordt niet gestreamed maar volledig in een buffer opgeslagen, wat geheugen vraagt. Deze nadelen worden opgelost door het spawn process te gebruiken dat twee weg communicatie toelaat. Via input- en output streams is interactie mogelijk tussen hoofd- en child-process. De interactie is gebaseerd op EVENTEMITTERS (en dus GEEN CALLBACKS) en kan aangevuld worden met standaard commands om bijvoorbeeld het child process te stoppen(kill).

In de achtergrond kan tussen hoofdprocess en childprocess gecommuniceerd worden met voorgedefinieerde signalen ("signals" genaamd). Zo wordt een bijvoorbeeld een signaal SIGKILL uitgestuurd wanneer een child.kill() uitgevoerd wordt. Sommige signalen kunnen door het child behandeld worden, andere signalen zijn alleen te verwerken door een specifiek operating systeem (linux).

Referentie: child_process.spawn(command[, args][, options])

```
var spawn = require('child_process').spawn;
//vergeet de [ ] (een array) niet -> unshift error
var child = spawn('myCommand', ['args'], ['options']);

//listener om data/error te ontvangen van child process
child.stdout.on('data', function(data) {
  console.log('myCommand output: ' + data);
});
child.stderr.on('data', function(data) {
  console.log('tail error output:', data);
})

//data versturen naar het child
child.stdin.write (" een boodschap van de parent voor het child");
```

Meer info over zowel exec als spawn is te vinden op http://nodejs.org/api/child_process.html

Voorbeeld:

1. Maak een javascript file aan met naam increment.js. Deze file accepteert een getal vanuit de console via stdin:
`process.stdin.on ("data", function (data) { });`
2. Controleer de ontvangen data of het wel degelijk om een getal gaat (validatie!)
`number = parseInt(data);`
3. Verhoog het getal met één en schrijf het uit in de console
`process.stdout.write()`
4. De file increment.js wordt ons child process. Roep dit child process op met een spawn. Om de javascript file uit te voeren maken we gebruik van het "node" process:
`var spawn = require ('child_process').spawn ;
var child = spawn('node', ['increment.js']) // de javascript file als arg;`

5. Roep om de seconde "child" op waarbij je een random getal doorgeeft aan increment.js met
`child.stdin.write(Math.floor(Math.random() * 1000));`
6. Schrijf het ontvangen getal van child naar de output met een callback functie:
`child.stdout.on ('data', function(data) { })`
7. Sluit na 10 seconden het volledige process af. Hierbij breekt je het child process af met
`child.kill();`
8. De methode kill zorgt eveneens dat het child een "signal" uitstuurt. Visualiseer dit signaal in de console.
`child.on('exit', function(exitcode, signal) { })`

9.3 De volgorde van callback taken beïnvloeden(flow control)

Om een correcte program flow te bekomen, moet elke callback functie de volgende functie of zijn parallelle callback functies kennen. Bij foutieve volgorde kunnen scope problemen ontstaan, waarbij variabelen nog niet gedefinieerd zijn. De duurtijd van een callback kan trouwens onderhevig zijn aan variaties in tijd. Combineer dit met geneste callbacks en het wordt snel onhandelbaar. Daarnaast beschikt elke callback over een error controle (if (error)). Dit betekent repetitieve code binnen elke callback. Het geheel kan zo weer heel complex ogen. Het kan eenvoudiger. Ofwel bouw je zelf een generisch flow control systeem, dat bijhoudt wat de volgende callback is en deze oproept. Ofwel wordt gebruik gemaakt van externe modules zoals

- [Async.js](https://github.com/caolan/async) (<https://github.com/caolan/async>) van McMahon. Bemerk dat Async.js één van de meest gebruikte node modules is, die toelaat de samenloop of volgorde van callbackfuncties te beïnvloeden.
- Step.js (<https://github.com/creationix/step>) van Caswell
- Asynquence.js (<https://github.com/getify/asynquence>) maakt gebruik van promises wat het geheel nog leesbaarder kan maken.

9.3.1 Generisch flow control systeem

Van een generisch flow control systeem verwachten we verschillende zaken:

- een manier om de volgorde van de callbackfuncties te bepalen,
- een manier om de resultaten van elke callbackfunctie bij te houden voor verdere processing,
- een manier om samenloop van het aantal callbackfuncties te beheren om voldoende systeem resources te behouden.
- een manier om te bepalen dat alle callbacks afgehandeld zijn.

Om dit uit te werken kan je bvb. een functie cascade aanmaken met als arg een array waarin alle callback functies terecht komen in de juiste volgorde:

`cascade ([function doeCallback1() { } , function doeCallback2() { } , ...]).`

Eén voor één kunnen de argumenten van de cascade functie opgehaald worden. De Array.prototype.shift() methode biedt deze mogelijkheid. Shift haalt één voor één de items op en

verwijderd deze uit de Array. Een "return" brengt het programma terug naar de eventloop na het beëindigen van een taak.

Een voorbeeld:

```
//asynchroon uit te voeren taak (return getal * 10 na 1 sec)
function asyncTask(input, callback) {
    //na één second wordt het resultaat aan de callback bezorgd
    console.log('doe iets met \'' + input + '\' en kom terug (timeout) na 1 sec');
    setTimeout(function () { callback(input * 10); }, 1000);
}
//finale taak: toon resultaat van alle taken:
function final() { console.log('Finale output:', output); }

//Test data: Haal de cijfers serieel of parallel op om met 10 te vermenigvuldigen.
//Na uitvoer worden de getallen (de taak) verwijderd.
var inputs = [1, 2, 3, 4, 5];
var output = [];
```

SERIES UITVOER (async na elkaar)	PARALLELE UITVOER (= real async)
<pre>function series(item) { if (item) { asyncTask(item, function (result) { //resultaat bijhouden output.push(result); // eerste item verwijderen return series(inputs.shift()); }); } else { return finalTask(); } } //returnt en verwijdert eerste element. series(inputs.shift())</pre> <p>Trager maar volgorde van async taken is gegarandeerd.</p>	<pre>inputs.forEach(function (item) { asyncTask(item, function (result) { output.push(result); if (output.length === inputs.length) { finalTask(); } }); });</pre> <p>Sneller maar zonder garantie van volgorde.</p>

Oefening

1. In het Loader object zorgen we met Math.random() voor een variërende delay om de users op te halen. De snelste oplading komt eerst in een zuivere async omgeving:

```
fout emitted in loader: ERROR in loadArrayAsync
Nieuwe user: P6 is loaded na 29 msec.
Nieuwe user: P4 is loaded na 152 msec.
Nieuwe user: P7 is loaded na 339 msec.
Nieuwe user: P3 is loaded na 368 msec.
Nieuwe user: P8 is loaded na 556 msec.
Nieuwe user: P9 is loaded na 620 msec.
Nieuwe user: P2 is loaded na 733 msec.
Nieuwe user: P1 is loaded na 820 msec.
ctor tijd bedraagt: 842
```

2. Door het gebruik van shift kunnen de taken in volgorde (maar nog altijd asynchroon opgehaald worden). De oorspronkelijke volgorde van ophalen wordt behouden. Het error wordt onmiddellijk teruggegeven.

```
Error emitted door loader: ERROR
Output array - seriële loader: in 4090 msec.:
[ 'P1 is loaded in 188msec.', 
  'P2 is loaded in 462msec.', 
  'P3 is loaded in 759msec.', 
  'P4 is loaded in 649msec.', 
  'P6 is loaded in 114msec.', 
  'P7 is loaded in 591msec.', 
  'P8 is loaded in 351msec.', 
  'P9 is loaded in 909msec.' ]
```

3. Zorg tot slot dat beide taken na elkaar uitgevoerd worden in eenzelfde programma.

```
Error emitted door loader: ERROR
Nieuwe user: P2 is loaded in 58msec.
Nieuwe user: P9 is loaded in 375msec.
Nieuwe user: P1 is loaded in 439msec.
Nieuwe user: P7 is loaded in 605msec.
Nieuwe user: P6 is loaded in 649msec.
Nieuwe user: P4 is loaded in 798msec.
Nieuwe user: P3 is loaded in 863msec.
Nieuwe user: P8 is loaded in 850msec.
Async tijd bedraagt: 892 msec.

-----
Error emitted door loader: ERROR
Output array - seriële loader: in 4437 msec.:
[ 'P1 is loaded in 505msec.', 
  'P2 is loaded in 769msec.', 
  'P3 is loaded in 10msec.', 
  'P4 is loaded in 238msec.', 
  'P6 is loaded in 886msec.', 
  'P7 is loaded in 224msec.', 
  'P8 is loaded in 173msec.', 
  'P9 is loaded in 654msec.' ]
```

Variaties van beide methodiekien (series / parallel), zijn zeker niet onbedenkelijk:

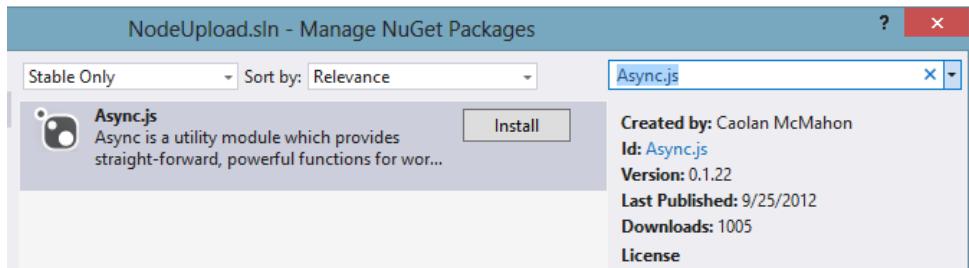
- Het aantal parallelle taken begrenzen op basis van een teller. Slechts een beperkt aantal taken worden parallel uitgevoerd.
- In plaats van inputs (variabelen) kunnen een reeks callbackfuncties aangebracht worden.
- Error controle (error first formaat) mag niet vergeten worden.

Het is echter onnoodig dit allemaal zelf in code aan te maken, gezien dit uitgewerkt is in de npm module "Async.js" met extra mogelijkheden.

9.3.2 Async.js

Vooraf dien je de library te installeren (npm install async) en op te halen in je module (var async = require("async")).

Referentie: <https://github.com/caolan/async>



Daarna zijn een twintigtal uitvoeringsmogelijkheden beschikbaar om de flow te beïnvloeden. Er wordt van deze mogelijkheden verder in de cursus gebruik gemaakt. De methodes van `async` bevatten twee argumenten. Een eerste argument "tasks" bevat een Array of een object van de uit te voeren functies, die elk een callback argument bevatten. Het tweede argument is de finale callback functie, die uitgevoerd wordt na de voltooiing van de series van functies of die bij een error opgeroepen wordt. Bij een fout worden alle nog niet uitgevoerde callback functies kortgesloten en wordt onmiddellijk de final callback opgeroepen. Met deze finalcallback functie wordt zo veel repetitieve code verhinderd (vb. error handling).

De meest gebruikte mogelijkheden:

<code>async.series(tasks, [finalCallback])</code>	De tasks worden na elkaar uitgevoerd <u>in de volgorde</u> van de tasks array/tasks object. De taken wachten op elkaar om uiteindelijk de final callback uit te voeren.
<code>async.parallel(tasks, [finalCallback])</code>	De tasks worden uitgevoerd in volgorde van de task array, <u>zonder dat gewacht wordt op de voltooiing</u> van een andere taak. Na voltooien van alle taken wordt final callback uitgevoerd.
<code>async.waterfall(tasks, [finalCallback])</code>	Het resultaat van een callback functie wordt via zijn argumenten <u>doorgegeven</u> naar de volgende callback functie.
<code>async.queue(worker(task, finalCallback), concurrency)</code>	Aan een worker object wordt een task en callback toegekend. De <u>workers komen in een queue</u> . Het aantal workers dat parallel verwerkt wordt, is bepaald door het concurrency getal.

Voorbeelden zijn te vinden op GitHub: <https://github.com/caolan/async>

Oefening

Gebruik de `async` library om stap 3 van vorige oefening uit te voeren.

9.3.3 Het gebruik van promises

Het is duidelijk dat node.js steunt op asynchroon werken en asynchroon ophalen van data. Binnen de workflow van asynchrone taken past het dan ook om javascript promises aan te brengen. Promises zorgen dat de flow en de volgorde van async callbacks leesbaar wordt samen met fout behandeling.

Info: <http://www.html5rocks.com/en/tutorials/es6/promises/>
<https://strongloop.com/strongblog/promises-in-node-js-with-q-an-alternative-to-callbacks/>

1.1.1.1 *Wat is een promise?*

Promises bevinden zich in 4 mogelijke stati:

- Een successvol 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 : `var promise = doSomethingAsync();`

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" wordt in code aangebracht om over te gaan naar een resultaatsactie van de promise. Dit kan een succesvolle "of" falende actie zijn. Slechts één van beide wordt geactiveerd:
`promise.then(onFulfilled , onRejected).`

Een promise zelf is bijgevolg op zich geen data maar kunnen we bekijken als *een wrapper of functie die het resultaat van een asynchrone taak returnt*.

1.1.1.2 *Waarom een promise gebruiken?*

De `async` syntax wordt vooreerst veel eenvoudiger bij gebruik van een promise. De volledige if structuur voor foutcontrole wordt vervangen door een `then` met twee callbacks:

`//traditionele calllback syntax`

```
fs.readFile(fileName, null, function(err,data) {  
    if (err) {return console.error(err)}  
    else { console.log (data)}  
});
```

`//vernieuwde en eenvoudiger promise syntax, wel met de errorhandler als tweede(!) argument.`

```
var promise = fsPromise.readFile();  
promise.then(console.log (data), console.error(err))
```

De "then" betekent eveneens een oplossing voor de onleesbaarheid door de "Pyramid of Doom" bij het uitvoeren van een workflow met meerdere of geneste callbacks. Pseudo code voor een reeks van async acties bij het lezen van een file zou er ongeveer zo uit zien:

```
var filedata = fsPromise.exists(fname)    //handler1: bestaat de file?  
  .then(fsPromise.stat(fname))           //handler2: file en file extensie OK?  
  .then(fsPromise.readFile(fname) {     //handler3: resultaat van stap2, lees file  
    console.log(data);  
  })  
  .catch (function (error) {  
    //één(!) catch handler behandelt elke fout van de bovenstaande callbacks  
  })  
  .done();
```

Een handler kan met een return een waarde teruggeven, zodat deze waarde kan behandeld worden door de daaropvolgende thenable functie. Een handler kan ook een promise returnen of teruggeven zodat we de keten van thenables kunnen opbouwen.

Noot: Bovenstaand voorbeeld illustreert een opeenvolging van verschillende taken. Een workflow van promises kan ook taken parallel afwerken. Hiervoor doet men meestal beroep op een library. De syntax kan wel verschillen naargelang de gebruikte library.

Voorbeeld: var parallelPromise = Q.all ([fsPromise.readFile(fname1),
 fsPromise.readFile(fname2)])

Een promise laat gecontroleerde foutbehandeling toe. Er is wel een onderscheid voor errorhandling tussen: promise.then(onFulfilled, onRejected) en promise.then(onFulfilled).catch(onRejected). Door de verschillende foutbehandeling kan een zeer specifieke keten van error handling aangestuurd worden.

- then(onFulfilled, onRejected): slechts één van beide callbacks wordt opgeroepen (een "of" functie). Bij een reject is dit callbackError en wordt callbackOK niet uitgevoerd. Bij een fulfilled callback is dit callbackOK.
- then(onFulfilled).catch(onRejected): de catch is gelijkwaardig aan een speciale then [then(null, onRejected)], die alle bovenstaande fouten opvangt en desnoods een aantal thens skipt (een "en" functie). De catch wordt altijd opgeroepen bij elke reject. Dus ook bij een eerder gebeurde reject. Een catch wordt hierom vooral gebruikt wanneer verschillende async functies in een workflow een gecombineerd resultaat moeten afleveren.

1.1.1.3 *Zelf een Promise maken*

In ES6 kan je zelf een promise aanmaken met de "thenable" javascript Promise constructor, waarbij het argument de fulfilled en rejected callback is. Om in node promises bruikbaar te maken moet gebruik gemaakt worden van een library zoals **Q.js** (<https://www.npmjs.com/package/q>) of **RSVP.js** (<https://github.com/tildeio/rsvp.js>) of een deel van RSVP (= polyfill te installeren via npm install **es6-promise**).

```
var Promise = require('es6-promise').Promise;  
  
/* 1. Maak een promise voor fs.readFile door gebruik van "resolve" en "reject" */  
function read(url) {  
  // Return(!) een promise (wrapper met twee cb's).  
  return new Promise(function (resolve, reject) {
```

```
fs.readFile(fileName, null, cbReadFile);

function cbReadFile(error, data) {
    if (error) {
        reject(new Error(error));
    } else {
        resolve(data.toString());
    }
});

/* 2. Consumeer de promisified fs.readFile() */
read(fileName)
// de resolve
.then(function (response) {
    console.log("Success!\n", response);
},
//de reject
.function (error) {
    console.error("Failed!\n", error);
});
```

1.1.1.4 Workflow met een promise library

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 het gebruik ervan eenvoudig is (lage instapdrempel). Elke library met promises kan wel verschillende syntaxen hanteren, zodat het lezen van de API vaak noodzakelijk is.

Er wordt ook blijvend gezocht naar een vereenvoudigde schrijfstijl voor de promises. Zo gebruikt de npm library "asynquence" promises en specifieke keywoorden om de flow van asynchrone taken op te bouwen. Met "val" wordt het fulfilled resultaat weergegeven, met "or" krijg je het rejected resultaat.

```
getSequencefileRead(fileName)
    .val(function (content) { console.log(content.toString()) })
    .or(function (err) { console.error("Error: " + err) });
```

Hierin kan je de val functie kan je zien als een verkorte then schrijfwijze waarbij de callback functie nog moet worden uitgewerkt :

```
.then(function (cb, content) { console.log(cb(content)) })
```

Oefening:

De loader module leent zich voor het maken van een Promise. Voeg een methode (loadArrayPromise) toe die een Promise returnt.

- De promise kent slechts 2 toestanden: ofwel wordt de users Array teruggeven ofwel niet en krijg je een error.
- Bij success zorgt een then sequence zorgt voor het stringifyen naar JSON.

- Een catch vangt alle mogelijke fouten op zonder de applicatie af te sluiten (test door te stringifyen op een niet bestaande arr).

De Promise runnen gebeurt nu als volgt. Bemerk 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);
  });
});
```

10 De API quick tour.

Nu de basiseigenschappen van een asynchroon programmeer model beschreven werden, komt er het op neer om de API van node en de modules beschikbaar via npm te bestuderen en te gebruiken. Algemeen kunnen we de node kernmodules onderverdeelen in vier grote types. De node modules samen resulteren in een API platform (<http://nodejs.org/api/>) voor de volgende aspecten:

- **Basis elementen:** Timers, Modules, Events met zijn events.EventEmitter
- **Processes:** De API's in deze categorie laten toe om een proces op het hoogste niveau te controleren. Zowel variabelen, memory gebruik als externe processen kunnen hiermee behandeld worden. De processen worden onderverdeeld in het hoofdprocess (process) en verschillende child_processes. Beide zijn instanties van EventEmitter.
<http://nodejs.org/api/process.html>,
http://nodejs.org/api/child_process.html#child_process_child_process.
- **Files, Buffers & Streams:** De API's van deze categorie laten toe om files te manipuleren(create, remove, load, write, read) - <http://nodejs.org/api/fs.html> .
Ook streaming is mogelijk. Via streaming kan input naar de output worden doorgestuurd, zonder veel tussenkomst. Hierbij kan tussenin buffering, filtering en transformatie worden toegepast - <http://nodejs.org/api/stream.html>
- **Networking:** laat toe om zowel een http server (class http, class https) als een tcp (class net) server aan te maken. Ook andere types en protocols zijn mogelijk zoals: TLS/SSL en UDP.
<http://nodejs.org/api/http.html>
- **Utilities:** bevat een console ("console") met string commando's en een utilities module ("util")
 - De **console** gebruikt 3 basis streams (stdin, stdout, stderr) , en je kan er natuurlijk json objecten in kwijt:

```
> var a= {1: true, 2 : false}; console.log(a);
{ '1': true, '2': false }
undefined
```


<http://nodejs.org/api/console.html> .
REPL (Read-Eval-Print-Loop) voorziet interactiviteit vanuit de command line.
(<http://nodejs.org/api/repl.html>)
 - De **util** module bevat handige functies zoals:
inspect om tijdens debuggen de inhoud van een object te inspecteren
inherits (*constructor*, *superConstructor*) voor overerven van objecten
<http://nodejs.org/api/util.html>
 - Met **Crypto** kunnen credentials geëncrypteerd worden.
(<http://nodejs.org/api/crypto.html>)
 - Testen en foutbehandeling kan onder andere met **Assert** (unit testing -
<http://nodejs.org/api/assert.html>)

11 Files en streams

11.1 File system

Node beschikt over de mogelijkheid om files te behandelen via zijn "*streaming API*". Dit laat toe om files in een continue vorm te behandelen. Streaming duidt erop dat data reeds behandeld wordt terwijl er nog data ontvangen wordt. Moet je echter op een bepaalde positie in de file aanpassingen doen, dan moet je gebruik maken van de "*file system API*". Filedescriptors (= gelijkaardig aan de UNIX file descriptors) laten dan toe om de file inhoud te behandelen (lezen, schrijven, error handler).

Global Objects "`_dirname`" & "`_fileName`"

Met respectievelijk `_dirname` en `_fileName` kan het absolute path van de map en file van de uitvoerende code opgevraagd worden.

```
console.log(_dirname + '/myText.txt');
```

Noot: Verwar `_dirname` niet met `process.cwd` (wat staat voor current work directory) of `./`:

- `./` returnt de directory van waaruit node runt.
- `_dirname`: het absolute path van de opgeroepen variabele, script of object waarin het zich bevindt.
- `process.cwd()`: het absolute path waarin het process runt/werkt. (= dat process kan gestart zijn vanop een andere folder). Dit kan bij sommige IDE's ingesteld worden als project property (VS2015).

Path beheer met "Path"

Om het filepath te beheren wordt de "Path" module gebruikt: <http://nodejs.org/api/path.html>
Enkele voorbeelden:

```
var path = require('path');

//normaliseren= correcte slashes volgens operating systeem (vb.: backslash windows)
//alternatief voor windows : escappen met dubbele \\
path.normalize('./node_modules//formidable/'); // node_modules\formidable\

//samenvoegen van pathnames
path.join('./node_modules', 'formidable', 'lib/file.js');
// \node_modules\formidable\lib\file.js

//bestaat de file wel
path.exists() is momenteel verwijderd uit de library. De 'fs' module (file system) moet hiervoor
gebruikt worden. ook fs.exists() is deprecated en werd vervangen door fs.stat() of
fs.access().

//naam ophalen van de directory/map van de file
path.dirname('/node_modules/formidable/example/readme.txt');
// /node_modules/formidable/example

//basisnaam ophalen(= zonder het suffix) van de file, eventueel met suffix filter
path.basename('/node_modules/formidable/example/readme.txt');
path.basename('/node_modules/formidable/example/readme.html', '.html'); //readme
```

Low level file manipulatie met "FS"

Voor het ondervragen en manipuleren van files gebruik je de "fs" module:
<http://nodejs.org/api/fs.html>.

Er kan zowel asynchroon als synchroon gelezen worden. Bij asynchroon werken is het laatste argument altijd de callback functie, en het eerste argument van callback is de error functie.

```
//asynchroon lezen (= met callback) met readFile (filename, [options], callback)
fs.readFile(fileName, 'utf8', function (err, data) {
  if (err) console.log(err);
  console.log("\n asynchroon: " + data);
});

//synchroon lezen (blokkeert output en komt eerst in de console)
var file = fs.readFileSync(fileName, 'utf8');
console.log("\n synchroon: " +file);
```

De fs module is een low-level API gebaseerd op file descriptors. Er kan gebruikt gemaakt worden van volgende commands: fs.open, fs.write, fs.read en fs.close. Ook readfile() en readFileSync() maken gebruik van deze commands. Deze commando's worden aangevuld met verschillende argumenten waaronder de flags: r, r+, w, w+, a of a+.

- De r verwijst naar het openen van de file om ENKEL te lezen vanaf het begin van de file. De w verwijst naar het ENKEL schrijven en zelfs aanmaken, mocht de file niet bestaan.
- De r+ en w+ openen de file en laten lezen EN schrijven toe.
- De a staat voor append en schrijft/leest vanaf het einde van de file.

Buffer en Errors

Javascript werkt met Unicode maar gaat minder gemakkelijk om met binaire data. Om wel met raw data en binaire data overweg te kunnen werd bij het Node het Buffer object extra aangemaakt. Node kan hierdoor data zoals een tekst file ontvangen in "*binary*" vorm. Voordat deze data kan ontvangen worden moet de buffer aangemaakt worden (het "*Buffer*" object), waarna met de read functie deze buffer opgevuld wordt. Buffer is te beschouwen als een extra primitief data type, dat niet standaard aanwezig is in raw javascript.

Default gebruikt de Buffer utf-8 maar ook dit kan overschreven worden:
var buffer = new Buffer("Wat tekst", "base64") of new Buffer("Wat tekst", "hex") Omzetten van de ingelezen data naar JSON kan vlot gebeuren: var json = buffer.toJSON(buf)

Node weet echter niet welke data ontvangen wordt in zijn Buffer object en stokeert al de gegevens "*binary*" als een blog in een geheugen ruimte, die bovendien niet gemanaged wordt door de V8 engine. De ontvangen cijfers stellen hex bytes voor. Met *toString()* kan de hex data omgezet worden naar leesbare tekst.

Een Buffer kan je vergelijken met een String object. Een Buffer is wel performanter maar kan beperkt worden door een voorafgedefinieerde omvang (new Buffer(size)) en beschikt over minder functies dan String. Buffer beschikt wel over een methode slice(start[,end]), copy(targetBuffer[,]) en toJSON(). (<http://nodejs.org/api/buffer.html>)

Tijdens het uitvoeren van de commando's moet veel aandacht gespendeerd worden aan de mogelijke errors, en dit op elke plaats waar een lees- of schrijfactie gebeurt. Vergewis er u van dat de filedescriptor acties beëindigd zijn, anders worden de gebruikte buffers niet opgekuisit. Vergeet ook

niet om na het lezen of schrijven een file te sluiten. Vooral bij grotere files is het van belang om de files, de file descriptor (fd) af te sluiten na het beëindigen van de taak, waardoor de geheugen ruimte weer vrijgegeven wordt.

```
fs.open(testFile, 'r', function opened(err, fd) {
//1. indien error
    if (err) { throw err }
//2.buffer aanmaken
    var readBuffer = new Buffer(1024),
        bufferLength = readBuffer.length,
        filePosition = 0; // start positie van het lezen
//3.async uitlezen volgens parameters met fs.read(fd, buffer, bufferoffset, length,
position, callback)
    fs.read(fd,
        readBuffer,
        bufferLength,
        filePosition,
        //callback
        function read(err, readBytes) {
            if (err) { throw err; }
            console.log('leesde een totaal van ' + readBytes + ' bytes');
            if (readBytes > 0) {
                console.log(readBuffer.slice(0, readBytes)); //<Buffer 3C 38 33 ...
            }
            //descriptor afsluiten bij grotere files
            fs.close(fd, function() {
                console.log("file closed");
            });
        }
    );
});
```

Oefening:

Vul de hierboven geschreven code aan (of herschrijf ze met benoemde callback functies) zodat telkens bij het openen van de file een tekst toegevoegd wordt aan de file . Gebruik de descriptor "a" van append en vul binnen de callback functie een writeBuffer op:

```
fs.open(myFile, 'a', function opened(err, fd){ ... });
```

De toegevoegde tekst bevat de datum van vandaag: "Laatst gelezen op ... "

Maak gebruik van de API documentatie. Schrijf de volledige code wel zodanig dat je de callback hell vermindert. Zorg voor error afhandeling.

Tip: Maak een functie `append_Date_To_File (myFile, callback)` die achtereenvolgens de volgende taken afhandelt: `open_file() >> fill_buffer() >> close_file()`

Oefening:

Maak een programma dat toelaat de files van een bepaald type (vb: *.txt) uit een bepaalde map weer te geven.

De map en file extensie worden meegegeven vanuit de console en opgehaald via `process.argv()`.

1. Haal de dirname op bij argv en zorg voor een default map.

```
args = process.argv[2] ? process.argv.splice(2): ["D:/BE-files/data"];
```

2. Controleer of de map bestaat en normaliseer het path (werk asynchroon)
`fs.stat(path.normalize(arg) , cbStat);`
3. Lees de map uit:
`fs.readdir(arg , cbReaddir);`
4. Controleer of het een *.txt file is:
`if (file.indexOf("txt" , this.length - "txt".length) !== -1) {}`
5. Return de opgehaalde filename via een callback en test op mogelijke errors

11.2 Lezen en schrijven van streaming data

Kleine teksten rechtstreeks binair uitlezen kan. Dit lezen gebeurt via een buffer. Geheugen ruimte moet hiervoor worden vrijgegeven om in één keer de file als een blog in te lezen. Een buffer kan hierdoor snel opgevuld en uitgelezen worden. Maar als het over grotere files gaat, kan al het beschikbare geheugen snel vol geraken, zeker als het nog over meerdere gelijktijdige gebruikers gaat of trage verbindingen. In dit geval gebruikt men beter streaming, waarbij data verwerkt wordt terwijl nog data ontvangen wordt. Default gebruikt node ook voor deze techniek buffers.

Node implementeert readable *streams* (inbound) en writable *streams* (outbound). De streams kunnen afkomstig zijn van bijvoorbeeld een TCP-server, HTTP-server, een database query of een file (als child process). Zo is een http request een readable stream. Een http response is een writable stream. Verschillende node objecten en dus niet alleen het filesystem, implementeren m.a.w. het stream object.

Het streaming kan (net zoals bij het lower level file system) gebeuren met synchrone of asynchrone functies:

		Synchroon	Asynchroon
File system	Gebruikt en vult de Buffer volledig	<code>fs.readFileSync(filename, [options])</code>	<code>fs.readFile(filename, [options], callback)</code>
Streaming	Gebruikt deel van de Buffer	<code>fs.readSync(filename, [options])</code>	<code>fs.readStream(), fs.createReadStream(fname,[options], callback)</code>

Streams zijn EventEmitters, wat betekent dat ze afgehandeld worden met events. Niet alle data moet zo in het geheugen vooraf opgeslagen worden. Readable streams emitten typisch volgende events en beschikken daarnaast over een aantal functies. Ze worden beschreven in de API voor stream consumers: <http://nodejs.org/api/stream.html>

READABLE STREAMS:	
EVENTS en EVENTCALLBACK functies	FUNCTIES (API voor reading consumers)

<code>on('data',callback)</code>	listener voor het lezen, bewerken van streams	<code>readable.pause()</code>	Wordt afgvuurd bij een volle Buffer. Pauseert alle data events.
<code>on('error',callback)</code>	listener voor fout behandeling	<code>readable.resume()</code>	Gaat verder met de data events.
<code>on('end',callback)</code>	listener bij ontvangst van een EOF (of FIN in TCP).	<code>readable.destroy()</code>	Sluit af waardoor de stream geen enkel event meer triggert.
		<code>readable.pipe()</code>	Streamt het leesresultaat in een writable (zie verder)

<u>WRITABLE STREAMS:</u>			
EVENTS en EVENTCALLBACK functies		FUNCTIES (API voor writing consumers)	
<code>on('drain',callback)</code>	Listener voor een ledige buffer	<code>writable.write()</code>	Schrijft readable weg. Gaat samen met aanwezigheid van readable data event.
<code>on('error',callback)</code>	listener voor fout behandeling	<code>writable.end()</code>	Stop het schrijven. Gaat samen met readable end event.
		<code>writable.destroy()</code>	Net zoals readable destroy.

Streams zijn duidelijk event emitters waar de readable en writable samen werken. Eigen streams aanmaken gebeurt door het implementeren van de API.

Readable Stream (als event emitter)	Writable stream
<pre>var stream = require("stream"); var sr = new stream.Stream(); sr.readable = true; //implementeert API sr.emit("data", "Dit is mijn data"); sr.emit("end", "Einde van mijn data.")</pre>	<pre>var stream = require("stream"); var sw = new stream.Stream(); sw.writable = true; //implementeert API sw.data = ""; sw.write = function (d) { sw.data += d }; sw.end = function (d) { if (sw.data){ } };</pre>

Streams lezen of schrijven

Node bevat zelf een reeks streaming objects met streaming methodes. Zo kan een file stream aangemaakt worden met `fs.createReadStream()`. Bij het lezen van de file kunnen verschillende opties meegegeven worden, waardoor de tekst bijvoorbeeld niet als bytes via een buffer gelezen wordt maar als geëncodeerde UTF8 tekts:

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

var readableStream1 = fs.createReadStream('testFile.txt');
var content = '';

readableStream1.on('data', function(chunk) {
    // chunck is beschikbaar bij een volle buffer als een reeks van bytes;
    //Default wordt de volledige file ingelezen.
    //Bij einde wordt het event on('end', ..) afgevuurd, wat kan
    // gebruikt worden om een callback functie aan te sturen.
    console.log('ontvangen data:', content +=chunck);
});

var readableStream2 = fs.createReadStream('testFile.txt', { flag: 'r',encoding: 'utf8',
start:22});
//readableStream2.setEncoding('utf8');
readableStream2.on('data', function(chunck) {
    // data is een 'leesbare' utf8-encoded string;
    console.log('chunck ontvangen utf8 data', content +=chunck);
});

readableStream2.on('end', function() {
    //indien callback : done(content)
    console.log('volledig ontvangen utf8 data', content);
});
```

Een readable stream moet niet aangemaakt worden vanuit een file. Een stream kan ook opgebouwd worden met eigen data en het push command. `push(null)` toont dat het aanmaken van de stream beëindigd is.

```
var Readable = require('stream').Readable;

var rs = Readable({encoding: 'utf8'}); //abstracte class (gn constructor functie nodig)
rs.push('Een eerste lijn tekst.\n'); //niet write. Push duwt data in de read queue.
rs.push('Een tweede lijn tekst. \n'); //enkel strings kunnen gepushed worden.
rs.push(null); //einde aanmaak Readable

rs.on('data', function(data) {
    // streaming data als een utf8-encoded string;
    console.log('\n Ontvangen utf8 data: \n', data);
});
```

Readable is een abstracte classe, die ook beschikt over zijn eigen `_read([size])` methode (= `stream.Readable.prototype._read`). Deze `_read` methode kan je aanbrengen en customizen naar eigen wensen. Net zoals bij EventEmitters kan je een object (`MyObject`) laten erven van `Readable` en er een `_read` methode aan toevoegen.

```
//MyObject erft van Readable
util.inherits(MyObject, Readable);
```

```
//Aanmaak MyObject met copy constructor
function MyObject(arg) { Readable.call(this, arg); }
MyObject.prototype._read = function () { //this.push(this.arg) ; this.push(null)...}

//Streamen start bij pipe oproep
var myObj = new MyObject("Einde");
myObj.pipe(process.stdout)
```

Pauzeren met readable.pause() voor een "slow cliënt" synchronisatie probleem

Naast een stream.Readable bestaat een stream.Writable. Waar een http request een inkomende of readable stream is, is http response een uitgaande of writable stream. De events waar een writable stream op reageert zijn: drain, error, (un)pipe en finish.

```
var ws = fs.createWriteStream(_fileName);
ws.on("finish", function () { })
```

Wanneer ontvangen data (read) onmiddellijk moet verstuurd worden (write) naar een "tragere" cliënt kan een synchronisatie probleem ontstaan. Dit wordt veroorzaakt doordat de trage cliënt de aangeboden informatie te traag verwerkt. Om dat te vermijden kan een stream gepauseerd worden.

```
var fs = require('fs');
require('http').createServer( function(req, res) {
    res.writeHead(200, { 'Content-Type': 'text/plain' });

    var rs = fs.createReadStream('testFile.txt');
    rs.on('data', function(data) {
        if (!res.write(data)) {
            rs.pause(); //pauseren vh lezen indien buffer vol
        }
    });
    res.on('drain', function() {
        rs.resume(); //verder lezen bij lege buffer
    });
    res.on('end', function() {
        res.end('\n Dit is het einde.');
    });
}).listen(1337);
```

readable.pipe(destination, [options]) gebruiken voor synchronisatie met tragere cliënts.

http://nodejs.org/api/stream.html#stream_readable_pipe_destination_options

Het veelvoorkomend synchronisatie probleem van read/write met verschillende snelheden kan nog eenvoudiger opgelost worden met het pipe() command. Source en destination worden op elkaar afgestemd door: `source.pipe(destination)` of `readable.pipe(writable)`. Dit laat toe om op eenvoudige wijze een kopieer functie of een backup file aan te maken:

```
var fs = require('fs');
var source = fs.createReadStream(_filename);
var target = fs.createWriteStream(_filename + '.backup');
source.pipe(target)
```

Er kunnen ketens van piping opgebouwd worden waarbij een pipe() een nieuwe source wordt:

```
sourceA.pipe(sourceB).pipe(destination);
```

Met pipe() wordt op het einde automatisch een res.end() opgeroepen. Wil je dat verhinderen dat

moet je als optie de end:false maken en rs.on('end') definiëren. Indien dit laatste onnodig is kan de volledig bovenstaande code met rs.pause() door één lijn vervangen worden:

```
rs.pipe(res, { end:true });
```

Meer voorbeelden: <http://nodestreams.com/>

Oefening:

1. Bouw het alfabet op als een readable stream.

Tip: String.fromCharCode(97) geeft het karakter a terug. String.fromCharCode(98) geeft het karakter b terug. Zo kunnen uit cijfers het alfabet opgebouwd worden.

De letters worden in een readable gepushed: rs.push(String.fromCharCode(i));

Pas bij de letter z wordt het resultaat naar de output gepushed : rs.push(null)

en als test uitgelezen in de console (process.stdout) met: rs.pipe(process.stdout);

2. Schrijf dit alfabet weg naar 'alfabet.txt' met een fs.createWriteStream('alfabet.txt').en toon in de console met on 'finish' de boodschap: "alfabet.txt is aangemaakt".

Pipe een transformed stream met stream.Transform

De recente API voor streaming laat ook duplex streaming en transforming toe.

http://nodejs.org/api/stream.html#stream_class_stream_transform.

Duplex streaming implementeert zowel een readable als writable stream. Het vormt de basis voor een transforming stream, die zich bevindt tussen het lezen en wegschrijven van de stream. De origineel gelezen stream kan worden aangepast (getransformeerd) voordat je deze weer weg schrijft. Om deze transformatie uit te voeren moet een _transform functie ingevuld worden op het Transform object. Daarna kan de getransformeerde file met een pipe(transformedFile) opgehaald worden.

```
//1. transformatie object aanmaken
var Transform = require('stream').Transform;
var uppercaseAndColor = new Transform({decodeStrings: false});

//2. _transform methode definiëren:
uppercaseAndColor._transform = function(chunk, encoding, done) {
  done(null,
    function() {
      return( chunk.toUpperCase().fontcolor('red'));
    }()
  );
};

//3. testen
var fs = require('fs');
var source = fs.createReadStream(_filename, {encoding: 'utf8'});
var target = fs.createWriteStream(_transformedFile);

//uitvoeren vh transformatie commando
source.pipe(uppercaseAndColor).pipe(target);
```

Node.js Stream Playground

Streaming is enorm handig en een belangrijk concept in node met vele mogelijkheden zoals bvb. het wegschrijven (pipe) naar een zip file. John Resig maakte een stream playground (<http://nodestreams.com>) waarop verschillende mogelijkheden (zippen, parse JSON, replace string, split string,...) gechained en in code aangemaakt worden.

Zippen kan als volgt:

Node.js Code

```
var fs = require("fs");
var zlib = require("zlib");

// Read File
fs.createReadStream("input/people.csv")
  // Gzip
  .pipe(zlib.createGzip());
```