



**NEW MEDIA &
COMMUNICATION
TECHNOLOGY**

BACKEND DEVELOPMENT



Inhoud

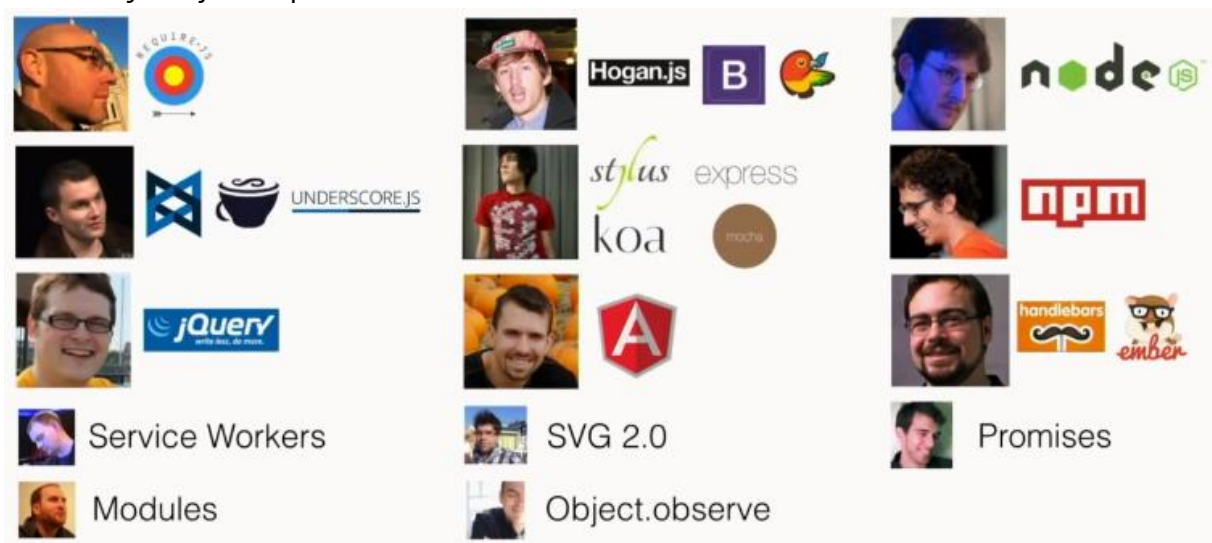
1	Het web (en javascript) wordt real-time	4
2	Introductie en kenmerken	5
3	Installatie, testen en debuggen van node.js	8
3.1	Installatie:	8
3.2	Command Line Interface (CLI)	9
3.3	File(*.js) execution:	10
3.4	Het global process object	11
3.4.1	global	11
3.4.2	process voorziet readable/writable datastreams	11
3.4.3	console	12
3.4.4	timers	12
3.5	Debuggen in node.js	12
4	Alternatieve installatie: IISNode	16
4.1	IISNode als handler	16
4.2	IISNode installeren	16
4.3	Configuratie van de IIS node server	17
5	Node.js in Visual Studio met NTVS	18
6	Asynchroon programmeren met de event loop	21
6.1	Asynchroon (=event-driven) programmeren	21
6.2	De event loop beheert de tasks	23
6.3	Javascript functies maken hun functiescope aan.	25
6.3.1	Functie scope != block scope	26
6.3.2	Closures	26
6.3.3	Zelfuitvoerende functies	28
6.3.4	Doorgeven van argumenten bij closures en zelfuitvoerende functies	28

6.3.5	Module patroon	29
6.3.6	Constructor patroon.....	30
6.3.7	Constructor functies of closures?	31
6.3.8	Objecten uitbreiden	32
6.4	Events en eventemittors.....	33
6.5	Javascript en node.....	36
7	Het module systeem	40
7.1	Node Packaged Modules (npm)	40
7.2	Het CommonJS module systeem.....	42
7.3	Modules en het Module pattern	45

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 technologieën 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 overal ondersteund en begrepen wordt. Nochtans, http werd niet gemaakt voor real time communicatie. Door de manier waarop klassieke http servers ontworpen zijn, is bij http voor iedere request/response cycle een thread nodig die de connectie aanvaardt, afhandelt en terugstuurt. Iedere thread die gestart wordt, heeft een zekere overhead, die in de context van realtime en massive scalability te zwaar en niet langer aanvaardbaar is.

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

2 Introductie en kenmerken



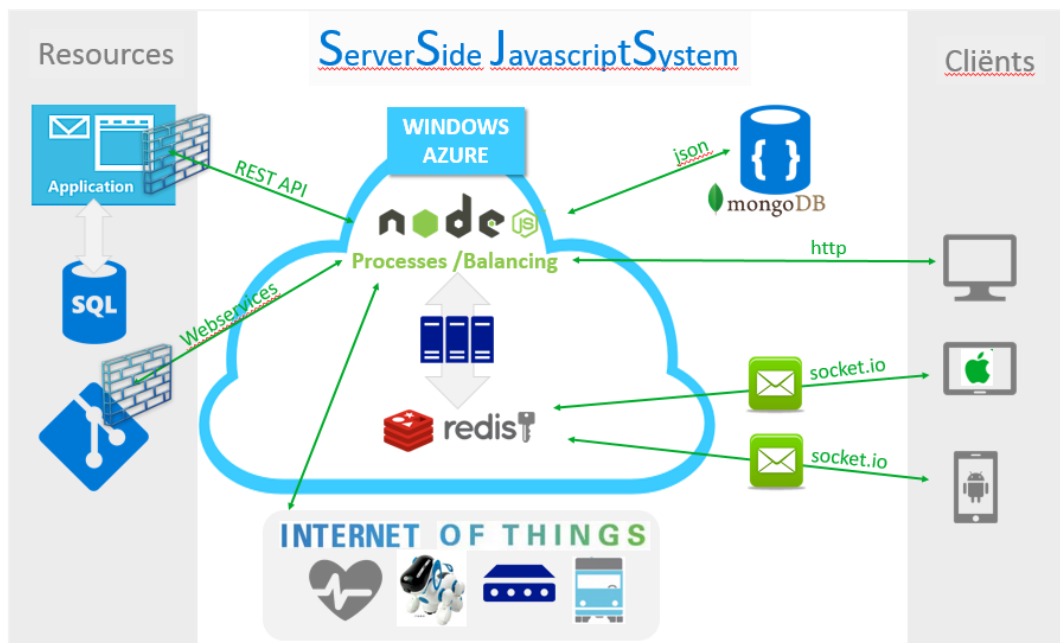
Een opsomming van de belangrijkste Node.js kenmerken:

- Ontstaan in 2009 onder impuls van Ryan Dahl met een presentatie op de Europese 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 omwille van zijn mogelijkheid om asynchroon te werken. Er werd niet gekozen voor een framework, omdat je anders eerst dit framework onder de knie moet krijgen.
- De interpreter maakt gebruik van V8 (Chrome engine), een javascript 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 luistert 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, XMLHTTP...
- Non blocking betekent dat node.js volledig event-driven is:

- Maakt gebruik van callback functies.
In een niet event driven omgeving wordt een programma lineair verwerkt. Zo wordt een database call afgewerkt, waarna het programma verder gaat. Event driven betekent 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 sterkte 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, of financial update applications;
 - updates ontvangen van social network apps.
- Deze sterkte op I/O gebied wordt technisch ondersteund door:
 - sockets voor realtime communicatie,
 - media servers en proxies voor custom netwerk services,
 - JSON webservices,

- cliënt geïntegreerde 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 relaties 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://github.info/>

3 Installatie, testen en debuggen van node.js

3.1 Installatie:

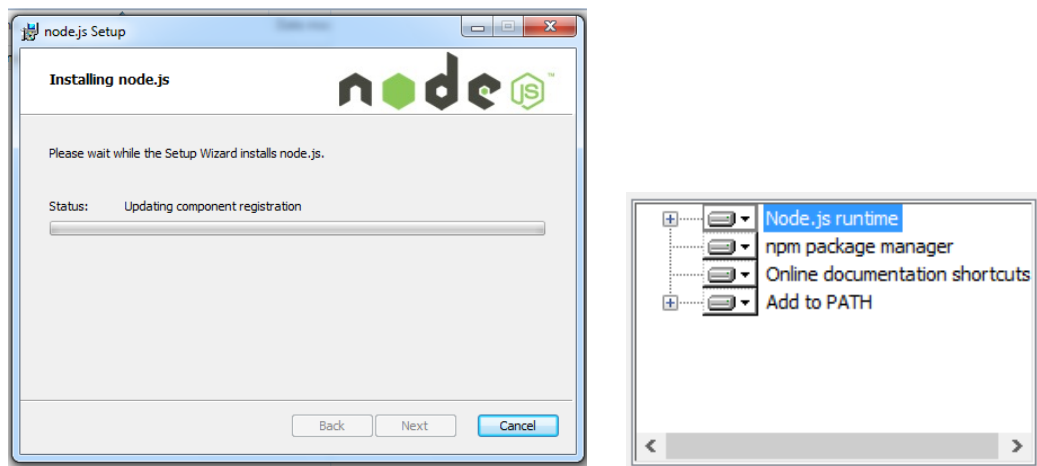
Raadpleeg de officiële site voor installatie (<http://nodejs.org/download/>). Zowel een windows als mac installer package zijn rechtstreeks beschikbaar op de site van node.js



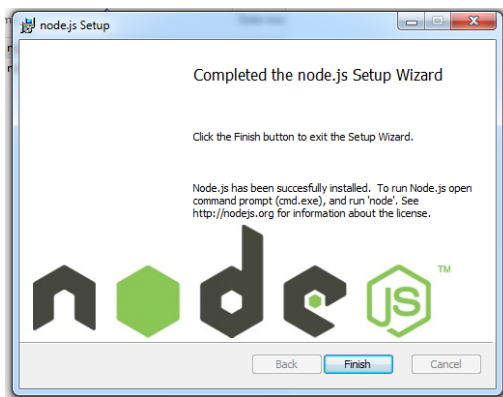
Download en installeer node.js

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.



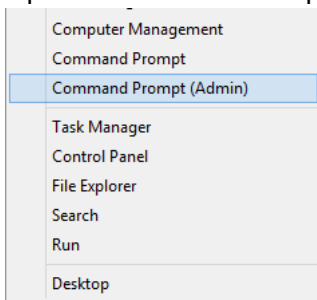
Default installatie map is "nodejs" onder Program Files waarbij zowel een runtime versie, een package manager en documentatie geïnstalleerd worden. Node wordt eveneens toegevoegd aan je PATH.



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\nodejs\

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 uitvoering 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.

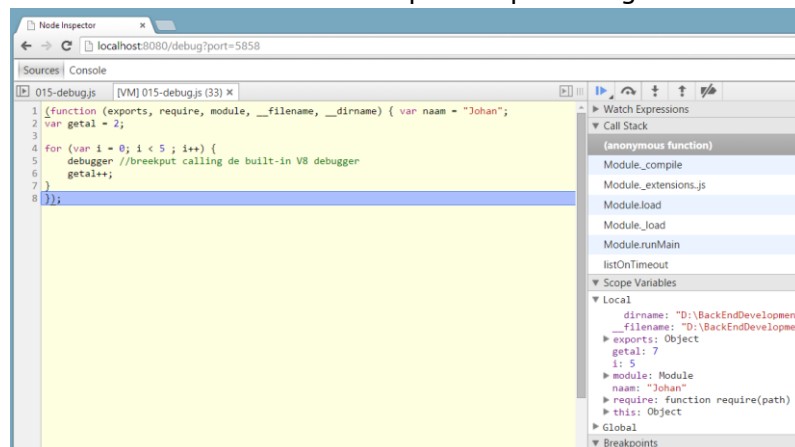
```
Node Inspector v0.7.4
Visit http://127.0.0.1:8080/debug?port=5858 to start debugging.
```

- 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)\node.js>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/):	Onbepaalde 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  Enide - Eclipse Node.js IDE (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 ontwikkeltijd.

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

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

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

c. JSLint of JS Hint:

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

online: testen op <http://www.jshint.com/> //vink nodejs aan

offline: npm install -g jshint//community variant op lint (esLint)
jshint helloworld.js



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 een zelfde 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.

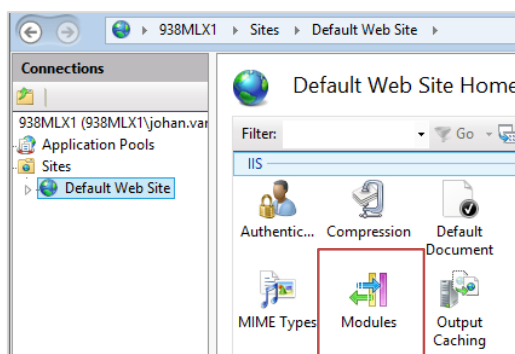
Hoe werkt de handler? Door IISNode als een HTTPHandler – geschreven in javascript- te implementeren worden *aspx requests naar ASP.NET gestuurd, terwijl de node requests bij node.exe terecht komen*. De IISNode handler werkt m.a.w. moeiteloos samen met andere content types, zoals bijvoorbeeld ASP maar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





Modules

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

Group by: No Grouping

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\modrqft.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:



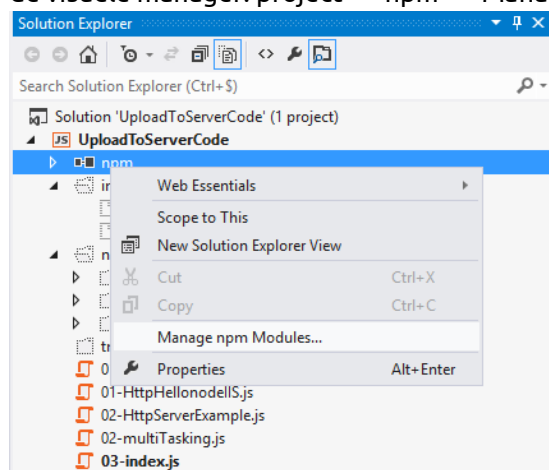
Web Essentials 2015
Adds many useful features to Visual Studio for web developers. Requires Visual Studio 2015 RTM



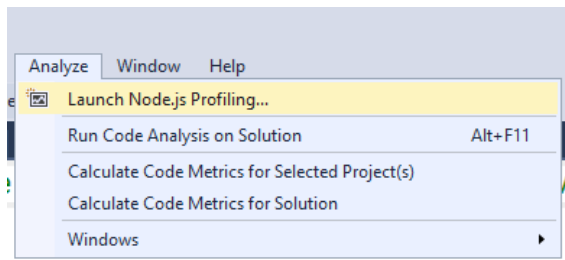
Node.js Tools
Provides support for editing and debugging Node.js programs.

2. Aanmaken van een node project:

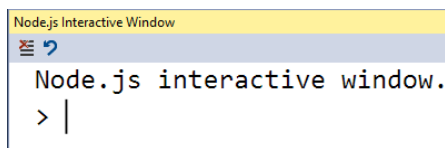
- a. Er kunnen verschillende types van nodejs projecten aangemaakt worden vanuit new project >> javascript:
 - i. een nodejs project, waarbij ofwel een nieuwe nodejs applicatie gemaakt wordt of waarbij een bestaande nodejs app kan geopend worden.
 - ii. een express project
 - iii. een Azure applicatie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



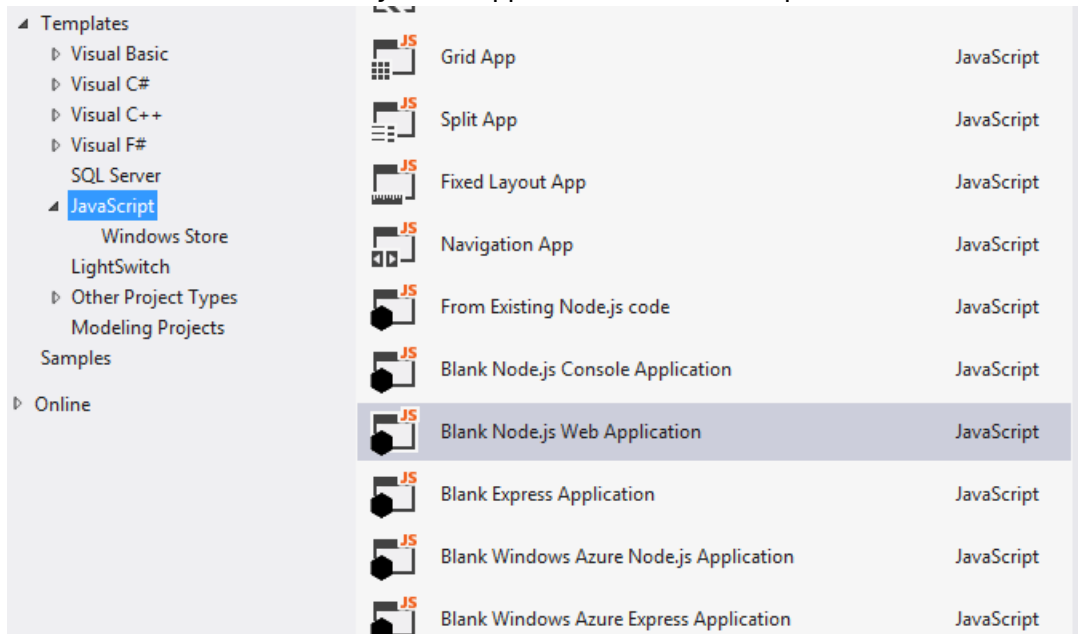
- c. Nodejs profiling kan via het Analyze menu of via een profiler onder het Debug menu.



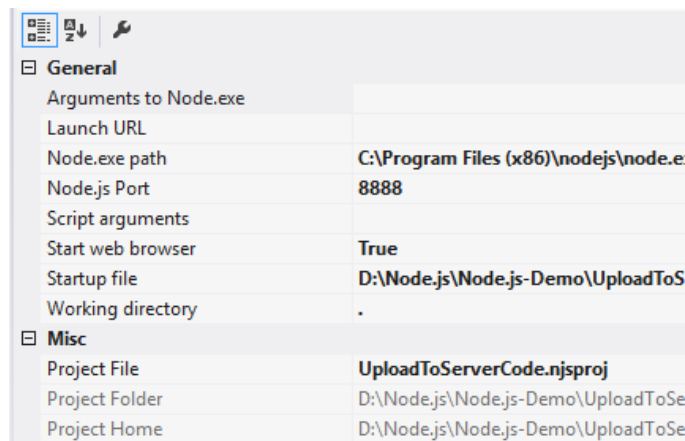
- d. View >> Other windows >> node js interactive window opent een interactieve console.



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.

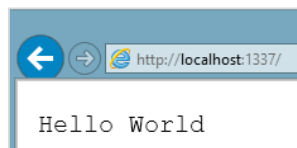
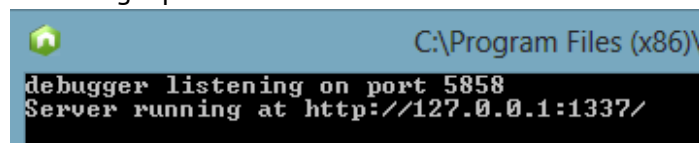


d. Run en debug de toepassing.

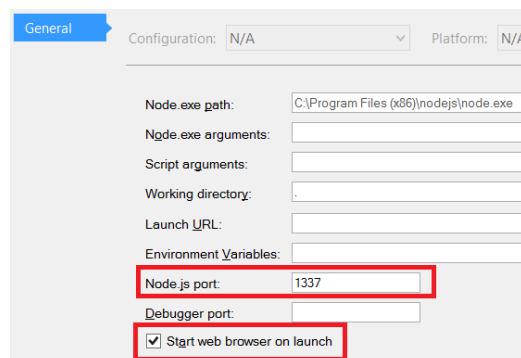
- i. De console opent automatisch maar sluit ook automatisch na voltooide taken. Je kan de console live houden met setTimeout:

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

- ii. Bij een webbrowser toepassing wordt de console en 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 voltooiën 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 simpelweg tijdelijk opgehouden tot zijn actie voltooid is. Voor de programmeur kan het moeilijk worden hierbij controle te houden over de synchronisatie van verschillende processen. 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>	<pre>getData(function (data) { console.log("Asynchroon: " + 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; }</pre> <pre>console.log(processData(2));</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); }); }</pre> <pre>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: Van sync naar async.

We wensen een synchrone functie (load) te herschrijven op een asynchrone manier. De load functie kopieert 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.

Verdere aanvullingen:

- De originele array bevat strings (als ids voor de users).
- Gebruik van een forEach
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).
- Toevoegen van errorcontrole volgens de error first schrijfwijze.

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, efficiënt geheugen gebruik omdat er minder moet geschakeld worden tussen geheugen plaatsen, scaling wordt eenvoudiger. Natuurlijk kan node in de achtergrond zijn eigen threads en afzonderlijk processen gebruiken, maar tussenkomst van de ontwikkelaar is onnodig. De ontwikkelaar kan zich concentreren op de applicatie software eerder dan op synchronisatie software.

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

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



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

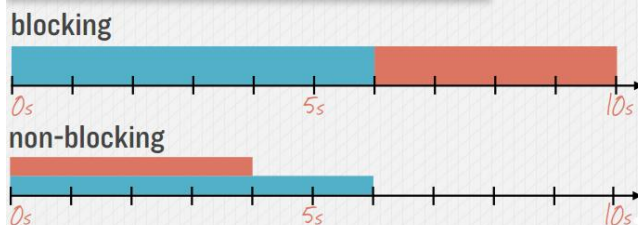
wordt asynchroon met een callback functie:

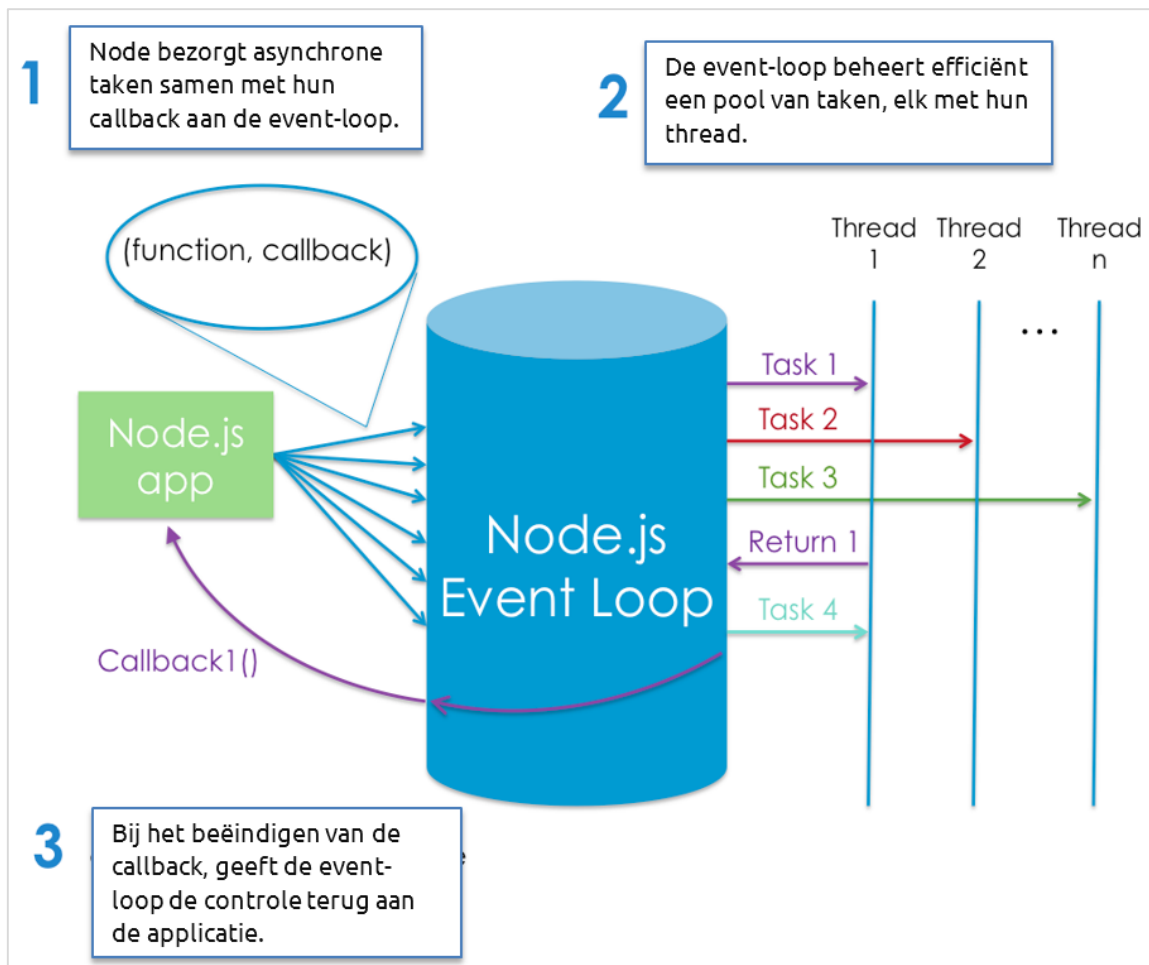
```
db.query("select..", function (result) {
  // use result
});
```

Tijds winst in de queue bij non-blocking:



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





Figuur 2: De Node.js Event-Loop lifecycle

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 variabele.

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 variabele `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 fungeert 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);
```

```

    })
  }
  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:

```

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

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

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

```

```
        duration: duration
    };
};
```

`var 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 asynchrone via het command:
`Loader.loadArrayAsync(users , usersIds, 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:

```
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?

Bij objecten 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. Daarnaast blijven de eigenschappen die via de constructor aangeboden worden toegankelijk. Beide punten, de scope van this en mogelijks toegankelijke eigenschappen, vragen de nodige aandacht van de ontwikkelaar. Anderzijds blijft een object eenvoudig om uit te breiden. Naar performantie bestaat er een te verwaarlozen verschil. Een eenvoudig object kan iets sneller zijn dan zijn evenwaardig module patroon.

Bij closures wordt de status van interne eigenschappen bewaard door de functional scope en blijven aangebrachte variabelen sowieso niet toegankelijk. Dit is iets veiliger naar het gebruik van private variabelen in een javascript omgeving waar alles gemakkelijk publiek wordt. Callback functies zijn een vanzelfsprekendheid voor closures. Vermoedelijk redenen die het gebruik van closures in node bevorderen. Uitbreidingen van closures daarentegen kunnen meer code en herschrijven van een nieuwe functie vragen.

Oefening

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 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 (){
```

```
        }();
```

```
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 encrypteert.

```
var usersIds = ["P1".encrypt(), "P2".encrypt(), "P3".encrypt() . . .
```


6.4 Events en eventemittors

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 methodes

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

Command	Beschrijving
<code>anObject.addListener("eventType", callbackFunction)</code> <code>anObject.on("eventType", callbackFunction)</code>	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.
<code>anObject.once("event", callbackFunction)</code>	Het event type kan slechts één keer opgeroepen worden.
<code>anObject.removeListener("eventType", callbackFunction)</code>	Verwijder een specifiek event type.
<code>anObject.removeAllListeners("event", callbackFunction)</code>	Verwijder alle listeners

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 uitsturen noemt men algemeen in javascript een " *evented object*". Node noemt een object dat events kan uitsturen een *event emitter*. De event emitters maken gebruik van callback functies, net zoals de standaard events, maar kunnen naargelang een status of ontvangen antwoord een andere event aansturen 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 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>)

De EventEmitter class beschikt over eigen static methodes (class methodes).

```
bvb.: events.EventEmitter.listenerCount(emitter, "addedUser")
```

maar laat ook toe om eigen listeners aan te maken (instance methode):

```
bvb.: emitter.on("newListener", function (eventName, listener){ } ).
```

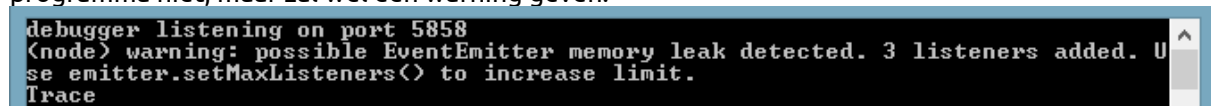
Je maakt je eigen event handlers aan met "emitter.on..." en vuurt het event af met "emitter.emit". Pas op: de volgorde waarin de handlers aangemaakt worden is van groot belang.

Een event kan pas opgeroepen worden NA registratie! De events worden bovendien afgehandeld volgens de volgorde van aanmaak.

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 luisteren naar afgevuurde events en naargelang de status een andere actie ondernemen.

Je kan vaak de applicatie gewoon afwerken met een reeks van callbackfuncties. 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 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
    (node) warning: possible EventEmitter memory leak detected. 3 listeners added. Use
emitter.setMaxListeners() to increase limit.

```

```
//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 evengoed kan dit met een eventemittor. Maak een eventemittor 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 andere zaken te kunnen doen met het resultaat, verwijderen we deze console.log om te vervangen door emitter.

```
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) { ...
```

6.5 Javascript en node

Meest gebruikte javascript methodes in een node applicatie:

Array	<code>some()</code> , <code>every()</code> : assertions <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> : kopiëren <code>splice()</code> : verwijderen operator in: 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 ECMAScript 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 <i>prototype</i>	<code>Array.isArray(array)</code> <code>array.filter(callback)</code> <code>array.forEach(callback)</code> <code>array.map(callback)</code>	Controleren of we een array hebben. 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	<code>Object.create(proto[, props])</code> <code>Object.defineProperty(obj, prop, desc)</code> <code>Object.preventExtensions(obj)</code> <code>Object.hasOwnProperty("aProp")</code>	Nieuw object aanmaken vanuit een ander (proto) Object eigenschappen aanmaken Laat niet toe om eigenschappen toe te voegen.

Get/Set	<pre>var obj = { get iets(){ return "aValue" }, set iets(){ "initialize" } }</pre>	getter en setter syntax
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 prototype	<pre>string.trim() string.trimRight()</pre>	Witruimte verwijderen
vars	<pre>{ const iets ="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

Voorals JSON wordt gebruikt (ipv XML) bij transport van data binnen een node applicatie. 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
```
- String:
Er wordt verwacht dat strings tussen double qoutes 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[0]).msg
```

- **Object :**
Kunnen net zoals arrays genest worden. Ook JSON objects kunnen genest worden:

```
var person = { "person1": { "adres": { "stad": "Kortrijk" } } }
```
- **null:**
JSON ondersteunt niet undefined, maar wel null (net zoals een lege string, lege Array, ..)

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

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

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

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

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 methode, omdat alle mogelijke betekenissen geëvalueerd worden. JSON.parse evalueert enkel valid JSON "strings". Het is een synchrone methode en wordt daarom ook best voorzien van een try/catch/finally om mogelijke fouten op te vangen. Het reviver argument is op zijn beurt een functie met twee argumenten (key/value). Tijdens het parsen kan de key waarde (= een eigenschap) verwerkt worden naar een nieuwe waarde.

```
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
    }
}

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

Oefening

Onze users array is een JSON array geworden.

```
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. ID6 bevat een fout en wordt in runtime omgezet naar P6.

7 Het module systeem

Node.js voorziet vanuit de command line maar 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 ...). Installatie van de modules verloopt vlot via een package manager met naam **Node Packaged Modules** (npm). Deze manager maakt gebruik van nuget. Het installatie commando voor een module is "install". Enkele voorbeelden:
 - Met "**npm install jshint**" installeer je jshint.
(info: jsLint kan je ook runnen vanuit de command line: jshint myfile.js).
 - Met "**npm install colors**" installeer je een module om de kleuren en tekst formaat van de console aan te passen.
 - 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]
bv.: --express -e, -ejs voegt support toe voor een EJS view engine
(<http://embeddedjs.com>), terwijl express default van jade gebruik maakt.
- Zelf modules toevoegen kan met "npm publish".
- 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.
- 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 meer dan 3500 andere npm modules..
- npm kan in twee modes 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 mode te activeren tijdens installatie is een “-g” vlag nodig:
“`npm install -g moduleName`”
- 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. Je gaat eerst naar de betreffende map met het command `cd` (change directory) en voert daarna de link uit (naar de map, naar een specifieke module)
Meer info: <https://www.npmjs.org/doc/>
- Een verzameling van modules kan eveneens geïnstalleerd worden via één json package met naam **package.json**. Deze file “package.json” wordt toegevoegd aan de root van de applicatie. Je kan deze file manueel toevoegen of je doet het via een node command: >`npm init`. Nadien 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 geïnstalleerd worden.

In de package.json file bepaalt de eigenschap “*main*” het startpunt van de applicatie. 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 terech in de eigenschap “*devDependencies*”. 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>

```
{
  "name": "MainApp",
  "main": "./lib/myModule.js",
  "version": "1.0.0",
  "description": "Een beschrijving van de module",
  "author" : {
    "name" : "Me FromHowest",
    "email" : "me@howest.be",
    "url" : "http://www.howest.be"
  },
  "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.

```

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>

```

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 de 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 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, een third party npm module, een zelf aangemaakte module.

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 de zelfde 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 voor het integreren van de modules zijn mogelijk:

Command	Beschrijving
<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.</p> <p>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>module.exports = function() { /* bvb. constructor fctie*/ function doeIets(a,b) { return a*b; } } /* Ook een module pattern kan geëxporteerd worden */ var Customer = (function() { var _login=function(pwd, callback) {}; return { login: _login } })(); module.exports = Customer; /* of alles als een reeks van functies afzonderlijk exporteren */ module.exports.doeIets = function doeIets(a,b, callback) { return a*b; }; module.exports.login = . . .</pre>	<p>Een <i>zelf aangemaakte</i> 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) .</p>
<pre>var http = require('http');</pre>	<p>Inladen van <i>een kern module</i> (= binair beschikbare modules van node) op basis van zijn module naam.</p>
<pre>//relatief adres: var myModule5 var myModule = require('..my_modules/my_module');</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</p>

<pre>var myModule2 = require('./lib/my_module_2.js'); //absoluut adres: var myModule5 var myModule3 = require('C:/lib/my_module_3.js'); var myModule4 = require('C:\\lib\\my_module_4'); //zoekt in node_modules: var myModule5 = require('my_module_5');</pre>	<p>file.</p> <p>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.</p> <p>Is de module beschuikbaar in een andere folder dan is het gebruik van <i>een puntje of twee puntjes een must (!) voor relatieve adressering.</i></p> <p>Wanneer de module beschikbaar is in de standaard "node_modules" folder moet de folder niet vermeld worden. (= geen puntje = zoek in node_modules)</p> <p>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 main gezocht met de naam van de opstart file.</p>
<pre>// in config.js var Config = { connection: 'localhost:test' }; module.exports = Config; // in app.js var config = require('./config.js'); console.log(config.connection);</pre>	<p>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.</p>

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 (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/JJJJ
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 met "require" en "exports" (zie hoger) en is een must do. Het is eveneens aangeraden om afgesplitste modules, die als zelfstandige blackboxes fungeren, af te splitsen in een eigen namespace.

Oefening (Herhaling):

- Maak vooraf een kleine toepassing met de module fs. Dit is een basis module van node voor het behandelen van files. Documentatie van de module vind je dan ook terug op <http://nodejs.org/api/fs.html> en niet op npm.
 - Lees met fs.readFile(filename, [options], callback) een willekeurige tekst uit.
 - 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) { })
```
 - Vul de callbackfunctie van forEach aan.
 - Toon het resultaat in de console.
- Omdat we het lijn nummeren nog willen hergebruiken als afzonderlijke module splitsen we de code ervoor af.

- Maak een file aan voor de module (LineNumbering.js)
- Maak gebruik van een constructor functie om het object 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).

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