



NEW MEDIA &
COMMUNICATION
TECHNOLOGY

BACKEND DEVELOPMENT



Johan Vannieuwenhuyse

Inhoud

1 Het web (en javascript) wordt real-time	8
2 Introductie en kenmerken.....	9
3 Installatie, testen en debuggen van node.js	12
3.1 Installatie:	12
3.2 Command Line Interface (CLI)	13
3.3 File(*.js) execution:.....	14
3.4 Het global process object	15
3.4.1 global.....	15
3.4.2 process voorziet readable/writable datastreams.....	15
3.4.3 console.....	16
3.4.4 timers	16
3.5 Debuggen in node.js	16
4 Alternatieve installatie: IISNode.....	20
4.1 IISNode als handler.....	20
4.2 IISNode installeren	20
4.3 Configuratie van de IIS node server	21
5 Node.js in Visual Studio met NTVS.....	22
6 Asynchroon programmeren met de event loop	25
6.1 Asynchroon (=event-driven) programmeren	25
6.2 De event loop beheert de tasks	27
6.3 Javascript functies maken hun functiescope aan.....	30
6.3.1 Functie scope !== block scope.....	30
6.3.2 Closures	30
6.3.3 Zelfuitvoerende functies.....	32
6.3.4 Doorgeven van argumenten bij closures en zelfuitvoerende functies	32

6.3.5	Module patroon.....	33
6.3.6	Constructor patroon.....	34
6.3.7	Constructor functies of closures?	35
6.3.8	Objecten uitbreiden	36
6.4	Events en eventemitters.....	37
6.5	Javascript en node.....	40
7	Het module systeem	45
7.1	Node Packaged Modules (npm)	45
7.2	Het CommonJS module systeem.....	48
7.3	Modules en het Module pattern	51
8	Process beheer in een asynchroon programmeer model.	53
8.1	Single threaded javascript	53
8.2	Async error handling	55
8.2.1	Try/catch vervangen door domain errors.....	55
8.2.2	Error argument in callback functies.....	56
8.3	Callback Hell = Pyramid of Doom = the Boomerang effect	57
8.3.1	Benoemen van callback functies	58
8.3.2	Distributed events	58
9	Flow control en scheduled processen	61
9.1	Sheduling node processen	61
9.2	Externe processen beheren	63
9.3	De volgorde van callback taken beïnvloeden(flow control)	65
9.3.1	Generisch flow control systeem	65
9.3.2	Async.js.....	67
9.3.3	Het gebruik van promises	69
10	De API quick tour.....	73

11 Files en streams	74
11.1 File system.....	74
11.2 Lezen en schrijven van streaming data	77
12 Servers en netwerking.....	83
12.1 TCP server (<code>net.createServer()</code>)	83
12.2 TCP cliënt (<code>net.connect()</code>)	84
12.3 HTTP server.....	86
12.3.1 HTTP Request in a nutshell:	86
12.3.2 HTTP response in a nutshell.....	87
12.3.3 De HTTP module in node	89
12.4 Node als REST API cliënt	94
12.4.1 Een node GET request met <code>http.request(options, cb)</code>	94
12.4.2 Een node POST request met de “ <code>querystring</code> ” module.	95
12.4.3 Meer request opties met de “ <code>request</code> ” module	98
12.5 Routing opbouwen met de node “ <code>url</code> ” module	99
12.6 Cookies in node	100
12.7 Herhalings oefening.....	101
12.7.1 Doel.....	101
12.7.2 Opgave	101
12.7.3 Oplossing	101
13 Andere server types en protocols.	107
14 Middleware vereenvoudigt een web applicatie	110
14.1 Middelware definitie in node.....	110
14.2 De middleware van <code>connect</code> of <code>express</code> gebruiken.....	112
14.3 Zelf middleware aanmaken.....	113
14.4 Oefeningen op middleware:	113



15 Frameworks: Express	118
15.1 Node Waarom express?.....	118
15.2 Opbouw van een express applicatie:	118
15.3 View Engine (Jade)	122
15.4 CRUD: Users toevoegen via een webform met express	123
15.5 Express en middleware	127
15.6 Oefeningen met Express	129
16 Unit testing	134
16.1 Type testen:.....	134
16.2 assert.js module	137
16.3 Extra asserting test functies met should.js.....	139
16.4 Testen met een dedicated request/response object, dedicated server opties	140
16.5 Testrunner: Espresso of Mocha?	141
16.5.1 describe ... it en asynchroon testen	142
16.5.2 tests runnen	143
16.5.3 Nog meer mocha mogelijkheden:.....	143
16.5.4 Testen automatisch oproepen:	145
16.6 Sinon faket reële returns	145
16.6.1 Spy	145
16.6.2 Stub	145
16.7 Oefeningen met Mocha	146
16.8 Meer mogelijkheden	147
17 Real time web & sockets	149
17.1 De nood aan websockets	149
17.1.1 Wat zijn websockets?.....	150
17.1.2 De Websocket API	153

17.2	Websockets in node.js	153
17.3	Basis webchat met socket.IO	154
17.3.1	Basis chat server en HTML5 chat cliënt.....	154
17.3.2	Socket.io server luistert naar http server	155
17.3.3	HTML cliënt verder afwerken.	156
17.4	Mogelijke uitbreidingen.....	156
17.4.1	json versturen.....	156
17.4.2	Meer message types toevoegen.....	157
17.4.3	socket object bevat rooms.....	157
17.4.4	Namespaces	157
17.4.5	Security.....	157
17.4.6	Multiserver/multiprocess	158
17.4.7	Waar sockets gebruiken?	158
17.5	Gebruik van frameworks en libs bij socket.io	159
17.5.1	Express framework en socket.io	159
17.5.2	Superset: express.io	160
17.5.3	Angular en socket.io	160
17.6	Oefeningen:.....	160
17.6.1	Oefening: images broadcasten	160
17.6.2	Oefening: Geselecteerd flickr image broadcasten.....	161
17.6.3	Oefening: multi-user white board	162
17.6.4	Oefening: Eenvoudige multi-user games	164
17.6.5	Oefening/enkel ter info: Opbouw van een socket game	164
18	Databases	167
18.1	Data stores en node.js	167
18.2	MySQL.....	168



18.3	MSSQL.....	168
18.4	MongoDB	168
18.4.1	Wat en waarom	168
18.4.2	Collections en documents	169
18.4.3	Node en Mongoose.....	170
18.4.4	CRUD mongo via node.....	176
18.4.5	Validatie bij Post en Put verbs:	181
18.4.6	Het gebruik van een repository	183
18.4.7	Meerdere gelinkte documenten	184
18.4.8	Gebruik van positionele operator \$ bij update, delete van collecties.....	187
18.4.9	Mongo, Express4 en socket.ioi	188
18.4.10	Horizontaal scalen (auto-sharding).....	189
18.4.11	Restore en backup een mongodb database.....	189

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



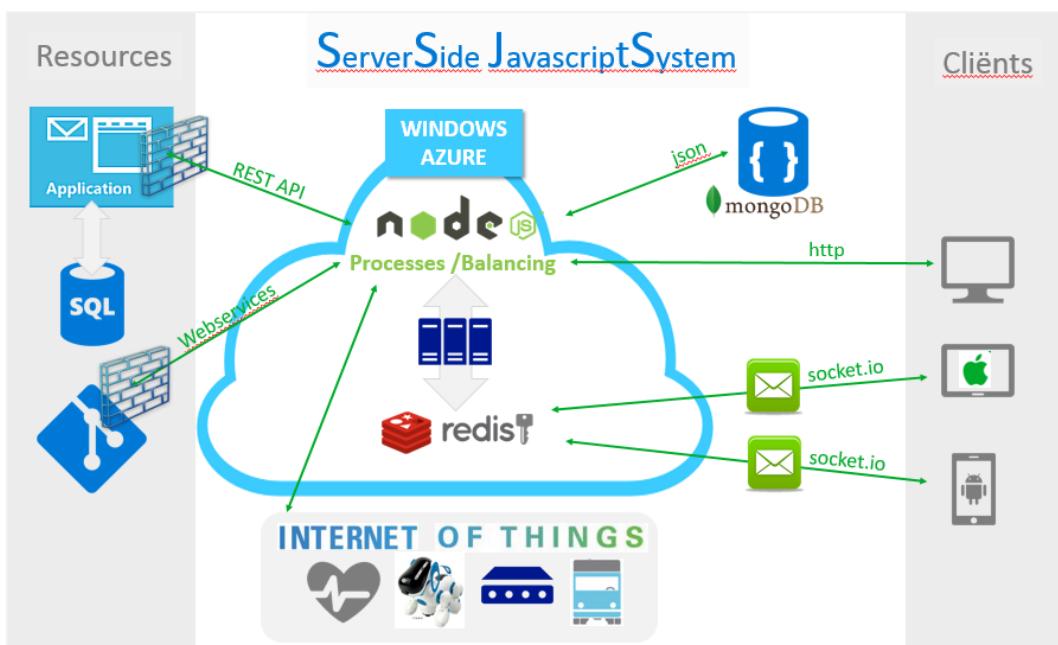
Een opsomming van de belangrijkste Node.js kenmerken:

- 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:

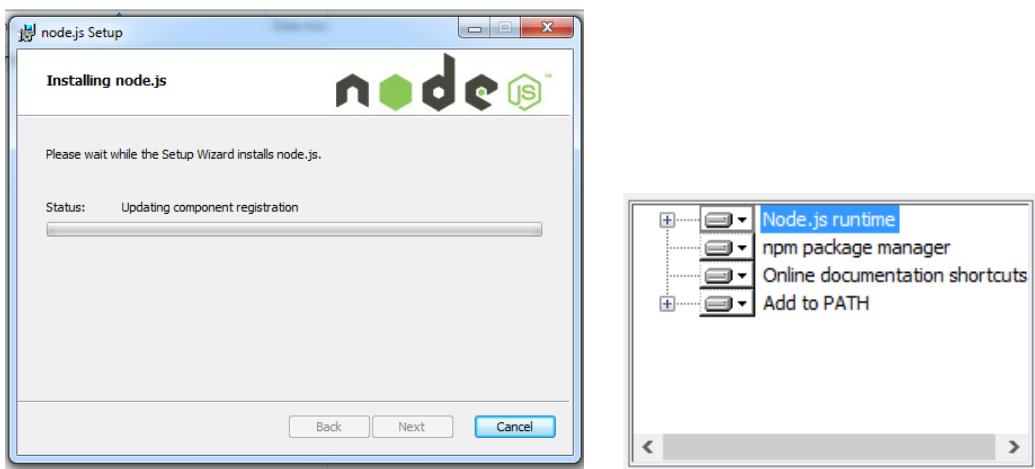
Raadpleeg de officiële site voor installatie (<http://nodejs.org/download/>). Zowel een windows als mac installer package zijn rechtstreeks beschikbaar op de site van node.js



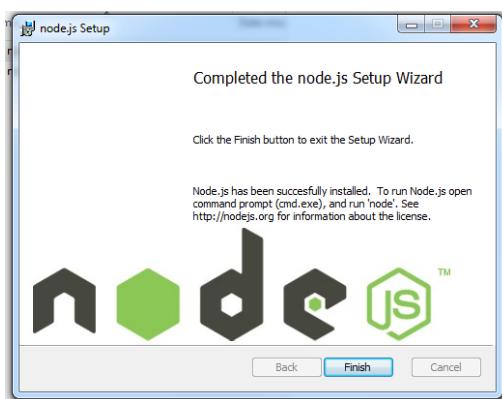
Download en installeer node.js

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.

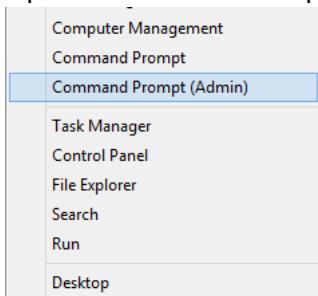


Noot: Open source programma's kunnen het moeilijk maken om downwards compatible te blijven.

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\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 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).

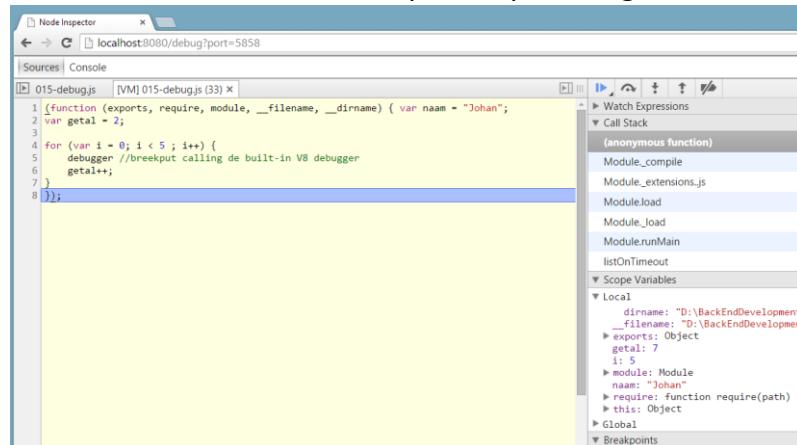
"node --debug-brk HelloWorld.js"

```
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 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 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

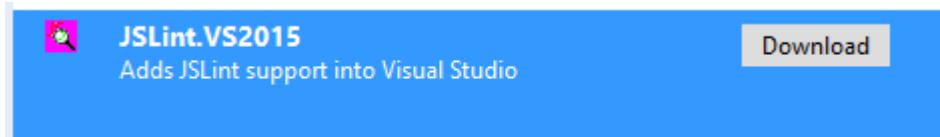
Ter herinnering: JSLint is een code kwaliteitstool voor javascript dat kijkt naar syntax fouten, stijl conventies en structurele problemen (zoals gedefinieerd in <http://javascript.crockford.com/code.html>). Je hoeft niet alle opmerkingen te volgen, soms is bvb. verantwoord om een variabele lager in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

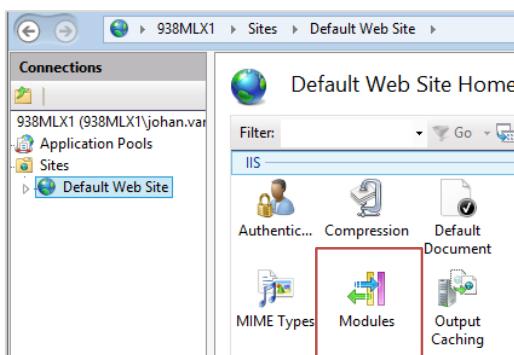
4.2 IISNode installeren

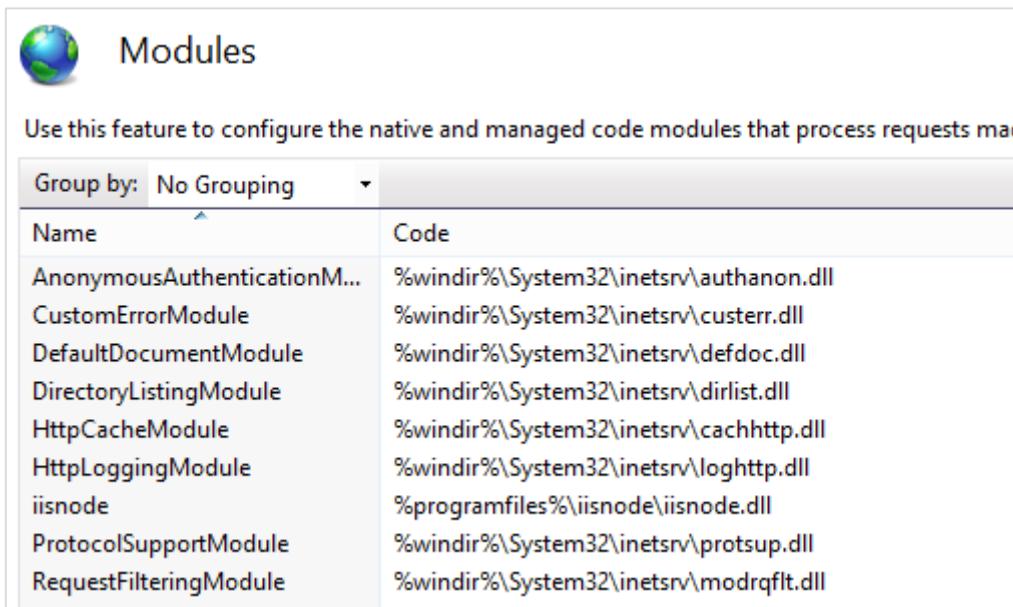
Het installeren van IISNode verloopt minder vlot dan het installeren van node.js. Een beschrijving van de installatie vind je terug op:

<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. At the top, there is a message: 'Use this feature to configure the native and managed code modules that process requests made to your website'. Below this, there is a dropdown menu 'Group by:' set to 'No Grouping'. 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

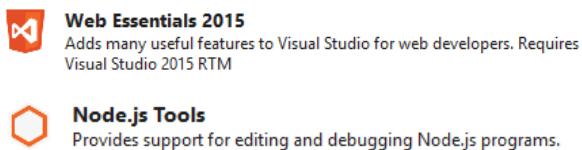
Wanneer de http server runt via iisnode worden poortnummer en domein bepaald vanuit IIS. De node server neemt deze waarden over via zijn process environment: process.env

```
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

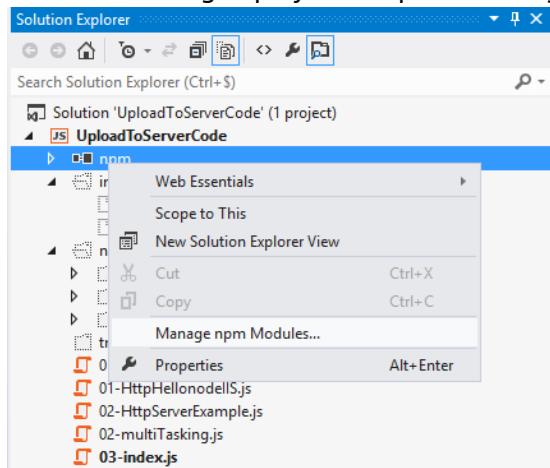
Node.js ontwikkelen kan complex worden, zeker als je er een volledige website mee wil bouwen. Dan komt de vraag naar ontwikkeltools. Combineren met IISNode als handler was een eerste stap naar meer overzichtelijkheid, maar in **nov 2013** zorgde Microsoft voor **NTVS: Node Tools for Visual Studio**. 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:

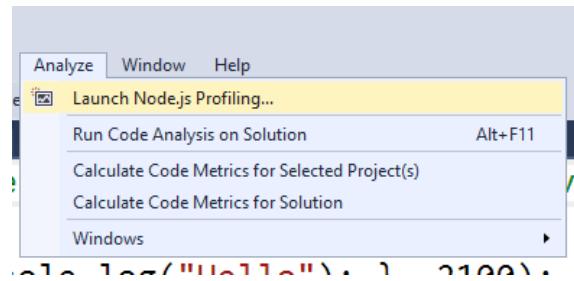


2. Aanmaken van een node project:

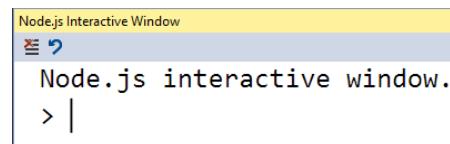
- a. Er kunnen verschillende types van nodejs projecten aangemaakt worden vanuit new project >> javascript:
 - i. een nodejs project, waarbij ofwel een nieuwe nodejs applicatie gemaakt wordt of waarbij een bestaande nodejs app kan geopend worden.
 - ii. een express project
 - iii. een Azure applicatie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



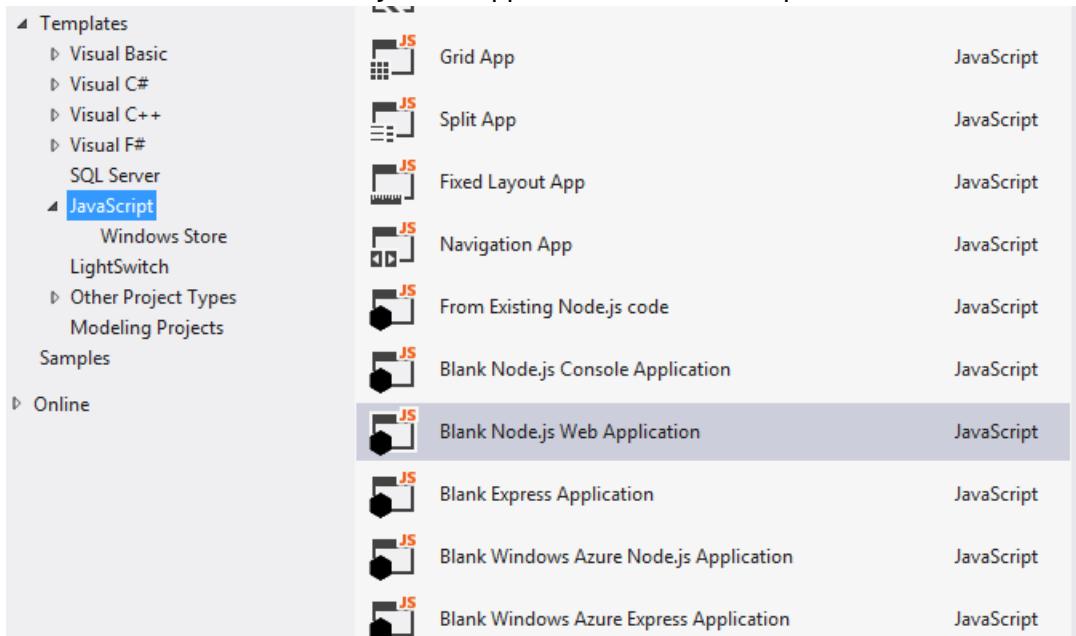
- c. Nodejs profiling kan via het Analyze menu of via een profiler onder het Debug menu.



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

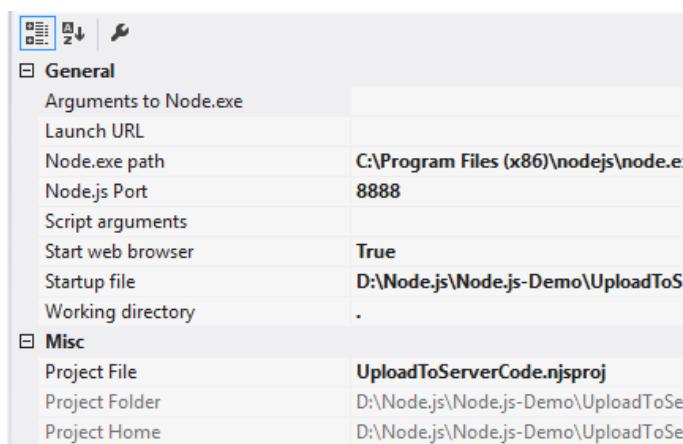


3. Maak een nieuwe "Blank Node.js" Web application aan in een map naar keuze.



4. Test uit door een eerder 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.

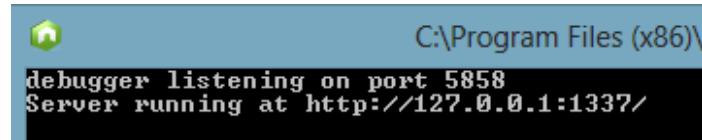


d. Run en debug de toepassing.

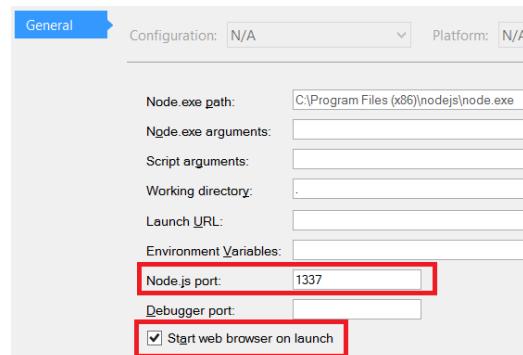
- i. De console opent automatisch maar sluit ook automatisch na voltooide taken. Je kan de console live houden met setTimeout:

```
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>	<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; } 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); }); } 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

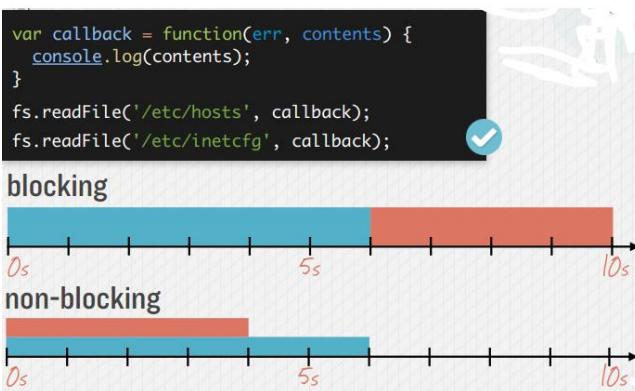
Node.js zorgt dat I/O taken op een asynchrone manier uitgevoerd worden door events.

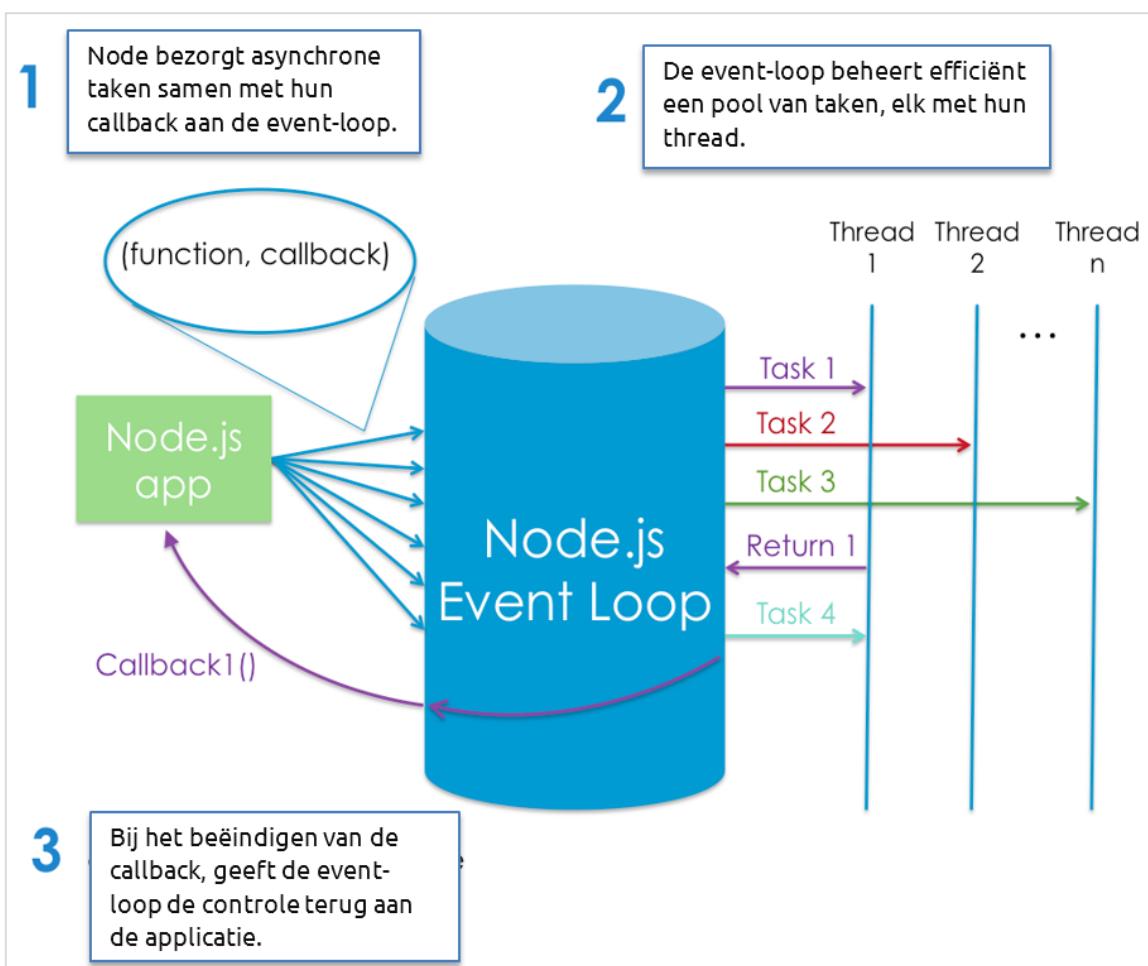
Het asynchroon programmeer model maakt gebruik van callback functies en wordt ondersteund door de event-loop. De event-loop start automatisch op, wanneer je node start en behandelt het volledig beheer van de pool van events en hun callback functies. De event-loop voert simultaan twee zaken uit:

- 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: 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 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 (){    }  
}());  
  
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();
```

Bemerк 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.

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

```
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>	<p>Witruimte verwijderen</p>
vars	<pre>{ const iets ="read only vasteWaarde"; // a geeft hier een ref.error let a = 10; }</pre>	<p>Het command 'let' laat expliciet "block scoping" toe. De variabele is enkele gekend binnen het block { } en is <i>pas gekend op de lijnen na zijn declaratie</i>. De garbage collector kuist sneller op. 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`
Alle string karakters zijn valid, maar sommige verwachten wel een escape (met backslash),

want het blijft javascript. Voorbeeld: single quote ' , double quote " , backslash \ , alle speciale chars zoals \t \n

- Boolean:

Enkel true en false (kleine letters) worden geaccepteerd.

```
{ "boolean": 1 } //is geen Boolean
```

- Array:

Wordt weergegeven met vierkante haakjes (niet new Array()) en kunnen genest worden.
Associatieve arrays resulteren bij stringify in een lege array. Indexed array worden wel gestringified. Je kan daarom best gebruik maken van array.push() om deze aan te maken.

```
{"arr1": [100, true, ["string1", "string2"] ]}
```

Er wordt binnen node veel gebruik gemaakt van zo genoemde JSON arrays. Hierbij verwijst met naar Array's die JSON objecten bevatten. Ze kunnen bvb.gebruikt worden bij sockets om validatie errors van server naar client te sturen.

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

De client 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 serialiseren:

```
//javascript object: {ID: 100, Name: 'Johan', City : "Brugge"}
```

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

De optionele argumenten replacer en space bieden extra mogelijkheden aan.

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

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

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

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

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

```
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" }];
```



NEW MEDIA &
COMMUNICATION
TECHNOLOGY

BACKEND DEVELOPMENT
NODE.JS

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 gelijknamige 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 impliciet *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 expliciet 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 + '/MyTextFile.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) { <u>return</u> process.nextTick(function () { callback(true); //asynchroon }); } //OK: asynchroon <u>return</u> 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:

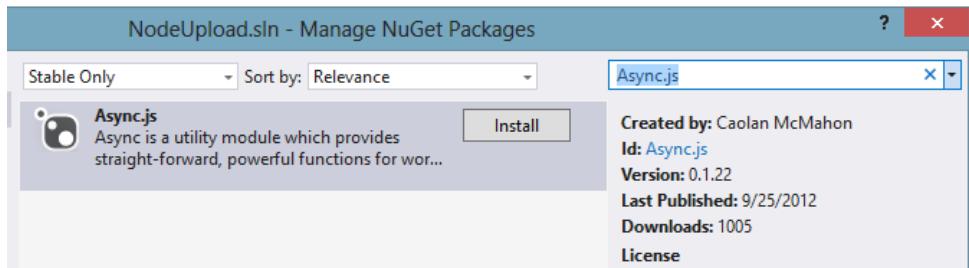
- Het aantal parallelle taken begrenzen op basis van een teller. Slechts een beperkt aantal taken worden parallel uitgevoerd.
- In plaats van inputs (variabelen) kunnen een reeks callbackfuncties aangebracht worden.
- Error controle (error first formaat) mag niet vergeten worden.

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 callback 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.

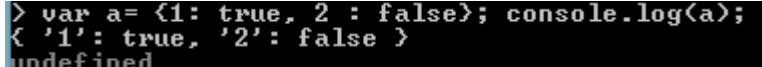
- Een catch vangt alle mogelijke fouten op zonder de applicatie af te sluiten (test door te stringifyen op een niet bestaande arr).

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:

<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 or 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.
- **s** duidt op een synchrone uitvoering
- **x** zorgt dat de actie faalt wanneer de file bestaat.

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 "*binaire*" 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 opgekust. 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) {
  //Het openen van een file return zijn filedescriptor, die de file verder identificeert.
  //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() >> fs.write >> 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 : fs streams, zlib streams, crypto streams, sockets.

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 een 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 (en luisteren) typisch naar volgende events en beschikken daarnaast over een aantal API functies. Deze worden beschreven: <http://nodejs.org/api/stream.html>

READABLE STREAMS:

EVENTS en EVENTCALLBACK functies <code>require("stream").Stream();</code>		FUNCTIES (API voor reading consumers) <code>require("stream").Readable();</code>	
<code>on('data',callback)</code>	listener voor het lezen, bewerken van streams <code>emit('data',...)</code>	<code>pause()</code>	Wordt afgevuurd bij een volle Buffer. Pauseert alle data events.
<code>on('error',callback)</code>	listener voor fout behandeling <code>emit('error',...)</code>	<code>resume()</code>	Gaat verder met de data events.
<code>on('end',callback)</code>	listener bij ontvangst van een EOF (of FIN in TCP). <code>emit('end',...)</code>	<code>destroy()</code>	Sluit af waardoor de stream geen enkel event meer triggert.
<code>on('close',callback)</code>	Wanneer stream of onderliggend object afsluit (vb. fs)	<code>pipe(destination [,options])</code>	Streamt het leesresultaat in een writable (zie verder)

WRITABLE STREAMS:

EVENTS en EVENTCALLBACK functies <code>require("stream").Stream();</code>		FUNCTIES (API voor writing consumers) <code>require("stream").Writable();</code>	
<code>on('drain',callback)</code>	Listener voor een lege buffer om verder te schrijven.	<code>write()</code>	Schrijft readable weg. Gaat samen met aanwezigheid van readable data event.
<code>on('error',callback)</code>	listener voor fout behandeling	<code>end()</code>	Stopt het schrijven. Gaat samen met readable end of writer end event.
<code>on('finish',callback)</code>	Listener voor writer.end();	<code>destroy()</code>	Sluit af en stopt event handling.
<code>on('pipe',callback)</code>			Bij ontvangst van een piped data stream.

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
<code>var stream = require("stream");</code>	<code>var stream = require("stream");</code>

<pre>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 sw = new stream.Stream(); sw.writable = true; //implementeert API sw.data = ""; sw.write = function (d) { sw.data += d }; s.end = function (d) { if (sw.data){ } };</pre>
---	---

Streaming objects

Node bevat zelf een hele reeks streaming objects met streaming methodes: fs, http response ... 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.
    // chunck is een leesbare strings door concatenatie met string .
    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);
});
```

Customised streaming & customised streaming object

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' module (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 beschikt via zijn prototype 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 willekeurig object (`MyObject`) laten erven van Readable. Zo wordt dit een object een Readable Stream en kan je een eigen `_read` methode definiëren op de klassieke manier:

1. Een subclasse aanmaken die erft van het Reabdalbe of Writable prototype.
2. Oproepen van de geërfde constructor met `Readable.call` of `Writable.call`
3. Eigen methodes of properties toevoegen.

```
//Aanmaak MyObject met copy constructor
function MyObject(arg) { Readable.call(this, arg); }

//MyObject erft van Readable (= overname van het prototype)
util.inherits(MyObject, Readable);

//bestaande _read method overschrijven naar eigen wens m.b.v. push
MyObject.prototype._read = function () { //this.push(this.arg) ; this.push(null)... }

//-----
//Streamen start bij pipe oproep
var myObj = new MyObject("EenArg");
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
    });
    rs.on('end', function() {
        //zonder "end" blijft de browser "runnen".
        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 });
```

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));`

Bij de letter z wordt de readable afgesloten: `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());
```

12 Servers en netwerking.

Node.js biedt verschillende netwerk mogelijkheden:

- Opzetten van TCP server en bijhorende cliënts;
- Opzetten van HTTP server en bijhorende cliënts;
- Gebruik van sockets objecten, cookies, form objecten;
- Gebruik van User Datagram Protocol (UDP);
- Beveiligen van de server (SSL of het nieuwere TLS)
- DNS service mogelijkheden

Naast de mogelijkheden, voorzien door node zelf, bestaan verschillende aanvullende third party modules en middleware voor communicatie mogelijkheden.

12.1 TCP server (net.createServer)

TCP (Transmission Control Protocol) is een fundamenteel *communicatie* protocol, bovenop IP (Internet Protocol) en voorziet een transport mechanisme voor de application layer. Het IP adres identificeert een device in het netwerk. Omdat TCP *connection-oriented* is, wordt gegarandeerd dat alle pakketten bij de ontvanger toekomen in de juiste volgorde.

IP4 adressen zoals bvb. 1.140.20.255 bestaan uit 32 bits (4 keer een 8 bit getal) gescheiden door een punt. IP6 adressen zoals bvb. 3ffe:1900:4545:3:200:f8ff:fe21:67cf zijn 128 bits lang (8 keer een getal met 4hex cijfers), waarbij de 8 getallen waarden zijn door een dubbel punt.

Het IP adres 127.0.0.1 wordt gebruikt voor de localhost, en is het default adres bij node.

Poort nummers zijn een 16 bits getal, waarbij de poorten van 0 tot 1023 gereserveerde poortnummers zijn. Poort 443 is gereserveerd voor HTTPS. Binnen het TCP protocol zoekt elk message pakket zijn weg.

De net module (<http://nodejs.org/api/net.html>) laat toe de TCP server aan te maken.

- `createServer([options], [connectListeners])` returnt een TCP server instantie en laat één optie toe (boolean): `allowHalfOpen`. Indien true kan enkel de server de connectie afsluiten. Default staat dit op false en kan zowel cliënt als server de connectie beëindigen.
- De `connectionListener` (of de `callback`) van `createServer` ontvangt als argument een "net.socket", en fungeert als `eventEmitter` voor `on data`, `on error`, `on end`.
- `server.listen()` accepteert 4 argumenten: `server.listen (port, [host], [backlog], [callback])`. Poort 0 gaat opzoek naar een random poort. De backlog verwijst naar het maximum aantal sockets in de connection queue (default 511)
- `de server.close([callback])` verhindert dat nieuwe connecties worden aangemaakt en sluit pas de server als de lopende connectie volledig uitgevoerd werd.

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

//1. aanmaken van de TCP server
```

```

var server= net.createServer({
    allowHalfOpen: false //cliënt of server sluiten verbinding
},
//3. Connection listener
function (socket) {
    //elke binnenkomende connectie verwerken
    console.log("server heeft nieuwe connectie");

    socket.setEncoding('utf8');
    socket.write("Dit is een customised boodschap voor de client.");
    socket.on("data", function (data) {
        //ontvangst van browser headers of van cliënt data.
        if (data) {
            console.log("ontvangen data: " + data )
        }
        return socket.end(); //beëindigt wel de socket
    });
    socket.on("end", function (data) {
        console.log("Goodbye. Client connectie is beëindigd.");
    });
}

//2. TCP server luistert naar poort 1337 (listening listener)
server.listen(1337, function () {
    console.log("luisteren naar poort" + server.address().port);
});

//4. error handling
server.on("error", function (error) {
    if (error.code === "EADDRINUSE") {
        console.log("Deze poort is reeds in gebruik");
    } else {
        console.log("Fout" + error.message);
        // server.close();
    }
});

```

12.2 TCP cliënt (net.connect)

Wanneer in een browser de TCP server opgeroepen wordt met `http://127.0.0.1:1337`, krijg je in de output de header van de webpagina te zien:

```

debugger listening on port 5858
luisteren naar poort1337
server heeft nieuwe connectie
ontvangen data: GET / HTTP/1.1
Host: localhost:1337
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/35.0.1916.153 Safari/537.36
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,nl;q=0.6,lt;q=0.4

```

In plaats van een browser client kan ook in een node een TCP cliënt aangemaakt worden:

```
var net = require("net");
```

```
var client = net.connect(1337, "localhost", function () {
```

```
console.log("client maakt verbinding");
//3. client (connectie) is een stream en kan dus read/write methodes uitvoeren
client.write("hier een boodschap van de TCP client");
});
```

```
server heeft nieuwe connectie
ontvangen data: hier een boodschap van de TCP client
Goodbye. Client connectie is beëindigd.
```

Oefening:

Vul de TCP server aan met volgende eigenschappen:

Zorg dat de socket niet beëindigd wordt op de server (geen socket.end()). Het afsluiten doen we verder vanuit de cliënt.

Vul de TCP cliënt aan met volgende eigenschappen:

- Zorg dat de ontvangen server messages (sockets) uitgeschreven worden in de client console. Gebruik daarvoor streaming met client.pipe(process.stdout);
- Zorg dat bij het intypen van tekst in de client console deze ingetypte tekst naar de server geschreven wordt. Maak hiervoor gebruik van process.stdin.on("data",function (data) { client.write(data); })
- Controleer of de ingetypte tekst niet "quit" is. Is dit wel het geval dan verbreekt de cliënt de connectie met client.end() en wordt dit meegedeeld in de console client.on("close",function () { ... })
- Bij het closen wordt een reconnect() functie opgeroepen. Deze functie zal na een timeout van 2 seconden een connect poging ondernemen. Maak gebruik van een self executing enclosure om net.connect() weer op te roepen.
- Na drie reconnect pogingen blijft de client definitief afgesloten

De TCP server behandelt tot nu toe slechts één socket. We breiden de TCP server uit zodat ontvangende informatie naar alle andere sockets gestuurd wordt (broadcasting). We maken m.a.w. een chat server.

- Maak een Array van sockets: var sockets = [];
- Voeg elke nieuwe socket toe aan deze Array: sockets.push(socket)
- Schrijf een ontvangen boodschap naar alle andere sockets.
sockets.forEach(function (currentSocket) {
 if (socket !== currentSocket) {
 currentSocket.write(data);
 }
});
- Verwijder de afgesloten socket uit het sockets Array bij socket close:
socket.on("close", function (data) {
 var index = sockets.indexOf(socket);
 sockets.splice(index, 1); //1 is het aantal te verwijderen

})

12.3 HTTP server

Het HTTP protocol maakt gebruik van TCP als transport protocol, zodat alles wat betrekking heeft op de TCP server ook geldig is voor de HTTP server. Ook HTTP is een connectie georiënteerd protocol.

Voor het werken met request/response van http maakt node gebruik van de http module:

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

Om uiteindelijk als server te luisteren naar requests op een host en port:

```
var http_serv = http.createServer(handleHTTP).listen(port, host);
function handleHTTP(req, res) { }
```

12.3.1 HTTP Request in a nutshell:

Request methodes:

Op te vragen in node via request eigenschappen zoals *request.method*, *request.url*.

GET	Basis lees operatie voor header en body.
HEAD	Identiek aan GET maar dan zonder de body.
POST	Versturen van nieuwe resources (form data) naar de server.
PUT	Aanpassen van bestaande resource. Bestaat de resource niet, dan wordt hij aangemaakt De volledige resource wordt overschreven.
DELETE	Verwijderen van een resource op de server.
TRACE	De server stuurt de ontvangen data terug.
OPTIONS	Server returns de mogelijke opties naar de client.
CONNECT	Voor het werken via een proxy, die in naam van de client voor de connectie zorgt.
PATCH	Gelijkwaardig aan PUT, maar slechts een deel van de resource kan aangepast worden.

HTTP Method	Idempotent	Safe
OPTIONS	yes	yes
GET	yes	yes
HEAD	yes	yes
PUT	yes	no
POST	no	no
DELETE	yes	no
PATCH	no	no

Idempotent = heruitvoer zonder gewijzigd resultaat.

Safe = caching kan zonder gevolgen voor de resource content gebruikt worden.

Meest gebruikt Request headers:

Op te vragen in node via `request.headers`

Accept	Specificeert de content types die de client kan aanvaarden (application/json, application/xml, tekst/html)
Accept-Encoding	Specificeert een lijst van encoderen en compressie types (gzip, deflate) die de client kan aanvaarden.
Cookie	Bevat alle cookies (tekst) van de client voor de opgevraagde server.
Content-Length	De lengte van het request (bytes).
Host	Domein en poort nummer van de server. (verplicht)
User-Agent	Identificatie van het type client.

12.3.2 HTTP response in a nutshell

Reponse codes

Op te vragen in node via de eigenschap `http.STATUS_CODES[acodenumber]` of `response.statusCode`

200 OK	Succesvolle request
201 Created	Een nieuwe resource werd succesvol aangemaakt
301 Moved permanently	De resource staat op een nieuwe url, die voor herhaalde requests te vinden is in de "Location" response header. De redirect

	gebeurt best niet automatisch en niet voor POST/PUT/DELETE
303 Redirect for undefined reason	Om bvb. een operatie op een ander adres verder af te werken De gevraagde resource kan met een GET opgevraagd worden en is te vinden in de "Location" response header. Redirect is ook voor POST/PUT/DELETE
304 Not Modified	Duidt aan de resource ongewijzigd is en de cache kan gebruikt worden.
307 Moved temporary	Tijdelijke redirect, zodat best de oude url blijvend gebruikt wordt.
400 Bad request	Foutieve request met foutieve of onvolledige data.
401 Not Authorized	De aangeboden credentials laten geen toegang toe.
404 Not found	De requested URL is niet te vinden
418 I'm a teapot	April Fool's day joke.
500 Internal Server Error	Server error.

Response Headers

In te stellen in node via `response.setHeader("Content-Type", "text/html")`

Cache-Control	De tijd in seconden dat de resource kan gecached worden.
Content-Encoding	Gebruikte encodering of compressie
Content-Length	Body reponse lengte in bytes
Content-Type	Het MIME type
Location	Target URL bij een redirect
Set-Cookie	Aanmaken van een nieuw cookie op de client

Terminologie voor een uri:

Protocol	hostname	port	pathname	querystring (search)	fragment (hash)
http://	localhost	:1337	/home	?name=johan	#history
http://	google.com		/search	?q=node	
https:/	www.howest.be		/		#q=nmct

host = hostname + port

path= pathname + querystring

query=verwijst naar de querystring zonder het vraagteken, en kan als een object aangeboden worden.

Volgende karakters worden escaped: < > " ' \r \n \t { } | \

12.3.3 De HTTP module in node

Met de module http kan de http server aangemaakt worden. De callback voorziet een http.serverrequest object en een http.serverresponse object.

Het serverrequest object is een inbound ReadStream van de cliënt en laat toe eigenschappen op te vragen (URL, headers), data events te verwerken en data te pipen in een andere stream.

Het serverresponse object is een outbound WriteStream naar de cliënt en laat toe headers en body te schrijven of kan een readstream in de body toevoegen(vb:piping een movie).

Om een continue responsestream uit te voeren kan het http chunked protocol gebruikt worden via de header eigenschap: {Transfer-Encoding: chunked}. Dit wordt default ingesteld wanneer geen "Content-Length" wordt opgegeven. Chunking kan vooral nuttig zijn bij het versturen van streaming audio of video. Het einde van de transfer wordt gekenmerkt door het versturen van een chunk met lengte nul, waardoor de cliënt afsluit.

Oefening: het basis "Hello World" voorbeeld van node

1. Test het basis voorbeeld van de node site. Dit is: een HTTP server aanmaken die op `http://127.0.0.1:1337/` een response "Hello world" terugstuurt door de instructie `res.end('Hello, world!');`
 - a. Je maakt een luisterende http server (IP adres 127.0.0.1 met poort 1337) in een javascript module en start deze module/server via de node command prompt.

Vervolgens kan je via de browser surfen naar 127.0.0.1:1337. De server antwoordt met "Hello World"

```

C:\Windows\system32\cmd.exe - node exampleHTTP.js
Error: Cannot find module 'C:\Users\Johan Van\exampleHTTP.js'
  at Function._resolveFilename (module.js:334:11)
  at Function._load (module.js:279:25)
  at Array.0 (module.js:470:10)
  at EventEmitter._tickCallback (node.js:192:40)
C:\Users\Johan Van>..
'..>' is not recognized as an internal or external command,
operable program or batch file.
C:\Users\Johan Van>cd
C:\Users\Johan Van
C:\Users\Johan Van>cd..
C:\Users>cd..
C:\>cd Program Files
C:\Program Files>cd nodejs
C:\Program Files\nodejs>node exampleHTTP.js
Server running at http://127.0.0.1:1337/

```

```

var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1337/');

```

We bemerken het gebruik van een externe library (`http`) en de `createServer()` methode. We zien hoe een anonieme callback functie als argument van `createServer` gebruikt wordt. We hebben met andere woorden een event driven, asynchrone en anonieme callback functie.

- b. We krijgen een server met een connection van het type "keep-alive" (voor streaming doeleinden) dankzij HTTP 1.1.
In de development console van een browser zien we dat zowel de Request- als de Response header van het type "keep-alive" zijn. Dit betekent dat we, zelfs bij toepassingen met langere response tijden, continu data kunnen blijven doorsturen naar de cliënt. Default wacht de server dan ook niet tot zijn antwoorden volledig zijn. Zo kan de server simultaan verschillende connecties aan. Concurrency is het sterkste punt van node.js. Anders uitgedrukt: verschillende requests kunnen los van elkaar uitgevoerd worden.

URL: http://127.0.0.1:8888/						
		Request headers	Request body	Response headers	Response body	Cookies
Key				Value		
Accept-Language				en-US		
User-Agent				Mozilla/5.0 (compatible;		
Accept-Encoding				gzip, deflate		
Host				127.0.0.1:8888		
DNT				1		
Connection				Keep-Alive		

- c. In de Response vinden we de eigenschap transfer-encoding: chunked terug. Dit duidt erop dat het antwoord in verschillende delen kan komen en dat er geen message length moet meegegeven naar de server. De cliënt is verantwoordelijk voor beëindigen van zijn connectie.

Request headers	Request body	Response headers	Response
Key	Value		
Response	HTTP/1.1 200 OK		
Content-Type	text/plain		
Date	Sat, 14 Dec 2013 13:36:17 GMT		
Connection	keep-alive		
Transfer-Encoding	chunked		

HTTP server voorbeeld: lezen van tekst files, waarvan adres in de *req.url* te vinden is.

Er wordt gebruikt gemaakt van een Buffer object, omdat Buffers uitermate geschikt zijn voor het werken met binaire data. Dit werkt sneller dan strings concatineren. Kortom: werk niet met binaire strings maar werk met buffers waar toepasbaar. Bij lezen haalt node typisch "chunks of bytes" op en stopteert ze tijdelijk in een buffer.

```

var http = require('http');
var fs = require('fs');
var path = require('path');
var util = require('util'); //analyseren van obj. properties

//http server aanmaken.
var server = http.createServer();
console.log("Http server running on 1337 ");
//event bij incoming client request
server.on("request", function (req, res) {

    //Header komt vóór body en kan slechts éénmaal.
    res.writeHead(200, {
        'Content-Type': 'text/html',
        'Cache-Control': 'max-age=3600',
        'Transfer-Encoding': 'chunked' //kan blijvend verbinding aanhouden
    });

    //buffer aanmaken en versturen naar cliënt.
    var buffer = new Buffer("<div>Omdat het response een stream object is, kan het ook buffers verzenden. </div>");
    res.write(buffer);

    //file URL ophalen uit request of een default URL gebruiken
    if (req.url == '/') {
        //relatief adresseren met een punt
        file = "." + path.normalize('/resources/testFile.txt');
    } else {
        //req.url start altijd met een /
        file = "." + req.url;
        console.log (" lezen van " + file);
    }

    //file uitlezen via streaming
    var readStream = fs.createReadStream(file, { flag: 'r', encoding:'utf-8' });
    console.log("\n streaming ");
})

```

```
readStream.pipe(res); // geen async on data, geen res.end bij regular pipe

//opvragen van serverrequest eigenschappen
res.write('<h3>req.url:</h3>' + req.url);
res.write('<h3>req.method:</h3>' + req.method);
res.write('<h3>req.headers:</h3>' + util.inspect(req.headers));
})

//server luistert naar requests van host/port
server.listen(1337, '127.0.0.1');
```

Oefening: http server vertaalt csv naar json

Een eenvoudige aan te maken csv file bevat een viertal titels van hoofdstukken. Bekijk deze file als een een key/value structuur die er als volgt uitziet:

- 1-Werken met Node.js.
- 2-De Node API voor files en servers. . .
- 3-Middleware. . .
- 4-Express framework

Zorg voor een http server, die deze csv file omzet naar een "valid" JSON file.

Oefening: http server zorgt voor streaming video

Zorg voor een http server, die streaming video aanbiedt (mp4). De filenaam wordt opgehaald uit de querystring. Voorzie ook error handling.

Oefening: lezen van statische HTML files.

1. Maak in de node applicatie een map aan met naam "public". Een map "public" wordt vaak aangemaakt om er de publieke files in te maken: html files maar ook script en css files.
Maak een index.html, test1.html en test2.html.
2. Maak een http server aan in de node applicatie. In de callback van createServer, haal je vanuit de url (req.url) de file naam op, of gebruik je default de index.html
`var filename = path.basename(req.url) || "index.html"`
Je hebt echter het volledige(!) path nodig zodat je gebruik maakt van __dirname of process.cwd():
`localPath = process.cwd() + "/public/" + filename`
3. Controleer of het localPath wel degelijk bestaat:
`fs.exists(localPath, function(exists) { ... });`
4. Indien het bestaat halen we de file op (=lezen met fs).
`fs.readFile(localPath, function(err, contents) { ... });`
Indien niet: `res.writeHead(404) ; res.end();`
Vergeet de `res.end()` niet. Zo niet blijft de browser wachten op het einde van de response.
5. Na lezen van de HTML file (= in de callback) kan deze file via de response verstuurd worden naar de client. Hierbij mag vooral niet vergeten worden om **het juiste "Content-Type"** mee te geven in de header !!!
`res.writeHead(200, { "Content-Type": "text/html" });`

```
res.end(myFile);
```

6. Voeg onder de map public een CSS map toe en plaats er een willekeurige stylesheet: public/css/style.css
7. De css file , die je oproept in de head van index.html, wordt echter niet opgehaald door de server. Oorzaak is een dubbele reden: de verkeerde map wordt opgehaald en het mime type van de response klopt niet.
Oplossing: controleer de extensie van de file met path.basename(filename), waarna het bijhorende "Content-type" met bijvoorbeeld extensions[".css"] opgehaald wordt.

```
extensions = {  
    ".html": "text/html",  
    ".css": "text/css",  
    ".js": "application/javascript",  
    ".png": "image/png",  
    ".gif": "image/gif",  
    ".jpg": "image/jpeg"  
};
```

Hetzelfde principe kan je gebruiken voor het ophalen van de correcte map.
Werk dit verder uit en test een html pagina met css en javascript waarbij het juiste mime type opgehaald wordt.

8. Onze server leest nu wel alle mogelijke files, maar de opbouw van de applicatie is eerder slordig voor hergebruik van deze static server. We hervormen daarom de opbouw.
(refactoring)

- a. In app.js (de eerst opgeroepen "main" file volgens package.json) roepen we de staticServer op als een module:

```
var staticServer = require("./scripts/staticServer.js");  
staticServer.init(8080);
```

- b. Maak de file staticServer.js aan in de map "scripts". We kiezen voor het module pattern, waarbij de zelf uitvoerende init de server initialiseert. Zo wordt de server de enige globale variabele:

```
var staticServer = function () {  
    http = require('http');  
  
    var httpListen = function (httpPort) {  
        var server = http.createServer(function(req,res) {  
            //filename ophalen en readFile oproepen  
        } );  
        server.listen(httpPort);  
    },  
    extensions = {  
        // plaats hier de extensies (opties)  
    },  
    localMaps = {  
        // plaats hier de lokale map voor elke extensie  
    },  
    readFile = function (res, filename) {  
        //haal extensie op  
        //lees de file met fs en verzend met correct mimeType  
    },  
  
    //publieke methodes (return)  
    init = function (httpPort) {
```

```
        console.log("server running on port :", httpPort);
        httpListen(httpPort);
    };

    return {
        init: init
    };
}();

module.exports = staticServer;
```

c. Werk bovenstaande verder af, waardoor we beschikken over een leesbare en onderhoudbare module met een static server. De module kunnen we verder hergebruiken.

12.4 Node als REST API cliënt

Omdat node sterk is in I/O handling wordt node vaak gekozen voor het opvragen van RESTfull API's. Hierbij kan node zelf fungeren als een cliënt op een API van derden. De opgehaalde data kan verder door de node server zelf gebruikt of gecombineerd worden tot een nieuwe pagina of nieuwe API.

12.4.1 Een node GET request met `http.request(options, cb)`

Met behulp van `http.request()` of `http.get()` kan een node client aangemaakt worden die een get request afvuurt. Als argument kunnen verschillende opties toegevoegd worden, die nodig zijn voor ondervragen van de REST API. Het resultaat is een `http.ClientResponse` dat een response eventemitter aanstuurt.

```
var http = require("http");

var fs = require("fs");

var options = {
  host: "jsonplaceholder.typicode.com", //fake data api
  port: 80,
  path: "/posts",
  method: "GET"
}

//verkorte schrijfwijze zonder end() kan met http.get (options... )
http.request(options, function (resp) {
  console.log('STATUS CODE: ', resp.statusCode + " " +
  http.STATUS_CODES[resp.statusCode]);

var data ="" ;

resp.on("data", function (chunck) {
  console.log("\n ontvangen buffer data: ", chunck);
data += chunck ;
  process.stdout.write("\n data: " + chunck); //weergave in console
});

resp.on("end",function() {} )
//stream|bewaar response in een file on disk:
  var file = fs.createWriteStream("./resources/response.txt");
  resp.pipe(file);
```

```
}).end(); //end zorgt voor de request uitvoer ( niet vergeten!!)
```

Voor het uitvoeren van zijn get request gebruikt node een agent. Deze agent onderhoudt een pool van live socket connecties. Bij het einde van een request, zonder nieuw pending requests, geeft de agent de socket automatisch vrij. Het aantal toegelaten open sockets per poort kan ingesteld worden via de options.

API's ondervragen in een voorgedefinieerde volgorde d.m.v een Bufferlist.

Het kan zijn dat je verschillende API's wil ondervragen, waarbij de data van een eerste API nodig is vóór een tweede API. Door het asynchrone gedrag is de volgorde niet zomaar verzekerd. Bovendien kan server twee sneller zijn dan server één. We kunnen dit oplossen (zoals eerder aangetoond) door de calls in een Array.shift() te plaatsen of door gebruik te maken van de `async` library.

Een alternatief op deze methoden die gebruikt wordt bij streams (zoals een server response), is het gebruik van een buffer list. <https://www.npmjs.org/package/bl>. Deze bufferlist is een duplex stream, die hierdoor thuis hoort in een "pipe()" rij. iets wat een Array niet kan.

```
var BufferList = require('bl');
```

Je haalt bijvoorbeeld alle data op (met `http.get()`) en voegt ze toe aan een bufferlist.

De callbackfunctie zorgt dat alle binnenvkomende informatie toegevoegd wordt aan de bufferlist.

```
response.pipe(BufferList(function (err, data) { }))
```

Of je doet het expliciet met :

```
var bl = new BufferList();
bl.append(data);
```

Als laatste stap lees je de bufferlist via een pipe() command uit op basis van zijn aantal ontvangen buffers (`bl._bufs.length`) in een `fileStream` of `responseStream`.

```
var file = fs.createWriteStream("./resources/response.html");
bl.pipe(file);

bl.pipe(res); //pipe roept automatisch end() op
```

12.4.2 Een node POST request met de "querystring" module.

Formdata POST met de module "querystring"

Referentie: Informatie over de module is te vinden op <https://nodejs.org/api/querystring.html>

Hoewel deze module zeer beperkt is, laat deze node module toe om volledig met querystrings te werken. Met deze module kunnen http POST values verstuurd worden via de `querystring` en worden deze querystrings (un)escaped. Hiertoe wordt de `querystring` module zowel in cliënt als server opgehaald:

```
var qs = require('querystring');
```

De module zorgt voor communicatie tussen de node client en de node server via de `querystring`:

- Node cliënt kant: formdata – onder de vorm van een javascript object - wordt geserialiseerd naar een `querystring` met `qs.stringify({//key1 : "value1" , key2: "value2" ... })`

- Node server kant: De ontvangen formdata wordt met `qs.parse(formData)` terug omgezet naar het oorspronkelijke javascript object.

```
/*--- Een node client ---*/
var http = require('http');
var querystring = require('querystring');

var request = http.request({
    //options
    hostname: "localhost",
    port: 8080,
    path: "/",
    method: "POST",
    headers: {
        "Content-Type": "application/x-www-form-urlencoded",
    }
}, function (response) {
    //callback

    process.stdout.write("Server bevestigt:\n")
    response.on("data", function (data) {
        process.stdout.write(data);
    });
});

//aanmaken van stringified formData voor de query string. Dit kan een Array zijn.
var formData = querystring.stringify({
    hogeschool: "HOWEST",
    naam: ["Johan", "VanHowest"]
});

//verzenden van formData als string: hogeschool=Howest&naam=Johan&naam=VanHowest
request.end(formData);
```

Oefening:

Als test maken we een http server voor de bovenstaande client. De server stuurt de formdata terug naar de cliënt maar dan wel omgezet naar hoofdletters.

```
HOGESCHOOL = HOWEST
NAAMCOMPLEET = JOHAN, VANHOWEST
NAAM[VOORNAAM] = JOHAN
NAAM[ACHTERNAAM] = VANNMCT
```

Hiervoor wordt gebruikt van twee listeners met callback functies: `on("data")` en `on("end")`. Reden hiervoor is dat data kan ontvangen worden in opeenvolgende chunks. Alle chuncks worden bij `on "data"` bijgehouden in een variabele. Bij `on "end"` gebeurt de verwerking (bvb. het terugzenden in hoofdletters):

```
var formData = "";

req.on("data", function (chunck) {
    formData += chunck;
});

req.on("end", function () {
    //de formdata kan met hulp van de qs module weer omgezet worden naar een obj
    var formResult = qs.parse(formData);

    //javascript obj. testen door obj properties uit te schrijven
```

```
for (var key in qs.parse(formData)) {
    res.write(key.toString().toUpperCase() + " = " + . . . . );
}
```

Noot: Controleer bij de fout “connect ECONNREFUSED” of de poortnummers wel overeenstemmen. En vergeet res.end() niet.

JSONP of JSON als response met de module “querystring”

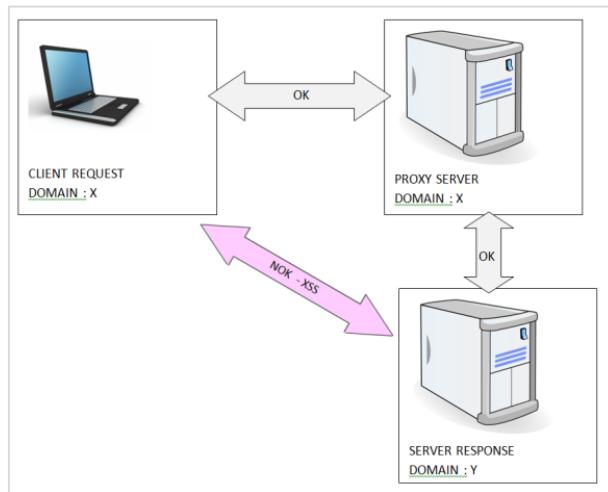
De querystring module laat toe om de querystring aan serverkant met één instructie om te zetten in key/value pairs:

```
var qs = querystring.parse(req.url.split("?")[1]).
```

Gegevens in key/value pairs zijn eenvoudiger te behandelen voor verschillende toepassingen:

- als pure data
- om de routing aan te sturen via bvb. een switch.


```
switch (qs['page']) {
    case 'product'.....
```
- om na te gaan of een API moet aangestuurd worden met een callback functie als parameter (JSONP) of zonder callback (JSON). Door het gebruik van JSONP worden Cross Site requests mogelijk gemaakt.



```
var http = require("http"), querystring = require("querystring");

http.createServer(function (req, res) {
    var qs = querystring.parse(req.url.split("?")[1]),
        username = qs.firstName + " " + qs.lastName,
        json;

    if (qs.callback) {
        // indien callback parameter dan deze callback terugsturen (JSONP)
        // http://api.mine.com/pictures.json?lastname=me&callback=?
        json = qs.callback + "({username:'" + username + "'});";
    } else {
        // zonder callback parameter dan raw JSON terugsturen
        json = JSON.stringify({ "username": username });
    }
})
```

```
res.writeHead(200, {
  // verander het MIME type naar JSON
  "Content-Type": "application/json",
  "Content-Length": json.length
});
res.end(json);
}).listen(8000);
```

12.4.3 Meer request opties met de “request” module

Referentie: Informatie over de module is te vinden op <https://github.com/mikeal/request>

Omdat de basis “querystring” module van node toch maar beperkte mogelijkheden heeft om url onderdelen op te vragen, wordt hiervoor veel gebruik gemaakt van third party modules. Deze extra modules voorzien mogelijkheden zoals: redirects, cookies, verzenden of parsen van query strings, verzenden van form data, verzenden van JSON objecten, streaming.

De module “request” wordt bestempeld als een “*Simplified http client*” en is één van de meest gebruikte modules. Na installatie van deze request module (npm install request) wordt het gebruik van de node “querystring” module zelfs overbodig.

De request module biedt ondanks zijn eenvoudige signatuur (`request({options} |url, callback)`) verschillende features aan. Het resultaat is een duplex stream, die zowel streams kan ontvangen (vb. van een file stream of zichzelf) of sturen (vb. naar een image) via pipe(). Er kan geluisterd worden naar on('response') en on ('error')

Opties worden aangebracht via een javascript property, waarbij de values meestal zelf een literal javascript object zijn:

- method : default GET , POST, PUT .. ,
- headers : default {}
- form : default header Content-type:application/x-www-form-urlencoded
- formData : default header Content-type: multipart/form-data
- auth : HTTP authenticatie toe via { "user": , "pass": }
OAuth : open autorisatie via { callback: "myURL", consumer_key:..., consumer_secret:... }
- agentOpties : voor het gebruik van het SSL protocol met certification keys.
- har : genereert een HTTP archive request

Een voorbeeld :

```
/*--- Een POST node client ---*/
```

```
var request = require("request");

request({
  uri: "http://localhost:8080/",
  method: "POST",
  headers: {
```

```
        host: "localhost:8080"
    },
    form: {
        hogeschool: "HOWEST",
        naamCompleet: ["JohanV", "VanHowest"]
    }
}, function (error, response, body) {
    if (error) { console.log("Error: " + error);}
    console.log("response body:" + body);
});
```

12.5 Routing opbouwen met de node "url" module

Referentie: Informatie over de module is te vinden op <https://nodejs.org/api/url.html>

Met `querystring.parse(req.url.split("?")[1])` kan je querystring delen opvragen. Wordt de url een SEO vriendelijke url (met slashes opgebouwd) dan moet je deze functie herschrijven. De node "querystring" module van node laat niet toe om vlot alle onderdelen van een willekeurige url op te halen.

Nochtans is de url is de basis voor het opstellen van een routing aan server kant. Om de routing efficiënt te kunnen verwezenlijken hebben we alle onderdelen van de url nodig (hostname, port, pathname, querystring, fragment). De "url" module van node helpt ons hierbij, door het aanbieden van een `parse()` methode waarbij (un)escape automatisch gebeurt. Het resultaat van deze `parse()` is een Array of een javascript object:

```
var arrPath = url.parse(req.url).pathname.split("/");
var objPath = url.parse(req.url, true).query ;
```

Van zodra je over de delen van het path beschikt kan de routing gedefinieerd worden:

Dit kan bvb. via een if: `if (req.method == "GET" && arrPath[1] == "product") { }`

Naast de `parse()`methode beschikt de url module nog over een `format()` en `resolve()`:

- `url.format(urlObj)`: zet een urlObject (objPath) terug om naar een string
- `url.resolve(part1, part2)`: concatineert beide delen met een / indien part1 NIET eindigt op een slash. Eindigt part1 wel op een slash, dan wordt het laatste deel van de routing (/) vervangen door part2.

Tip: Een wijziging van het node programma of de routing verwacht dat je de server opnieuw start om dit actief te maken. Node cache zijn programma. Het herstarten kan je automatisch laten gebeuren door de "supervisor" of "forever" module. Je start je applicatie dan via:
`>> supervisor myApp.js` in plaats van `>> node myApp.js`

Oefening:

Een http server returnt de actuele tijd in een JSON formaat.

- Via een browser client die werkt met querystrings (GET). Afhankelijk van de URL wordt een zomeruур of een winteruurr gegeven. Een html file (in de

"public" folder) laat toe met radiobuttons te kiezen tussen wintertijd of zomertijd.

Zomertijd - Wintertijd

Selecteer wintertijd of zomertijd:

Zomertijd

Wintertijd

kies zomer of winter

- Via een node client die werkt met een POST request. Afhankelijk van het meegegeven argument in de console (process.argv[2]) wordt zomertijd of wintertijd gegeven. Wordt niets ingegeven dan krijg je het zomertijd.

> node Timeclient winter

Oplossingsmethodiek:

1. Maak de node cliënt aan. Je kan gebruik maken van de request module en zijn POST method.
2. Zet een http server op die bij req.on("data", cb) op basis van data.toString() een winter of zomertijd terugstuurt naar de node cliënt. Een winteruur kan je uitrekenen via `new Date(new Date().getTime() + 60 * 60 * 1000);`
3. Maak een html cliënt aan met een formulier dat data stuurt via de querystring..
Je kan bvb. gebruik maken van jQuery en \$.ajax.

```
$('input[type="submit"]').on('click', function () {
    $.ajax({
        dataType: "json",
        url: "http://localhost:3000",
        data: $("#form").serialize(),
        success: function (data) {
            //data verwerken en weergeven
        }
    });
    return false; //stilleggen van submit
});
```

4. Pas de server aan met behulp van een `switch` case, die zich baseert op de ontvangen querystring data:

De reeds aangemaakte code kan je in `default:` onderbrengen

De ontvangen querystring resulteert in twee bijkomende `cases`. Ofwel wordt een zomertijd teruggestuurd, ofwel een winteruur. (`res.end(JSON.stringify(body))`)

12.6 Cookies in node

In de response header van een node request kunnen met Set-Cookie zowel enkelvoudige als meervoudige cookies aangemaakt worden. Voorbeeld:

```
server.on("request", function (req, res) {
    res.writeHead(200, {
        'Set-Cookie': 'afdeling=NMCT ; Max-Age=60',
        'Set-Cookie': 'naam=Johan; HttpOnly'
```

```
})
res.end("<div>Cookie ingesteld</div>");//opent browser
});
```

Het lezen van cookies gebeurt analoog met `request.headers.cookie`.

12.7 Herhalings oefening

12.7.1 Doel

- Asynchroon werken met node.js
- Gebruik van module pattern (closures) en constructor pattern.
- (Gebruik van domains voor foutbehandeling= deprecated).
- Gebruik een `options={}` object voor het instellen van basis configuraties.
- Gebruik van zelfaangemaakte modules voor oa creatie van een server, die statische files leest (html, css, javascript).
- Module laten erven van `EventEmitter` om een event kenbaar te maken aan een andere of meerdere modules.
- Gebruik van een API met JSON communicatie.
 - Ontvangen JSON opkuisen naar de standaard norm.
 - Een unobtrusive HTML client met jquery voor de json weergave.
- Het resultaat cleanen met lint en minifiënen.

12.7.2 Opgave

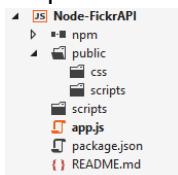
Zet een http server op, die dienst doet als client voor flickr (<https://www.flickr.com/services/api/>).

In een HTML formulier wordt een zoekwoord ingegeven, waarna de http server de bijhorende foto's ophaalt en door geeft aan de HTML cliënt.

Een teller op de server (afzonderlijk object) houdt bij hoeveel succesvolle requests er per dag gebeuren.

12.7.3 Oplossing

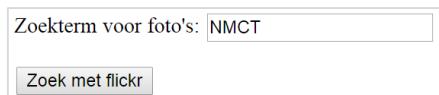
1. Maak een nieuw project. Met de volgende structuur.: `public` , `public/scripts/` , `public/css` , `scripts`:



2. Zorg ervoor dat in package.json de file app.js de startfile is:
`"main": "app.js"`

3. Aanmaken en weergave van public/index.html:

- a. Maak "index.html" aan in de map public.



Zoekterm voor foto's: NMCT

Zoek met flickr

- b. We maken gebruik van de eerder aangemaakte staticServer. Voeg de file toe aan het project en toon index.html als test.

4. Toevoegen van clientscript en jQuery in index.html.

- a. Doe het nodige in de head van index.html om jquery te kunnen gebruiken. (lokaal of via CDN)

```
<script  
src="//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>  
<!--<script src="scripts/jquery-2.1.1.js"></script>-->
```

- b. Maak start.js aan in public/scripts en test het opstarten van jQuery.

```
$(document).ready(function () {  
    console.log("jQuery loaded");  
})
```

Controleer bij fouten of het juiste content-type meegegeven werd (application/javascript) en de juiste map.

5. Een ajax call naar de server api client opzetten.

- a. Voeg een click event toe aan de submit button "Zoek met flickr".

```
$('#input[type="submit"]').on('click', function () { })
```

- b. Maak een ajax call met de inhoud vh zoekveld:

```
$.ajax({  
    dataType: "json",  
    url: "http://localhost:8080/apiData",  
    data: $("#myForm").serialize(),  
    success: function (items) { }  
});  
return false; //Verhindert submit!
```

Vergeet de return false niet. Zoniet wordt de ajax call niet uitgevoerd, maar krijg je steeds een submit van de form.

Noot de url kan universeler worden aangemaakt via

```
url: location.protocol + "//" + location.host+ "/apiData"
```

6. De node static server moet nu overweg kunnen met de url (apiData). Deze url informeert node om als een get client te fungeren met een API call naar flickr.

- a. De server verwerkt de binnenkomende url niet goed , want path.basename(req.url) returnt telkens de volledige filename zoals index.html , start.js of apiData?search= .

We hebben echter delen van het path nodig zoals de querystring of een fragment afzonderlijk. Om dit te realiseren maken we gebruik maken van de url module om de pathname (apiData) op te halen..

```
var uri = url.parse(req.url).pathname;
```

- b. Met een switch op uri kunnen we alles in de juiste richting sturen:
`case "/": //lees de file` `case "/apiData": //maak REST call`
....
- c. Bij ontvangst van "apiData" kan de querystring data van het formulier (de zoekterm) opgevraagd worden met url.parse(req.url).query.

De string "na" het vraagteken komt hierbij beschikbaar en kan gebruikt worden bij de API cliënt, die flickr ondervraagt.

Noot: Bemerk hoe het routing verhaal relatief veel code bevat. Iets om verder bij refactoring in een afzonderlijke module onder te brengen, zodat hergebruik mogelijk wordt?

7. Aanmaak van een flickr API cliënt.

- a. Maak getAPIData.js aan in de server scripts map. Hierin komt de API client. Verwar niet met de map scripts op de cliënt.
- b. Ook nu maken we gebruik van het module pattern. De module getAPIData.js kan er ongeveer uit zien zoals hieronder vermeld. Vervolledig de code en vervang de vaste zoekterm NMCT door de zoekterm aangeboden via het formulier.
Uiteindelijk roep je vanuit de switch case de module op. Geef ook het current request en response object mee: callAPI('hierdezoekeerm', req, res); dit is http.

```
var getAPIData = function () {  
    options = {  
        method: "GET",  
        port: 80,  
        host: 'api.flickr.com' ,  
        path:  
            '/services/feeds/photos_public.gne?format=json&tags=NMCT&jsoncallback=?'  
    },  
  
    callAPI = function (search, req, res ) {  
        //Voer een http.request uit, die als EventEmitter naar een antwoord luistert  
    };  
  
    return {  
        callAPI : callAPI ,  
        options : options // wordt zo configurerbaar  
    };  
  
}();  
  
module.exports = getAPIData;
```

- c. Voor het uitwerken van de callAPI moet je er rekening mee houden dat de resultaten van de flickr service in chuncks binnen komen. De chunks worden samengebracht in een variabele json. De webservice levert immers json af (geen xml) , die we verder parsen om de gewenste data naar de http client te sturen.

```
http.request(options, function (response) {
    var json = "";
    response.on("data", function (chunck) {
        json += chunck
    });

    response.on('end', function () {
        console.log("\n ontvangen json data: ", json);
        //hier json nog parsen voor gewenste cliënt data
    });
});

}).end(); // end niet vergeten -> zorgt voor de uitvoer
```

8. Parsen en verzenden van de JSON data, die ontvangen werd van flickr

- a. De ontvangen JSON data willen we on "end" parsen, om zo de image url's er uit te halen.

Bij `JSON.parse(json)` krijgen we echter een error die vertelt dat de JSON string niet correct geformateerd is. We krijgen een "unexpected end" error.

Maak daarom een clean functie aan, die niet publiek komt, maar die wel het probleem zal oplossen van een invalid json file:

```
var jsonObj = JSON.parse(clean(json))
```

```
⚠ String is not JSON formatted
{
  "title": "Recent Uploads tagged mountains",
  "link": "http://www.flickr.com/photos/tags/mountains/",
  "description": "",
  "modified": "2014-10-18T15:06:26Z",
  "generator": "http://www.flickr.com/",
  "items": [
    {
      "title": "Ben Nevis via Carn Mor Dearg (41)",
      "link": "http://www.flickr.com/photos/walksindreams/15378462690/",
      "media": {"m": "http://farm4.staticflickr.com/3954/15378462690_b4cc9c3e3c_m.jpg"},
      "date_taken": "2013-06-04T16:50:17-08:00",
      "description": " <p><a href=\"http://www.flickr.com/people/walksindreams/\">Walks in",
      "published": "2014-10-18T15:06:26Z",
      "author": "nobody@flickr.com (Walks in Dreams)",
      "author_id": "78126169@N06",
      "tags": "mountain mountains walking landscape scotland bennevis munro kevincpool"
    }
}
```

Of : Uncaught SyntaxError: Unexpected token (

- b. Voeg de clean methode toe en bouw ze uit met volgende informatie. Het vraagt wat zoekwerk, maar de reden van de foutief gevormde JSON is te vinden in:

- a. ongewenste json start karakters (vb. ronde haken). Dit kan worden opgelost met de eerste karakters te verwijderen:
`json.substring(1, json.length - 1)`

- b. een aantal enkele quotes ' en backslash \ die voor verwarring kunnen zorgen. Dit kan worden opgelost met een reguliere expressie (vervangen door lege string of extra escape karakters)
`json.replace(/\\"/g, '')`
 - c. Verzend de gecleande json naar de cliënt. Hiervoor gebruik je de origineel door cliënt ontvangen response "res". Dit is niet de response van de api call :
`res.write(JSON.stringify(jsonObj)).`
Vergeet uiteindelijk hierbij de `res.statusCode` en `res.end()` niet.
9. Ontvangen json weergeven op de cliënt.

In de success callback functie van \$.ajax wordt de json data ontvangen. Zie de inhoud ervan na met de browsers tools, zodat je de juiste of gewenste data gebruikt. We willen de foto's weergegeven in een `` tag. De src is terug te vinden in json onder de eigenschap `this.media.m`

```
success: function (items) {
    $.each(items, function (items) {
        // uitwerken met $("<img />").attr('src', this.media.m)
    })
}
```

10. Waarschijnlijk heb je nu altijd het keyword NMCT opgevraagd. Pas de code aan, zodat een willekeurig zoekwoord kan meegegeven worden vanuit het formulier.
11. We wensen een teller bij te houden, die het aantal aangevraagde succesvolle requests bijhoudt. We plaatsen deze code niet in getAPIData. Ze hoort eerder thuis in de switch case van de server. (vb. dan kunnen ook andere zaken al dan niet afgehandeld worden na verkrijgen van API data).

Kortom: er is een eventemitter nodig, die informeert wanneer de API data opgehaald is. Binnen de callAPI (en zijn "end") kunnen we deze emit afvuren

```
getAPIData = new EventEmitter();
getAPIData.emit('apiData', jsonObj.items);
```

De staticServer is ingeschreven in het event en zal de teller ophalen in de module "requestCounter". Bouw zelf de requestCounter module uit.

```
getAPIData.on('apiData' , function (jsonItems) {
    console.log("de counter bevat nu :" , requestCounter.getCount())
})
```

12. Volgende stap laat extra foutmanagement toe. Het gebruik van domeinen is momenteel deprecated, zodat deze stap er enkel staat ter info. Je hoeft deze niet te realiseren, hoewel het voorlopig nog werkt. Zorg wel dat de callback functies allen een error controle uitvoeren.

Foutmanagement met gebruik van domeinen (deprecated):

Door het gebruik van verschillende domeinen, weten we waar de fout zich afspeelt zonder dat de applicatie vast loopt.

Vooral voor de cruciale in/out acties willen we error domain controle. We passen het toe op createServer() in de module staticServer.js:

```
var domain = require('domain'),
    httpDomain = domain.create()

var httpListen = function (httpPort) {
    httpDomain.on('error', function (err) {
        console.log('Error in the http domain:', err);
    });

    httpDomain.run(function () {
        // hier alles van createServer invoegen
        var server = http.createServer(function (req, res) { });
        server.listen(httpPort);
    });
}
```

Op dezelfde manier kan ook een domain aangemaakt worden voor de API oproep in de module getAPIData.js: apiDomain = domain.create();

13. Wanneer we onze staticServer nader bekijken, zien we dat routing instructies (switch case) tussen de create van de server staan. Je zou een afzonderlijke module (router.js) kunnen aanmaken waardoor de volledige applicatie nog meer uitwisselbaar wordt. Dezelfde bedenking kan vervolgens gemaakt worden bij het toevoegen van Sessions, Cookies...
Wanneer de applicatie echter zo blijft groeien, overweeg je het gebruik van middleware (bvb.: componenten van "connect.js") of het gebruik van een framework (bvb.: het express framework). Beide worden verder behandeld.

13 Andere server types en protocols.

TLS/SSL

TLS (*Transport Layer Security*) , de nieuwe vorm van SSL (Secure Socket Layer), zorgt voor encryptie boven de transport layer waardoor een veiliger communicatie ontstaat met message authenticatie. Node maakt hiervoor gebruik van de "TLS" module (<http://nodejs.org/api/tls.html>).

Er wordt gebruik gemaakt van twee key's, waarvan één publiek is. Eén key wordt gebruikt voor encryptie en de andere voor decryptie. Een public key certificaat (ook digitaal certificaat genoemd), wordt ondertekend door een Certificaat Authority (CA) en zorgt voor de koppeling met een identity (= een persoon, een organisatie). De CA's zijn typisch betrouwbare third parties. Om te communiceren is het signed certificaat nodig (=publiek) en een private key.

De keys worden gegenereerd met behulp van de SSL library of de OpenSSL library (<http://www.openssl.org>). Binaries zijn terug te vinden op:

<https://www.openssl.org/related/binaries.html>. *De keys komen terecht in twee files* (bvb.: my_key.pem en my_certificate.pem) die tijdens initialisatie (= niet in een callback!) synchrono ingelezen worden door node als extra serveropties. Via extra properties kan een verplichte authenticatie van de client opgelegd worden (requestCert:true) en kunnen niet geautoriseerde personen toegang verhinderd worden (rejectUnauthorized:true). Voor de rest verloopt het proces gelijkaardig aan een TCP server, maar met andere namen voor de properties.

```
var tls = require("tls");
var fs = require("fs");
var port = 4010;

var options = {
  key: fs.readFileSync("my_key.pem"),
  cert: fs.readFileSync("my_certificate.pem")
}

var server = tls.createServer(options, function (clientreq) {
  clientreq.on("data", function(data) {    });
})

server.listen(port, function () {
  console.log('server listens on port: ', server.address().port);
})
```

HTTPS

Het is logisch dat voor het verzenden van gevoelige data (bvb.: credit card informatie) over het net een beveiligd protocol, zoals HTTPS, nodig is. *HTTPS voegt het TLS transportmechanisme toe aan het HTTP protocol*. Node maakt hiervoor gebruik van de module "HTTPS" (). De HTTPS module erft dan ook van de http module.

```
var https = require("https");
var fs = require("fs");
var port = 4010;

var options = {
```

```
key: fs.readFileSync("my_key.pem"),
cert: fs.readFileSync("my_certificate.pem"),
requestCert: true,
rejectUnauthorized: true
}

var server = https.createServer(options, function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end("Welkom bij HTTPS");
})

server.listen(port, '192.168.1.100');
```

UDP

UDP(User Datagram Protocol) is een transport layer gelijkaardig aan TCP om messages (datagrams genoemd) te versturen. UDP is niet connectie georiënteerd en voorziet dan ook *geen error checking* op de volgorde van, het verliezen van, of het twee keer weergeven van een datagram. UDP vindt zijn toepassing bij *kleine queries met zeer veel gebruikers, waar de boodschap zelf minder belangrijk is*. De lengte van een datagram kan wel beperkt worden door het netwerk (bvb.: 1500 bytes over Ethernet. Routers chunken op 512 bytes).

UDP ondersteunt broadcasting (= alle cliënts inlichten) en multicasting (= one-to-many of many-to-many distributie). Andere toepassingen zijn: DNS (Domain Name server), VoIP (Voice over IP), IPTV (Internet Protocol Television voor streaming video), TFTP (Trivial Transfer Protocol voor eenvoudige file transfer zonder authenticatie tussen servers), SNMP (Simple Network Management Protocol voor beheer van devices zoals routers en switches).

Node maakt gebruik van de module "dgram" (<http://nodejs.org/api/dgram.html>). Zowel cliënt als server maken gebruik van createSocket("udp4" of "udp6, [message callback handler]). UDP gebruikt wel een random port number voor het aanmaken van een socket. Is dit niet gewenst dan kan de socket aan een specifieke poort gebonden worden met socket.bind(port, [address], [callback]). De socket kan ook boodschappen versturen met socket.send(buffer, offset, length , port , address, [callback])

```
var dgram = require("dgram");
var port = 4020;
var socket = dgram.createSocket("udp4", function (message, rinfo) {

});

socket.bind(port, function () {
  console.log("bound to:", server.address());
});

socket.on('message', function (message, rinfo) {
  // lees message
  var msg = "Er zijn " + rinfo.size + " bytes ontvangen van "
  msg += rinfo.address + ":"+rinfo.port;
  msg += "\n Boodschap : \n " + message.toString();
  console.log (msg)
});
```

Naast het message event , kan ook nog gebruik gemaakt worden van het listening event.
Voor de uitwerking van de client wordt ook gebruik gemaakt van `createSocket()`.

Oefening (optioneel):

- Maak een UDP cliënt voor bovenstaande server.
`var client = dgram.createSocket("udp4");`
- De client geeft de booschap van de server weer:
`client.on("message", function (message) { })`
- Maak er een command-line cliënt van door gebruik te maken van `process.stdin.on()`. Wat je intypt wordt hierdoor naar de server gestuurd.
`process.stdin.on("data", function (data) {
 client.send(data, 0, data.length, port, 'localhost')
})`

Door de `socket.on('message')` op de server krijgt de cliënt zijn eigen message terug.

Enkele netwerkutilities

Opvragen van een DNS (Domain Name Server) kan met behulp van de dns module en de methode `dns.resolve(eenDomainNaam, [rrtype], callback)`. Default wordt een IP4 adres opgehaald. Met rtype (recordType) kunnen ook andere types worden opgehaald. Omgekeerd kan met `dns.reverse()` vanuit het IP adres een domain naam opgehaald worden. (<http://nodejs.org/api/dns.html>)

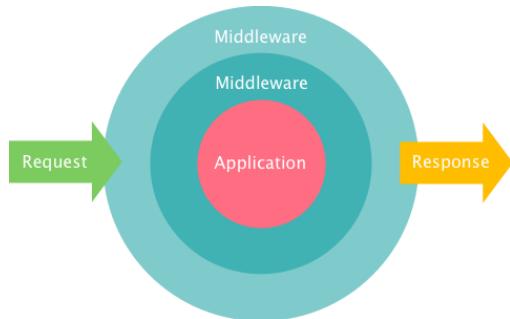
```
var dns = require("dns");
var domain = "www.google.be";

dns.resolve(domain, "A", function (error, addresses) {
  if (error) {
    console.log("fout: " + error.code);
  } else {
    console.log (domain + " heeft volgende IP4 adressen " + addresses) ;
  }
});
```

Opvragen van een valid IP adres kan met de module "net" en de methode `net.isIP(eenIPAdres)`

14 Middleware vereenvoudigt een web applicatie

Alle http server applicatie logica zelf programmeren met de basis node modules kan wel maar is een intensief werk. Middleware is een alternatief. Deze term maakt duidelijk dat middleware zich bevindt tussen het operating systeem en de applicatie met als hoofdbedoeling het werk van developers voor input/output communicatie gemakkelijker te maken. In een juiste definitie beschikt middleware dan ook maar over één input en returnt één output, die meestal een bewerking is op de ontvangen input. Middleware kan hierbij in cascade gebruikt worden:



14.1 Middleware definitie in node

Middleware componenten in node voor http, wrappen de request/response cycle en verwerken of bewerken het *request object of response object*. Het request object wordt verwerkt voordat het de applicatie bereikt. Bij het response object kan bvb. de header aangepast worden, een deel van de body geschreven worden, of de response kan beëindigd worden. Typische middleware voorbeelden bij een request zijn authenticatie, caching of logging.

Dergelijke middleware component bestaat uit niets meer dan hro die het request object ontvangt en na verwerking teruggeeft. Een request kan volledig verwerkt worden (complete processing) of er kan een bewerking op gebeuren waarna het request doorgegeven wordt aan de volgende middleware component. Vergeet hierbij errorhandling niet. Bij een fout in de middleware wordt de output op de hoogte gebracht (vb. foutieve authenticatie) en/of wordt de fout geregistreerd.

De programmeur kan zo verschillende middleware componenten na elkaar chainen. Hierin wordt vaak *zelf geschreven middleware* afgewisseld met *bestaande middleware* van third party libraries. Middleware in node is in essentie een functie met 3 parameters: request, response en next.

Door het gebruik van middleware kan veel code uitgevoerd worden op basis van "one-liners". Keerzijde hieraan: het vraagt wel een inspanning om eigen herbruikbare middleware aan te maken of om bestaande middleware instructies te bestuderen, conventies te volgen, en bovendien binnen de basis ideeën van een framework te blijven. Vernieuwende zaken toevoegen aan een applicatie vraagt meer creativiteit van de ontwerper, vooral omdat iedereen dezelfde one-liners gebruikt.

Aanmaken van middleware

Een basis middleware functie ziet er als volgt uit. De aangemaakte functie wordt geëxporteerd zodat ze bruikbaar wordt voor andere modules. Het geheel wordt in een javascript file gesaved (bvb: middleware.js). Zo kan bij createServer bvb. een externe middleware functie opgeroepen worden waarbij request en response als argumenten doorgegeven worden, al dan niet met de volgende middleware functie. Voor deze laatste gebruikt men meestal het keyword next.

```
function middleware(request, response, next) {
    //doe verwerking van request en response
    return next(); //roept de volgende middleware (= een callback dus) op.
}
module.exports = middleware;
```

De callback functie kan ook als functie in de middleware geregistreerd worden. Je maakt m.a.w gebruik van asynchrone middleware. De algemene opbouw hiervan ziet er als volgt uit. De return zorgt dat de callback slechts één keer op de eventloop komt.

```
function middleware(arg1, arg2) {
    return function (request, response, next) {
        //doe verwerking van request en response,
        //gebruik eventueel arg1 en arg2
        next(); //roept de volgende middleware op
    }
}
module.exports = middleware;
```

Consumeren van middleware

Het oproepen van de middleware kan vanuit elke module met de klassieke require methode, dank zijn de module.exports. Indien meerdere middleware componenten nodig zijn, kan de volgorde van het ophalen belangrijk zijn. Zo kan je geen middleware routing gebruiken, wanneer er geen voorafgaande middleware opgehaald was voor het definiëren van deze routings. Bij een foutieve volgorde blijven typisch het request of response object undefined.

```
var middleware = require("middleware");
var nextMiddleware = require("nextMiddleware");
```

Bestaande of opgehaalde middleware (= de externe file) kan meerdere bruikbare functies (middleware componenten) bevatten. Je hoeft ze niet allemaal te gebruiken.

De gewenste middleware functie kan rechtstreeks opgeroepen worden maar wordt meestal meegegeven als argument van een constructor. Vaak gebruikt men de variabele met naam "app" in plaats van de naam "server" om aan te duiden dat de server opgewaardeerd werd met middleware. Voorbeeld: middleware als argument van createServer:

```
var app = http.createServer(middlewareFunction("eenArg"))
```

In combinatie met een callbackfunctie, wordt ervoor gezorgd dat de middleware functie eerst uitgevoerd wordt vóór de callback (logisch). Zo kan in bovenstaand voorbeeld, de middleware functie zijn resultaat doorgeven aan het request en response object van create server.:

```
function middlewareFunction(someText) {
    return function (req, res) {
        res.write("<div>" + someText + "</div>");
        res.end();
    };
}
```

Middleware kan wel aan speciale voorwaarden onderworpen zijn. Dit kan een aanpassing vragen bij

gebruik van third party middleware. Zo haalt bij het express framework de methode bodyParser() de key/value form data op. Maar dat kan alleen maar werken bij een post () en een correct content type (form-urlencoded of application/json):

```
app.post('/send', express.bodyParser(), function(req, res) { })
```

14.2 De middleware van connect of express gebruiken.

Node heeft een http module die met createServer() **een http.Server object** returnt om http requests te behandelen. De Http module is op zich zeker een bruikbare module, maar vaak wordt de "request" en "connect" module erbij gehaald. De request module werd reeds kort aangehaald als "de simplified http client". Connect is een verzameling van middleware componenten.

Info: <https://www.npmjs.org/package/connect>

De "Connect" module zorgt met zijn eigen createServer() methode voor een uitbreiding van de node http.Server mogelijkheden. Connect zie je niet onmiddellijk staan in het npm overzicht, maar connect wordt wel gebruikt in het populaire express framework, dat verder behandeld wordt. "Express" beschikt eveneens over zijn createServer() methode met een nog uitgebreider versie van http.Server en nog meer mogelijkheden voor het eenvoudig parsen van bijvoorbeeld cookies, bijhouden sessions, logger voor specifieke request en response data, uitgebreide error handler, behandelen van statische files ... Redenen genoeg waarom express één van de meest succesvolle modules geworden is.

Het maken van een connect server of een express server is gelijkaardig aan http.createServer. Je installeert natuurlijk vooraf connect en/of express ((npm install connect) en brengt connect aan als argument in de constructor van createServer(). Daarna kunnen verschillende middleware componenten opgehaald worden met het "use" command. Ze worden met next() na elkaar opgeroepen.

```
var http = require("http");
var connect = require("connect") //kan ook express zijn
var app = connect(); //initialiseer middleware -> opgewaardeerde createServer

app.use(function (req, resp, next) {
  //eerste middleware is puur raw node als voorbeeld
  resp.setHeader("Content-Type", "text/html");
  resp.write("<div>Een middleware boodschap</div>");
  next(); //volgende middleware (= volgende use) ophalen
});

app.use(function (req, resp, next) {
  resp.end("<div> Hello World van een tweede middleware</div>");
})

http.createServer(app).listen(1337) // equivalent met app.listen(1337)
console.log("server listening on 8000");
```

Enkele bruikbare connect componenten zijn : app.use (connect.logger()) ,
app.use(connect.errorHandler()), app.use(connect.static(__dirname+'/public')),

```
app.use(connect.query()), app.use(connect.bodyParser()), app.use(connect.cookieParser()),  
app.use(connect.session()).
```

Connect en express zijn niet de enige succesnummers onder node middleware. Er bestaat daarnaast nog middleware voor o.a. CSRF (Cross-site Request Forgery) bescherming, Gzip compressie, JSONP handling, LESS support enz...

14.3 Zelf middleware aanmaken

De middleware componenten zijn niets meer dan een functie die geëxporteerd wordt voor algemeen gebruik door een andere module. Niets belet ons om zelf een middleware functie aan te maken en te exporteren.

```
function echoText(someText) {  
    return function (req, res) {  
        res.write("<div>" + someText + "</div>");  
    };  
}  
  
module.exports = echoText;
```

De aangemaakte middleware kan ook hier meegegeven worden *als argument in de constructor van createServer()*. Door de return komt de callback functie (met zijn req en res) beschikbaar voor verdere verwerking.

```
var http= require("http")  
var echoText = require("./MyMiddleware"); //Puntje ./ niet vergeten!  
  
var app = http.createServer(  
    echoText("Iets") //in constructor van createServer  
);  
  
app.listen(8080);
```

Het resultaat van een middelware bewerking wordt ook vaak *toegevoegd aan het request object*. (vb. req.user = user). Een request kan zonder probleem verstuurd worden met deze aangevulde gegevens. Dit request object is bovendien beschikbaar en wordt gebruikt in heel veel andere modules, zodat het resultaat vd middelware vlot beschikbaar komt in de applicatie.

14.4 Oefeningen op middleware:

Oefening: middelware voor het lezen van een tekst file

We maken gebruik van connect (= de use syntax) om 3 eigen middleware componenten na elkaar op te roepen:

- "writeHeader.js" schrijft het mime type van de header en maakt een cookie met de actuele datum aan.
- "saveRequest.js" schrijft enkele request eigenschappen weg naar een txt file. (bvb.: req.method, req.url, req.headers ...)
- "echoText()" zorgt dat een tekst vanuit de app door de module uitgeschreven wordt.

Voor de uitwerking worden de modules als middleware meegegeven via de constructor van `app.use()`. Voor de eerste module ziet dit er bvb. zo uit.

```
//in app.js
app.use(
  writeHeader("Content-Type", "text/html")
)

//in writeHeader.js()
function writeHeader(key , value) {
  return function (req, res, next) {
    res.setHeader(key, value);
    res.write("writeHeader done");
    next();
  }
}
module.exports = writeHeader;
```

Zorg verder dat er een cookie aangemaakt wordt in `writeHeader`.
Vul verder aan met module 2 en 3.

Oefening: requestHandlers als middleware

Een mini applicatie waarmee images gekozen worden om ze op te laden op de server ([file-upload](#)).

Om de requests uit te werken maken we deze keer gebruik van middleware voor de `requestHandlers` (= haal api data, lees html, lees script...). We maken een file met naam `requestHandlers.js`, die een verzameling wordt van geëxporteerde functies. Het hoeft dus niet als een module aangemaakt te worden.

```
exports.myFunctionA = function(req, res) {};
exports.myFunctionB = function(req, res) {};
```

De middleware wordt gebruikt door de router (= router kiest de juiste handler).

```
var requestHandlers = require("./requestHandlers")
```

Traditioneel wordt de middleware meegegeven in de `app.js` file bij opstart van de server. Zo is ook duidelijk, op één plaats, welke middleware de applicatie gebruikt:

```
staticServer.init( router , handlers ,8080 );
```

Oplossing:

1. Ofwel start je een blanco applicatie waarbij je de static server deels herbouwt, ofwel vertrekken we van de eerder gemaakte oefening (static server) en refactoren we deze applicatie naar een aantal afzonderlijke files. We kiezen het laatste en maken volgende server scripts aan:
 - a. requesthandlers.js: Zorgt voor de verwerking van de requests op basis van een aantal handlers. Dit kan een handler zijn voor het lezen van de root , een handler voor het lezen van een file (`readFile`), een handler voor het ophalen van API data(`apiData`) , een handler voor het opladen van een file (`uploadFile`). De verzameling van handler functies in `requestHandlers.js` ziet er als volgt uit:

```
//public
var root = function (req, res) {
    getFile( 'index.html' , res);
}
var apiData = function(req, res) { // uit te werken }

var getFile = function (uri ,res) { ....
    //haal extensie op
    // lees file
};

// exports request handlers--
exports.root = root;
exports.apiData = apiData;
exports.getFile = getFile;
```

- b. router.js: roept de juiste request handler van punt a op naargelang de ontvangen url (=de ontvangen pathname). De router moeten we herwerken.

Dit kan bvb.als volgt:

```
var getRoute = function (handlers, request , response) {
    var basename = path.basename(request.url) || "index.html",
        ext = path.extname(basename);
    pathname = "/" + basename.split("?")[0];

    //ophalen van de juiste handler
    handlers[pathname](request, response);
}

module.exports.getRoute = getRoute;
```

Hierbij is handlers een literal object (of een Array) , dat we aanmaken in app.js, om alle middleware functies logisch te groeperen. Dit is een veel gebruikte techniek voor het groeperen van middleware:

```
var handlers = {}
handlers["/] = requestHandlers.root;
handlers["/apiData"] = requestHandlers.apiData;
```

Vergeet ook niet error controle toe te voegen. Wat indien:

```
typeof localMaps[ext] === 'undefined' → return een 404
```

2. Hoe brengen we deze middleware functies aan in onze static server?

We haalden reeds aan dat dit via de constructor dependancies gebeurt van staticServer.init(). Zo gebeurt alle initialisering mooi in één file: app.js. De volledige app.js , met alle initialisaties, ziet er nu zo uit:

```
//1. Nodige middleware en scripts in app.js.
var requestHandlers = require("./scripts/requestHandlers.js");
var router = require ("./scripts/router.js")
var staticServer = require("./scripts/staticServer.js");
//2. Alle handlers definiëren (leesbaar) in een literal object
var handlers = {}
handlers["/] = requestHandlers.root;
handlers["/apiData"] = requestHandlers.apiData;
handlers["/getFile"] = requestHandlers.getFile;
```

```
    . . .
//3. Initialisering van static server
staticServer.init( router , handlers ,8080 );
```

3. Natuurlijk moet de staticServer.js module (!) nu aangepast worden om vanuit zijn init() de server op te starten. Dit wordt bvb. (deels)

```
//private
var httpListen = function (router, handlers, httpPort) {
    var server = http.createServer(function (req, res) {

        //routing ophalen
        var uri = url.parse(req.url).pathname;
        console.log("routing ophalen voor uri : " + uri)

        router.getRoute(handlers, req, res, function(err, contents,mimeType) { })
    })

    server.listen(httpPort);
};

//public
var init = function (router, handlers , httpPort) {
    console.log("server running on port :", httpPort);
    httpListen(router, handlers, httpPort);
};

return {
    init: init
}
```

4. Test uit of de server correct werk door bvb. een html file uit te lezen waar je een menu in onderbengt. Dit laat ons toe een nieuwe pagina op te roepen voor de gevraagde file upload.

```
<nav>
    <ul>
        <li><a href="index.html">INDEX pagina met CSS</a></li>
        <li><a href="fileUpload.html">File upload</a></li>
        <li><a href="flickrAPI.html">Foto's zoeken met flickrAPI</a></li>
    </ul>
</nav>
```

5. Aanmaken van de fileUpload.html. We maken gebruik van het type="file"

```
<form action=".upload" enctype="multipart/form-data" method="post">
    <input type="file" name="upload" multiple="multiple">
    <input type="submit" value="upload één of meerdere files" />
</form>
```

6. Definieer deze nieuwe handler in app.js:

```
    handlers[ "/upload" ] = requestHandlers.upload;
```

Maak hem aan in requestHandlers.js:

```
    var uploadFile = function (req, res) { }
```



En zorg dat hij beschikbaar wordt buiten zijn module:

```
exports.upload = uploadFile;
```

7. De upload handler verder uitwerken:

a. Post data lezen:

We dienen de post data van het formulier op te halen. Er kan hiervoor gebruik worden gemaakt van een third party library. We kiezen voor bvb. Multiparty. Voorbeeldcode en informatie: <https://github.com/andrewrk/node-multiparty>

```
multiparty = require('multiparty');

var form = new multiparty.Form();
form.parse(req, function (err, fields, files) {
    filename = files.upload[0].originalFilename;
    //Hier filename verwerken als ReadStream en WriteStream
})
```

b. Omdat we het geheel willen verwerken met pipe hebben we een inbound en outbound stream nodig om de file op disk te schrijven.

```
var inStream = fs.createReadStream(files.upload[0].path);
var outStream = fs.createWriteStream("./public/images/uploads/" +
filename);
inStream.pipe(outStream);
```

8. Tot slot wordt als bevestiging het opgeladen beeld uitgelezen en getoond.

We kunnen een bewaard beeld direct ophalen, maar doen het hier eveneens via een handler, die aangestuurd wordt met een querystring.

```
res.write("<div>Volgend beeld werd opgeladen:</div>")
var imgURL = 'http://localhost:8080/show?img=' + filename;
res.write("<img src='" + imgURL + "' width='250' alt=''/>");
```

9. De handlers ["/show"] moet hiervoor worden uitgewerkt. Doe dit nu zelf, haal de filename op uit de querystring en maak gebruik van:

```
fs.readFile("./public/images/uploads/" + filename , function (err, data) { });
```

Noot: Heb je een warning gekregen voor een max. aantal eventListeners (default ingesteld op 10), kijk dan of een event niet continu luistert binnen de server listener (server.listen(httpPort)). Plaats de listener erbuiten, en geef de nodige argumenten mee vanuit de emitter.emit. Vaak worden daarom het response en request object meegegeven als argumenten.

15 Frameworks: Express

15.1 Node Waarom express?

We weten reeds dat het gebruik van middleware ons programmeerwerk verlicht. Connect voorziet verwerkingsmogelijkheden voor http request en response, net zoals ook het "express" framework dit doet. Maar express kan meer.

Express is een webapplicatie framework voor node dat een oplossing aanbiedt voor alle low level code nodig om deze nodeapplicatie op te zetten. Express is te beschouwen als een klein MVC framework voor nodejs dat (REST) routing afhandelt en callbackfuncties hierbij ophaalt. De routingsinstructies zijn vergelijkbaar met controller acties in het MVC patroon. De HTML pagina's zijn natuurlijk de views. Hierbij kan Express ook overweg met templates zoals view engine templates of less templates voor CSS instructies.

Documentatie van Express is te vinden op: <http://expressjs.com/>
Storify en MySpace maken onder andere gebruik van het Express framework.

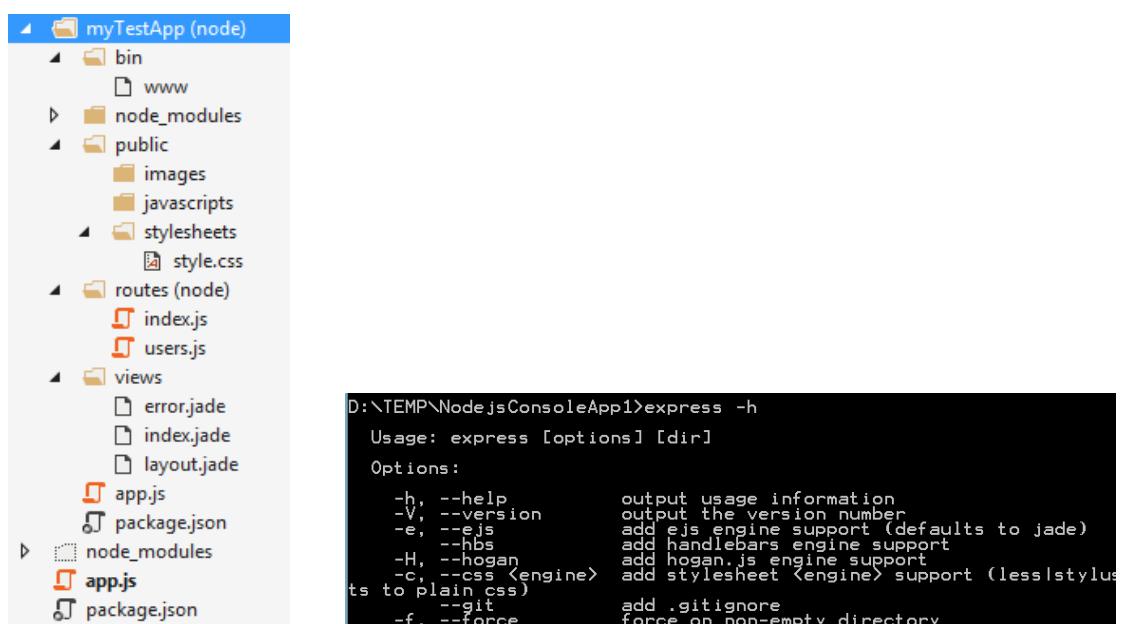
Naast Express bestaan nog frameworks voor node: Geddy, Yahoo!Mojito (richt zich op een mobiele omgeving), Flatiron (zorgt dat enkel de gewenste componenten kunnen opgenomen worden). Elk van deze frameworks maakt gebruik van het MVC patroon.

15.2 Opbouw van een express applicatie:

1. Installeren van express :

```
npm install -g express
```

2. Na het installeren van de express module kan een skeleton applicatie gevormd worden (7 directories en 9 files) . Dit kan zowel via de "Express generator" als via een IDE.



- a. Express skeleton applicatie aanmaken via een node command en de "Express application generator":

npm init npm install express -- save	maakt package.json aan via aanbevelingen zorgt voor een lokale express installatie
npm install express-generator -g express -h express myTestApp cd myTestApp && npm install SET DEBUG=myTestApp: & npm start	globaal installeren van de generator help voor express maakt het skeleton aan in de map myTestApp installeert alle dependancies run TestApp

- b. Express skeleton applicatie aanmaken via NTVS:

Via NTVS kan een leeg express project aangemaakt worden. Bij aanmaak zal VS een reeks modules installeren (terug te vinden in package.json). De modules samen met hun dependencies worden opgehaald en geïnstalleerd.

3. Entry point: app.js

app.js is het entry punt van de applicatie. In app.js worden afhankelijke modules toegevoegd met "require", middleware wordt opgehaald met "app.use(myMiddleware())" en "app.set(key, value)" zorgt voor een basis configuratie. Daarna staat de Express server klaar om te luisteren op de ingestelde poort (app.set('port', process.env.PORT || 3000);

Bij connect werd de server als volgt aangemaakt:

```
var connect = require('connect');
var app = connect();
app.listen(3000) // zelfde als http.createServer(app).listen(3000);
```

Voor express is dit gelijkaardig:

```
var express = require('express');
var app = express(); // voert express.createServer() uit
app.listen(3000)
```

Er is wat verschil in de uitwerking tussen express3 en express4. Wel wordt bij beide het listen event uitgevoerd nadat alle andere (app) methoden als correct geëvalueerd werden. De middleware methoden omvatten o.a. bodyParser, logger, cookieParser en natuurlijk express zelf.

Bij express3 start de applicatie bij app.js :

```
http.createServer(app).listen(app.get('port'), function(){ });
```

Bij express4 is het start script terug te vinden in ./bin/www. Daartoe is app.js als module geëxporteerd. Opstarten gebeurt via:

```
app.listen(app.get('port'))
```

Voor de migratie naar een nieuwere versie (intussen wordt aan Express5 gewerkt) voorziet

de documentatie een migratie guide : "Migrating from 3.X to 4.X"

4. Routing bepalen:

In app.js worden een aantal belangrijke taken geïnitialiseerd zoals de routing. Natuurlijk gebeurt dit op een asynchrone manier.

```
var routes = require('./routes/index');
app.use('/', routes); zorgt voor routings terug te vinden in de file index.js.
Deze index.js file maakt gebruik van de express.Router();
```

5. Express.Router():

Om de routes te configureren worden event listeners gebruikt. Deze listeners worden niet rechtstreeks toegevoegd aan het request object van node, maar dit gebeurt met de express methoden en de express.router(). Deze router kan rechtstreeks de http verbs accepteren. Best handig!

Vb.: Bij ontvangst van een get request komt het path als eerste argument. Door in app.use() een extra stap toe te voegen (app.use('/extra/', routes); worden alle routes voorafgegaan door /extra/. Ieder path van deze routing komt m.a.w. terecht in de namespace 'extra'.

Dit path kan als een URL opgegeven worden maar ook als een reguliere expressie.

```
router.get('/', function(req, res) {
  //stuurt automatisch de header mee. Was manueel werk bij nodejs
  res.send('Welkom op de root van de applicatie')
})
router.get('/Welcome', function(req, res) {
  res.send('Welkom bij EXPRESS')
})
//ophalen van benoemde parameters zoals productID kan m.b.v. een dubbel
punt
//opvragen kan via req.params object
router.get('/products/:productId', function(req, res) {
  // stuurt automatisch de header mee
  res.send('Gevraagd product is' + req.params.productId)
})
```

bij editering:

router.get '/edit/:id', (req, res) -> met een callback functie res.send('iets' + req.params.id)

6. Enkele routing voorbeelden, waarbij duidelijk wordt hoe een javascript object zorgt voor specifieke renderings

Foutpagina oproepen	router.get('/error', function (req, res) { res.status(500); res.render('error'); });
Extra context meegeven zoals een cookie of een session kan via het request object (req)	router.get('/userPreference', function (req, res) { res.render('preferences', { userColor: req.cookie.usercolor, userName: req.session.username, }); });

	}
Negeren van layout pagina	res.render('specialLayout', { layout: null });
Optionele routing wordt met een ? aangeduid.	router.get('/:page/:admin?', function (req, res) { var admin = req.params.admin; if (admin) { . . . } })
Ook wild characters kunnen gebruikt worden.	router.get('/:page/*', function (req, res) { })

7. Express maakt gebruik van zijn eigen middleware om static files aan te brengen in een default map 'public':
`app.use(express.static(path.join(__dirname, 'public')));`

Dit zorgt ervoor dat files in public behandeld worden als statische files. Maak je in de map public een map images, dan worden de images opgehaald met een `src='/images/imgName.jpg'`. Let op de absolute adressering zonder punt.

8. Index.js in de map routes.

Het response object in de router files is door de middleware uitgebreid met de render methode. Deze render methode laat toe om met een javascript object databinding te realiseren met data in de views folders.

```
res.render('index', {
    'title': 'Express'
})
```

In de views wordt deze data opgehaalt met een spoorwegteken : `#{title}`

9. View template engine: index.jade: <http://jade-lang.com/>
 Express laat het gebruik van willekeurige template engines toe.

- a. Zonder template engine (puur html):

```
Ref: http://expressjs.com/4x/api.html#res.sendFile
res.sendFile('../views/index.html'); //vanaf express 4.8
res.send('<p>Hier HTML </p>');
```

Initialiseren van de mappen voor script, stylesheets, images kan met :

```
app.use("/styles", express.static(__dirname +
    '/public/stylesheets'));
```

- b. Met een template engine:



```
res.render('index')
Express maakt default gebruik van jade als view engine.
app.set('view engine', 'jade');
```

De map waar de views staan wordt eveneens gedefinieerd in app.js met
`app.set('views', path.join(__dirname, 'views'));`

Een basis HTML layout (master layout voor alle pagina's) is gedefinieerd in layout.jade.
 De expressie "block content" (in layout.jade maar ook in de andere jade files) toont waar de specifieke pagina inhoud terecht komt.

`block content`

10. Er wordt gebruik gemaakt van Stylus als CSS preprocessor:

<https://learnboost.github.io/stylus/>

De Stylus files met suffix .styl zijn terug te vinden in de map public>> stylesheets.

app.use(require('stylus'))

In dezelfde map wordt tijdens het starten van de app de styl files gerendered naar een gelijknamige css file. Wijzigingen die je aanbracht in de css file worden hierbij overschreven.

Noot: express wordt nog regelmatig aangepast. Een update kan op een applicatie uitgevoerd worden via: "npm update -g express". Ook de generator, die het basis website skelet aanbrengt met templates, kan aangepast worden met "npm update -g generator-express". Dit is vooral nuttig mocht je express global op je server geplaatst hebben.

15.3 View Engine (Jade)

Express voorziet een aantal verschillende view template engines. Geen echte voorkeur. Veel gebruikte: EJS en Jade.

- Jade : <http://jade-lang.com/>
- EJS: <http://blog.imaginea.com/ejs-with-node-js/>

Naar conventie worden de Views default in de "views" folder geplaatst. Partial views komen meestal in een subfolder "partial" van "views" terecht.

Dedicated karakters in de views tonen waar de javascript variabelen (lees.viewmodel data) moeten toegevoegd worden. Om een onderscheid te maken tussen het versturen van een response (res.send()) en het verwerken van een view template wordt res.render() gebruikt. De instructie res.render() verzorgt de template binding.

We kiezen Jade omdat het iets meer gebruikt wordt dan EJS bij node. De basis principes van Jade zijn de volgende:

1. Door het inspringen van de tekst wordt de HTML boom opgebouwd. Inspringen gebeurt met spaties of met tabs. Beide systemen mogen/kunnen niet(!) gemixed worden!
2. Html tags worden zonder het teken < of > geschreven. Attributen komen tussen haakjes.
`input(id="inpClient", type="text")`
3. Commentaar: commentaar kan met // of <!-- -->
Commentaren aangemaakt met //-- zijn niet zichtbaar in de gerenderde HTML
4. Met # wordt een aangeboden model gerendered:
`#{header} #{user.username}`
Met # wordt ook divs aangemaakt met de tekst als id:
`#myId` rendert als `<div id="myId"></div>`
Met een punt . wordt een div aangemaakt met de tekst als class
`.myClass` rendert als `<div class="myClass"></div>`
5. Met het gelijkheidsteken wordt een variabele opgehaald: `h1= title`
6. "Niet gelijk aan" als != haalt een variabele op zonder te escapen (om bijvoorbeeld de html tags weer te geven van een partial template)
7. Koppelteken (-) laat toe javascript te schrijven: - `for(var username in users) { }`

8. Linken van externe files zoals javascript:
`script(src="/javascripts/chat.js")`
9. Inline script invoegen, kan door een punt na script:
`script.
var userID = "#{userID}";`
10. Een directive voor het toevoegen van basis layout: extends ../layout.
In layout.jade gebeurt het inserteren via: block content
11. Het gebruik van partials kan op twee manieren:
 - a. via partial:
`!= partial(template file name/options')`
De != zorgt dat de inhoud als HTML gerendered wordt.

Een ! zonder het gelijkheidsteken zorgt dat HTML (tags, tekens) niet geëncodeerd worden.
 - b. via include:
`include ../includes/footer`
12. Het omgekeerde van include is is extend. Dit laat toe om vanuit een pagina "html block delen" te sturen naar bijvoorbeeld de layout pagina:
`extend layout`
13. Concateneren doe je met het + teken of met het # teken. De + wordt gebruikt in code (javascript). Het # wordt gebruikt in HTML en rendert de inhoud:
Met +: info = myData[student].name + " uit " + myData[student].city
Met #: div De naam is : #{myData[student].name}
14. Meer info : <http://jade-lang.com/reference/>

Noot: Jade kan ook gebruikt worden op front-end (in de browser) en op de server. Hiervoor gebruik je een library zoals jade-browser (<https://www.npmjs.org/package/jade-browser>), waarbij server side templates aan de browser bezorgd worden.

15.4 CRUD: Users toevoegen via een webform met express

Als oefening voegen we users toe aan de standaard express app. Voorlopig maken we de applicatie volledig in het geheugen met een json file en zijn de user gegevens niet persistent. De json file, wordt dus niet overschreven om aangebrachte wijzigingen te bewaren..

1. Op root niveau wordt een map "data" aangemaakt met daarin een users.json file.

```
{  
  "johan": {  
    "username": "Johan",  
    "name": "Vannieuw",  
    "profession": "docent"  
  },  
  
  "lambik": {  
    "username": "Lambik",  
    "name": "Vandersteen",  
    "profession": "student"  
  }  
}
```

2. De routes plaatsen we niet in de app.js file (!) maar wel in een user.js file onder de map routes. Zo blijft alles overzichtelijker en blijft ook app.js leesbaar.

Om alle users op te vragen roep je de deze file op en pas je de routing aan.:

```
var users = require('../data/users.json');
```

Bemerk de adressering voor de router, die verder bouwt (concatineert) op de configuratie app.use('/users' , users) uit app.js. De volledige URL wordt hierdoor http://localhost:****/users/:

Het oproepen van de view gebeurt realtief (start niet met /) en concatineert met de ingestelde views map. Hier wordt de file views/users/index opgehaald

```
router.get('/', function (req, res) {  
    res.render('users/index', { title: 'Users', users: users });  
});
```

3. Het opstarten van de website gebeurt logischer wijze bij de root (of index) van de website. Volgende instructie zorgt dat de index routes bereikt worden:

```
app.get('/', routes.index);
```

Wil je opstarten bij /users , dan kan je de routing hiervoor aanpassen of in de root index een redirect plaatsen:

```
res.redirect('/users');
```

4. Maak de map users aan in view >> map users. Daarin komt de template index.jade met het overzicht van alle users onder de vorm van links. Bovenaan komt een link om een nieuwe user toe te voegen. Let op een juiste inspringing (tabs), anders zal jade verkeerd nesten. Schrijf zonder spaties. Zorg ook dat met encodeURIComponent() script injectie verhinderd wordt.

```
extends ../layout
```

```
block content  
    h1= title  
    p  
        a(href="/users/new") Maak een nieuwe gebruiker aan.  
    ul  
        - for(var user in users) {  
            li  
                a(href="/users/" +  
                    encodeURIComponent(user)) = users[user].username  
        - };
```

Noot: Een verzameling kan ook met een "each" getoond worden:

```
ul  
    each user in users.list  
        h2 #{user.name}
```

5. Test met "localhost:****/users" of je alle users te zien krijgt en de routing zijn werk doet.
6. De routing in routes>>user.js om één gebruiker op te vragen ziet er als volgt uit. Let op het dubbele punt zodat willekeurige namen geaccepteerd worden. Bij url's waarbij een variabele inhoud kan meegegeven worden spreekt men vaak over "variable url's".

```
router.get('/:name', function (req, res, next) {
```

```
var user = users[req.params.name];
if (user) {
    res.render('users/details', { title: 'User profile', user: user });
} else {
    next();
}
});
```

Noot: Ook een verzameling (i.p.v. een enkelvoudig object) kan meegestuurd worden in render. Volgende fictieve code zou bijvoorbeeld toelaten om alle users die docent zijn (of een ander beroep, vermeld in de url) door te sturen naar een view:

```
var parts = req.originalUrl.split('/');
var profession = parts[parts.length - 1];
res.render('users/profession', {
    users: {
        list: users[profession],
        profession: profession
    }
});
```

7. In views>>users>>details.jade wordt het detail van de user getoond.

```
h1 gegevens voor <i>#{user.name}</i>
p  #{user.username} met beroep #{user.profession}
```

Vul dit eventueel zelf verder aan met een foto en voeg een terug link toe.

8. Voeg de routing toe voor een nieuwe user:

```
router.get('/new', function (req, res) {
    res.render('users/create', { title: "New User" });
});
```

9. Maak een users>>create.jade template ervoor aan. Dit is een post formulier. Vergeet extends en block niet, met bijgevolg een extra indent voor de form tag.

```
form(method="POST", action="/users")
p
    label(for="username") Username<br />
    input#username(name="username")
p
    input(type="submit", value="Create")
```

10. De uitwerking van de POST om een nieuwe user toe te voegen vraagt de verwerking van form data. We gebruikten reeds middleware hiervoor, maar express zelf voorziet ook zijn gekozen middleware hiervoor. Hiertoe roept app.js de bodyParser en nodige middleware functies op:

```
var bodyParser = require('body-parser');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
```

en zien hoe in app.js de nodige middleware functies opgehaald worden:

```
app.use(bodyParser.urlencoded({
```

```
        extended: true
    }));
});
```

11. Vul de routes>user.js aan voor deze POST en test uit:

```
router.post('/', function (req, res) {
    if (users[req.body.username]) {
        res.send('Conflict', 409);
    } else {
        users[req.body.username] = req.body;
        res.redirect('/users');
    }
});
```

Noot: Krijg je in Express3 een 500 fout “cannot read property of undefined” dan doe je vermoedelijk de require('/routes/user')(app) nog vóór je de bodyParser middleware oproept. Hierdoor ontbreekt de middleware in je app (req.body is undefined)

12. Op de details pagina voegen we een delete knop toe; Bemerk hoe de delete methode kenbaar gemaakt wordt via het _method attribuut.

```
form(action="/users/" + encodeURIComponent(user.username), method="POST")
    input(name="_method", type="hidden", value="DELETE")
    input(type="submit", value="Delete")
```

13. In routes>>user.js gebeurt de delete verwerking.

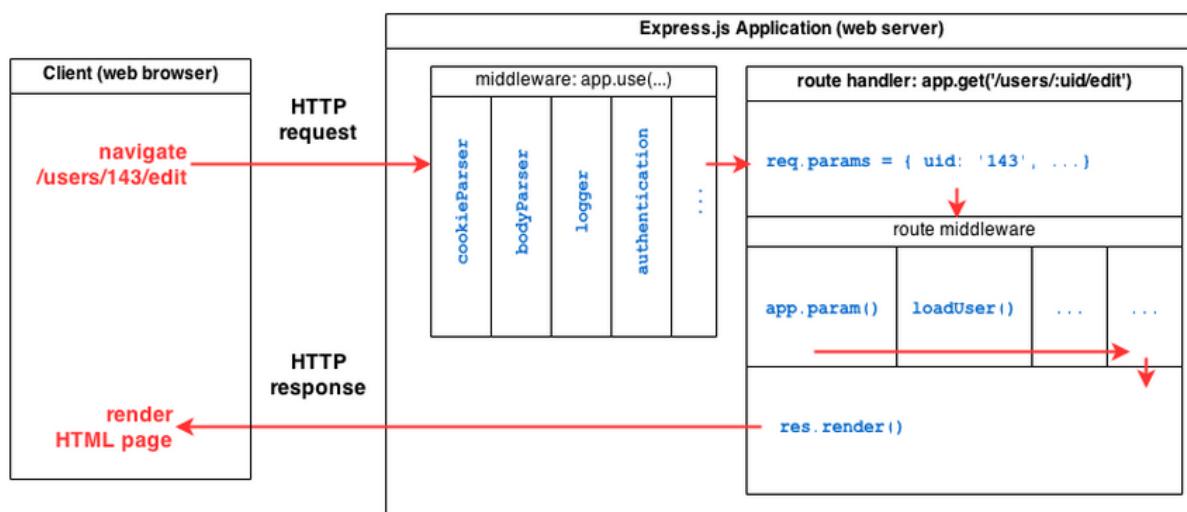
Pas op: “delete” is een RESERVED word in javascript (gebruik bijvoorbeeld “del”). In express werd dit naam probleem opgelost door de router te voorzien van een delete (router.delete(function (req, res, next) { }))

De router herkent echter alleen zijn delete wanneer die gestuurd wordt via XMLHTTP. Hierdoor zijn we bij HTML5 forms (waar momenteel PUT en DELETE in de spec in vraag gesteld worden) aangewezen op een universeler oplossing met POST en een manueel toegevoegde “_method” formdata met waarde “DELETE” of “PUT”..

```
router.route('/:name').post(function (req, res, next) {
    if (users[user_name]) {
        switch (req.body._method) {
            case "DELETE":
                delete users[user_name];
                break;
            case "PUT":
                users[user_name] = req.body;
                break;
            default:
                next();
        }
        res.redirect('/users'); ;
    }
});
```

14. Optioneel: Zorg voor het editeren van een user via een PUT request.

15.5 Express en middleware



Middleware wordt toegevoegd als argument aan de constructor en zorgt voor een extra bewerking, voorafgaand aan de callbackfunctie. Bij express kan middleware toegevoegd worden aan de methodes app.get(), app.post(), router.get(), router.post() vanaf het tweede argument (eerste argument bevat router expressie). We houden de callbackfunctie steeds als laatste element. Het request object wordt gebruikt als container voor het bijhouden van data.

Voorbeeld waarbij een req.content aangemaakt wordt vanuit een tekst file:

```
//Module middleware.js in de routes map -----
fs = require('fs');

var middleware = function(req, res, next) {
    //dirname returnt application root
    fs.readFile( './data/Test.txt', function( err, data ) {
        req.content = data;
        next();
    });
};

module.exports = middleware;

//routing die gebruikt maakt van middleware.js-----
//eerst wordt de middleware gerund, daarna gaat get verder met de callback
app.get('/readFile', middleware, function(req,res) {
    res.send("Dit is de tekst via middleware : " + req.content );
});
```

Test dit en zie of je de inhoud van de tekstfile in de webpagina krijgt.

Middleware wordt zo vlot ingeschakeld bij bvb. een post voor extra validaties:

```
app.post('/products/:id', authenticateUser, validateProduct, addProduct);
function authenticateUser(req, res, next) {
    //authenticatie
    next()
}
function validateProduct(req, res, next) {
    //validatie
```

```

        next()
    }
    function addProductr(req, res, next) {
        //toevoegen aan file of database
    }
}

```

De techniek wordt ook regelmatig gebruikt voor foutafhandeling, waarbij een error meegegeven wordt naar een default aanwezige error handler (`new Error()`). Zonder de handler breekt de site. Met de handler kan de fout opgevangen worden.

```

app.get('/doeIets', function (req, res, next) {
    //bij fout
    next(new Error('oproep vd error handler'));
    res.send("om het even");
});

//custom error handler
app.use(function (err, req, res, next) {
    console.error(err.stack);
    res.send(500, 'Er gebeurde een server fout');
});

```

In app.js wordt om die reden ook de default error handler opgeroepen NA de geconfigureerde routes.

```

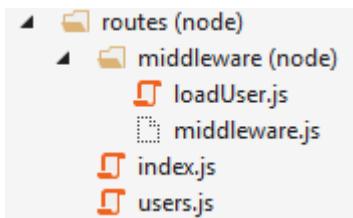
app.use(function (req, res, next) {
    var err = new Error('Not Found');
    err.status = 404;
    next(err);
});

```

Oefening: update van de user uitwerken met loadUser middelware

1. Vervolledig de user toepassing met een update (_method put) mocht dit nog niet gebeurd zijn.

2. Maak onder de routes een map middelware aan met daarin de file "loadUser.js":



3. Een te exporteren (!) middelware methode returnt het volledige user object op basis van zijn key:

```

function loadUser(req, res, next) {
    var user = req.user = users[req.params.name];
    if (!user) {
        res.send('Niet gevonden of onbestaande gebruiker', 404)
        next();
    } else {
        next();
    }
}

```

4. Zorg dat de editering van een user de middleware gebruikt.
`router.get('/:_name',loadUser , function (req, res) { . . .`
5. Pas deze middleware ook toe op andere CRUD acties waar de user opgehaald wordt.

15.6 Oefeningen met Express

Oefening: Meer met opmaak (stylus en bootstrap):

1. Toevoegen van css link:

Dit gebeurt natuurlijk in de layout pagina.

Stylesheets in de head:

```
link(rel='stylesheet', href='/stylesheets/style.css')
```

2. De css style sheet kan met stylus gescaffold worden vanuit een *.styl file. De syntax van een *.styl file is identiek aan de jade syntax. Bij het opstarten zorgt style ervoor dat de CSS file in de achtergrond aangemaakt wordt.
Stylus is geïnspireerd op SASS. Meer info: <http://learnboost.github.io/stylus/>
Een tabel via styl kan er bvb. als volgt uitzien:

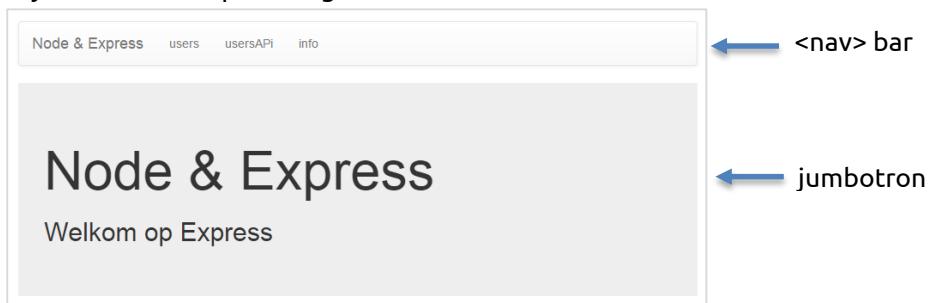
```
table
    background:#CCC;

    th
        background:#EEE;
        padding:10px 20px;
        text-align:center;

    tbody
        padding:0; margin:0;

        td
            background:#FFF;
            padding:5px 10px;
            text-align:center;
```

3. Wil je liever gebruik maken van bootstrap, dan kan bootstrap opgeroepen worden in de layout.jade pagina. Dit geeft ook de gelegenheid een <nav> bar toe te voegen, waarbij de stijl door bootstrap verzorgd wordt:



- a. Voeg in de head de files van bootstrap toe. Je kan ze downloaden of een CDN gebruiken.

```
script(src='//cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/3.2.0/js/bootstrap.min.js')

link(rel='stylesheet', href='//cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/3.2.0/css/bootstrap-theme.min.css')
```

```
link(rel='stylesheet', href='//cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/3.2.0/css/bootstrap.min.css')
```

Noot: ook via npm kan bootstrap geïnstalleerd worden met inbegrip van een startproject: <https://www.npmjs.org/package/twitter-bootstrap-node>

- b. Maak een block aan met daarin de navbar. Respecteer de indentation (inspringing);
Dit is stylus :

```
block navBar
  nav(class='navbar navbar-default', role='navigation')
    .container-fluid
      .navbar-header
        button(class='navbar-toggle', data-toggle=..)
```

Voor de rest van de code die je kan omzetten naar jade, zie :
<http://getbootstrap.com/components/#navbar>

- c. Maak gebruik van blocks om meer flexibiliteit te hebben. Voeg bijvoorbeeld een

```
block pageTitle
```

```
  .jumbotron
    h1= title
```

- d. Om partial views in te laden kan je gebruik maken van include. Denk eraan dat je ook een jade file kan maken met raw html.

```
block footer
  include footer.jade
```

4. Om de site mobieler te maken voegen we een viewport toe aan layout.jade. Bootstrap doet verder het nodige bij kleinere schermen en klapt het menu samen.

```
meta(name='viewport', content='width=device-width, initial-scale=1.0')
```

Oefening: Meer met javascript (ajax calls)

De pagina's worden gerendered door express met het commands zoals

```
res.render('users/index', { title: 'users', users: users })
res.send('Conflict', 409)
```

In plaats van de pagina's volledig te renderen kunnen we ook gebruik maken van XMLHttpRequest, waarbij de cliënt een ajax call uitvoert. In dit geval dient de server een REST api te voorzien voor de users behandeling:

De server moet nu json versturen (of XML) Dit kan op twee manieren:

```
res.json(users); // express
```

```
res.end(JSON.stringify(users)); //raw javascript
```

Maak op de server hiermee de nodige api routing (= een api controller 'usersAPI.js') aan:

```
router.get('/', function (req, res) { res.json(users); })
```

De client zorgt met javascript voor ajax calls.

1. Toevoegen van (CDN) javascript voor AJAX:

jQuery in de head

```
script(src='http://ajax.googleapis.com/ajax/libs/jquery/2.0.3/jquery.min.js')
```

Javascript als laatste in de body om de GUI thread niet te onderbreken:

```
script(src='/javascripts/index.js')
```

Noot: jQuery of een customised versie van jQuery met enkel de nodige modules kan ook geïnstalleerd worden via npm: <https://www.npmjs.org/package/jquery>.

2. Het gebruik van namespaces voor client script is sterk aangeraden (zoniet verplicht) omdat de volledige toepassing in javascript gemaakt is. Globale variabelen kunnen snel de scope vervuilen of voor collision zorgen. Een namespace aanmaken volgens het module pattern kan ook hier voor de oplossing zorgen en ziet er deels als volgt uit:

```
var client.api = {};
client.api = (function () {
    //blijft local binnen de client.api namespace
    var start = $(document).ready(function () { ... });
    return {
        //is een global javascript object binnen de namespace
        start: start
    }
})();

client.api.start;
```

3. Het ophalen van de users met cliënt ajax calls gebeurt bvb. met jQuery als volgt

```
$.getJSON('/api/users/', function (data) {
    var tableRows = "";
    $.each(data, function () {
        tableRows += '<tr>';
        tableRows += '<td><b>' + this.username + '</b></td>';
        tableRows += '<td>' + this.email + '</td>';
        // . . .
        tableRows += '<td><a href="#" rel="' + this._id + '">details</a></td>';

        tableRows += '</tr>';
    });

    // Tablerows toevoegen aan tbody
    $('#userList table tbody').html(tableRows);
```

4. Deze routing kan verder op eenzelfde manier aangevuld worden met een create, update, delete. De express4 router herkent nu wel de put en delete verbs omdat er gebruik gemaakt wordt van XMLHTTP via \$.ajax calls.

Werk dit uit. Onderstaand voorbeeld maakt gebruik van <http://cwbuecheler.com/web/tutorials/2014/restful-web-app-node-express-mongodb/> maar dan voorlopig zonder een database.

[Maak een gebruiker aan](#)

Gebruikers overzicht

gebruikersnaam	beroep	email			
johan	docent	johan@howest.be	delete?	details	update
steven	docent	steven@howest.be	delete?	details	update
george	student	superman@howest.be	delete?	details	update

```
//---- API controller
router.get('/', function (req, res) [...]);

router.get('/details/:name', function (req, res) [...]);

router.delete('/delete/:name', function (req, res) [...]);

router.post('/create', function (req, res) [...]);

router.put('/update', function (req, res) [...]);
```

Typisch bestaat de routing uit 4 delen:

- Haal de nodige informatie op uit de url (bvb. req.body)
- Zorg voor server validatie (mocht cliënt validatie disabled zijn)
- Pas de collectie aan (array, json list, database)
- Stuur een http response (in JSON!) terug naar de cliënt met het resultaat.

Voorbeeld bij een post:

```
router.post('/api/users/create', function (req, res) {
    //1. user info ophalen vanuit request
    var formData = req.body;
    var user = {};
    user.email = formData.email;
    user.username = formData.username;
    user.profession = formData.profession

    //2. validatie aan server kant
    if (typeof user.username === "undefined") {
        res.statusCode = 400;
        res.send(" een gebruikersnaam is verplicht");
    }

    //3. json collectie aanvullen indien nieuwe user
    if (users[user.username]) {
        res.send('Conflict', 409);
    } else {
```



```
users[user.username] = user;
//4. cliënt HTTP message als json
res.send({msg:user.username});
}
});
```

16 Unit testing

16.1 Type testen:

Maak eerst een onderscheid tussen end-to-end testen (ook integratie testen genoemd) en unit testen. Beide zijn nodig wil men via testen een goed geschreven applicatie deftig onderhouden.

- Bij end-to-end testen test men *alle* componenten van een applicatie flow van UI tot database. End-to-end test is duidelijk een integratie test (integreert verschillende app. onderdelen).
- Bij unit testen concentreert men zich op één geïsoleerd (!) onderdeel of component van de applicatie. *Geen enkel* ander onderdeel wordt betrokken in de test. Unit testing leent zich perfect voor TDD (Test Driven Development). Voorbeeld: Heb je voor het testen van jouw unit (zoals een repository) data nodig, dan maak je geen gebruik van de database maar maakt de unit test gebruik van faked data.
- Integratie testen testen meer dan een single unit code. Verschillende unit kunnen *interageren met elkaar* om het testresultaat te bekomen. Voorbeeld: Men wil een unit testen maar heeft nood aan een sockets. Voor de sockets maak je gebruik van de bestaande socket.io library. Je integreert socket.io in jouw unit test.
Integratie testen kunnen omgezet worden naar verschillende unit testen door extra gebruikte methodes of modules te faken met spy's, stubs of mock (komt verder aan bod).

Testen of werken volgens TDD (Test Driven Development) is niet de favoriete bezigheid van veel ontwikkelaars. Nochtans, blijft bij complexe applicaties of bij applicaties met meerdere developers TDD wel sterk aangeraden. Het zorgt voor automatisering van testen (i.p.v manueel testen) en hergebruik van testen (i.p.v steeds hetzelfde manueel testen) en robuustere code (typische eigen fouten maak je minstens twee keer), verhindert fouten bij refactoring, zorgt dat de wensen van de klant of een moeilijke redenering correct geprogrammeerd wordt, laat vele developers samenwerken ...en is daarom op lange termijn geen tijdverlies. We vermelden hier ook BDD (Behavior driven development), dat gebaseerd is op TDD en vooral de samenwerking tussen verschillende product eigenaars beoogt met aandacht voor de business regels.

Unit testing heeft vele facetten. Het testen gebeurt niet alleen op (complex) business logica maar ook op testen op exceptions, testen om bugs op te sporen, testen op I/O & async operaties. We herkennen verschillende types van testen:

- Assertion testing: testen op true, false, gelijkenis, verschillen tegenover *objecten*, functies of waarden.
- Behavior testing: testen hoe een object reageert op *events* van een ander object.
- Functional testing: black box testing van de verkregen *output volgens input*. Dit kunnen zowel fouten en *exceptions* zijn als verwachte data.
- Regression testing: testen indien na code aanpassing nog alles identiek werkt.

Om vlot te testen heb je tools nodig. Denk aan het stubben van een server api calls, het mocken van database gegevens. Deze tools verwachten meestal dependancies, zodat je testdata kan inbrengen in plaats van productie data. Dependancy verwacht schrijven in een specifiek patroon. En dit kan herhaaldelijk tot refactoring leiden. Eénmaal je een patroon en testtool goed onder de knie hebt

wordt ook jouw design en code robuuster tot eenvoudiger om te onderhouden.
Enkele testtools die vaak gebruikt worden bij Node:

- Mocha: testframework en testrunner
 - o mogelijkheid tot async testen op een eenvoudige manier!!!
 - o bevat een reporter om een test rapport weer te geven volgens verschillende formaten (ook html)
 - o kan via een watcher de testen opstarten bij elke aangebrachte wijziging in de test
- Sinon: voor het opstellen van spies, stubs of mocks in een willekeurig javascript framework.
 - o Spy: De uitvoer van een 'functie' garanderen zonder werkelijke return. Er wordt enkel gefaked dat de functie opgeroepen werd zonder controleren van return data. Wordt gebruikt om te controleren of een callback "call" wel degelijk gebeurde.
 - o Stubs: Werkt met stub-objecten. Deze objecten kunnen methodes of eigenschappen bevatten. De uitvoer van de 'objects methodes' wordt gefaked. Er wordt gecontroleerd of de vooraf gedefinieerde fake value van die ene methode.correct gereturned worden.
 - o Mocks: Werkt met mock-objecten en fakes de uitvoer van 'objects' methodes door het vooraf opzetten van "meerdere" expectations zoals aantal verwachte functie oproepen in combinatie met verwachte query data.

Meestal zijn stubs voldoende maar complexe zaken kunnen met mocks afgehandeld worden. Sinon stubs en mocks worden veel gebruikt voor het faken van database queries (zonder dat de query op de database komt) of het faken van een externe API call (zonder dat de API een hit ontvangt).

- Istanbul en blanket worden beide gebruikt om de coverage van je testen te bepalen. Het coverage percentage duidt aan hoeveel % van je code door testen bewaakt wordt. Dit wordt uitgedrukt als een % van SLOC (Source Lines Of Code)
- Gulp in combinatie met gulp-mocha en gulp-util gebruiken als test runner. In de "gulpfile.js" worden verschillende gulp.tasks aangemaakt met een callback. De te testen source (testfile) wordt als een stream naar mocha gestuurd (pipe command). Mocha returnt het resultaat volgens aangebrachte opties.

gulpfile.js :

```
var gulp = require('gulp'),
    mocha = require('gulp-mocha'),
    gulputil = require('gulp-util');

gulp.task('mocha_test_uitvoeren', function () {
    return gulp.src(['tests/*.js'], { read: false })
        .pipe(mocha({ reporter: 'list' }))
        .on('error', gulputil.log);
})
```

Een gulp watch kan toegevoegd worden als tweede gulp.task zodat bij elke wijziging van een test deze test gaat runnen:

```
gulp.task('watch_mocha', function () {
    gulp.run('mocha_test_uitvoeren');
```

```
gulp.watch(['tests/*.js', './**/*.js'], [ 'mocha_test_uitvoeren']);  
}  
  
gulp.task('default',[ 'mocha_test_uitvoeren']);
```

- Soms geïnstalleerd, meestal om bepaalde elementen eenvoudiger te maken qua syntax of opbouw.:
 - o Chai : <http://chaijs.com/> maakt schrijven van asserts leesbaarder met "should" of "expect". Chai vereenvoudigt assertion syntax.
 - o Moment:<http://momentjs.com/> om met datums te werken (formateren en rekenen)
`moment().format('MMM Do YYYY, h:mm:ss a');`
 - o Webstorm voorziet de mogelijkheid voor het continu runnen van de testen om de X seconden met realtime feedback. Testen kunnen ook gedebugged worden in Webstorm. Voor het debuggen in Visual Studio moet mocha lokaal (*niet globaal*) geïnstalleerd zijn: <https://github.com/Microsoft/nodejstools/wiki/Test-Explorer#discovering-mocha-tests>
 - o Nedb laat toe een database te mocken. De database wordt volledig aangemaakt in memory en is op zich een mongo nosql database.
 - o Nock als mocking module wordt gebruikt om niet een werkelijke API te moeten aanspreken.
 - o Node-uuid module voor het aanmaken van unieke ids: person.id = uuid.v4();

Opbouw van een unit test

Praktisch wordt voor een module (of unit) een test programma aangemaakt (unit testing) dat de verwachtingen van de module moet valideren bij elke wijziging van de code ervan. Eerst wordt de test geschreven (die dus faalt) en nadien komt de implementatie van de module. Typisch wordt gebruik gemaakt van de A A A techniek: Arrange-Act –Assert. Het is niet ongebruikelijk om dezelfde arrange code (voorbeeld : maak gebruiker A aan) in de test zelf te herhalen. Het verhoogt leesbaarheid binnen het AAA principe

- *Arrange* verzorgt setup. (vb: maak gebruiker A aan)
- *Act* voert de test uit met als gevolg een status verandering (vb. voeg de gebruiker toe aan de database)
- *Assert* evalueert het resultaat.(vb. bestaat hete aangemaakte id, tel aantal gebruikers...)

Een zo genoemde “assertion module” wordt gebruikt voor het uitvoeren van een test waarbij een bewering of veronderstelling geëvalueerd wordt. (assertion = een bewering). Om alle testen geautomatiseerd na elkaar uit te voeren is bijkomend een *testrunner* nodig die de testen afloopt en een *testrapport* aanmaakt over de geslaagde en niet geslaagde testen. Bijkomend wordt een “test coverage” opgevraagd, die aanduidt hoeveel van je code door testing afgedekt is.

Een goede unit test bevat de volgende kenmerken:

- De test is voorspelbaar en returnt altijd dezelfde output voor een specifieke input.
- Een test kan alleen maar slagen (pass) of falen (fails).
- De testsuite documenteert zichzelf zonder dat bijkomende uitleg nodig is.
- Een test onderzoekt slechts één zaak. Moeten meerdere zaken getest worden, maak dan meerdere testen.
- Unit testen mogen niet interferen met integratie testen. Unit testen kunnen geen meerdere zaken testen.
- Unit testen komen als eerste aan bod in een testprogramma. Na unit testen kunnen integratie testen komen, daarna validatie testen om tot slot end-to-end testen te runnen.

16.2 assert.js module

Express zelf bevat geen complete testrunner maar wel een basis assert module. Deze assert module kan wel gecombineerd werken met een testrunner zoals Espresso of Mocha:

```
var http = require('http'),  
    assert = require('assert')
```

Sommigen vinden de assert.js module nog te basis en verkiezen dan weer Jasmine of Chai. Deze laatste module (Chai) richt zich specifiek op Node met een BDD houding (= Behaviour Driven Development = TDD met een vlot leesbare syntax).

Een basis voorbeeld met enkel assert.js:

In een map models maken we Calculator.js aan.

```
var Calculator = (function () {  
    function Calculator() {}  
  
    Calculator.prototype.add = function (operand1, operand2) {  
        return operand1 + operand2;  
    };  
  
    Calculator.prototype.subtract = function (operand1, operand2) {  
        return operand1 - operand2;  
    };  
  
    return Calculator;  
})();  
  
module.exports = Calculator;
```

In de map tests maken we calculatorTests.js aan met enkele "asserts".

```
var assert = require("assert");  
var Calculator = require('../Models/Calculator');  
var calculator = new Calculator();  
  
assert.strictEqual(calculator.add(1, 1), 2);  
assert.strictEqual(calculator.subtract(3, 1), 2);  
assert.strictEqual(calculator.add(0.2, 0.1), 0.3, "Javascript is niet gemaakt om  
met floating points te rekenen");//resulteert in een exceptie door klein verschil  
assert.equal( {a:1}, {a:1} )//resulteert in exceptie (deepEqual == zou ok zijn)
```

De module assert voorziet verschillende vergelijkingsmethoden zoals beschreven op <http://nodejs.org/api/assert.html>:

<code>assert.equal(a, b, [message])</code> <code>assert.deepEqual(a, b, [message])</code>	Bij een reguliere equal (shallow vergelijking) worden geen nested objecten vergeleken en wordt gebruik gemaakt van de <code>==</code> operator. Bij een deepEqual gebeurt <u>nested vergelijken</u> wel, en kunnen enumerable properties vergeleken worden. Bij een deepEqual worden <i>de prototype eigenschappen van een object niet vergeleken want die zijn niet enumerable</i> .
<code>assert.notEqual(a, b, [message])</code> <code>assert.notDeepEqual(a, b, [message])</code>	
<code>assert.strictEqual(a, b, [message])</code> <code>assert.notStrictEqual(a, b, [message])</code>	strictEqual gebruikt de <code>===</code> operator. equal gebruikt de <code>==</code> operator.
<code>assert.throws(block, [error], [message])</code> <code>assert.doesNotThrow(block, [error], [message])</code>	Controleert of een gegeven functie (block) wel een exceptie genereert. In error kan de exceptie gecontroleerd worden. De de testen functie moet wel in een extra "function(){ //hier de test}" gewrapped worden. Indien niet heeft test al de throw uitgevoerd.

Wanneer we de module calculatorTests.js oproepen via bvb. een routing `localhost:1337/calculatorTests` dan zal een assert die een foutieve vergelijking heeft een exceptie genereren van het **500 type Assertion Error**.

Goed dat we de exceptie krijgen maar toch is dit niet echt handig. We hebben een drietal praktische problemen:

- a. De Assertion errors kunnen de toepassing onderbreking. Volgende testen worden zo ook niet uitgevoerd. Niet aangenaam, ook al werken we om te testen in een test omgeving en niet in productie.
Oplossing: try/catch per test met output naar `stdout` of een `testrunner` gebruiken.
- b. De testen zijn vaak gebaseerd op een request en bijhorend response. De testen gebeuren op dezelfde server specificaties als de productie functies: zelfde request/response /poort nummer.
Oplossing: dedicated server options gebruiken om zelf vanuit de test een eigen request te plaatsen. Hier kan ook aan request mocking gedaan worden: een testmodule gebruiken die request/response acties simuleert zoals bvb. de nock module.

- c. De vergelijking tussen objecten is eerder omslachtig

Oplossing: abstracties via een extra [chai module](#): <http://chaijs.com/>

Should	Expect	Assert
<pre>chai.should(); chai.should.be.a('string'); foo.should.equal('bar'); foo.should.have.length(3); tea.should.have.property('flavors') .with.length(3);</pre>	<pre>var expect = chai.expect; expect(foo).to.be.a('string'); expect(foo).to.equal('bar'); expect(foo).to.have.length(3); expect(tea).to.have.property('flavors') .with.length(3);</pre>	<pre>var assert = chai.assert; assert.typeOf(foo, 'string'); assert.equal(foo, 'bar'); assert.lengthOf(foo, 3) assert.property(tea, 'flavors'); assert.lengthOf(tea.flavors, 3);</pre>
Visit Should Guide	Visit Expect Guide	Visit Assert Guide

16.3 Extra asserting test functies met should.js

De testfuncties in assert zijn niet altijd voldoende zodat modules zoals "should.js", "expect.js" of "node-tap.js" bijkomend geïnstalleerd worden.

Should.js is een asserting test module die gebruikt wordt voor *het testen van objecten*. Bij node zorgt dit voor het gemakkelijker controleren van o.a. de request en response properties.

bvb.: `res.should.have.status(200);`

Daarnaast kunnen ook de properties van modellen en hun type eenvoudiger getest worden.

bvb.: `myObj.should.be.a('object').and.have.property('firstname', 'johan');`

Volgende functies zijn opgenomen in should.js en staan beschreven in npm:

<https://www.npmjs.org/package/should>

<code>(myActualValue = 'test').should.be.ok</code> <code>myComparison.should.not.be.ok</code>	Een expressie (tussen de haakjes) met actual en expected value wordt getest.
<code>myObj.should.be.true</code> <code>myObj.should.not.be.true</code> <code>myObj.should.be.false</code> <code>myObj.should.not.be.false</code>	Testen op booleaanse waarden
<code>myObj.should.be.empty</code>	Testen op (geen) inhoud.
<code>myObj.should.equal('test')</code> <code>myObj.should.be.within(10, 20)</code> <code>myObj.should.be.above(100)</code> <code>myObj.should.be.below(5)</code>	Vergelijkend testen op zowel strings of numerieke waarden.
<code>"562".should.match(/[0-9]{3}/)</code> <code>"abcdef".should.include.string('json')</code> <code>"adcdef".should.be.type("string")</code> <code>obj.should.have.property('name')</code> <code>obj.should.have.keys('name',</code>	Reguliere expressies. Datatypes vergelijken. Objecten vergelijken

```
'firstname');
user.should.respondTo('emailfunction')
```

16.4 Testen met een dedicated request/response object, dedicated server opties

Met de `http.request()` factory methode kunnen we zelf ons eigen request object aanmaken dat voor de testen moet zorgen. Twee argumenten zijn nodig: de opties (`host, url`) en de callback functie, die de response moet verwerken. Tijdens het ontvangen van de response callback wordt assert testing uitgevoerd.

Voorbeeld: testen van een json bericht

We wensen een test op te stellen om de werking van een formulier te testen. Dit formulier laat toe om via een post een tekstbericht te versturen. In een testmodule controleren we of de post bericht wel degelijk een "message" key bevat en of dat een antwoord "Message received" in JSON formaat terugkomt.

Maak een module `messageTests.js` aan in de tests map. Deze module maakt een eigen post request aan `(/send)` met een bericht in een "message" key.

```
var assert = require("assert");
var http = require('http');
var should = require('should'); //om response obj te testen

//1. Opties instellen voor server
var opts = {
  host: '127.0.0.1',
  port: process.env.PORT || 3000,
  path: '/sendMessage', //zorgt voor verzending vd test bericht
  method: 'POST',
  headers: { 'content-type': 'application/x-www-form-urlencoded' }
}

//2. Een http request/response op basis vd post opties
var req = http.request(opts, function (res) {
  res.setEncoding('utf8')

  var result = ""
  res.on('data', function (d) {
    console.log(d);
    result += d;
  });

  res.on('end', function (err) {
    // 4. Assert testing op de response
    assert.strictEqual(JSON.stringify(JSON.parse(result)),
      '{"status":"ok","message":"Message test"}',
      'fout bij messageTests response');
    if (err) {
      throw err;
    }
  });
});
```

```

        }
    //5. Gebruik van should.js:
    res.statusCode.should.match(200);
    JSON.parse(result).should.have.property("message");
});
//3. Uitvoeren van een het request met een "message" key ( als echo)
req.write("message=Message test"); //post data
req.end()

```

De routing /send zorgt dat bij ontvangst van de post message de server een specifieke JSON boodschap terugstuurt. Hier bevat JSON een status en message property.:

```

app.post('/sendMessage', function (req, res) {
//controleert op een message key
if (req.body && req.body.message) {
    //verstuurt success message in JSON formaat
    res.send({ status: "ok", message: req.body.message })
} else {
    //verstuurt error message in JSON formaat
    res.send({ status: "nok", message: "No message received" })
}
res.end()//einde response niet vergeten
})

```

Nu de test bestaat kan ook het formulier aangemaakt worden. (bvb. Toevoegen aan index.jade) en kan getest worden of de JSON message ontvangen wordt:

```

<form action="/sendMessage" method="POST">
<input type="text" length="150" name="message">
<input type="submit" value="Stuur bericht ">

```

16.5 Testrunner: Espresso of Mocha?



Als testrunner wordt bij express zowel gebruik gemaakt van Espresso als Mocha. Beide worden geïnstalleerd via npm install. Beide maken gebruik van een "**tests**" map onder de root, met daarin de test files. Het activeren van de tests om een rapport te genereren gebeurt via de command console met oproep van de te testen module (bvb: espresso tests/moduleX.js of mocha tests/moduleY.js)

Info over Espresso is te vinden op : <https://github.com/espresso/espresso>
 Info over Mocha is te vinden op <http://mochajs.org/>

Assert.js en Should.js kunnen perfect gecombineerd worden met beide test runners.

In frontdevelopment wordt Mocha eveneens gebruikt naast qUnit, en Jasmine. We kiezen dan ook Mocha.

16.5.1 describe ... it en asynchroon testen

De Mocha testen volgen een specifiek formaat met "describe" en "it". Alle testen worden aangemaakt binnen een "describe" en worden na elkaar uitgevoerd. Testen kunnen ook genest worden. Met de term "test suite" wordt verwezen naar de verzameling van alle testen.

Met describe() wordt een titel aan de test gegeven en een callback voor de eigenlijke test(en). Met it() wordt een begrijpbare titel aangemaakt voor een individuele test. Traditioneel start men deze beschrijving met de woorden 'should return'

```
describe("Add", function () {
    it("should return the sum of both operands", function () {
        assert.strictEqual(calculator.add(1, 1), 2);
    });
});
```

Om asynchroon te werken wordt aan it een callbackfunctie als argument toegekend. Traditioneel noemt men dit done(). Door dit argument, is het eenvoudig om te tonen waar de test beëindigd is. Het is dank zij deze done() dat kan getest worden op asynchrone werking. Want: deze done zorgt dat de test pas een resultaat returnt nadat hij volledig is uitgevoerd.

```
describe('overkoepelende testnaam, meestal module naam', function () {
    //verschillende usertesten hier aanmaken met telkens describe()
    describe('Hier de beschrijving van de test', function () {
        it('should return .... when....', function (done) {
            //breng de test aan, gebruik eventueel dedicated req en res
            var opts = { . . . };
            var req = http.request(opts, function (res) { . . .
                res.on('end', function (err) {
                    assert.strictEqual( . . . );
                    this.statusCode.should.match(200);
                    done(); //niet vergeten = indicatie test uitgevoerd
                });
            });
        });
    });
    //Hier eventueel meerdere subtesten met it(' ', function() {})
}

describe('setTimeout ', function () {
    it('should return TRUE after 1000 milliseconds', function (done) {
        setTimeout(function () {
            assert.equal(1, 1);
            done();
        }, 1000);
    });
});
```

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

Voorafgaand aan de describe kan ook een before() functie opgeroepen worden. In deze before kunnen alle nodige setups gebeuren die nodig zijn voor een test. In before kan je bijvoorbeeld een database connectie opzetten.

Er bestaat een gelijkaardige beforeEach(), die vóór elke test opgeroepen wordt of een after() die na alle tests opgeroepen wordt, en een afterEach().

Met de term "**test hooks**" verwijst men vaak naar alle bijkomende functies die vóór of na de werkelijke testen uitgevoerd worden.

```
before(function(done) {  
    //Hier alle vóórinstellingen  
    // met foutopvang waar nodig  
    mySomething.on('error', function () {  
        throw new Error('fout bij mySomething');  
    });  
    done(); //async callback  
})
```

16.5.2 tests runnen

Nu kunnen alle testen gerund worden vanuit een (tweede) console door het oproepen van de testmodule.

```
> mocha myModuleTests.js
```

Een fout kan zich voordoen waarbij de socket afgesloten wordt. Oorzaak is te zoeken in de app die standaard op een poort draait (vb. 1337) terwijl vanuit de console de test op een port 3000 draait.

Oplossing: haal de app op in de testmodules met : `var app = require('../app').app;`

Een require wordt immers gecacht en nooit twee keer ingeladen.

Nu kan je zonder problemen (en zonder het opstarten van een tweede console) de testen runnen. Zorg wel dat je het volledige path invoert, tenzij je de default mocha map "test" (zonder s) gebruikte.

Het resultaat van `> mocha tests/calculatorTests` is een mini test rapport.

```
Express server listening on port 3000  
Testing the small message sender:  
  
Stuur bericht, ontvang JSON  
POST /send 200 15ms - 53b  
  ✓ should return OK when receiving the success JSON msg  
  
1 passing (31ms)
```

16.5.3 Nog meer mocha mogelijkheden:

- Mocha zoekt default naar een map test , of naar een file test.js
- Overzicht van de reporters in mocha : `> mocha -r reporters`
Met het nesten van describe kunnen meerdere testen gegroepeerd worden.
Testrapport in specifiek formaat (ook html):

```
> mocha repoTest      -R nyan
> mocha repoTest      --reporter doc>testReport.html
    mocha repoTest      open testReport.html
```

- Een test skippen of enkel op één test werken:
it.skip (" ", cb) in plaats van it(" ")
describe.skip ...
Een x voor de testnaam plaatsen heeft hetzelfde effect als skip
it.only("....", cb) in plaats van it (" ")
- Met mocha -w voert mocha een watch uit naar elke wijziging van een testprogramma en voert vervolgens automatisch de test uit.

- Mocha kan je runnen met een reeks opties via > mocha [options].
Enkele voorbeelden:
-h of --help : help informatie
-V of --version : gebruikte versie nummer
-u of --ui<name> : een user interface reporter ophalen (met name <name>)
-g of --grep <pattern> : run alleen de testen die beantwoordden aan het "pattern"
-c of --colors : enable kleur
-b of --bail : exit na de eerst falende test
--check-leaks :check voor leaks in globale variabelen

Deze opties kan je ook vooraf aangeven in een file (Makefile genoemd). Dit is oorspronkelijk een unix tool dat hier fungeert als een batch file voor het uitvoeren van de testen. Deze file kan gerunned worden via de console : met een > make command. Meer info hierover: <http://www.cprogramming.com/tutorial/makefiles.html>

- Met een opts file (mocha.opts) een zelfde testmap oproepenmap testen :
test/testmap
 - -recursive
Via dezelfde mocha.opts een coverage.js opstarten: - -require tests/coverage.js

Een HTML file maken als coverage rapport: > mocha testfile -R html-cov>coverage.html
Bij een coverage rapport verwijst SLOC naar "Source lines of code" of het aantal geteste code lijnen.

- Mocha combineren met JSHint.
JSHint genereert onmiddellijk fouten binnen node door niet gekende variabelen zoals "require", "before", "after".

Via de package.json kunnen deze fouten geneutraliseerd worden.

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

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

```
nodemodules
assets
coverage
```

Meer info : www.JSHint.com

16.5.4 Testen automatisch oproepen:

Het automatiseren om de testen te starten kan met gulp of kan vanuit de package.json file:

- Met gulp : door het combineren van een gulp.task('default ') en een gulp.watch() in plaats van een gulp.run(). Zie code hoger.
- Met package.json:
In het "scripts" element kunnen javascript instructies opgegeven worden. Bij express zorgt "start" voor het aanduiden van de start file van de applicatie.

```
"scripts": {  
  "testMocha": "./node_modules/.bin/gulp watch_mocha",  
  "start": "node ./bin/www"  
},
```

Testen kan manueel vanuit de npm console met npm run-script testMocha

16.6 Sinon faket reële returns

16.6.1 Spy

Een spy garandeert alleen dan een functie opgeroepen wordt. Er wordt verder geen data gecontroleerd. Dit wordt toegepast om te controleren of een callback werd opgeroepen:

```
describe('SPY: callApi with faked callback', function () {  
  it("runs a http call", function () {  
    var spy = sinon.spy();  
    getData.callAPI("howest", spy); //geen controle van de API data  
    assert.equal(spy.called, true); //enkel getest dat de cb opgeroepen is.  
  });  
});
```

16.6.2 Stub

Bij een stub wordt naast het controleren van de oproep ook een return waarde gecontroleerd. De stub duidt aan dat het om een fake ingevoerde waarde gaat. Dit wordt bijvoorbeeld toegepast (in integratie of end-to-end testing) voor het ophalen van data uit een repository of business applicatie.

```
it('should return a stubbed profession', function () {  
  var stub = sinon.stub(dataAccessor);  
  stub.getProfession.returns("nendocent"); //initialiseren stub waarde  
  
  dataAccessor.getProfession(stub); //real resultaat wordt niet gecontroleerd  
  
  assert.equal(stub.getProfession.called, true);  
  assert.equal(stub.getProfession("JVN Stub"), "nendocent"); //controleert stub waarde  
});
```

16.7 Oefeningen met Mocha

Oefening: de eerder gemaakte calculator tests en message tests via Mocha runnen.

1. Installeer mocha.js en should.js (npm). Mocha wordt meestal globaal geïnstalleerd in de dev dependencies
> npm install -save-dev-g mocha
2. Bouw de mocha testen uit met de describe- en it-methodes.
Tip: Bij objecten zoals de calculator maakt men meestal gebruik van beforeEach om de instantie aan te maken:

```
var calculator;
beforeEach(function () {
    calculator = new Calculator();
});
```
3. Voeg zelf een willekeurige test toe.(vb. max message length)
4. Start de applicatie zoals gebruikelijk (in een eerste node shell).
5. Run de mocha tests vanuit de root map in een tweede shell met
> mocha tests/messageTests.js
6. Gebruik een package.json file om de testen op te starten

Oefening: Maak een testfile "usersTests.js" voor de eerder gemaakte users API.

Zorg dat volgende testen aangemaakt en correct uitgevoerd worden:

```
it('should return the API user collection upon own request', function (done) {  
});  
  
it("should throw an exception when looking for unexisting user" , function (done) {  
});  
  
it('dataAccessor should get users ' , function (done) {  
    var DA = new usersDA();  
    DA.getUsers(function (err, data) {  
        var obj = JSON.parse(data);  
        assert.equal(Object.keys(obj).length , 3);  
        done();  
    });  
});  
  
it("dataAccessor should create a testUser" , function (done) {  
});  
  
it('dataAccessor should return a Promise upon a user.name' , function (done) {  
    //voldoende om de returned state te checken van een promise.(fullfilled of rejected)  
    //dit laat toe volledig async te werken-> de returned Promise is immers thenable, eindigt met done().  
  
    var DA = new usersDA();
```

```

DA.findUserByName("johan" , function (userPr) {
    var userPrState = userPr.inspect().state; //fulfilled - rejected
    assert.equal(userPrState , "fulfilled");
    done();

    userPr
        .then(function (user) {
            assert.equal(user.profession , "docent");
        })
        .done(null, function (error) {
            //bij promises komt error op de tweede plaats
            done(error); //assert throw error opvangen
        })
        ;
    });

});

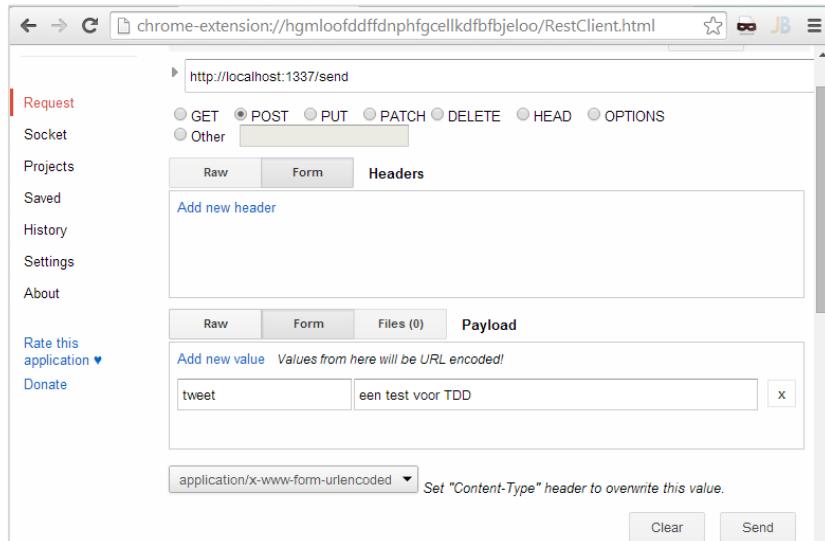
```

Zorg dat je de testen kan runnen met `>gulp default`.

16.8 Meer mogelijkheden

- **Chrome test client:**

In combinatie met server testing en bvb. een assert module beschikt Chrome over een extensie om een advanced REST client te simuleren. Na het installeren van de extensie kunnen get, post, put en delete messages verstuurd worden:



- **Code coverage:**

Niet dat hiermee de kwaliteit van de test bepaald wordt maar wel dat een library zoals "blanket" of "istanbul" kan aanduiden welke delen van je code niet getest worden.

> `npm install --save-dev blanket`

```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

D:\BE-Node-2014\092-Q-Labo\20140-ParkingsReserv\ParkingsOpl-exp4>mocha test/repositoryTest

  Repository Test
1611
count 13
    ✓ should return reservations with getAllReservations

  1 passing (34ms)

D:\BE-Node-2014\092-Q-Labo\20140-ParkingsReserv\ParkingsOpl-exp4>run test-cov
'run' is not recognized as an internal or external command,
operable program or batch file.

D:\BE-Node-2014\092-Q-Labo\20140-ParkingsReserv\ParkingsOpl-exp4>npm run-script
test-cov
> Parkings@0.0.0 test-cov D:\BE-Node-2014\092-Q-Labo\20140-ParkingsReserv\Parkin
gsOpl-exp4
> mocha --require blanket test/repositoryTest -R html-cov --recursive > coverage.html
```

Na configuratie (via de package.json of mocha.opts) wordt een coverage.html aangemaakt met weergave van coverage % en het aantal SLOCs (Source Line Of Code)

50% coverage 16 SLOC

17 Real time web & sockets

17.1 De nood aan websockets

HTTP is half duplex & pull

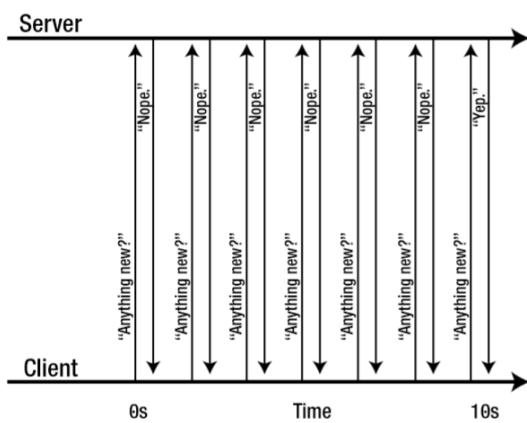
Websockets horen thuis in de categorie "connectiviteit van webbrowsers". De protocols HTTP/1.0 en HTTP/1.1 zijn typische request response protocols, waarbij de cliënt het request initialiseert. Bijgevolg is dit puur half duplex werking. HTTP/1.0 maakte een afzonderlijke en nieuwe connectie voor iedere request. HTTP/1.1 zorgde voor "reusable connections", zodat bvb. het ophalen van beelden en scripts over eenzelfde connectie gebeurde.

Het web bestaat al lang niet meer uit statische pagina's. Met AJAX werd het web dynamischer en dient enkel een klein deel van de pagina te worden vernieuwd. Toch is dit nog steeds een http technologie die een pure pull technologie is en bovendien half duplex werkt. Het wordt noodzakelijk om continu nieuwe real time informatie te hebben voor bijvoorbeeld: sport uitslagen, headline nieuws, chat programma's, stock evoluties, social media, newsfeeds ...

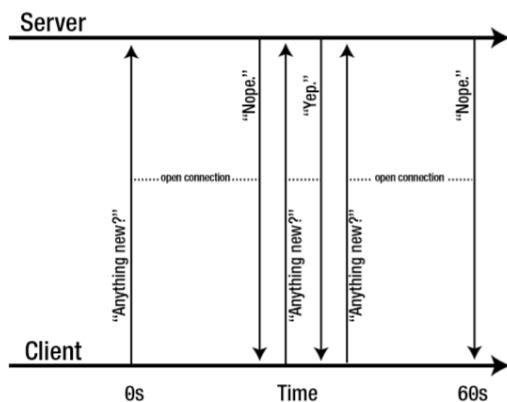
Polling & streaming

Tussentijdse oplossingen werden gezocht in periodisch pollen, ook long polling of continuous polling genoemd. Bij long polling blijft de connectie open op vraag van de client tot de server nieuwe info stuurt en afsluit.

Polling (= setInterval): *De client stuurt met een vaste regelmaat (polling interval) requests en dit gedurende een voorgedefinieerde periode. Als er geen informatie is gedurende deze periode, sluit de cliënt de lijn af op basis van het negatief server antwoord. Is er wel een antwoord, dan sluit de server de lijn vervroegd af.* Bijgevolg kunnen veel nodeloze connecties het web vervuilen.



Long Polling: de client opent de lijn voor een langere (long) periode zonder extra requests. *De server houdt de lijn open tot er een nieuw antwoord is. De server zal na het zenden van een antwoord de lijn afsluiten.*



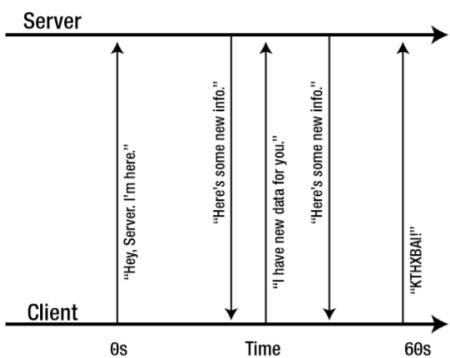
Streaming is gelijkaardig aan polling. In streaming wordt de connectie niet afgesloten bij nieuwe data. De nieuwe data wordt vanuit de server gepushed en de server houdt de connectie lange tijd open. Streaming kan als een oplossing voor realtime aanzien worden maar besef dat de server nooit signaleert dat http moet beëindigd worden. Dat betekent ook dat proxies en firewalls het antwoord kunnen gaan bufferen, waardoor het realtime antwoord toekomt met vertraging.

Websockets

Een betere oplossing kwam er pas in HTML5 met push services zoals websockets. In een push service start de cliënt een negoziatie met de server. De cliënt informeert de server wat verwacht wordt en wacht gewoon op het antwoord van de server. Push services in HTML5 werden mogelijk door het begrip "origin". Een "origin" werd gedefinieerd als een scheme of protocol + host + port (bvb. <https://www.voorbeeld.com:8080>) en baseerde daarop een aantal "connectiviteits methodieken". Connectiviteit in HTML5 werd gerealiseerd door technologien zoals *Websockets*, *Server-Sent Events* en *Cross-Document Messaging*.

17.1.1 Wat zijn websockets?

Wat zijn websockets en wat zijn de voordelen van dit websockets protocol?



- Zorgt voor continu "connected" informatie stromen (cfr: telefoonlijn). Hierdoor krijg je een hogere belasting op de server in vergelijking met http. De hogere belasting zit hem in het *scalen over verschillende ports*, eerder dan een grotere bandbreedte.

- Werkt *bidirectioneel* en *full duplex* met streamed data over één enkele TCP/IP connectie. HTTP vult de TCP connectie aan door te zorgen voor het handshaking tussen de webcliënt en de webserver. In tegenstelling tot polling hebben we bij sockets *slechts één request* vanuit de cliënt om de connectie te initialiseren.
- Permanent open connectie naar alle geconnecteerde partijen (men spreekt over **geconnecteerde sockets**). Door het permanent open houden van een socketconnectie in full duplex wordt de vertraging van boodschappen onbestaand. In http kan een roundtrip request 500msec tot 800msec bedragen. Met sockets wordt dit herleid naar 50msec tot 100msec. Binnen een webomgeving wordt dit als RealTime ervaren.
- Meer responsive en daardoor sneller dan “successive ajax calls”.
- Maakt gebruik van het “ws”-protocol, dat beschreven staat in RFC6455. Het protocol werd eerst gepubliceerd in *december 2011* en is te vinden op <http://tools.ietf.org/html/rfc6455> :
`var socket = new WebSocket ("ws://chatserver.com");`
Naast dit hoofdprotocol bestaan ook subprotocols zoals XMPP (Extensible Messaging and Presence Protocol), STOMP (Streaming Text Oriented Messaging Protocol) of SOAP (Simple Object Access Protocol). Je kan een eigen subprotocol aanmaken. Het subprotocol kan optioneel worden meegegeven in de constructor:
`var socket = new WebSocket ("ws://chatserver.com", "myCustomHomeMadeProtocol");`
Bijkomende bestaat een beveiligde geïncrypteerde “wss” versie van het protocol, die herkenbaar is in het URL schema:
`var socket = new WebSocket ("wss://chatserver.com");`
- De cliënt initieert de connectie met één http 1.1 request en vraagt een upgrade aan de server. Wanneer de server sockets ondersteunt kent het de upgrade toe en is handshaking met request/response niet langer nodig.

Client Request met vraag naar upgrade voor “websockets”. Er wordt gebruik gemaakt van een unieke key om de client te identificeren.	Server Response met upgrade toegekend: volgens http (200 OK , 401 Unauthorized , 503 service unavailable) Bij status 200 zal de server een Sec-WebSocket-Accept uitgeven, gebaseerd op de ontvangen key, en zal een status code 101 versturen bij succesvolle connectie op een specifieke URL: <code>ws(s)://host/namespace/version/transportMechanismeId/sessionId</code>
--	--

```

Request URL: ws://echo.websocket.org/echo
Request Method: GET
Status Code: 101 Web Socket Protocol Handshake
▼ Request Headers
  ▲ Provisional headers are shown
  Cache-Control: no-cache
  Connection: Upgrade
  Host: echo.websocket.org
  Origin: null
  Pragma: no-cache
  Sec-WebSocket-Extensions: permessage-deflate
  Sec-WebSocket-Key: nVmCBCiPVd4jn52R3c320w=
  Sec-WebSocket-Version: 13
  Upgrade: websocket
  
```

Na een succesvolle upgrade handshake komt socket messaging beschikbaar in een multi-frame formaat. Dit formaat wordt gespecificeerd het gebruikte socketprotocol en bevat

MessageId : [incrementId bij Ack]:[endpoint]:JSONdata.

Volgende messageld zijn bruikbaar

Messageld	Message type	Info
0	Disconnect	Verplicht de socket om af te sluiten.
1	Ack	Bevat een endpoint om de connectie op te zetten
2	Heartbeat	Van client naar server binnen een servertimeout van de handshake. De server antwoordt met heartbeat (geen data)
3	Message	Bevat de message en optioneel een endpoint
4	JSON message	Bevat een JSON message (stringified)
5	Event	Bevat event informatie volgens een formaat: <code>{"name": "eventName", "args": {"some": "content"}}</code> Naast willekeurige eigen eventnamen zijn een reeks eventnamen gereserveerd voor connectie doeleinden: <u>message</u> , <u>connect</u> , <u>disconnect</u> , <u>open</u> , <u>close</u> , <u>error</u> , <u>retry</u> , <u>reconnect</u>
6	ACK	Wordt verstuurd bij ontvangst van een message. Bevat het messageld
7	Error	Wordt verstuurd door de server met een foutbericht (bvb. Unauthorized)
8	NOOP	Geen verwerking. Sluit een poll af.

17.1.2 De Websocket API

Een Websocket API zorgt voor een interface, die toelaat het Websocket protocol te gebruiken. Deze API werd ontwikkeld door het W3C. Het protocol komt van IETF (Internet Engineering Task force). De API voor websockets is puur event driven en maakt gebruik van zijn eigen status codes en zijn eigen events. Het asynchroon programmeer model wordt hierbij gevuld, wat betekent dat éénmaal een connectie open is de applicatie naar events luistert.

Een websocket object dispatcht over het algemeen 4 events:

- open event: wordt afgevuurd na het uitvoeren van de handshake bij het beschikbaar zijn van de socket,
- message event: data wordt afgevuurd als de server beschikbaar is. Deze data kan ook binair zijn en dus Blobs bevatten (Blob object in javascript) of een Array van binaire data (ArrayBuffer object in javascript)
- close event: wordt afgevuurd als de socket connectie afgesloten wordt.
- error event: voor het afhandelen van de fouten en eventueel een nieuwe connectie te initialiseren.

Met de `socket.readyState` eigenschap kan de connectie status van de socket uitgevraagd worden. Er zijn 4 stati:

1. `socket.connecting` met waarde 0,
2. `socket.open` met waarde 1,
3. `socket.closing` met waarde 2
4. `socket.closed` met waarde 3.

Het socket object voorziet een aantal *methodes* waaronder `socket.close()` en `socket.send("someData")` waarmee data naar de server verstuurd wordt. Vergeet niet dat er pas data kan verstuurd worden nadat de connectie open is! (Lees als: Doe een `socket.send()` enkel binnen de onopen callback handler). Hoeveel data ineens verstuurd wordt is ook niet vanzelfsprekend. Ook al wordt de send direct uitgevoerd, vooraf wordt een gedeelte van de data gebuffered. Send kan je verschillende keren na elkaar oproepen met verzenden van een gebuffered deel. Met de eigenschap `socket.bufferedAmount` kan je gebufferde hoeveelheid opvragen.
Meer info over de events: <http://www.w3.org/TR/websockets>

Er zijn verschillende websockets API's beschikbaar voor Real Time apps waaronder: Socket.io, Faye, SignalR client voor asp.net SignalR, PubNub, Realtime.co, Pusher (info: <http://www.leggetter.co.uk/real-time-web-technologies-guide>)

17.2 Websockets in node.js

1. Om websockets te activeren op node.js moeten we een extra module installeren. We hoeven niet zelf (kan wel) de socket server aan te maken. Eénmaal sockets geïnstalleerd zijn, kunnen we deze module gebruiken om bijvoorbeeld een chat server aan te maken.
2. We hebben de keuze tussen verschillende socket modules: **socket.IO**, WebSocket-Node. Oorspronkelijk werd WebSocket-Node het meest gebruikt maar vandaag is socket.IO de

defacto standaard aan het worden. Node is natuurlijk ook niet de enige server die websockets implementeert. Zo bestaat ook Apache_mod_pywebsocket, Jetty, Kaazing's Websocket Gateway. We kiezen voor Socket.IO.

Socket.IO voorziet een abstractie met meerdere mogelijkheden bovenop sockets en zorgt dat oudere browsers (die geen sockets supporten) terugvallen op de oudere technieken zoals AJAX polling, JSONP polling of long-polling. Via www.caniuse.com of <http://html5please.com> kan nagezien worden of de browser websockets ondersteunt.

Bovendien kan socket.IO geïntegreerd worden met de http server van node en werkt het in express.js.

17.3 Basis webchat met socket.IO

17.3.1 Basis chat server en HTML5 chat cliënt

In zijn eenvoudigste vormt luistert socket.io op een poort naar de mogelijk events. Hierbij kan gebruik gemaakt worden van het Event Emitter patroon, zodat willekeurige events kunnen aangemaakt worden zowel op de server als op de client.

Echo Server:

```
var io = require("socket.io").listen(4000);

io.sockets.on('connection', function (socket) {
  socket.on('clientMessage', function (data) {
    socket.emit('serverMessage', 'The client said: ' + data);
  });
});
```

HTML Client:

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title>simple socketio client</title>
  //cliënt socket.io.js wordt dynamisch bezorgd door de server
  <script src="/socket.io/socket.io.js"></script>
  <script>
    //constructor met url als arg (doet dienst als namespace)
    var socket = io.connect("http://localhost:4000");
    //luister naar connection event
    socket.on("connected", function () {
      console.log(("connected"));
    });

    //stuur message event naar server
    socket.emit("clientMessage", "Hello from client socket")
    //bij ontvangst van echo event vd server
    socket.on("serverMessage", function (data) {
      document.write ("server says:" + data)
    })

    //luister naar disconnection
```

```
socket.on("disconnect", function () {
    console.log ("client disconnected")
})

</script>
</head>
<body>
    <h1>socket.io cliënt </h1>
</body>
</html>
```

Noot: via javascript kan host inclusief poort opgehaald worden uit de URL:

```
var connection = window.location.host; //of window.location.origin
var socket = io.connect(connection);
```

17.3.2 Socket.io server luistert naar http server

Om dit in werking te krijgen dienden we de html file te starten op zijn fysisch path. We hebben echter een webserver nodig. Willen we de socket server combineren met een webserver (die static files leest), dan kunnen we de socket server laten luisteren naar de http server. De http server wordt aangebracht als argument van listen(httpServer). De HttpServer fungeert als middleware voor de socketserver.

De applicatie kan nu gestart worden met `http://localhost:4000/` en functioneert nu als een socket server die luistert naar de webserver.

```
var httpServer = require('http').createServer(handler);
httpServer.listen(4000);
//socket server luistert/steunt op naar httpserver
var io = require('socket.io').listen(httpServer);
```

De callbackfunctie (hier "handler" genoemd) van `createServer` , vervult dezelfde functie als een eerder aangemaakte `staticServer`: ophalen van filename uit de url >>> met `fs.readFile()` de file uitlezen met het juiste contenttype, te vinden in de file extensie.

We maken dan ook gebruik van de eerder aangemaakte `staticServer`. Om het overzichtelijk te houden plaatsen we de `socketHandlers` best in een afzonderlijke module (bvb. `socketHandlers.js`)

De message event handler kan aangevuld worden voor broadcasting. Vele third party websockets modules verwachten dat je een array bijhoudt met de boodschap en id van elke cliënt. Socket.io maakt voor elke binnenkomende cliënt zelf een `socket.id` aan, en laat toe om met `socket.broadcast` alle cliënts aan te spreken. Ook een verzameling van sockets wordt bijgehouden in zowel het `socket.client` als `socket.server` object.

```
var getHandlers = function (httpServer) {

    io.sockets.on('connection', function (socket) {
        socket.on('clientMessage', function (data) {
            socket.emit('serverMessage', 'cliënt said: ' + data);
            socket.broadcast.emit('serverMessage', socket.id + ' said: ' + data);
        });
    });
}
```

```
    });
}

module.exports.getHandlers = getHandlers;
```

Deprecated: Na het toevoegen van de module domain (npm install domain) kan het socketDomain voor foutcontrole ingesteld worden:

```
var getHandlers = function (httpServer) {
    var socketDomain = domain.create();

    socketDomain.on('error', function (err) {
        console.log('Error caught in socket domain:' + err);
    })

    socketDomain.run(function () { //hier de handlers })
}
```

17.3.3 HTML cliënt verder afwerken.

De server is grotendeels af, maar de gui kan beter met een tekst input veld voor de boodschap, met verschillende kleuren per cliënt enz...

```
var inpClient = document.getElementById('inpClient');
inpClient.onkeydown = function (keyboardEvent) {
    if (keyboardEvent.keyCode === 13) {
        socket.emit('clientMessage', escape(inpClient.value));
        inpClient.value = '';
        return false;
    } else {
        return true;
    }
};
```

Vergeet de escape niet om script injection attacks te verhinderen

17.4 Mogelijke uitbreidingen

17.4.1 json versturen.

De socket is een object dat bijgehouden wordt met een id voor elke geconnecteerde gebruiker. Dit object kunnen we willekeurig dynamisch uitbreiden (javascript eigenschap). Zo kan elke client een willekeurig kleur toegekend worden:

```
socket.color = getRandomcolor();
```

Om gegevens naar de client te sturen (zoals dit kleur bijvoorbeeld) maken we vervolgens beter gebruik van een json object.

```
var obj = { color: socket.color , id : socket.id , content: content}
socket.broadcast.emit('serverMessage', JSON.stringify(obj));
```

Aan client kant wordt de ontvangenJSON string weer geparsed naar het object.

```
socket.on("serverMessage", function (json) {
    showMessage(JSON.parse(json));
})
```

17.4.2 Meer message types toevoegen

De toepassing is tot nu toe eenvoudig met een beperkt aantal message types: on("serverMessage"), on("clientMessage").

Uitbreiding blijven best overzichtelijk door extra message types toe te voegen. Voorbeeld bij een login:

```
socket.on('login', function (username) {
    socket.username = username;
    socket.broadcast.emit("serverMessage", username + " is ingelogd");
}

//server login event triggert de gebruiker éénmalig om in te loggen
socket.emit('login');
```

De client moet natuurlijk een username bezorgen via socket.on('login', function () { })

17.4.3 socket object bevat rooms.

Het socket object heeft meerdere eigenschappen, methodes en handlers. Eén van de handlers is socket.on("disconnect") en laat toe een boodschap te versturen als een client de chat sessie verlaat. Met de eigenschap "rooms", kan een client toegekend worden aan een specifieke room.

De client stuurt een join request: socket.emit('join', aRoom).

Waarop de server on("join") een room toekent: socket.room = aRoom ; socket.join(aRoom)

En nadien enkel broadcast naar deze room: socket.broadcast.to (aRoom)

17.4.4 Namespaces

Op eenzelfde server kunnen verschillende chatrooms runnen. Om collision te verhinderen is het gebruik van namespaces geen overdreven luxe. Socket.io laat toe om namespaces te gebruiken.

Client : krijgt een specifieke namespaced url.

```
var socket = io.connect("http://localhost:4000/myNamespace");
```

Server: moet enkel de sockets aannemen van deze namespace:

```
var io = require('socket.io').listen(httpServer);
var myNameSpaceio = io.of('myNameSpace');
myNameSpaceio.on('connection', function (socket) { . . . });
```

17.4.5 Security

Websockets connecteren op dezelfde server als HTTP. De gemeenschappelijke handshake die zorgt voor de upgrade naar een socket connectie kunnen we als save beschouwen. Daarna heeft de cliënt toegang tot alles op de server. Binnen de http context kan bij elke request een security controle gedaan worden met bvb? session cookies. Maar bij websockets wordt geen nieuw request geplaatst. Integendeel, éénmaal de connectie upgrade voltooid is kan de cliënt pogingen ondernemen om bvb. een session cookie te hijacken.

Extra beveiliging blijkt nodig. Hiervoor stuurt men meestal bij elke socket communicatie een JWT boodschap.

17.4.6 Multiserver/multiprocess

Sockets communiceren tussen een specifieke server en een specifieke client. Dat betekent dat een message gestuurd wordt naar de server (het current servergeheugen en “zijn” users) die de socket opzette. Heb je een toepassing met meerdere processen of meerdere servers, die in loadbalancing opgezet worden, dan zullen gebruikers op een tweede server niet langer de broadcast boodschappen van de eerste server ontvangen.

Oplossing hiervoor is het gebruik van een **message broker**. Een event wordt naar een centrale locatie(=de broker) gepublished, waar iedereen naar luistert. Deze centrale locatie houdt alle servers op de hoogte, zodat ze kunnen antwoorden naar hun cliënts.

Bruikbare message brokers zijn:

Redis: eenvoudig in setup. Men spreekt over een pub/sub technologie (publish/subscribe). De publish functie op de server publiceert (meestal in JSON) naar een centrale Redis locatie via zijn pubsub channel. Om te luisteren naar een event (=je moet dus live geconnecteerd zijn) moet je inschrijven (subscribe) op het kanaal van dit event. Dit kanaal kan een wildcharacter kanaal zijn. Een listener luistert naar deze binnenkomende events voor verdere verwerking.

De Redis site “<http://redis.io/>” zelf beschrijft zich als :

“Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache and message broker.”

De voorziene data structures in Redis zijn : Strings, Lists, Sets, Hashes en SortedSets.

- AMQP (RabbitMQ): meer mogelijkheden, iets lastiger setup
- ZeroMQ: minder mogelijkheden, minder complex dan AMQP

17.4.7 Waar sockets gebruiken?

Sockets kunnen gebruikt worden voor elke vorm van cliënt-server communicatie. Buiten het http handshaking protocol om een socket verbinding op te zetten, kan een volledige applicatie met sockets uitgebouwd worden. HTTP calls en AJAX calls voor posting of het ophalen van data kunnen perfect worden uitgevoerd met sockets.

Anderzijds steunen sockets op een permanente verbinding en een statefull design (niet stateless zoals http!). States (of sockets), die bewaard blijven, zorgen voor een moeilijker design proces bij distributed computing. Door het bewaren van verschillende stati kan wanorde en verwarring ontstaan voor het beheer ervan over verschillende processen of verbindingen. De hoger vermelde broker architectuur is één van de mogelijke hulptools hiervoor. Maar toch zorgt het statefull design bij sockets ervoor dat momenteel http niet volledig vervangen worden door sockets bij nieuwe toepassingen. Andere bronnen vermelden dan weer dat peer to peer verbindingen meer en meer hun intrede zullen doen bij nieuwe webtechnologien.

Sockets worden zo vooral interessant en efficiënt

- wanneer slechts kleine hoeveelheden data of slechts kleine veranderingen op de webpagina nodig zijn. Voorbeeld: mobiele toepassingen of toepassingen met veel kleine data uitwisselingen.

- wanneer veel real time interactie nodig is (games, notifications vanop de server, progress updates)

17.5 Gebruik van frameworks en libs bij socket.io

17.5.1 Express framework en socket.io

De combinatie van het express framework en socket.io is niet ongewoon. Er is geen verschil in het gebruik van socket.io met express in vergelijking met de httpserver only. Ook hier luistert de socketserver naar de httpserver.

```
var express = require('express');
var http = require('http');
var socket = require('socket.io');

var app = express();
var server = http.createServer(app); //httpServer is afhankelijk van app
var io = require("socket.io").listen(server); //sockets luistert naar de server
```

In Express 4 start de applicatie vanuit de www file met dependancy van app. Het is dan ook logisch om in deze www file de socket.io te initialiseren.

In www roepen we sockets.js op, die alle socket handlers bevat.

De configuraties (keys, connectionstrings, port) plaatsen we in een config.js file. Natuurlijk moet je de routing nog aanpassen en een jade file maken.

Express4 www :

```
#!/usr/bin / env node
var debug = require('debug')('Express4Chat');
var app = require('../app');

//0. configuraties via afzonderlijke file:
var config= require('../config');
app.set('port', process.env.PORT || config.PORT);

//1. sockets.io activeren met server dependancy
var server = app.listen(app.get('port'), function() {
  debug('Express server listening on port ' + server.address().port);
});
var io= require('socket.io')(server);

//2. listeners voor (zowel server als) sockets
var sockets = io.listen(server);
console.log('io luistert naar de express server op port ' + app.get('port'));

require('../sockets.js')(io); //bevat alle socket events
```

sockets.js:

```
module.exports = function (io) {

  //socket connectieve maken
```

```
io.sockets.on('connection', function (socket) {  
    console.log("sockets connected")  
  
    //start boodschap  
    socket.emit ('chat','Welcome to the Chat Service');  
  
    //events  
    socket.on('chat', function (data) {  
        socket.emit('chat' , obj);  
    });  
  
});
```

Voorbeelden met socket.io:

Uitgewerkte voorbeelden zijn vlot te vinden op het net of bij github:

- Basis chat voorbeeld: <http://code.tutsplus.com/tutorials/real-time-chat-with-nodejs-socketio-and-expressjs--net-31708> .
- Basis voorbeeld hoe socket.io zou kunnen gebruikt worden voor het aansturen van een hardware device (arduino, raspberry pi, usb board): <http://cdn.chrislarson.me/blog/nodejs-socketio-and-real-time-web-hmi-example>

17.5.2 Superset: express.io



Het veelvuldig combineren van express met socket.io zorgde voor het ontstaan van een nieuwe module met naam "express.io". De module is te vinden op <https://github.com/techpines/express.io>. Dergelijke combinaties van bestaande modules noemt men vaak een "superset".

17.5.3 Angular en socket.io

Frontend developers combineren graag socket.io met een MVVM structuur aan de client kant. Angular zorgt voor databinding bij de client, en bevordert zo ook het hergebruik van client script modules. Een chat applicatie voorbeeld hiervan is uitgewerkt op:
<http://www.html5rocks.com/en/tutorials/frameworks/angular-websockets/>

17.6 Oefeningen:

17.6.1 Oefening: images broadcasten

Opgave : Het basis chat voorbeeld uitbreiden om met de javascript FileReader images op te halen van je laptop en deze te broadcasten naar de andere chat cliënts.

Oplossing:

1. De cliënt krijgt een input type file: met bijhorend javascript
`input(id="selectImg", type="file" , multiple)`

2. Javascript aan cliënt kant zorgt dat een image gelezen wordt in **base64 formaat**. Dit laat toe het image te versturen via de socket:

```
selectImg.addEventListener("change", function (evt) {  
    for (var i = 0; i < evt.currentTarget.files.length ; i++) {  
        var file = evt.currentTarget.files[i];  
        var reader = new FileReader();  
        //Lezen van image in base64  
        reader.readAsDataURL(file);  
  
        reader.onload = function (evt) {  
            //werk af: verzenden van image via socket.emit()  
            //evt.target.result bevat image in base64 format  
        };  
    }  
});
```

3. De server stuurt het base64 image naar alle cliënts:

```
socket.on('user Image', function (baseImg) { //broadcast baseImg })
```

4. Alle ingeschreven clients tonen het ontvangen image:

```
socket.on('user Image', function (base64Image) {  
    //img element aanmaken en appenden  
    // var img = document.createElement("img");  
    // img.src = base64Image;  
})
```

17.6.2 Oefening: Geselecteerd flickr image broadcasten

Opgave: In een eerder gemaakte oefening werd gebruik gemaakt van de flickr API om een beeld te zoeken op basis van een zoekterm.

Breed deze toepassing uit zodat een willekeurig beeld (vb. uit de eerste socket) om de 10 seconden verstuurd wordt naar alle andere aangesloten sockets. Je kan het bijvoorbeeld beschouwen als het favoriete beeld van de dag, dat in de banner terecht komt.

Oplossing:

1. Door dat gewerkt werd met een eventemitter bij getAPIData.js kan, als mogelijke oplossing, deze emitter gebruikt worden om de nodige actie te triggeren. Om de 2 seconden wordt bijvoorbeeld een random beeld gebroadcast.

```
getAPIData.on('apiData' , function (jsonItems) {  
    var socketHandlers = require("./socketHandlers");  
    socketHandlers.broadcastPicture(jsonItems); //initieel beeld tonen  
    timerID = setInterval(socketHandlers.broadcastPicture, 2000 , jsonItems);  
}
```

2. Handler uitwerken in socketHandlers.js

```
var broadcastPicture = function (jsonItems) {  
    // een random image uit jsonItems ophalen  
    var randomPic = . . .;  
    // random beeld broadcasten
```

```
        sockets[0].broadcast.emit('broadcastImage', jsonItems[randomPic])
    }
```

3. De client toont het ontvangen beeld:

```
socket.on("broadcastImage", function (img) {
    //<img /> aanmaken met als source img.media.m
}).
```

17.6.3 Oefening: multi-user white board

Opgave: Het aanmaken van een white board waar meerdere personen op hetzelfde moment kunnen tekenen.

Oplossingen en verschillende voorbeelden zijn terug te vinden op het net, waar al dan niet gebruik gemaakt wordt van een canvas, en waar de gebruikte socket module kan variëren. Een mogelijke oplossing met socket.io en een canvas ziet er als volgt uit:

1. Het start met het aanmaken van een socket server. We kiezen socket.io, waarbij de server zoals gewoonlijk luistert naar de http server.
`var io = require('socket.io').listen(http);`
2. Bij connectie wordt er geluisterd naar een drawEvent. De resultaten van het drawEvent worden via socket.emit('draw') naar de alle ingeschreven cliënts gestuurd. De data die verstuurd wordt naar de cliënts is natuurlijk een verzameling van tekenparameters zoals x positie, y positie, kleur, dikte, patroon. Dit kun je aanvullen naar eigen wensen. Deze data versturen we logischer wijze via een javascript object. Wat (welke eigenschappen) we versturen hangt wel af van de gebruikte drawing module aan cliënt kant. Wat heeft de cliënt nodig om te tekenen.

```
io.sockets.on('connection', function (socket) {
    console.log("sockets connection established")
    socket.on('drawInSocket', function (data) {
        //emitted object is afhankelijk van gebruikte client drawing library
        //Type is een event zoals dragstart, drag, dragend
        socket.broadcast.emit('draw',
            { x: data.x, y: data.y, type: data.type, color: data.color});

    });
});
```

3. De code aan cliënt kant is uitgebreider dan deze aan server kant en begint met het aanmaken van de canvas.

HTML:

```
<script src="http://localhost:4000/socket.io/socket.io.js"></script>
. . .
<body>
    <section><!-- canvas hier dynamisch toevoegen --></section>
</body>
```

SCRIPT: wordt binnen een namespace App Draw geplaatst
`var AppDraw = {}`

```
AppDraw.init = function () {

    //canvas dynamisch toevoegen
    AppDraw.canvas = document.createElement("canvas");
    AppDraw.canvas.height = 400 ; // geen px !
    AppDraw.canvas.width = 800;
    document.getElementsByTagName('section')[0].appendChild(AppDraw.canvas);

    //context vd canvas initialiseren (een lijn of stroke)
    AppDraw.ctx = AppDraw.canvas.getContext("2d");
    AppDraw.ctx.fillStyle = "solid";
    AppDraw.ctx.strokeStyle = "#000000"; //eventueel random kleur
    AppDraw.ctx.lineWidth = 5;
}

AppDraw.init();
```

4. Aanbrengen van socket.io. dit kan binnen de AppDraw namespace of je kan socket.io zelf in een andere namespace plaatsen.

```
Appdraw.socket = io.connect('http://localhost:4000');
```

5. Het tekenen aan client kunnen we volledig zelf uitwerken. Dit zou iets kunnen worden in de zin van:

```
//listeners voor mouse events voor ophalen van x,y coördinaten
window.addEventListener("mousedown", startDraw, false);
window.addEventListener("mousemove", startDraw, false);
window.addEventListener("mouseup", startDraw, false);

//listeners voor canvas touch voor ophalen van x,y coördinaten
AppDraw.canvas.addEventListener('touchstart', startDraw, false);
AppDraw.canvas.addEventListener('touchmove', startDraw, false);
AppDraw.canvas.addEventListener('touchend', startDraw, false);
```

In startDraw wordt afhankelijk van het ontvangen type event het tekenen in de canvas gestart.

Het is echter eenvoudiger om een bestaande module te gebruiken om de nodige coördinaten op te halen. jQuery beschikt over voldoende bruikbare modules. Download en installeer de gekozen module.

```
<script type="text/javascript" src="js/jquery.event.drag-2.0.js"></script>
```

Dit pakket laat toe om aan een object (hier de canvas) mouse drag events toe te kennen. De coördinaten die voortvloeien uit het draggen kunnen gebruikt worden om te tekenen in de canvas. Meer info over deze jquery-ui module: <http://threedubmedia.com/code/event/drag>

6. Het tekenen kan nu lokaal op de cliënt geïnitialiseerd worden. Op de canvas worden de tekenevents dragstart, drag en dragend geïnitialiseerd met bind.

```
$(document).ready(function () {
    $('canvas').bind('drag dragstart dragend', function (evt) {

        var offset, type, x, y;

        type = evt.handleObj.type; //Noot: evt.type is het mouse event.
        offset = $(this).offset();
        x = event.clientX - offset.left;
        y = event.clientY - offset.top
```

```
//teken lokaal
AppDraw.draw(x, y, type, color);
}
```

7. Rest nog om de handlers van de events uit te werken.

```
AppDraw.draw = function (x, y, type, color) {
    //teken lokaal op basis van emitted data
    App.ctx.strokeStyle = color;

    //teken naargelang ontvangen event type
    //indien type dragStart dan AppDraw.ctx.beginPath();App.ctx.moveTo(x, y)
    //indien type drag dan AppDraw.lineTo(x,y);
    //indien ander type sluit het tekenpath af...

    if (type === "dragstart") {
        App.ctx.beginPath();
        return App.ctx.moveTo(x, y);
    } else if (type === "drag") {
        App.ctx.lineTo(x, y);
        return App.ctx.stroke(); //teken
    } else {
        return App.ctx.closePath();
    }
}
```

8. Tot nu toe werd enkel lokaal getekend. Willen we ook tekenen op de remote cliënts, dan worden de draw parameters (x, y, type, color) emitted naar de server, waar sockets de rest doen.

```
//teken in remote sockets
AppDraw.socket.emit('drawInSocket', {
    x: x,
    y: y,
    type: type,
    color: "#000000"; //eventueel random kleur
})
```

17.6.4 Oefening: Eenvoudige multi-user games

Eenvoudige games, die enkel kleine status uitwisselingen nodig hebben met andere gebruikers zijn snel gemaakt met sockets. Meestal volstaat het om posities, of een CSS eigenschap, of woorden te broadcasten.

Een voorbeeld waarbij gebruikers zo snel mogelijk muisclicks uitvoeren is te vinden op:
<http://krasimirtsonev.com/blog/article/Real-time-chat-with-NodeJS-Socketio-and-ExpressJS>

17.6.5 Oefening/enkel ter info: Opbouw van een socket game

Een volledig spel uitbouwen in een labo is hier niet de bedoeling. Wanneer een spel meer omvattend wordt, wint vooral een duidelijke opbouw aan belang. Een paar basisprincipes, die bij quasi bij elk game te vinden zijn, worden kort beschreven.

Game opbouw

Terugkerende elementen bij een multi user game zijn o.a.:

- Server kant handlers:
 - o netwerking via sockets om wijzigingen in de spel status te broadcasten,
 - o beheer van login/logout,
 - o unieke spel parameters beheren (id – kleur-image van spelers, moeilijkheidsgraad ..)
 - o ranking en bijhouden van spelresultaten
- Cliënt kant handlers:
 - o Monitoren van input (mouse, keyboard, socket, playerobjects) op status wijzigingen.
 - o frame per frame game status bijhouden en wijzigingen aan de server kenbaar maken (emit).
 - o al dan niet gebruik van svg (= retained mode = status bewaard door de app waardoor het volstaat om via bvb.setAttribute een bestaand image, player te wijzigen) of gebruik van canvas (= immediate mode = elke game cyclus de volledige canvas hertekenen met directe uitvoering).

Server netwerking bij games steunt op WebRTC of websockets.

WebRTC staat voor *Web Real Time Communication* en richt zich vooral op audio/video streaming en browser-to-browser (peer-to-peer) data uitwisseling via UDP. Bij RTC wordt via een signaling protocol (een server dus) de metadata voor een connectie aangebracht, waarna peer to peer op eenzelfde webpagina actief is. Een basis beschrijving voor WebRTC is te vinden op <http://www.html5rocks.com/en/tutorials/webrtc/basics/>

Websockets hebben als API voor HTML5 blijvend een server nodig, zijn betrouwbaarder (gebruiken geen UDP) en zijn eenvoudig voor full duplex data uitwisseling (zoals bvb. het uitwisselen van game status). Meestal worden websockets gebruikt.

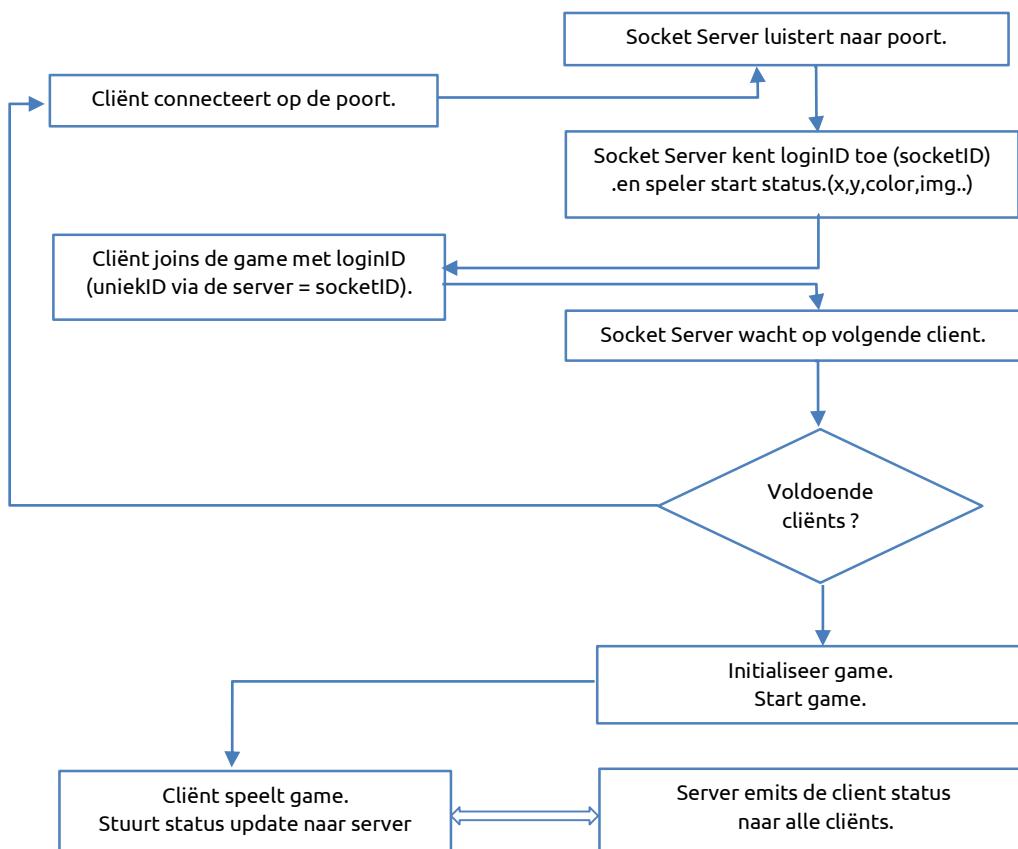
De client: Het grotere werk van een game zit hem vaak bij de cliënt: aantrekkelijk design, status wijzigingen die getriggerd worden door input events, events handlers gebaseerd op animaties.

De uitwerking gebeurt frame per frame. Per frame wordt het game (browser, canvas) hertekend en komt het game in zijn nieuwe status. Zowel setInterval, setTimeout of requestAnimationFrame (RAF) worden hiervoor gebruikt. RequestAnimationFrame geniet vaak de voorkeur omdat RAF timing bijstuurt voor de nog komende frames. Vertragingen worden zo weggewerkt en er wordt gegarandeerd dat eenzelfde afstand wordt afgelegd over eenzelfde tijdsinterval. Vraagt een refresh cycle echter teveel updates van de browser dan wordt requestAnimationFrame verder uitgewerkt met bvb.webworkers, XMLHttpRequests of worden combinaties met setInterval, setTimeout uitgezocht. Naast de updates aan cliënt kant, die met een zo hoog mogelijke refresh rate gebeurt (50hz, 60hz), kan de update die de server naar alle cliënts stuurt iets trager ingesteld worden (25hz, 30hz) om bandbreedte te sparen. Timing wordt hier niet verder behandeld. We gebruiken bij voorkeur RAF, maar spenderen er wel aandacht aan om zo weinig mogelijk data over het net te sturen.

Games zijn niet nieuw en eenvoudige HTML API's (bestaand of gepland) kunnen/zullen de ontwikkelsnelheid nog verhogen: Using_full_screen_mode, Gamepad, audio of Web_Audio_API, Pointer_Lock_API, WebGL...

Spelverloop:

LOGIN: Het spel zelf start met het toevoegen van cliënts (login). De login start onmiddellijk na het bestaan van de socket (`io.sockets.on("connection", ...)`) en kan zowel vanuit de server als vanuit de cliënt opgestart worden met een `socket.emit("login", user)`. Een typische login:



Een toepasselijk code voorbeeld is te vinden op: <http://rawkes.com/articles/creating-a-real-time-multiplayer-game-with-websockets-and-node.html>

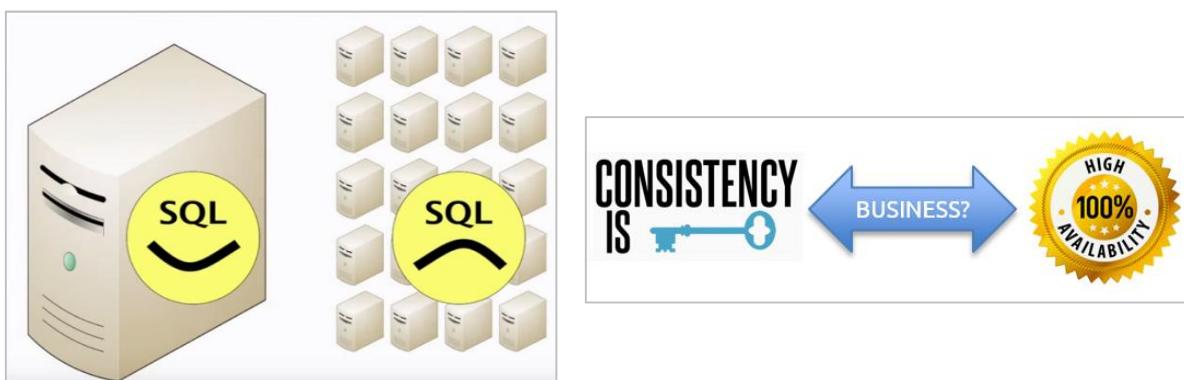
Ander game met socket.io : <https://github.com/HTML5Hub/>. Live te vinden op <http://mario.ign.com>

18 Databases

18.1 Data stores en node.js

Geen websites zonder datastores. Meestal wordt gewerkt met relationele databases (RDBMS zoals MSSQL, MySQL) maar ook NoSQL databases (wat staat voor "not only" SQL) hebben hun specifieke toepassingen.

De relationele databases voorzien een betrouwbare en snelle oplossing voor gestructureerde data, waarbij transacties als 100% ACID (Atomicity, Consistency, Isolation, Durability) bestempeld worden. NoSQL databases verwerpen niet alleen de query language maar houden ook geen relationeel model of schema bij (schema-less). Meestal zijn ze open source. Transacties worden er soms als BASE bestempeld (Basically Available, Soft state, Eventual consistency) wat de veelheid van users, response snelheid van data of "high-available service" benadrukt, inclusief database replicatie voor distributed processing en sharding voor het splitsen van databases over machines. Alles wordt eenvoudig gehouden en zo wordt bijvoorbeeld JSON gebruikt om objecten te stoken met javascript als query taal waarbij joins vermeden worden. Hierdoor is geen extra data mapping van gestokenneerde data naar memory nodig. Data wordt geaggregeerd opgenomen en bewaard waardoor multi-machine of multi-process installaties eenvoudiger worden.



Voor het stoken zelf onderscheiden we verschillende NoSQL (data-)types als vervanging van de klassieke tabellen en joins:

- Stoken via *key/value* (Redis, Riak) – meest eenvoudig maar met beperkte query mogelijkheden. Een key is niet op de hoogte van wat in de waarde zit (kan een object zijn) maar wordt enkel gebruikt als de zoeksleutel. Redis is een in memory database dat snelheid ten goede komt in toepassingen met zeer veel gebruikers met small data trafiek in een high I/O workload zoals: gerichte analytics, search engines, advertise targeting, forums, messaging, geo searching.
- Stoken in *kolommen*. Data wordt verzameld in families van kolommen (Cassandra, Apache HBase) – ideaal voor heel veel data over meerdere machines. Een row key bevat een verwijzing naar een "column family". Een column family bevat meerdere samenhangende kolommen zoals bijvoorbeeld een klant kolom met zijn bestellingen kolom. Een column is te bekijken als een tabel, die bestaat uit een verzameling van [colum key/column value].
- Stoken in *documenten* (CouchDB, MongoDb, Raven DB) – verbeterde JSON key/value techniek met betere query technieken. Data wordt bijgehouden in JSON documenten (of

andere complexe datastructuren), die vlot ondervraagbaar zijn en ook wel een uniek id hebben (maar niet noodzakelijk een schema).

- Stoken in een flexibel *graph* model (Neo4J, InfoGrid) – waarbij rij/kolom/index structuren vervangen worden door uitgebreide grid structuren met meerdere nodes, die stoker plaatsen voorstellen. Nodes worden voorzien van model properties en zijn gelinkt met andere aangrenzende nodes via edges. Data wordt minder samen genomen maar wordt eerder opgesplitst in nodes met heel veel edges (relaties) tussenin en snel ophalen van correlaties.

De node community zorgde voor een beschikbaarheid van drivers voor verschillende van deze databases. Vermoedelijk gaat de toekomst naar een organisatie waar verschillende topics door verschillende database types ondersteund worden. Elk type database lost een ander type probleem op: Shopping Carts in Redis, Marketing recommendations in Neo4J, Productcatalog in mongoDB en financiële transacties in MS SQL.

18.2 MySQL

Verschillende drivers zijn beschikbaar. Meest gebruikt binnen RDBMS is waarschijnlijk de module: "MySQL".

De drivers laten typisch 2 zaken toe: eerst de connectie aanmaken en daarna de nodige queries uitvoeren. De documentatie op GitHub is zeker voldoende om een test uit te voeren. (<https://github.com/felixge/node-mysql>)

Het MySQL protocol is sequentieel, wat betekent dat bij een connectie de queries na elkaar uitgevoerd worden. Een applicatie met veel MySQL operaties kan resulteren in wachttijden. In dat geval kan een connectie pool gebruikt worden.

Andere mogelijkheden zijn: pooling clusters gebruiken (meerdere hosts), transacties, stored procedures, een SQL resultaat onmiddellijk streamen vanuit het query object wat piping toelaat,

18.3 MSSQL

Verschillende drivers zijn beschikbaar. Eén van de meest gebruikte is "tedious". De naam verwijst naar TDS, wat staat voor Tabular Data Stream, dat een application layer protocol is voor MSSQL.

<http://weblogs.asp.net/chanderdhall/microsoft-sql-server-driver-for-nodejs>

18.4 MongoDB



18.4.1 Wat en waarom.

Data consistent onderhouden in een relationeel model vraagt overhead. Voor applicaties waar deze consistentie minder belangrijk is het relationeel model onnodig en kan gewerkt worden met een

NoSQL database. Denk aan een sociaal netwerk, waar het verschijnen van dezelfde newsfeeds, maar dan anders geformuleerd of vrijgegeven op een ander tijdstip zeker geen ramp is. Denk aan een object, dat door zijn verschillende links, noodzakelijk met meerdere tabellen moet gemapped worden. Objecten afgeleid van een parent object kunnen compleet andere tabellen nodig hebben dan deze van het parent object. Een bijkomende typische eigenschap van NoSQL applicaties is dat data op zich alleenstaand al zin heeft. Het gebruik van joins en transacties over tabellen wordt uitermate zelden tot nooit gedaan.

Scalability in een relationele database wordt bemoeilijkt omdat alle data consistent aan strikte voorwaarden en aan normalisatie moet voldoen. Tabellen kunnen bij updates tijdelijk gesloten worden. Het tijdelijk sluiten van tabellen betekent tragere toepassingen. NoSQL toepassingen leggen geen verplichtingen op in verband met relaties. Voor zover tabellen en relaties bestaan...

Een NoSQL database wordt gekenmerkt door eenvoud van schaalbaarheid (scalability), een instelbaar niveau van consistentie met snelheid als hoofddoel en dit door geen schema op te leggen, en geen tabellen te gebruiken.

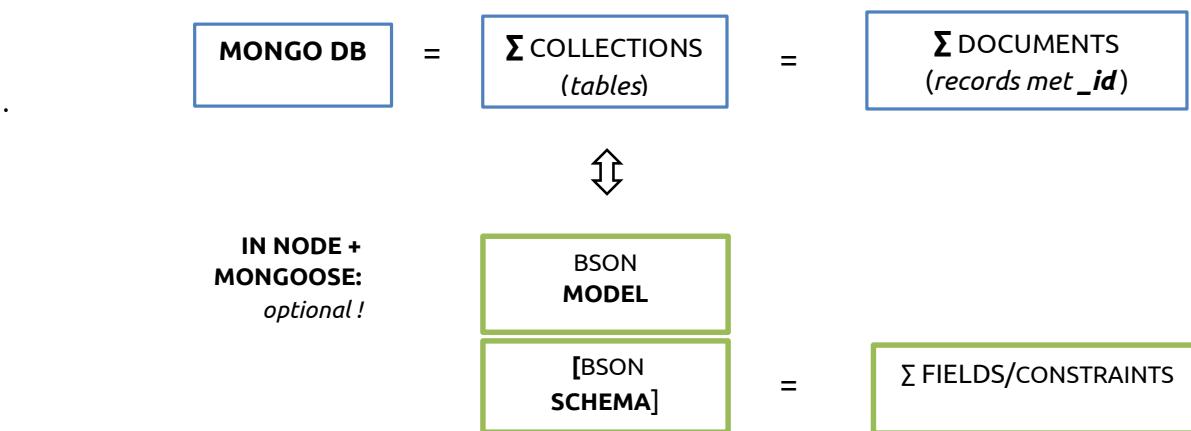
18.4.2 Collections en documents

MongoDB is een document georiënteerde database.

Een database is opgebouwd uit *collections* (te vergelijken met tables). Deze collections kunnen zo simpel of zo complex mogelijk zijn. Verschillende collections zijn niet aan elkaar gelinked volgens een ER diagramma, waardoor locking onnodig is. Het niveau van consistentie ligt in de handen van de developer. De developer kiest tussen snelheid, consistentie en duurzaamheid. Wanneer eenzelfde collectie over verschillende servers dient te komen (replicatie van data vanuit een primary database) doet een NoSQL wel het nodige hiervoor, maar er kan wat tijd overgaan. Men noemt dit "eventual consistency".

De collections bevatten documenten. De data van *documenten* wordt gestokeerd in een BSON (Binary JSON) formaat (<http://bsonspec.org>). De max size is momenteel 16Mbytes. Een document beschikt altijd over een _id. Ken je dit zelf niet toe dan doet Mongo dit. Verder zijn er geen regels of verplichtingen. Wel kunnen documenten opgemaakt worden (zonder verplichting!) volgens een specifiek schema, dat de nodige constraints en datatypes beschrijft. Een schema hoeft wel niet, zodat bvb. adres velden in een document totaal verschillend kunnen opgebouwd worden.

Een document beantwoordt aan een model, waarbij enkel het _id verplicht is. Het document wordt rechtstreeks in memory gemapt. Een veld in dit document is een key/value pair in de BSON file. De key is een string en de value kan van verschillende datatypes zijn. BSON is een binaire serialisering van JSON en zijn javascript objecten die, in tegenstelling tot JSON, ook datatypes voorzien zoals bvb. Date



De query taal is javascript. De query is een javascript literal uitdrukking, die niet alleen statische data hoeft te bevatten. Ook reguliere expressies kunnen gebruikt worden.



Vaak wordt MongoDB gebruikt in de MEAN combinatie. Met MEAN verwijst men naar MongoDB + Express + Angular.js + Node.js. (<http://mean.io/#/>)

18.4.3 Node en Mongoose.

Om een database aan te spreken vanuit een applicatie is een driver nodig. Er bestaan verschillende MongoDB drivers voor node. De meest gebruikte zijn mongodb en mongoose. Mongoose is perfect integreerbaar met Express en voorziet object mapping met de collections in mongodb. ORM (Object Relational Mapping) dus. Daarom starten we met express en voegen Mongoose toe.

1.1.1.5 Installeer de MongoDB server:

MongoDb is geschreven in C++ en is Open Source. Het is beschikbaar in 64 of 32 bit versies. Voor productie kies je natuurlijk voor de 64 bit versie. Voor development is de 32 bit versie voldoende snel.

Download de files op: <http://www.mongodb.org/downloads>. Unzip de download om een reeks executables te bekomen. Er zijn naast de executables voor het opstarten van de server ook executables voor bijvoorbeeld het scalen van de server.

Noot: Voor productie kan de database gehosted worden op Azure maar ook andere omgevingen voorzien mongodb –as-a-Service: MongoLab en Compose bieden een 512MB gratis sandbox aan.

Open de command console voor volgende commando's:

> De data vd database komt default in een map terecht: \data\db. Maak deze map aan via de command console met: md \data\db

```
C:\Program Files\MongoDB 2.6 Standard\bin>mkdir .\data\db
```

Noot: Een alternatief path kan ingesteld worden via een mogod instructie:

```
mongod --dbpath c:\node\eenMap\data
```

Het deleteen van de data inhoud om fresh te herstarten kan met
del data*

> run vanuit deze map als administrator show `mongod.exe` (= `mongodemon` = de database server initialiseren)

Bij het starten wordt een pid en poort nummer (meestal 27017) toegekend. Met Ctrl+C sluit je de server af.

Mogelijke opties kan je opvragen met “`mongod --help | more`”. Deze opties kunnen opgenomen worden in een configuratie file (`mogod.conf`), die je later runt met een commando: `mongod -f mongod.conf`

Je kan mongo ook als een service starten (ipv vanuit console commando's) met “`mongod -install`”. Je start de service met “`net start mongod`” en stopt deze met “`net stop mongodb`”

Andere mogelijkheden voor in productie is het aanmaken van een replica set, dat voorziet in automatic recovery, mocht je Primary database falen. Minimaal worden hierbij 3 databases aangemaakt: een Primary DB (write only), één of meerdere Secondary DB's (read only met een priority index) en een Arbitrary DB (helpt bij falen een secondary te kiezen).

> run eveneens `mongo.exe` (de interactieve `mongod shell` om met mongo te communiceren) in een tweede command console. Met Ctrl+C of exit verlaat je de shell.

```
C:\Program Files\MongoDB\Server\3.0\bin>mongo.exe
MongoDB shell version: 3.0.6
connecting to: test
```

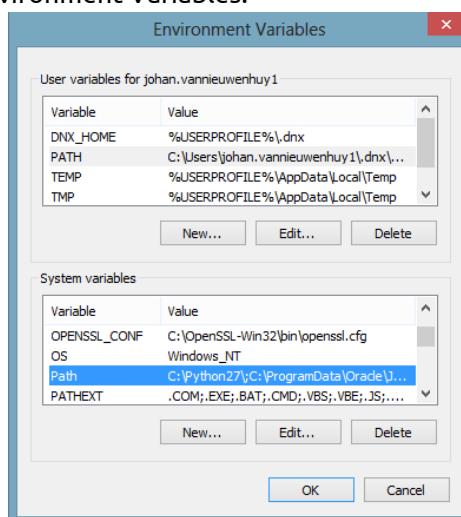
De “connecting to:test” is een default prompt, die aanduidt dat mongo deze database zal gebruiken indien geen andere bestaat.

>Bekijk met het command “`show dbs`” de geïnstalleerde databases.

> **db** toont de database momenteel in gebruik

De mongo shell is te bekijken als een externe applicatie die toelaat commando's af te vuren. Deze commando's zijn geschreven in javascript, zodat ook administratieve taken kunnen uitgevoerd worden (controle of aanpassen van documenten, omvang uitvragen...) op basis van een externe script file, die in de achtergrond draait (“`mongo [myServer] myScript.js`” of “`load ('myScript.js')`”).

Noot: voor eenvoud van gebruik kan je eventueel het MongoDb path toevoegen aan het windows path : Control Panel >dir /p System and Security > Advanced system settings > Advanced > Environment Variables:



1.1.1.6 Maak een mongo database en collecties aan vanuit de Mongo shell

Referentie: <https://docs.mongodb.org/manual/reference/mongo-shell/>

- > **use usersDB** maakt een database aan met naam "usersDB"
- > **help** toont meer shell commando's.

Na dit user command bestaat de database nog niet. Aanmaak gebeurt pas tijdens toevoegen van een collectie(= eentabel) met BSON documents (= data). Het aanmaken van een collectie met een document gebeurt via één "db" command. Bvb.: een **collectie "users"** op de database "usersDB" krijgt als volgt een extra **document**:

- > **db.users.insert({ "username": "Johan", "name": "Vannie", "email": "johan.vannie@howest.be"})** maakt de collectie aan.
- > **db.help()** toont meer commando's

Mongo stopt zijn binary collections default in een map data\db. Voor windows gebeurt dit default onder de root: c:\data\db. Elke collectie bestaat er uit twee binaire files.

Om collecties en documenten te bevragen kan een **find()** zonder argumenten toegepast worden in de command console. Een document **_id** wordt automatisch gegenereerd, indien je zelf geen aanmaakt, en wordt gebruikt om indexes aan te maken in db.system.indexes. Dank zij de indexen wordt het zoeken versneld. Een index kan van verschillende datatypes zijn (zelfs in eenzelfde collectie!): string, floating point, date, json... Een document **_id** dat aangemaakt wordt door mongo bevat inherent een timestamp. Deze timestamp kan opgevraagd worden met **ObjectId().getTimestamp()**.

Met **pretty()** worden de nodige linebreaks aangemaakt voor weergave.

```
> use usersDB
switched to db usersDB
> db
usersDB
> db.users.insert({ "username": "johan", "name": "Van", "email": "johan.van@howest.be"})
WriteResult({ "nInserted": 1 })
> db.users.find().pretty()
2014-08-10T18:01:26.927+0200 SyntaxError: Unexpected token .
> db.users.find().pretty()
{
    "_id" : ObjectId("53e7973e64f62a4ffa0e8edd"),
    "username" : "johan",
    "name" : "Van",
    "email" : "johan.van@howest.be"
}
```

Noot: Default beschikt een Mongo database over een collectie "Users". Deze collectie (met hoofdletter U) verzorgt de authenticatie van de Mongo database, wat resulteert in een andere connectionString.

1.1.1.7 CRUD acties in mongo:

CRUD acties in mongo vragen enige voorzichtigheid. Er is immers geen schema die over de collecties de consistentie bewaart. Het enige wat verplicht aanwezig is, en indien niet aanwezig door mongo zelf aangemaakt wordt is het **_id**.

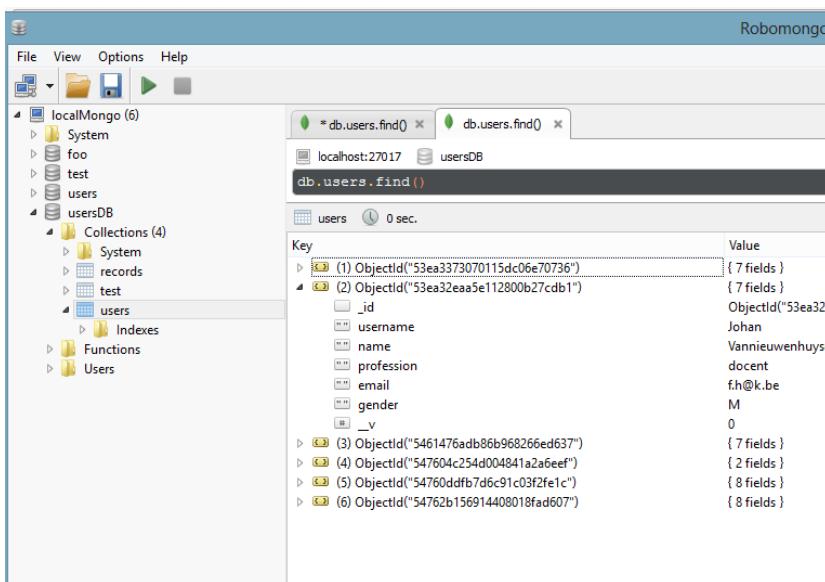
ACTIE	METHODES
Create	<p>insert, save <code>db.foo.save ({_id:1, name: "Johan"})</code></p> <p>Wanneer je een document <u>saved</u> met een eerder <u>zelf(!)</u> aangemaakt _id dan wordt het eerste <u>document overschreven</u> en dit zonder warning! Gebruik daarom insert, dan gebeurt dit niet maar krijg je een “duplicate key” warning. Bij het saven of inserten kan de waarde van een document key een totaal ander formaat of opbouw hebben (geen schema).</p>
Read	<p>find, findOne, findById <code>db.foo.find (query, [projection])</code> waarbij projection de vermelde velden teruggeeft indien een waarde 1 vermeld wordt voor dit veld (een 0 betekent verbergen). De query kan opgebouwd worden met operatoren (<code>\$gt, \$lt, \$gte, \$not, \$exists,...</code>) en steunt op <u>strict equality en case sensitivity</u>.</p> <pre>var doc = db.foo.findOne({_id:1}, {_id:1 , name:1, info:0}) var docs = db.foo.find({_id:\$gt:2, \$lte50}) var arrResults = db.foo.find({myArray: 'Howest'})</pre> <p>Ook reguliere expressies zijn bruikbaar: <code>db.users.find({name: new RegExp("chars-to-find")})</code></p> <p>Met dot notatie tussen quotes(!) kunnen nested items opgehaald worden: <code>db.schools.find({ "students.created_at" : {\$gte : 2015, \$lt : 2017} }, { _id : 0 , "students.name": 1 }) .toArray();</code></p> <p>Het resultaat van een find is één of meer documenten en kan in een variabele ondergebracht worden. Deze <u>variabele of pointer</u> noemt men in Mongo: de CURSOR. Op deze cursor kunnen verschillende (callback)methodes worden uitgevoerd.</p> <pre>docs.forEach(function(doc) { }); var count = docs.size(); var fiveResults = docs.limit(5) docs.sort({name:1 , "info.type":"txt":-1}) // -1 is descending, 1 is ascending</pre> <p>Om zoekopdrachten te versnellen kunnen verschillende soorten INDEXES worden toegekend. Ze worden bewaard in de collectie system.indexes en zoeken op een veldnaam</p> <pre>db.foo.ensureIndex({name:1, age:-1, "info.favorite":1}) db.foo.ensureIndex({name:1},{unique:true}) db.foo.dropIndex({"name"})</pre>
Update	<p>update, save , findAndModify <code>db.foo.update(query, updateoperator, [options:One,Many,Upset]);</code></p>

	<p>db.foo.save(doc.x + 1) -> kan concurrency problem geven. Daarom gebruik je beter een update. Update voorziet verschillende operatoren (starten met \$), die concurrency problemen verhinderen:</p> <pre>db.foo.update({ _id:1 }, { \$inc:{x:1} }) db.foo.update({ _id:1 }, { \$set:{NieuwOfUpdateVeld: 'iets'} }) db.foo.update({ _id:1 }, { \$unset:{VerwijderVeld: ''} }) db.foo.update({ _id:1 }, { \$rename:{ 'oldName': 'newName' } })</pre> <p>Er bestaan gelijkaardige operatoren voor een Array { "_id:1", "myArray": [] }:</p> <ul style="list-style-type: none"> \$push voegt de waarde onvoorwaardelijk toe, db.foo.update({ _id:1 }, { \$push: {myArray : 10} }) \$addToSet voegt de item toe, mocht hij nog niet bestaan, db.foo.update({ _id:1 }, { \$addToSet: {myArray : 10} }) \$pull verwijdert de elementen met de key: db.foo.update({ _id:1 }, { \$pull: {myArray : 10} }) \$pop verwijdert het laatste element. <p>Een query kan een _id of een veld zijn: { name: 'johan' }</p> <p>Een query {myArray:3} update een array, waarin een waarde 3 voorkomt.</p> <p>Een query { } update alle documenten (alle _ids dus)</p>
Delete	<p>remove, drop</p> <pre>db.foo.remove({ }) db.foo.remove({ status: "D" })</pre>

Meer mongo commands, inclusief een mapping tussen SQL commands en Mongo commands, zijn te vinden op: <http://docs.mongodb.org/manual/>

1.1.1.8 Robomongo als MongoDB management tool.

Werken met de console kan voldoende zijn. Is meer interactie nodig met MongoDB dan voorziet RoboMongo een handige database manager: <http://robomongo.org/>



1.1.1.9 Node connecteren met mongo

mongoose



Er bestaan verschillende mongodb drivers. Er bestaan zelfs native mongodbdriivers voor C++, C#, Java, PHP en natuurlijk ook voor Node. Toch kiezen we voor Mongoose als driver. De reden is te zoeken in de extra functionaliteiten zoals validatie, collecties met pseudo joins, een eenvoudige query builder en dit op basis van een **Object Document Mapper** (lees: ODM in plaats van ORM). (<http://docs.mongodb.org/ecosystem/drivers/node-js/>)

Mongoose docs: <http://mongoosejs.com/index.html>

1. Installeer een express app en voeg mongoose toe: `npm install mongoose`
Visual Studio: start een express app.
Webstorm: `npm install express -g`, `npm install express-generator -g`
2. Voeg Mongoose dependency toe in package.json (indien dit niet automatisch zou gebeurd zijn met een `install -- save` instructie)

```
"dependencies": {
  "express": ">=4.9.8",
  "jade": "*",
  "stylus": "*",
  "mongoose": ">=3.0.0"
}
```

1.1.1.10 MongoDB, Mongoose en Express4:

In Express4 gebeurt alle opstart en configuratie vanuit de www file. We volgen die structuur en roepen in de www file een zelf uitvoerende module op in de data map: connectDBService.js

```
var DBService = require("../data/connectDBService.js");
```

De service wordt gebruikt om de connectie op te zetten op basis van de gekozen driver (hier mongoose). Daarnaast wordt gebruik gemaakt van een config.js file voor bvb. de connectionstring.

```
var connectDB = DBService (config.MONGODBURL,require('mongoose'));
```

```

/* config.js */
var config = {
  HOST: 'http://localhost',
  PORT: getEnv('PORT') || 3000,
  MONGODBURL : process.env.MONGO_URI || 'mongodb://localhost/usersDB'
  //andere:'mongodb://username:password@localhost/usersDB?options...');
};

/* /data/connectDBService.js */
module.exports = (function (configURL, database) {
  //var mongoose = database;
  var db = database.connect(configURL)// connecteer de database

  database.connection.on("open", function ()[...]);
  database.connection.on("error", function (error) [...]);
  database.connection.on("close", function ()[...]);

  return database; // mongoose connected
});

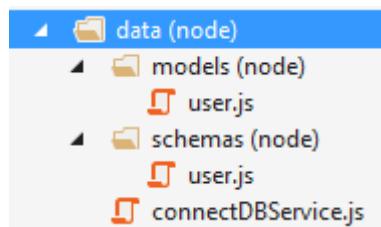
```

18.4.4 CRUD mongo via node

Eénmaal de connectie bestaat is het onnodig (maar het mag) om de database en collections zelf aan te maken. Bij een eerste save zorgt MongoDb daar zelf voor.

CRUD acties vanuit node naar mongo, steunen op het maken van een model en een schema dat verwijst naar de respectieve collectie. Enkel de zaken in het model zullen behandeld worden door mongo. Velden, die niet bestaan in het model en die je toch oproept in node worden niet behandeld door mongo. Dit resulteert in tijdswinst en een sneller antwoord maar genereert geen error. Dit een duidelijk verschil met het 100% consistent ER model in een relationele database.

1.1.1.1 Model en schema aanmaken



In de map data (waar de mongoDB connectie gebeurt) voegen we een map models en een map schema's toe. Denk niet over dit schema als een volwaardig ER diagramma. Het gaat over het opleggen van een beperkt aantal regels, die gevalideerd worden voordat mongoose de data aanbiedt aan de database.

Schema:

De benaming bestaat uit de collectienaam gevolgd door "Schema" en wordt aangemaakt met `new mongoose.Schema({//hier bson schrijven})`. Het schema benoemt in een key value alle velden van de documenten. Validatie kan toegevoegd worden op basis van bvb. reguliere expressies. Een schema laat de volgende datatypes toe: String, Number (geen long, geen double), Date (in ISO formaat: "2014-11-11T12:00:00.006Z"), Boolean, Buffer (stoken van binary data), ObjectId (een unieke identifier zoals bvb een foreign key), Array of Mixed (= een verzameling van de voorgaande).

<http://mongoosejs.com/docs/guide.html>

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

var emailRegExp = /.+@\.+\.+/.;

var UserSchema = new mongoose.Schema({
  username: {type:String, unique:true},
  name: {type:String, index:true},
  profession: String,
  email: {
    type:String,
    required: false,
    match: emailRegExp
  },
  gender: {
    type: String,
    required: true,
    uppercase: true,
    trim: true,
    enum:[ 'M', 'F' ]
  },
  createdOn: {type:Date, 'default':Date.now }
});
```

Noot:

- Andere datatypes zijn Number ({ type: Number, min: 18, max: 65 }), Boolean, Buffer. Ze kunnen ook als collectie aangebracht worden: [] of [Number]
- Alle datatypes kunnen op "required" gevalideerd worden. Strings laten "enum" en "match" toe. Numbers kunnen op "min" en "max" controleren.
- Een error E11000 wijst erop dat het veld niet "unique" is.
- Bij het opvragen van een document kan een _v verschijnen. Dit is een versie nummer voor een document dat automatisch door Mongoose aangebracht wordt.
- In tegenstelling tot relationele databases hoeft het schema niet af te zijn om je applicatie aan te maken. Het schema kan meeëvolueren gedurende de lifecycle van de applicatie. Het schema is wel geen garantie voor integriteit van de applicatie. De applicatie is belangrijker dan het schema met vooral aandacht voor een hoge performantie (geen overhead).
- Schema's kunnen ook genest worden. Het oproepen van een ander schema gebeurt op basis van zijn naam en is een collectie :

```
var LineupSchema = new mongoose.Schema({
  artist: {type: String}
})

var FestivalSchema = new mongoose.Schema({
  name: { type: String, unique: true },
  lineups : [LineupSchema]
})

module.exports = FestivalSchema; //voldoende de parent te exporteren
```

- Preserved words kunnen voor problemen zorgen. Een herhaling van het woord in een javascript object kan de oplossing zijn: var addressSchema = { type: {type:String} , street: ...}
- Om geen autogenerated _id te hebben: new Schema({ _id:false })
- Bepaalde delen van een Schema kunnen zelfs toegevoegd worden op basis van javascript codes (vb. een if())

Model:

In het model (waarvan de naam bij conventie op het schema is afgestemd) is natuurlijk het schema nodig (require). De naam van de collectie wordt toegevoegd en het model wordt beschikbaar gesteld voor andere module (oa. Routing handlers) met een export.

```
var mongoose = require("mongoose");
var UserSchema = require("../schemas/user");
var User = mongoose.model('User', UserSchema, "users"); //model, schema, collection
module.exports = User;
```

Noot: de collection naam bestaat normaal uit het meervoud van het model, maar dan in kleine letters. Je hoeft het dus niet te vermelden als derde argument. Het laat wel toe om zelf een naam voor de collectie te definiëren (case sensitive). Dit kan alternatief ook in het schema gebeuren. In dat geval vul je als tweede constructor argument de property collection toe met de gewenste naam.
var UserSchema = new mongoose.Schema({username: } , collection: 'usersList' }

Vergeet ook niet om het model te exporteren! Vergeet je dit, dan wordt geen fout geworpen maar krijg je als zoekresultaat vaak alleen het _id.

Document:

Eénmaal model en schema gemaakt zijn, kunnen we aan instanties of documents gaan denken. Dit zijn niets meer dan javascript objecten, die je toevoegt aan usersDB. Een bijhorend document zou er zo kunnen uitzien:

```
var userA = new User({
  username: 'userA', name: 'Van', // ...
});
var userB = new User({
  username: 'userB', name: 'Ver', //...
})
```

1.1.1.2 De routing handlers

De routings volgen de naam conventie (hier dus routes/user.js). Het volstaat om het model te requiren. Het model zorgt immers voor de database verbinding. De methode find() is de query methode van mongodb en laat toe om bvb alle users op te vragen.

```
var User = require('../data/models/user');
module.exports = function (app) {

  router.get('/', function (req, res) {
    User.find({}).sort('username').sort('address').exec(function (err, docs) {
      res.render('users/index', { title: 'Users overzicht', userlist: docs });
    })
  });
}
```

1.1.1.3 Een repository voor de mongoose queries.

1.1.1.4 Mongoose Queries met de find functie

Mongoose voorziet een reeks static helper methodes om data op te zoeken. Deze methodes zijn gelijkaardig aan de methodes in MongoDb. Deze methodes (bvb. find() functie) laten chaining toe met andere aanvullende methodes van mongoose en voorzien in veel zoek mogelijkheden, die aan SQL statements doen denken (<http://mongoosejs.com/docs/queries.html>):

where()	Additionele zoekfuncties, die kunnen gechained worden: <code>User.find({}).where("lastName").equals("Van") where("age").gt(55)</code>
limit()	Neemt een integer als argument <code>User.find({}).limit(10)</code>
sort()	Sorteren (kan net als de andere functies in chaining gebruikt worden) <code>User.find({}).sort('username').sort('lastName firstName')</code>
select()	Retrunt een subset van fields van de gevonden documenten. <code>User.find({}).select('lastName email');</code>
exec()	Voert de query uit en roept de callback functie op. Vergeet de exec niet bij gebruik van mongoose zoek functies.
find(conditions, [fields], [options], [callback]) find({ }) findOne({ }) findById({ObjectID})	Tussen de accolades van find({ }) kan een literal object, maar ook string of een reguliere expressie aangebracht worden voor het zoeken in de docs, of in arrays. De resultaten komen binnen via een callback functie. <pre>User.find({ name: {first:"johan", last:"van"}}, function(err, results) { }); User.find({ name.last: "van"}, function(err, results) { }); User.find({ lastName: /^V/ })//begint met een V User.find({ age: { '\$gt': 18, '\$lte': 25 } }) // 18<age<= 25 User.find({ name: {"johan":1,'name city':1}, function(err, results) { }})// enkel de velden name en city worden returned User.find({ name: {"johan":1,'-city':1}, function(err, results) { }})// alle velden worden returned buiten city</pre> Naast find() bestaat ook een findOne(), wat de eerste match returnt en een findById({ObjectID})
Update(condition, update, [options], callback)	Find en save kunnen gecombineerd worden in een update (condition, update, function(err, results) { }): <code>User.update ({name: "johan"}, {lastName:"vannie"}), function() { }</code> De options laten verschillende update methodieken toe door het opgeven van een true of false: safe: bij true worden errors teruggegeven door callback functie upsert: maak het document aan indien onbestaand

	multi: meerdere docs mogen aangepast worden strict: alleen data in het model wordt gesaved overwrite: al dan niet overschrijven
remove(condition, callback)	Verwijderen van document(en)
findByIdAndRemove(id, [options],[callback])	

Vanaf Mongo4+ beschikken de queries over een "then" methode. Ze kunnen m.a.w. als promises gebruikt worden.

1.1.1.5 *Middleware om DRY te werken*

Bij het opvragen via een query kan handig gebruikt gemaakt worden van eigen middleware om hergebruik te bevorderen en het overzicht te behouden. Hieronder een voorbeeld van middleware die een user ophaalt op basis van zijn naam.

```
//routing handler in de map: routes>> users.js gebruikt middleware
var loadUser = require('./middleware/load_user'); //ophalen van één user

router.get('/:_name', loadUser, function (req, res, next) {
    res.render('users/details', { title: 'User profile', user: req.user })
})

//middleware loadUser in de map routes>>middleware>>load_user.js
var User = require("../data/models/user");

function load_user(req, res, next) {
    User.findOne({ username:req.params.name }, function (err, user) {
        if (err) { return next(err) }
        if (!user) {
            return res.send('Not Found', 404);
        }
        //user als eigenschap van req aanbrengen, waardoor vlot uitvraagbaar via req
        req.user = user;
        next();
    });
}

module.exports = load_user;
```

1.1.1.6 *De HTTP verbs PUT en DELETE*

//delete functie (maar ook update) kan de hierboven vermelde middleware gebruiken:
Het http "Delete" keyword is enkel beschikbaar in express4 bij het gebruik van de API. Als oplossing gebruikten we al eerder een reguliere POST met een switch case als universele oplossing

```
router.route('/:_name').post(loadUser , function (req, res, next) {
    switch (req.body._method) { . . .
```

Noot: Als alternatieve oplossing voor het niet gekend HTTP verb "delete" of "put" kan bestaande middleware worden gebruikt. De aanpassing bestaat erin om een standaard POST te overschrijven met de middleware '**method-override**' :

De routing ziet er als volgt uit.

```
router.route('/:name').delete(loadUser, function (req, res, next) { });
```

Voeg de middleware toe aan **app.js**. juist vóór het oproepen (app.use) van de routes

```
var methodOverride = require('method-override');
```

Initialiseer de middleware juist vóór het oproepen (app.use) van de routes

```
app.use(methodOverride(function (req, res) {
  if (req.body && typeof req.body === 'object' && '_method' in req.body) {
    // look in urlencoded POST bodies and delete it
    var method = req.body._method
    delete req.body._method
    return method
  }
}))
```

Noot: Vergeet niet om deze router te exporteren. De router wordt opgevraagd in app.js. Zoniet krijg je een fout in de zin: *Router.use() requires a middleware function but got an Object*.

18.4.5 Validatie bij Post en Put verbs:

Maak zoals gebruikelijk vooraf de action methode put, post , delete ...kenbaar in de view:

```
input(name='_method', value='PUT', type='hidden')
```

Validatie & Custom validatie

De validatie wordt aangestuurd vanuit het schema. Het mongoose schema laat een aantal standaard validaties toe (zie hoger). Tijdens het saven (user.save(function(error, user) { })) zorgt een foutieve validatie voor een error in de callback functie. Dit error is een Javascript object dat een message, name en een opsomming van errors bevat.

Custom validatie is mogelijk door zelf een "validate" functie aan te brengen met de eigenschap validator in het schema. Je kan een eigen functie en een eigen boodschap aanbrengen.

```
validate: {
  validator: checkLength, //de validating functie
  message: "Niet meer dan 10 karakters" //de custom errormessage
},  
  
var checkLength = function (val) {
  if (val && val.length < 10) {
    return true;
  } else {
    return false;
  }
}
```

Verschillende customized validaties functies (op bvb. een zelfde eigenschap) kunnen ook na elkaar aangebracht worden. Dit kan handig gebruikt worden om de interne aanwezige mongoose validaties zelf aan te maken in bvb. een eigen taal.

```
UserSchema.path('name').validate(function (value) {
    return checkRequired(value); //validatie functie uit te werken
}, 'Naam verplicht.');
```

Code voorbeeld.

Een editeer voorbeeld, die nu niet de loadUser middleware, gebruikt maar de User zelf opzoekt in de database:

```
router.put('/users/:name', function (req, res, next) {
    User.findOne({ username: req.body.username }, function (err, user) {
        if (err) {
            return next(err);
        }
        if (user) {
            user.update(req.body, function (err) {
                if (err) {
                    if (err.name === 'ValidationError') {
                        //error weergave op client verder afwerken
                        return res.send("validation error");
                    }
                    return next(err);
                }
                res.redirect('/users');
            });
        }
    });
});
```

De post is gelijkaardig van uitwerking met een user.create(req.body, function(err) { }):
app.post('/users', function (req, res, next) { });

Om alle errors te returnen kan de methode map van javascript helpvol zijn:

Array.prototype.map(callback[, args]) maakt een nieuwe array aan en roept voor elk element de callback functie op. De array met errors haal je op met Object.keys(err.errors). Vervolgens wordt voor elk element een callback mapping gedaan. De foutberichten kunnen met een join en een nieuwe lijn (br) aan elkaar gekoppeld worden. De resulterende foutbericht kan samen met de render teruggestuurd worden naar de view als een errorMessage =

```
var errorMessage = Object.keys(err.errors).map(function (errField) {
    return err.errors[errField].message;
}).join('<br/>')
```

Let wel: in jade moet de break gerendered worden en niet als
 worden weergegeven .
Dit kan met != errorMessage

Custom validatie messages aanbrengen kan via het model. Even goed kunnen de basismessages van mongoose overschreven worden, waarbij de betreffende documentnamen kunnen opgeroepen worden met {PATH}. Met messages.general overschrijf je alle default messages voor deze fout.:

```
mongoose.Error.messages.general.required = "Veld '{PATH}' is verplicht.";
mongoose.Error.messages.Number.min = " Veld '{PATH}'{{VALUE}} is te klein.";
```

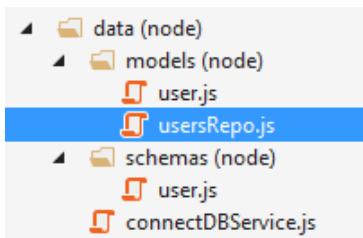
http://mongoosejs.com/docs/api.html#schematype_SchemaType-validate

Uitwerken met een API en JSON communicatie:

Evengoed kan het voorbeeld uitgewerkt worden als API met JSON communicatie. Een code voorbeeld is te vinden op: <http://webaplog.com/express-js-4-node-js-and-mongodb-rest-api-tutorial/>.

18.4.6 Het gebruik van een repository

Het kan noodzakelijk zijn om verschillende keren dezelfde queries (= dezelfde dataaccessors) op te roepen. Queries horen niet echt thuis in de routings. We verplaatsen ze daarom naar het model. Soms worden ze toegevoegd aan het model zelf (`users.js`) maar evengoed kunnen ze ondergebracht worden in een afzonderlijke file (`usersRepo.js`).



De route in `routes/users.js` wordt:

```
//met repository:
User.getUsers=function (error, users) {
  if (error) { //handleError }
  res.render('users/index', { title: 'Users overzicht', userlist: users });
});
```

Het model in `models/users` krijgt de dataaccessor, die met een callback de data ophaalt:

```
User.getUsers = function (callback) {
  User.find({}).sort('name').exec(function (err, docs) {
    if (err) { console.log(err); callback(err, null) }
    callback(null, docs);
  })
}
```

Alternatieve mogelijkheden:

- Alle mongoose queries in een afzonderlijke javascript module onderbrengen onder >> Models >> `usersRepo.js`. Dit repository maakt gebruik van het Model:

```
//models > UsersRepo.js
UsersRepo = (function () {
  var User = require("./user.js");

  var getAllUsers = function (callback) {
    User.find({}).sort('name').exec(function (err, docs) {
      if (err) { console.log(err); callback(err, null) }
      callback(null, docs);
    })
  }
  return {
    User : User ,
    getAllUsers: getAllUsers
  }
})
```

```
})();
```

Ophalen in de routes gaat via : UsersRepo.getAllUsers(`function (users) { . . . };`);

2. De dataaccessors (mongoose queries) opstellen met Promises in deze usersRepo module:

```
var q = require('q'); //promise library
var getAllUsersAsPromise = function (User) {
    return q.Promise(function (resolve, reject) {
        User.find({}).sort('name').exec(function (err, docs) {
            //Promise error retournt de foutbericht
            if (err) { reject(err) }
            //Promise resolve
            resolve(docs);
        })
    })
}
```

In de router wordt de Promise met done() verwerkt. De error komt als tweede (!) argument bij Promises:

```
usersRepo.getAllUsersAsPromise(User)
    .done(function (docs) {
        console.log("Alle gebruikers: ", docs)
        res.json(docs);
    },
    function (err) {
        serverError(err, 'Er gebeurde een fout bij ophalen van
gebruikers!');
    })
}
```

3. Soms kiest men ervoor om de repository als statistische methodes onder te brengen bij het Schema. Een schema bevat een statics eigenschap, die dit toelaat

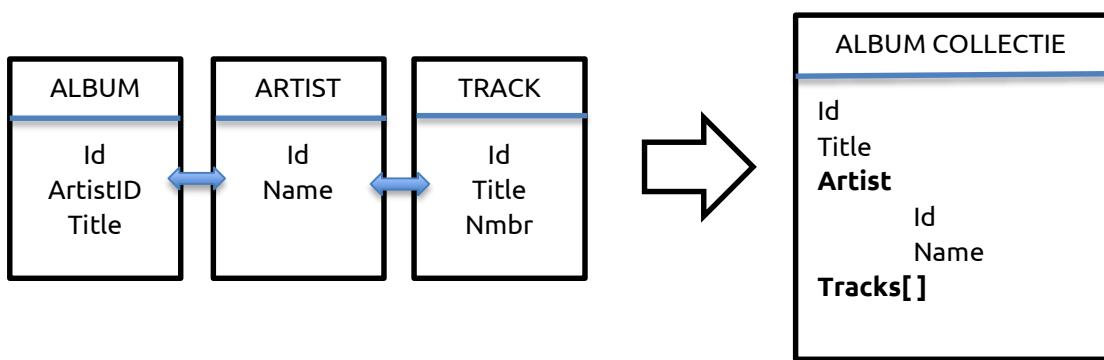
```
UsersSchema.statics.getTheUsers = function (callback) {
    var User = require('../models/user');// Hier als statics scope
    User.find({ }, 'addresses', { sort: 'addresses.city' },
    callback(null));
}
```

Oefening: Werk de CRUD acties (post, put) verder af voor MongoDb. Brente alle queries onder het model, al dan niet in een afzonderlijke file (usersRepo.js). Maak gebruik van de Mongoose documentatie.

18.4.7 Meerdere gelinkte documenten

Collectie Design = Σ documenten = documenten + arrays + subdocumenten

In tegenstelling tot relationele databases wordt de informatie zoveel mogelijk verzameld in één collectie met documenten en subdocumenten. Men spreekt over embedded documenten en embedded subdocumenten. Zeker als het over een kleiner schema gaat wordt embedding gebruikt. In plaats van 3 tabellen bij een entiteit relatie diagramma (album – track – artist) krijgen we één collectie met arrays en subdocumenten en veel minder id velden. Een collectie op zich heeft alleenstaand alle informatie inzich. Er is slechts één query nodig om data op te halen maar die kan wel iets complexer worden.



Als gevolg kan met één transactie een collectie en zijn subdocumenten aangepast worden. Je wint in snelheid (omdat niet over tabellen moet gewerkt worden) maar queries kunnen complexer worden met gevaar van fouten. Bij meerdere schrijf en leescliënts kunnen cliënts zelf lezen voordat alle schrijfinformatie verwerkt is.

Het komt er daarom op neer om de juiste toepassingen met MongoDB uit te werken. *Toepassingen die scalability vragen met kleine documenten maar veel trafiek voor kleine wijzigingen zijn de uitgesproken toepassingen voor MongoDB en Node.* MongoDB beperkt dan ook de max. grootte van een document tot 16MB (minder dan 250KB wordt aangeraden). Indexes kunnen toegevoegd worden aan collecties (max. 64 indexes per collectie) om het zoeken nog te versnellen.

Linked collections en verwante documenten : one-to-one

Documenten (vb. authorId in een article collection) kunnen verwant zijn met een document in een andere collectie (vb. de authorId is een userId in de collectie user). Dan kan de informatie in het andere document opgehaald worden in drie stappen:

1. Gebruik van de "ref: ' . . .' " eigenschap in het oproepende schema-document met als type: Schema.ObjectId
2. De routing moet het nodige _id bevatten, waarvan het detail moet worden opgehaald.
3. Ophalen van de detail data op basis van het _id met de methode populate:
`article.populate('author')`

Het resultaat (one-to-one link author –user):

//article.js in de map “schemas”:

```
var ArticleSchema = new mongoose.Schema({
  title: { type: String, unique: true },
  content: { type: String },
  author: {
    type: Schema.ObjectId,
    ref: 'User', // om populate('author') te kunnen gebruiken
    required: true
  }
});

module.exports = ArticleSchema;
```

```
//routing:article.js in de map "routes/":
app.get('/articles/:title', loadArticle, function (req, res) {
    res.render('articles/details', {
        title: req.article.title,
        article: req.article });
})

//middleware: load_article.js in de map "routes/middleware/":

var Article = require('../data/models/article');

function loadArticle(req, res, next) {
    Article.findOne({ title: req.params.title })
        //ophalen van author object en beschikbaar stellen via article.author
        .populate('author')
        .exec(function (err, article) {
            if (err) { return next(err) }
            if (!article) { return res.send('Not found', 404) }
            //article bijhouden in req object
            req.article = article;
            next();
        })
}
}
```

Model	View	Controller
<ul style="list-style-type: none"> data <ul style="list-style-type: none"> models <ul style="list-style-type: none"> article.js user.js schemas <ul style="list-style-type: none"> article.js user.js connectDB.js 	<ul style="list-style-type: none"> views <ul style="list-style-type: none"> articles <ul style="list-style-type: none"> create.jade details.jade edit.jade index.jade users <ul style="list-style-type: none"> create.jade details.jade edit.jade index.jade index.jade layout.jade 	<ul style="list-style-type: none"> routes <ul style="list-style-type: none"> middleware <ul style="list-style-type: none"> load_article.js load_user.js loggedIn.js article.js index.js session.js user.js

Embedded documenten : one-to-many

Embedded documents kunnen een verzameling bevatten. (one-to-many relatie). Zo kan een user verschillende adressen bevatten:

```
{
  "_id" : ObjectId("54774ec78c61f15c01b3621f"),
  "username" : "Johan",
  "addresses" : [
    {
      "type" : "thuis",
      "street" : "Studieweg",
      "number" : 25,
```

```
        "city" : "Brugge"
    },
{
    "type" : "kot",
    "street" : "Nodestraat",
    "number" : 123,
    "city" : "Kortrijk"
}
],
"gender" : "M",
"createdOn" : ISODate("2015-11-27T16:18:15.862Z"),
"__v" : 0
}
```

Een schemas > address.js file bevat een schema zoals gewoonlijk. Het ophalen van de adressen - embedded in de user- kan via een repository in het model. De mongoose find query wordt hiervoor aangevuld met de verwachte collectie naam 'addresses':

```
var _findByIdUserId = function (userid, callback) {
    User.find({ _id: userid }, 'addresses',
        function (err, docs) {
            if (err) { console.log(err, null); }
            callback(null, docs[0]._doc.addresses)
        });
}
```

Oefening: Toon op de user details pagina de embedded adressen van deze user door gebruik te maken van bovenstaand repo en en een ajax call.

1. De view start de call met een aangeboden userID:

```
script.
    var userID = "#{user._id}";

script(src="/javascripts/user.js")
```

2. Het script user.js start de ajax call:

```
$.ajax( '/address/byuser/' + userID,
{
    datatype: 'json',
...    success: function (data) {      }
});
```

3. De routing stuurt de json file:

```
res.json(_docs);
```

4. Het javascript visualiseert de ontvangen json.

18.4.8 Gebruik van positionele operator \$ bij update, delete van collecties.

Wanneer een array collectie moet aangepast worden (bvb. één van de adressen van een gebruiker, die behoren tot de array addresses) kan gebruik worden gemaakt van de positionele operator \$.

De positionele operator onthoudt de positie van de item in de array. Gecombineerd met een update of delete methode, die hun eerste argument de zoek query bevatten, zorgt dit voor een verkorte en handige manier van schrijven.

Het \$ teken fungeert als de eerste match van de aangeboden query.

bvb. Opzoeken van een "city" en "userid" om een aanpassing te doen aan de gevonden match (de gevonden city van de user):

```
User.update({ _id: userid , "addresses.city": city },
    {$set: { "addresses.$.city": "Gent" } },callback); //positionele operator
```

De callback van update is een document met informatie : nMatched (aantal matches), nModified (aantal wijzigingen). Bij Mongoose kan een nested collectie zonder _id hierbij problemen geven.

18.4.9 Mongo, Express4 en socket.io

Bij express4 gebeuren initialisaties van libraries in de bin/www file. Zo worden zowel Mongo als Socket.io als dependancies toegevoegd:

```
#!/usr/bin/env node
var debug = require('debug')('festivals');
var app = require('../app');

//----- 1. server en sockets variabelen (orig: app) -----
var http = require('http');
var server = http.createServer(app);
var io = require('socket.io')(server);

//----- 2. initialisaties -----
app.set('port', process.env.PORT || 3000);

//2.1. (self) executing database connectie
var DBService = require("../data/connectDBService.js");
var connectDB = DBService (config.MONGODBURL,require('mongoose') ); //connect

//----- 3. listeners voor server en io -----
server.listen(app.get('port'), function () {
  console.log('Express server listening on port ' + app.get('port'));
});
var sockets = io.listen(server);

//----- 4. Socket handling initialiseren (server en DB dependency) -----
require('../sockets.js')(io, connectDB);
```

18.4.10 Horizontaal scalen (auto-sharding)

MongoDB laat eenvoudig horizontal scaling toe (= het toevoegen van extra machines) wat staat in contrast met verticaal scalen (= de machine krachtiger maken). Sharding technieken kunnen hiervoor gebruikt worden. Dit is het opsplitsen van de database in verschillende geclusterde databases. Dit laatste is veel moeilijker te realiseren bij relationele databases in vergelijking met document databases. Zonder extra efforts voorziet MongoDB sharding.

18.4.11 Restore en backup een mongodb database

Er zijn verschillende methodieken terug te vinden in de documentatie (<http://docs.mongodb.org/manual/core/backups/>). Backup/restore kan met Data Files, mongodump, door het gebruik van MongoDB Tools of MongoDB Management Services (MMS). Met deze laatste kunnen continu replica sets aangemaakt worden met een recovery punt.

Backup met [mongodump](#) kan snel en eenvoudig vanuit de console:

Controleer vooraf de beschikbare collections:	<pre>db.getCollectionNames() db.myCollectionName.stats()</pre>
Voer het mongodump.exe command uit vanop de mongodb bin folder. Omdat dit een dump folder aanmaakt, start je de console best als administrator. Dit resulteert in een index files en twee files per collectie in de onderliggende "dump/ <i>dbName</i> " folder: <pre>myCollectionName.bson myCollectionName.metadata.json system.indexes.bson</pre> De metadata file bevat schema informatie in een json formaat. Voor meer info over beschikbare commands gebruik je het command: <code>:mongodump --help</code>	<pre>mongodump --db myCollectionName</pre> <div style="background-color: #f0f0f0; padding: 5px;"> <pre>C:\Program Files\MongoDB 2.6 Standard\bin>mongodump -d users connected to: 127.0.0.1 2014-12-17T12:18:02.310+0100 DATABASE: users to dump\users C:\Program Files\MongoDB 2.6 Standard\bin>mongodump --db users connected to: 127.0.0.1 2014-12-17T12:22:28.162+0100 DATABASE: users to dump\users 2014-12-17T12:22:28.162+0100 users.system.indexes to dump\users\system.indexes.bson</pre> </div>
In de mongo console: Drop (als test) de originele database en controleer het resultaat met stats.	<pre>db.myCollectionName.drop() db.myCollectionName.stats()</pre> <div style="background-color: #f0f0f0; padding: 5px;"> <pre>> db.users.drop() true > db.users.stats() { "ok" : 0, "errmsg" : "Collection [users.users] not found." }</pre> </div>
Restore de database vanuit de dump folder met behulp van een mongorestore.exe in de command window. Controleer het resultaat met "show dbs".	<pre>mongorestore dump/myCollectionName</pre>