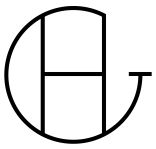


HoGent

Faculteit Bedrijf en Organisatie

Monitoring mogelijkheden in Node.js toepassingen voor feilloos debuggen in 2019

Michiel Leunens

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Tom Antjon
Co-promotor:
Michiel Cuvelier

Instelling: beSports bvba - Kayzr

Academiejaar: 2018-2019

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Monitoring mogelijkheden in Node.js toepassingen voor feilloos debuggen in 2019

Michiel Leunens

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Tom Antjon
Co-promotor:
Michiel Cuvelier

Instelling: beSports bvba - Kayzr

Academiejaar: 2018-2019

Tweede examenperiode

Woord vooraf

Backend debug toepassingen zijn vaak slecht onderhouden, incompleet of gewoonweg verschrikkelijk duur. Daarom is deze bachelorproef gemaakt met als doel een open-source debug tool te schenken aan iedereen die hun project in NodeJS en Express ontwikkelt. Mede mogelijk gemaakt dankzij Kayzr, is het de bedoeling om deze tool online te zetten zodat deze effectief in de praktijk gebruikt kan worden. Graag bedank ik ook Dhr. Tom Antjon, voor altijd paraat te staan, snel mijn vragen te kunnen beantwoorden en mij een goed inzicht te geven in het werk dat nog gedaan moest worden.

Ook wil ik Kayzr bedanken, voor een onvergetelijke stage, waar ik als developer EN als fotograaf enorm heb kunnen groeien. Zonder hen had ik nooit deze bachelorproef kunnen verwezenlijken.

Samenvatting

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

1.1 Probleemstelling

Het zoeken naar fouten in je software kan een frustrerend, hectisch en vermoeiend proces zijn. Daarom zijn, als ontwikkelaar zijnde, toepassingen die dat proces voor jou kunnen automatiseren of beter analyseren altijd welkom. Kayzr, een Belgische start-up die zich bezighoudt met het mainstream maken van E-Sports in de Benelux en het organiseren van toernooien ervan, wenst zulke software zodat hun productieproces versneld kan worden. Het concreet probleem gaat als volgt: Kayzr's processen werden op verschillende plekken gemonitord. De tools zijn incompleet, en daardoor heeft men een gefragmenteerde collectie aan logs, foutmeldingen, informatie, enz... Per API call moet men foutmeldingen controleren in de terminal, IP-adressen opzoeken in Google Cloud platform en andere nuttige informatie staat dan weer eens ergens anders opgeslagen. Hierdoor duurt het lang om fouten te analyseren en te komen tot een mogelijke oplossing. Een degelijke tool zoals PM2, een monitoringstoepassing met veel meer functionaliteiten, meer dan dat Kayzr nodig heeft, is meteen een kost dat ze zich niet kunnen permitteren. Aan €50 per server per maand, kost hen dit met een vijftigtal servers al te veel. Deze kost zou schalen zeer moeizaam of zelfs onmogelijk maken.

1.2 Onderzoeksvraag

De onderzoeksvraag gaat dus als volgt: bestaat er een mogelijkheid om de monitoringstechnieken van Kayzr, en mogelijks andere bedrijven, op een goedkope manier te stroomlijnen en te verbeteren binnen een NodeJS omgeving, zodat hun debugproces geoptimaliseerd kan worden?

1.3 Onderzoeksdoelstelling

Kayzr is tevreden met de toepassing en gaat in hun project hiervan gebruik maken. De gemiddelde tijd om een probleem op te lossen is gedaald en dit proces verloopt efficiënter. De data die per proces is verzameld kan op een goede manier gevisualiseerd worden zodat deze ook voor andere doeleinden kan gehanteerd worden. Er kan meer data ontleed worden dan ervoor, en de software voldoet aan alle opgesomde requirements.

1.4 Opzet van deze bachelorproef

Het vervolg van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 volgt een stand van zaken over JavaScript, Node.js, Express en Express middleware. Ook wordt er onderzoek gedaan naar verschillende monitoringssoftware. Ook wordt het huidige debug proces bestudeerd. Hoe verloopt dit? Wat kan er beter? Wat kan er sneller? Welke functies moet de nieuwe toepassing bevatten?

In Hoofdstuk 3 schrijven we software naargelang de requirements, en toetsen we die op verschillende factoren, zoals later zal worden besproken, aan de oude processen.

In Hoofdstuk 4, wordt er gekeken of er een oplossing bestaat op deze onderzoeksvragen. We kijken of Kayzr deze gaat gebruiken naar de toekomst toe en wat er gaat gebeuren met de eigendom van dit softwareproject.

2. Stand van zaken

Dit hoofdstuk bestaat uit een zeer uitgebreide literatuurstudie, waarin de kern van het probleem wordt opgesplitst in verschillende delen. In het eerste deel wordt er uitleg gegeven over JavaScript zelf en diens evolutie. Vervolgens volgt er een groot deel over de frameworks Node.js en Express, erna bekijken we het belang van testen en monitoren van software en breiden we uit naar een analyse van zulke monitoringssoftware. In het tweede deel wordt er onderzoek gedaan naar de vraag van Kayzr. Wat wordt er verwacht, welk inzicht moet er gegeven worden, welke features moeten worden uitgewerkt.

2.1 Javascript

Javascript is een programmeertaal gemaakt voor het web, waarmee statische websites kunnen omgezet worden naar dynamische en interactieve websites. Doordat het een enorm krachtige scripttaal is dat speciaal werd ontwikkeld om de functionaliteiten van een doorsnee HTML/CSS-pagina uit te breiden, wordt het bijna onmogelijk om nog iets in te beelden dat niet geïmplementeerd kan worden m.b.v. Javascript. De taal is weakly-typed, functioneel, event-driven en dynamisch. JavaScript bevat een bibliotheek van standaard-objecten samen met specifieke taalelementen zoals operatoren, controlestructuren, enz... Deze kern of 'core' kan makkelijk uitgebreid worden met extra objecten waardoor men JavaScript makkelijk kan specifieren als *Client-side JavaScript* en *Server-side JavaScript*. Client-side JavaScript wordt bijvoorbeeld gebruikt om interactie met de gebruiker te verzorgen zoals muiskliks, keyboard-hits en nog veel meer. Server-side JavaScript houdt zich eerder bezig met het achterliggende van een site, bijvoorbeeld om een webapplicatie te doen communiceren met een databank. Een zeer bekende en veel gebruikte Server-side JavaScript variant is Node.js, waarover later meer uitleg wordt gegeven. (**Javascript2019**)

2.1.1 Tijdlijn van JavaScript

Javascript heeft echter een hele evolutie achter de rug. Ontstaan in mei 1995, wordt de taal na vele updates nog steeds dagdagelijks gebruikt en kan het wereldwijde web niet meer ingebeeld worden zonder. Brendan Eich, de auteur van de taal, werkte in 1995 samen met Netscape Communications, de makers van de eerste grote webbrowser genaamd Netscape Navigator, om een taal te implementeren in hun browser waar webontwikkelaars gebruik van zouden kunnen maken. Java, een zeer zware programmeertaal met tal van functionaliteiten was de eerste keuze van Netscape, maar Eich schreef uiteindelijk zijn eigen idee uit van een scripttaal in nog geen 10 dagen en overtuigde Netscape om de lichte, schaalbare en Java-complementerende-taal te adopteren (**Rangpariya2019**). Javascript, toen onder de naam Mocha en vervolgens LiveScript, was geboren en zette de wereld van webontwikkelaars op zijn kop.

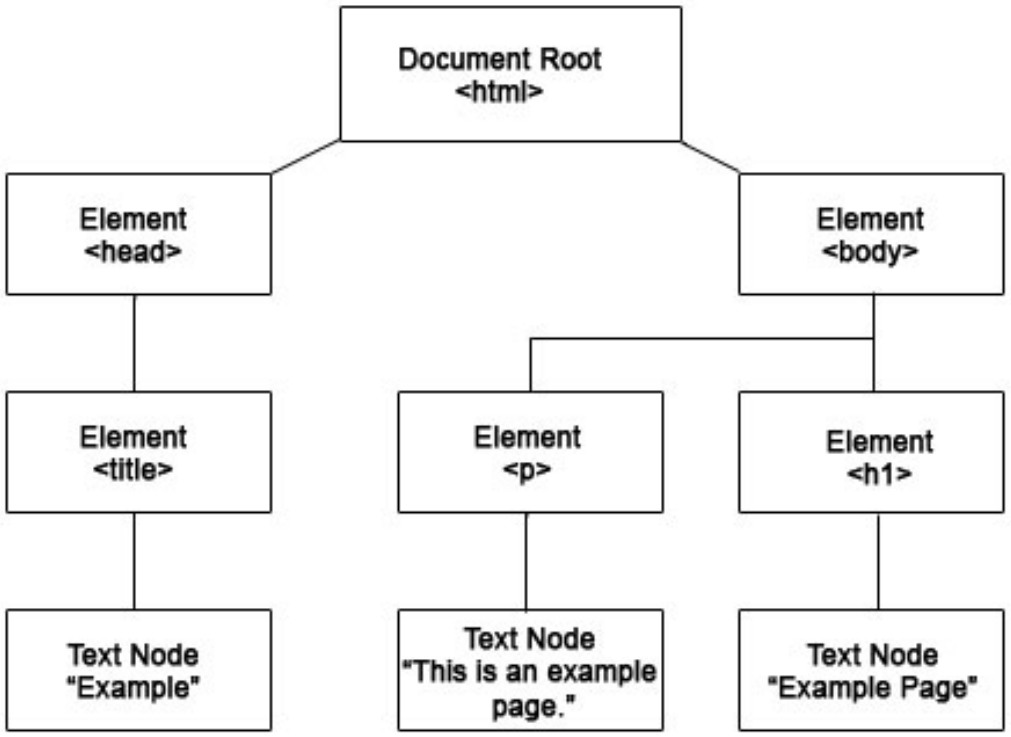
Netscape bracht nieuwere versies van JavaScript uit, maar Microsoft dreigde die te onttronen. Niet zoveel later, bracht Microsoft Internet Explorer 3 uit met een eigen variant op Javascript, namelijk JScript. Dit was een zware slag voor Netscape, aangezien Microsoft hen hierdoor zou voorbijsteken, maar was wel een grote stap in de evolutie van JavaScript zoals wij het nu kennen. Dit bracht echter problemen met zich mee, want bedrijven zouden telkens eigen versies van JavaScript uitbrengen wat voor veel compatibiliteitproblemen zou zorgen. Uiteindelijk, in 1997, werd JavaScript 1.1 gestandaardiseerd dankzij het European Computer Manufacturers Association en werd omgedoopt tot ECMAScript. (**Wiley2016**) Deze standaard, namelijk ES1 (versie 1), zou dan vertakkingen verhelpen. Implementaties van ECMAScript, waaronder JScript, ActionScript, maar dus ook JavaScript zelf, zouden dan telkens ECMAScript implementeren waardoor de core van de varianten telkens hetzelfde blijft. Nu, in 2019; is JavaScript nog steeds enorm populair en implementeert ondertussen al de 9e versie van ECMAScript: ES2018. Na ES5 is ECMASCRIPT overgeschakeld naar jaarlijkse releases, startend vanaf ES2015.

2.1.2 Waarom JavaScript?

JavaScript is niets voor niets enorm populair, zelfs nog steeds na al die jaren, dankzij de vele voordelen dat de taal met zich meebrengt. En die lijst van voordelen wordt per iteratie alleen maar groter. Hieronder worden er enkele kernvoordelen opgesomd.

Dynamisch en weakly-typed

Een dynamisch getypeerde taal slaat neer op het feit dat het type van waarden kan veranderen tijdens uitvoertijd. Dit betekent bijvoorbeeld dat een variabele string plots kan gebruikt worden om een som te maken met een getal. Dit maakt uitvoertijd en compileertijd aanzienlijk sneller doordat er niet voortdurend controles moeten plaatsvinden. Dit zorgt voor hoge performantie, maar slechte nauwkeurigheid.



Figuur 2.1: Voorbeeld van een DOM structuur.

DOM-manipulatie

De DOM, ofwel Document Object Model, is kortweg het skelet van een HTML-pagina. De structuur van het skelet beschrijft een boomstructuur, waardoor een element een ouder, broer, kind, kleinkind, achterkleinkind, enz... van een element kan zijn. Het HTML element is de wortel van de boom. Client-side JavaScript wordt grotendeels gebruikt om deze boomstructuur te manipuleren. Hierdoor kunnen we HTML elementen toevoegen, verwijderen, stijlen manipuleren, enz... allemaal tijdens uitvoertijd. **Kantor2017**

Prototype-based

In JavaScript kan een object eigenlijk aanzien worden als een Array. Hierdoor kunnen ze heel gemakkelijk aan de eigenschappen van een object aan en kunnen we deze ook makkelijk tijdens uitvoertijd aanpassen. Ook kunnen we op deze manier makkelijk eigenschappen aan bestaande objecten toevoegen. In JavaScript wordt gebruikt van de 'dot-notatie' of van de 'haakjesnotatie'.

```
foo.bar = 10;
foo['bar'] = 10;
const bar = foo.bar;
foo['bar2'] = bar;
```

Event-driven

JavaScript is event-driven, wat betekent dat de code vooral kijkt naar acties van de gebruiker. Bijvoorbeeld: Wanneer een gebruiker op een knop klikt, wil je dat met een animatie de knop heeen en weer beweegt, tekst op je scherm verschijnt, en een server-request wordt gestuurd.

Functioneel

Sinds ES2015 is JavaScript niet enkel en alleen een object-georiënteerde taal, maar ook een functionele taal. JavaScript was zeer snel met het introduceren van functioneel programmeren (waaronder de populaire arrow-functions). Dit vergemakkelijkte de taal aanzienlijker, aangezien veel minder code geschreven moest worden om hetzelfde resultaat te bereiken.

Universeel

JavaScript wordt sinds ES5 ondersteund door alle moderne browsers! Of er nu in Chrome, Firefox, Safari, Edge of het oude Internet Explorer[1] wordt gesurft, ze ondersteunen allemaal JavaScript. D.m.v. de console in de browser te openen kan er al geprogrammeerd worden.

2.2 Node.js

Node.js is een open-source JavaScript runtime omgeving dat wordt gebruikt voor Server-side toepassingen. Node.js maakt het mogelijk, (of eerder *makkelijker*), om JavaScript nu ook te gebruiken buiten websites maken. Aangezien JavaScript zo'n krachtige taal is, hadden de ontwikkelaars van Node.js een ding in gedachten: JavaScript niet enkel voor browsers, maar ook voor alleenstaande applicatie. Sindsdien evenaart JavaScript aan andere scripttalen, zoals Python. **Patel2018**

2.2.1 Tijdlijn van Node.js

Node.js' verhaal start in 2009, toen een ontwikkelaar genaamd Ryan Dahl het niet zo goed stelde met Apache Http Servers.

Zoals beschreven in (**Chaniotis2015**), was (en is in grote mate nog steeds) Apache HTTP, in combinatie met PHP, jarenlang de go-to-taal om webapplicaties te laten communiceren met databases en server-side functionaliteiten toe te voegen aan webapplicaties, zoals authenticatie, file-en-ftpservers, logging en nog veel meer. PHP is echter nooit in het achterhoofd ontwikkeld geweest om hedendaagse, complexe webapplicaties te schrijven. Ten tweede was Apache incapabel om te schalen naar meerdere processorkernen, waardoor de performantie enorm daalde. PHP in combinatie met Apache was gewoonweg niet

[1] Internet Explorer wordt niet meer verder ontwikkelt, dus ondersteunt het enkel ES2015.

gemaakt voor de volgende generatie webapplicaties. Andere struikelblokken zijn cross-platform mobiele applicaties en real-time communicatie.

Meer en meer applicaties worden gemaakt met cross-platform compatibiliteit in het achterhoofd, aangezien de toestroming van verschillende nieuwe besturingssystemen en apparaten het meer en meer onmogelijk maken om native apps[2] te ontwikkelen. Hierdoor worden bibliotheken en frameworks ontwikkeld zoals React Native, Flutter, Xamarin, Ionic, PhoneGap, enz... die deze nadelen kunnen overbruggen. Deze zijn echter zo krachtig geworden en schenken zoveel nieuwe voordelen, dat server-side talen en frameworks zoals Apache HTTP/PHP eerder een bottleneck vormden.

Real-time communicatie is nog een factor dat niet ondersteund wordt door Apache. Apache was niet ontwikkeld om berichten te sturen naar de client-side. In de wereld van vandaag, waarin sociale media nog nooit zo'n grote invloed heeft gehad op ons dagelijks leven, is het haast ondenkbaar dat real-time communicatie niet zou kunnen bestaan.

Dit alles zorgde voor de ontwikkeling van Node.js. Dahl's project werd met open armen ontvangen. Na enkele struikelblokken, kende Node.js in 2011 voor het eerst het licht en wordt nu zo'n tien jaar later, gebruikt in vele moderne applicaties en is nog steeds één van de meest begeerde vaardigheden van een programmeur. (**Patel2018**)

Hoewel Apache en PHP nog steeds overduidelijk op plaats één staan, hebben ze hun populariteit enkel nog te danken aan bedrijven die op verouderde software werken, en hun populariteit bij senior ontwikkelaars. PHP blijft een geliefde taal bij velen, maar in 2.2 zien we dat Node.js's populariteit duidelijk die van Apache aan het inhalen is. (**SimilarTech**)

2.2.2 Waarom Node.js?

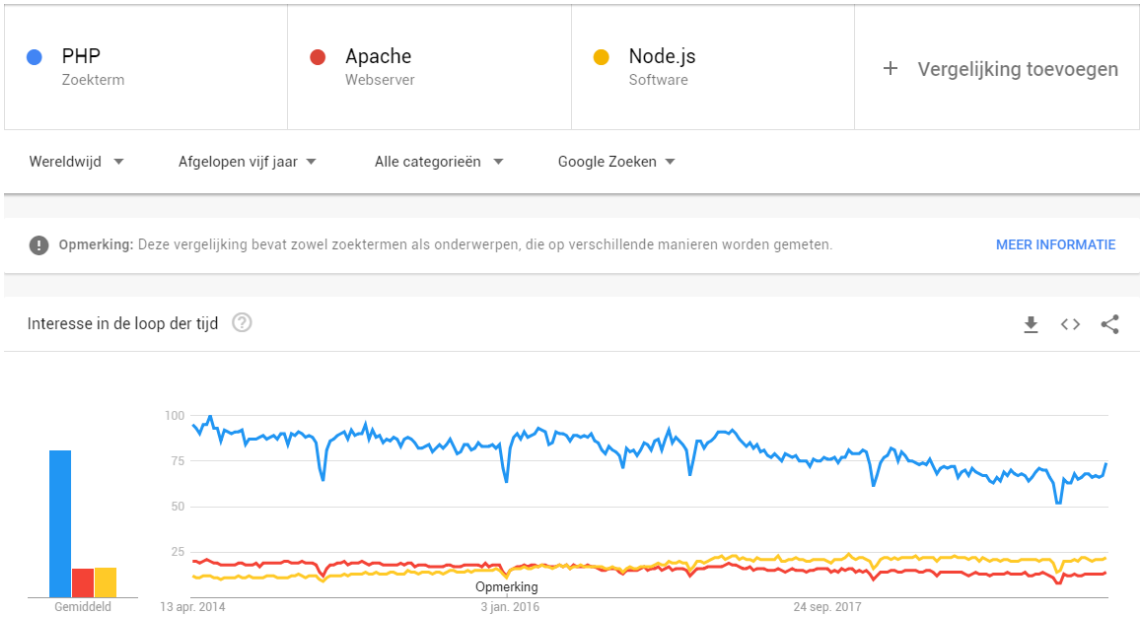
Node.js is reeds een volwassen server-side framework. Youtube, Yahoo, Google, Amazon, Netflix, Ebay, Reddit, LinkedIn, Paypal, Github, Forbes, Walmart, Uber, NASA, Slack,.. zijn slechts enkele van de duizenden websites die overgeschakeld zijn naar Node.js. En dit zijn geen kleine namen.

Illustrator **Mehmet2016** toont op een illustratieve wijze de voordelen van Node.js aan. LinkedIn kon hun 15 servers reduceren naar slechts 4 en terwijl de verkeerscapaciteit nog eens verdubbelen. Walmart kon al hun Client-side JavaScript processing naar hun servers verplaatsen, en Ebay kon hun verkeerscapaciteit enorm doen verhogen en hun verbruik enorm doen dalen. De voordelen en functionaliteiten worden hieronder nog eens opgesomd zoals aangegeven door **Chandrayan2017**.

nieuw

JavaScript is relatief gezien een nieuwe taal dat voortdurend geüpdatet wordt. In tegenstelling tot traditionele server-talen zoals Python, PHP, enzovoort. Vele dialecten van

[2]Native Apps zijn applicaties die gemaakt worden met één besturingssysteem in het hoofd. Bv: Android, IOS.



Figuur 2.2: Google searches van PHP (blauw), Apache (rood) en Node.js (geel) in de afgelopen 5 jaar.

JavaScript (zoals TypeScript, CoffeeScript, ClojureScript,...) compileren dan gewoonweg in JavaScript waardoor overschakelen zeer simpel wordt. Doordat Node.js ook in JavaScript geschreven is, maakt dit het makkelijker om de client-server-side-brug over te steken als frontend ontwikkelaar of vice versa. (**ExpressMozilla**)

Asynchroon

Node.js is sterk afhankelijk van asynchrone en continue programmeerstijl. I/O-bewerkingen worden uitgevoerd door middel van oproepen naar asynchrone functies waarbij een callback moet worden doorgegeven om aan te geven hoe de berekening wordt voortgezet zodra de genoemde I/O-bewerking asynchroon is voltooid. Het Node.js executiemodel bestaat uit een hoofdgebeurtenislus die wordt uitgevoerd op een single-threaded proces. Met behulp van deze *event loop* moet een Node.js server nooit wachten op antwoord, het kan gewoon blijven doordraaien na een oproep van een API Call en kan dankzij een notificatiesysteem op een later moment het antwoord terugsturen. Dit maakt Node.js enorm schaalbaar, in tegenstelling tot Apache dat slechts een gelimiteerd aantal threads kan starten om handelingen uit te voeren.

Het is daarom niet altijd makkelijk om deze soort frameworks te debuggen, en het wordt uiteindelijk een uitdagende taak. Gelukkig zijn hiervoor dan ook weer verschillende hulpmiddelen en tools uitgebracht om dit te vereenvoudigen. Zoals vermeld in (**Runtime2017**), kan asynchrone code subtiele bugs opleveren die niet meteen zichtbaar zijn. Hier is nog niet echt onderzoek over gedaan. Waar er honderden vergelijkende studies bestaan over de frameworks zelf en of Node.js een goede optie is, zijn er geen of amper studies over

de beste manier om dit te debuggen en hoe men dit het best aanpakt. (**Runtime2017**) vertelt ons meer over het identificeren van schaalbaarheidsproblemen en het aanrijken van mogelijke oplossingen. Ze maken gebruik van parametrische uitdrukkingen voor runtime monitoring van Node.js toepassingen, maar ze geven toe dat dit nog maar de eerste stap is en dat hier nog meer onderzoek naar kan gedaan worden. Hier wordt later in dit onderzoek dieper op ingegaan.

Performant

Google Chrome's JavaScript engine, V8, is bijzonder snel. Dahl heeft dit verder ontwikkeld voor Node.js en dit maakt het framework enorm performant en efficiënt in het compileren en uitvoeren van code. Veel sneller dan Python, Ruby of Perl.

npm

Node.js kent een enorm bloeiende community. Deel van Node.js is zijn Node Package Manager, ofwel npm, en komt meegeleverd bij de installatie. Via npm kan men uit honderdduizenden pakketjes kiezen voor een JavaScript/Node project makkelijk uit te breiden met extra functionaliteiten in één oogwenk. Dagelijks worden er nieuwe pakketjes toegevoegd aan de packet manager, en is makkelijk één van de redenen waarom Node.js zo populair is, aangezien het ontwikkelen van een webapplicatie enorm vereenvoudigd wordt.

2.3 Express

2.3.1 Waarom Express?

ExpressMozilla vertelt ons dat echter niet alles rechtstreeks door Node.js wordt ondersteund. Als men bijvoorbeeld routing aan een webapplicatie wenst toe te voegen, zoals HTTP *GET, POST, PUT, DELETE,* enzovoort... of URL-paden met specifieke requests, statische bestanden weergeven... moet men die code zelf schrijven. Herinner echter het npm-systeem van Node.js. Eén van die honderdduizenden pakketjes is namelijk Express, en is makkelijk *het* populairste Node framework en is zelfs de grondlegger van duizenden andere pakketjes. Express wordt om deze reden door bedrijven dan ook vaak in combinatie gebruikt met Node.js, waaronder Kayzr.

Request Handlers

Dankzij Express wordt het zeer gemakkelijk om routering in je applicatie toe te voegen. Indien een GET request gemaakt wordt in de frontend naar een bepaalde URL, is dit zeer gemakkelijk om op te vangen m.b.v. Express. Voor elke soort request en URL kan Express logica uitvoeren en code makkelijk verder delegeren.

Views

Express kan statische bestanden zoals HTML, CSS en afbeeldingen laten weergeven in de browser dankzij render engines. Makkelijk om bijvoorbeeld foutpagina's te weergeven.

Modulair

Modulariteit is een kerneigenschap van Express. Express is *unopinionated*, wat wil zeggen dat er niet echt een gouden regel is om een doel in Express te bereiken. Doelen kunnen op meerdere manieren bereikt worden, waardoor het makkelijker is voor ontwikkelaars om een bepaalde taak uit te voeren. Dankzij modulariteit kan het framework ook volledig ingesteld worden naar wens, van poorten en routers, het instellen van een databank tot het toevoegen van Express middleware.

Middleware

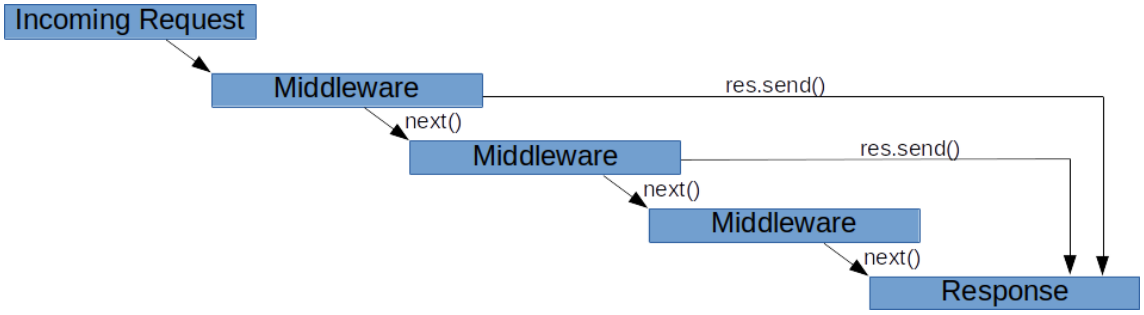
Wellicht de krachtigste functie van Express is de eigenschap tot het gebruik maken van middleware. Middleware wordt voortdurend gebruikt in Express. Wanneer een request binnenkomt, wordt deze doorgestuurd door nul, één of meerdere middleware die elk iets doen met de request, tot deze uiteindelijk wordt afgehandeld. De volgorde waarin een request de middleware doorloopt, is volledig afhankelijk van de programmeur. Net zoals Node.js, kent Express ook een enorm grote bibliotheek aan middleware die men kan gebruiken in een applicatie en is alsook helemaal verkrijgbaar via npm. Hierdoor kunnen ontwikkelaars zelf hun middleware schrijven, bestaande middleware downloaden en deze inpluggen in hun applicatie. Middleware kan gaan van Http-request loggers tot cookie-parsers en veel meer. De mogelijkheden zijn praktisch oneindig. Foutmeldingen zijn een goed voorbeeld van Express Middleware. Wanneer ergens iets fout gaat, roept Express de Error-Handler op, die dan zelf de request afwerkt. Monitoringstoepassingen in Node.js maken vaak gebruik van middleware. Hierover volgt meer info in het volgende hoofdstuk.

Listing 2.1: Express code flow voorbeeld.

```
1  var express = require('express');
2  var app = express();
3
4  // An example middleware function
5  var a_middleware_function = function(req, res, next) {
6  // ... perform some operations
7  // Call next() so Express will call the next middleware function in the
       chain.
8  next();
9  }
10
11 // Function added with use() for all routes and verbs
12 app.use(a_middleware_function);
13
14 // Function added with use() for a specific route
15 app.use('/someroute', a_middleware_function);
16
17 // A middleware function added for a specific HTTP verb and route
18 app.get('/', a_middleware_function);
```

```
19
20          app.listen(3000);
```

Middleware schrijven is op zich niet moeilijk. Men kan een enkele functie schrijven, of een heel takenpakket, die men dan als het ware injecteert in een request afhandeling. In een middleware functie verkrijgt men dan een req object, dat van de router kan komen of van een andere middleware, waar dan iets mee kan gedaan worden. Wanneer alles is uitgevoerd, roept de middleware de next-functie op, die dan het nieuwe res, of result, object doorstuurt naar de volgende middleware. Het kan ook zijn dat de middleware de next-functie niet oproept, wat betekent dat dit de laatste stop was van router.



Figuur 2.3: Schema van de werking van middleware, gebruikt van **middleware**

2.4 Testen en Monitoring

Testen schrijven. Of men nu alleen aan software werkt, of in team in een bedrijf, elke programmeur met een gezond verstand zal kunnen vertellen over het belang van testen schrijven. Iets waar vaak wordt tegenop gekeken, maar het is nodig voor een een geslaagd werkend product op te leveren. Meer en meer bedrijven gaan dit dan ook verplichten in hun implementatieproces.

Testen en monitoring zijn twee termen dat dicht bij elkaar horen. Bijna elke programmeur kent dan ook het belang van testen schrijven. D.m.v. testen te implementeren kan men kwaliteitsvoller code schrijven, fouten sneller opsporen en is de software een pak robuuster. Het toevoegen van nieuwe functionaliteiten kan code niet meer breken en daardoor kan software langer meegaan.

2.4.1 Agile Testing

Agile testing is een methodologie, ontstaan rond 1990, waarin software incrementeel en iteratief wordt uitgewerkt en dat op de dag van vandaag in vele bedrijven wordt gehanteerd. Vooraleer er effectieve code wordt geschreven, zullen er eerst testen gemaakt worden, waarop men dan de code gaat schrijven. Zo verkrijgt men code die robuust, aanpasbaar en efficiënt is. Tijdens elke iteratie van het projectproces, zullen er nieuwe testen worden geschreven, oude testen worden aangepast, en zo hoeft men telkens maar kleine gedeeltes

code aan te passen of te schrijven. Een goede handhaving van Agile testing doet niet alleen de slaagkansen van het project vergroten, het kan het project ook goedkoper doen uitdraaien en de effectieve loopduur stabiel en klein houden. Per sprint, of ontwikkelcyclus, worden er telkens eerst testen geschreven voordat men dan uiteindelijk begint met programmeren. (**CHAKRAVORTY2014536**)

Dit kan echter ook problemen met zich meebrengen. Agile vergt voorbereiding. **CHAKRAVORTY2014536** vertelt ons dat de kost en tijd van het project kunnen uit balans geraken telkens de klant iets wilt veranderen en daardoor komt er veel werk op de programmeurs te liggen. In (**Borland2012**) wordt dit ook nog eens samengevat. Veel bedrijven doen aan de Agile methodologie, maar niet aan Agile Testing. Dit komt omdat in Agile de verantwoordelijkheid om beslissingen te nemen naar het team van ontwikkelaars wordt verschoven. Dit breekt met de traditionele manier van de waterval methode waar de belangrijkste keuzes door de leiding en niet door het team worden genomen. Dit breekt ook met de traditionele hiërarische structuur waarin bedrijven werken. De leiding verliest een deel van de controle over de richting van hun projecten. Het is dus veilig om te zeggen dat het implementeren van de Agile methodiek niet alleen het werkproces van het bedrijf aanpast maar ook de mentaliteit van het bedrijf. Veranderingen in een bedrijf gaan log vooruit en zeker als het een gehele mentaliteitsverandering inhoud. Daarom dat er veel bedrijven zijn die er voor opteren om de Agile methodiek gedeeltelijk te implementeren. Zo laten ze een paar minder belangrijke componenten van de Agile methodiek achterwege zoals bijvoorbeeld Agile testing. Dit kan verklaren waarom er niet zo veel bedrijven zijn die Agile testing implementeren. Een andere grote hindernis bij Agile testing is dat er niet veel tools bestaan die geavanceerde testing methodes ondersteunen.

Wat men ook in de afgelopen tien jaar meer en meer terugziet is de problematiek van het continue releasen van software. Software-reuzen zoals Google, Facebook, Mozilla, enzovoort... kozen om software telkens opnieuw uit te brengen in zeer korte periodes, ook wel *rapid releases* genoemd. Mozilla kende bijvoorbeeld een release-cyclus van maanden, waarin elke grote versie vele nieuwe dingen met zich meebracht. Vanaf Firefox 5.0 zijn ze overgeschakeld naar een rapid release systeem en daardoor brengen ze nieuwe, maar kleinere updates om de 6 weken uit. Dit heeft zijn voordelen, maar kent echter ook zijn nadelen in het test-proces. Aangezien er meer en meer sneller moet uitgebracht worden, moeten testen meer geautomatiseerd worden, wordt er minder getest en is de software minder robuust doordat fouten sneller tevoorschijn komen (**Maentylae2013**). Kayzr kampt met dezelfde problemen. Daarom is er meer en meer nood aan monitoringstoepassingen die de duur van het debugproces van zo'n rapid release model aanzienlijk kunnen doen verminderen. Hierover volgt later meer info.

2.4.2 Soorten Agile Testing

Er bestaan natuurlijk verschillende manieren om te testen, maar het wordt grotendeels opgesplitst in drie categorieën, namelijk unit testing, integratie testing en systeem testing, maar er zijn er nog anderen. Sanity testing, smoke testing, interface testing, regression testing, end to end testing, performantie testing,... zijn allerlei soorten opgesomd door **Pittet**. Dit zijn slechts de functionele testen. Men heeft ook niet-functionele testen, die

dan betrokken zijn tot andere aspecten van het project, zoals beveiliging en lokalisatie (taal),... Dan wordt er nog eens gekozen of men deze soorten testen wenst te automatiseren of niet. Daarin kruipt meer tijd, maar goed geschreven geautomatiseerde testen zijn een sleutelcomponent tot continue integratie en releases. Hier wordt echter niet verder in detail op ingegaan aangezien dit niet tot de scope van het onderzoek behoort.

2.5 Monitoring

Monitoring is een proces om info en metriek te analyseren van een lopend proces. Zo kan monitoringssoftware er op letten dat alles verloopt volgens de afspraken, en indien dit niet het geval is, actie ondernemen (**Rouse2018**).

2.5.1 Hoe werkt monitoring?

Real-time monitoring is een monitoringstechniek die dagelijks gebruikt wordt in de IT-wereld. Zo'n proces kan data verzamelen door voortdurend checks te doen op de lopende software, zoals het opvragen van CPU-verbruik, geheugen-verbruik, foutmeldingen controleren,... Monitoringsoftware kan gebruik maken van *agenten*, die dan specifiek geprogrammeerd kunnen worden om op een intelligente manier zaken te onderzoeken en desnoods af te handelen (bijvoorbeeld: een specifieke fout die opduikt).

Dankzij monitoring kan een applicatie draaiende gehouden worden, of kan een IT-medewerker tijdig opgeroepen worden om fouten op te sporen. Monitoring kent ook andere voordelen, zoals het verzamelen van data die dan gebruikt kan worden voor andere doeleinden, bijvoorbeeld voor marketing - of managementafdeling.

```
vb:  Stuur telkens een bericht naar de Administrator wanneer een IP-adres
uit Amerika onze site bezoekt, en visualiseer deze op een kaart.
```

2.5.2 Monitoring, Node.js en Express

In de vorige hoofdstukken werd de kracht aangehaald van Node.js's npm package manager en Express's middleware. Monitoringssoftware kan effectief worden geïnjecteerd in Express als middleware, wat ons zeer veel mogelijkheden schenkt. Zo kan van elke API Call enorm veel info worden opgevraagd, zoals foutboodschappen, geolocatie, cpu-verbruik en zoveel meer, wat opnieuw bewijst dat middleware van Express en het asynchrone gedrag van Node.js zeer krachtige functionaliteiten zijn. Er bestaan al enkele monitoringsmiddleware voor Express, maar geen software dat aan de vereisten voldoet van Kayzr. Hiervoor wordt in het volgend hoofdstuk nog wat onderzoek naar gedaan, maar we sommen alvast enkele populaire packages en standalone software op.

2.5.3 Tools

Hier worden enkele middleware en standalone tools beschreven. Alle info hieronder beschreven is terug te vinden op npm of op de respectievelijke website van de tool. Tools die werden weggelaten zijn tools die onrelevant zijn voor Kayzr, niet meer ondersteund worden in de voorbije twee jaar of niet populair genoeg zijn om een goede ondersteuning aan te bieden.

Morgan

Morgan is een http request logger middleware voor Express en Node.js, en wordt gebruikt om details van een request object te loggen. Morgan komt standaard gratis meegeleverd met Express, en is daarom één van de populairdere tools. Het is niet krachtig, maar eenvoudig en efficiënt om snel enkele kleine zaken te weergeven aan de administrator.

Enkele voorbeelden van logbare zaken zijn.

- Remote Address
- Remote User
- HTTP Method (GET, POST,...)
- URL
- Statuscode
- Request header
- Result header
- Response-time
- ...

Express-Status-Monitor

Een eenvoudige monitor om simpele metingen van een Node.js server gemaakt met Express weer te geven. Deze tool analyseert onderstaande data en kan deze in mooie lijngrafieken visualiseren. Deze tool is gratis te downloaden via npm.

- CPU-verbruik
- Geheugen-verbruik
- Response-time
- Requests per seconde
- Load average
- Status codes

Prometheus

Prometheus is een open-source, gratis monitor voor Node.js dat uitblinkt in data-compressie en het snel opvragen van tijd-series data. Deze tool is zeer handig voor statistische analyse van een Node.js server en diens metingen. Ook is deze tool in staat om ontwikkelaars te verwittigen bij het bereiken van een bepaalde ingestelde limiet (zoals bijvoorbeeld het

oververhitten van een processorkern na een bepaalde tijdsperiode). Het is een krachtigere versie van Express-Status-Monitor, maar wordt voor andere doeleinden gebruikt. Indien statistische analyses gemaakt willen worden, is dit één van de go-to tools, al zit er een grote leercurve achter.

Bedrijven die gebruik maken van Prometheus zijn onder andere DigitalOcean, Docker, Soundcloud, Argus en veel meer.

New Relic

New Relic is een betalende monitoringssoftware gemaakt door New Relinc, Inc. Het ondersteunt Node.js servers maar kan ook gebruikt worden voor Ruby, Java, PHP, .NET, Python en Go. Ook is de ondersteuning met andere frameworks zeer uitgebreid, aangezien het databanken en platformen zoals mongoDB, Redis, MySQL, Express, Http, KrakenJs en nog veel meer ondersteunt.

New Relic ondersteunt de basisfunctionaliteiten van Express-Status-Monitor en Morgan, breidt deze wat meer uit en ondersteunt volgende zaken bovenop:

- Service Maps geeft een mooi dashboard-overzicht van de server
- Fouten opsporen tot op de exacte lijn van ontstaan en hulpmiddelen om deze fouten te vinden
- Database-monitoring en meer
- Applicatie-monitoring en meer
- DevOps Team Collaboratie
- ...

We gaan niet te diep in op de functionaliteiten van New Relic, aangezien dit niet binnen de behoeften ligt van Kayzr. New Relic kent enorm veel functionaliteiten, maar daar hangt dan ook een zeer hoog prijskaartje aan. Op het moment van schrijven betaalt men per server maandelijks tussen de **€17.00** en **€600.00.** Aangezien Kayzr ongeveer 50 verschillende servers heeft, is dit onbetaalbaar. Ook bevat dit veel functionaliteiten dat ze niet nodig hebben, wat het de investering niet waard maakt.

Bedrijven die New Relic gebruiken zijn Mlbam, Ryanair, Hearst, REI, Condé Nast...

Retrace

Retrace, een monitoringsoplossing van Stackify, probeert zich te onderscheiden van de andere oplossingen door meer features aan te bieden in één strakke applicatie dat een heel stuk goedkoper is dan de concurrentie. Naast Node.js ondersteunen ze ook .NET, PHP, Ruby en Java. Hun top-features zijn als volgt:

- App Performance Monitoring, een krachtige tool dat trage hindernissen van een applicatie kan opsporen, zoals sql queries, dependencies, requests,...
- Centralized Logging: Alle logs van verschillende frameworks zijn beschikbaar op één plek.

- Geavanceerde Error Tracking zorgt ervoor dat fouten opsporen eenvoudiger wordt, dankzij gerelateerde logs, exception rates, identificatietools en meer.
- Dankzij Code Profiling kan op een lichte manier gekeken worden naar wat een applicatie aan het doen is op elk moment. Dit is waar Retrace in uitblinkt, vanwaar de naam ook vandaan kom, aangezien alles op te zoeken valt van code dat wordt uitgevoerd.
- Ondersteuning voor Server Metingen, waardoor alle servers en applicaties tegelijkertijd in de oog gehouden kunnen worden. Ook ondersteunt het notificaties voor wanneer er fouten of waarschuwingen zouden voorkomen.

Retrace kan ook geïntegreerd worden met vele bestaande tools waaronder Jira en Slack, waar Kayzr enorm gebruik van maakt. Op het eerste zicht lijkt Retrace een goede oplossing, maar daar wordt later nog op teruggekomen. Retrace kent een prijskaartje van **€50** per maand, al kan dit voor kleine en preproductie servers verlaagd worden naar **€10-€25**. Dit komt een pak goedkoper uit dan New Relic, maar kan afhankelijk van het aantal servers nog steeds prijzig uitkomen voor een start-up. Retrace is echter relatief nieuw, sinds 2017, en wordt nog niet gebruikt bij zeer grote bedrijven.

Dynatrace

Dynatrace van Dynatrace LLC is opnieuw een enorm grote tool met een uitgebreid pakket van hulpmiddelen voor een server. Dynatrace blinkt uit in het ondersteunen van wel liefst meer dan 130 technologieën en 63 integraties, en wordt regelmatig uitgebreid. Ze ondersteunen technologieën van databases en cloud infrastructuren tot webtechnologieën en veel meer. Enkele functies:

- Een enorm sterke User Interface met een zeer mooi en krachtig dashboard.
- *Alle* soorten performantiemetingen bekijken in real-time en meer. Zelfs de Node.js event loops.
- Node.js details bekijken zoals Heap Memory, Garbage collection time, web requests, response-time, crashes, throughput en veel meer.
- Dynatrace is in staat om problemen op te sporen tot code level, rekening houdend met gemonitorde heap en geheugen metingen en tal van andere middelen. Het kan zelfs fouten opsporen dat niet afkomstig zijn van Node.js
- Handige visualisatiemiddelen om dependencies en applicaties overzichtelijk te houden
- Database Queries monitoring
- Enorm veel meer functies voor de gehele IT-infrastructuur

Deze software wordt gebruikt door Adobe, Samsung, Ebay, Experian en meer, wat meteen grote klanten zijn. Een van de grootste spelers, maar opnieuw met een veel te groot prijskaartje van €216 per server per maand.

PM2

PM2 is één van de populairste tools in Node.js dat gratis gebruikt kan worden om de server te monitoren en meer. Beschikbaar via npm, is dit binnen de minuut geïnstalleerd. In tegenstelling tot de vorige betalende software, is PM2 speciaal ontwikkelt door de Node Community voor Node.js. PM2 kent de volgende features:

- Watch and Reload
- Log management
- Max memory reload
- Startup Scripts
- **monitoring**
- Cluster Mode
- Hot reload
- Keymetrics monitoring
- Procesbeheer
- ...

PM2 kent vele functies, maar kent ook een betalende PM2 Plus versie. Deze versie kost **€79** per server per maand, wat ook al meteen buiten de prijsklasse valt van Kayzr, maar is echter speciaal gemaakt voor monitoring. Het enige nadeel aan de gratis versie van PM2, is dat deze voor Kayzr net niet genoeg functies bevat. Realtime logs, exception tracking en histogrammen van de data maakt het meteen een enorm duur opstapje.

2.6 Kayzr's Probleem

Kayzr's probleem geldt als volgt: Kayzr's data staat op verschillende servers verspreid, en wenst deze data te centraliseren. Zo hoeven ze niet voor sommige zaken naar Google App Engine te gaan, andere naar Google Cloud Service, andere zaken in Morgan, of simpelweg de Node Console, enzovoort. Dit zou het best kunnen opgelost worden via een Express middleware dat alle data verzamelt, van geolocatie tot foutmeldingen, en omzet in .json formaat. Deze .json zou dan kunnen uitgelezen worden om zo een mooie visualisatie van de opgevangen data te hebben. Men zou het kunnen aanzien als een enorm versimpelde versie van alle voorgaande betalende monitoring tools.

Monitoringssoftware is te duur en veel te uitgebreid voor hen, maar de gratis bestaande tools op npm zijn te primitief, dus hebben ze de opdracht gegeven om iets te ontwikkelen dat het debugproces aanzienlijk zou kunnen verhogen.

2.6.1 Effectieve requirements

Hieronder worden de effectieve requirements, na overleg met Kayzr, weergeven.

- Hoeveel keer wordt een bepaalde API call uitgevoerd in een specifieke tijdspanne? Kan Kayzr hierdoor bekijken welke API calls tijdens drukke uren te veel worden

opgeroepen en de applicatie vertragen?
- In Google Cloud Stack Driver kan men mooi de errors terugvinden die voorkwamen in de backend server. Maar hier staat nergens de URL van de opgeroepen API call bij. Waar is deze fout beginnen optreden? Kan men die URL verkrijgen opdat men niet moet zoeken naar een naald in een hooiberg?
- Kunnen deze fouten ergens opgeslagen worden opdat men later deze fouten makkelijker kan terugvinden?
- Kan men de (gemiddelde) tijd monitoren tussen het versturen van een bepaalde API call en een verkregen antwoord?
- Monitoren van deze Node.js processen hun taxatie op de server waar ze op draaien (CPU, RAM, netwerk,. . .)
- Middleware als Morgan toont veel te weinig, Google Cloud Stack Driver toont veel maar geeft geen context mee. De voorgaande opgesomde betalende monitoringsoftware bevatten te veel functies die Kayzr niet meteen zou gebruiken, maar wel handig *zou kunnen* zijn. Dit verantwoordt echter het grote bedrag niet, en wenst dus deze som geldt er niet aan te spenderen. Bestaat er geen goede middenweg?
- Afhankelijk zijn van betalende services betekent ook dat Kayzr's data terecht komt bij (dure) third party oplossingen. Is die data veilig? Volgen ze de GDPR regels? Ze hebben liever hun data in eigen handen.
- Men wil niet enkel realtime zaken kunnen bekijken, maar ook data opslaan zodat die op een later moment nog eens bekeken kan worden.

Doel testen: Is de tool een meerwaarde, is er meer data uit te halen, reqs bereikt hierboven

Kayzr wenst dus een middleware oplossing, gratis te downloaden via npm, dat uit binnenkomende verzoeken praktisch alle informatie kan halen. Deze informatie moet vervolgens ergens opgeslagen worden en op een visuele manier ook bekeken kunnen worden.

Men kan vervolgens controleren of het doel bereikt is door te kijken of de middleware een meerwaarde is voor het ontwikkelingsteam, er meer data is uit te halen dan vóór de verkregen middleware, en aan alle voorgaande requirements is voldaan.

3. Methodologie

Nu de vereisten van Kayzr gekend zijn, kan het ontwerpen van de middleware toepassing van start gaan. Hiervoor zal er een bepaalde werkwijze worden gehanteerd, waarover zo dadelijk meer uitleg wordt gegeven. Tijdens het ontwerpen van de middleware zal echter niet enkel met Kayzr's requirements rekening gehouden worden. Het is de bedoeling dat dit geen hardcoded stuk software in Kayzr's backend wordt, maar een abstracte, universele middleware dat gedownload en geïnstalleerd kan worden via Node.js package manager.

3.1 Werkwijze

Men is als volgt te werk gegaan: eerst werd een middleware ontwikkeld dat op een mooie manier elke request kon loggen in de console. Erna werd een server aangemaakt op één van de servers van Kayzr, en werd daar Grafana en InfluxDB op geïnstalleerd. Tenslotte werd de middleware geabstraheerd, waarna er getracht werd om connectie te maken met de databank en telkens na een bepaald interval, data naar InfluxDB weg te schrijven. Wanneer de data van de middleware tenslotte Grafana bereikte, werd de middleware geïnstalleerd op Kayzr's backend. Op het einde werden er dan grafieken aangemaakt in Grafana die de requirements van Kayzr zouden volbrengen.

3.1.1 Multilogger

De te ontwikkelen middleware werd omgedoopt tot Multilogger (en later officieel express-influx-multilogger). Als eerste werd een basis express applicatie aangemaakt, waarin de eerste stapjes middleware software werden geschreven. Als eerste werd er geprobeerd

om per keer dat een API call werd gemaakt, de methode van deze call naar de console te loggen.

In app.js

Listing 3.1: app.js eerste stap

```
var app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static(path.join(__dirname, 'public')));

app.use(multilogger.multilog); // Custom middleware

app.use('/', indexRouter);

app.listen(3000);
```

In multilogger.js

Listing 3.2: multilogger.js eerste stap

```
const multilogger = {
        multilog: (req, res, next) => {
        console.log('\n=====- Multilogger v0.1 -=====');
        console.log('--- Basic ---\n');

        res.on('finish', () => {
                console.info(`${req.method} --- ${res.statusCode} ---  ${res.
                    statusMessage}  at ${new Date().toLocaleString()}`);
                console.info(`Response-time: ${res.getHeader('X-Response-Time')}`);
                console.info(`URL: ${req.hostname} --- ${req.url}`);
                console.info(`Client: ${req.ip} --- ${req.header('User-Agent')}`);
        });


        next();
        },
};

module.exports = multilogger;
```

Dit werd uiteindelijk meer uitgebreid, zodat er een mooi breed overzicht naar de console werd gelogd. Natuurlijk wensen niet alle ontwikkelaars dat er per API all naar de console zou worden gelogd met zeer veel overbodige informatie, aangezien dit kan oplopen van (uiteraard afhankelijk van de software) honderden tot duizenden calls per enkele seconden. Daarom werd er een parameter toegevoegd om deze functionaliteit uit te schakelen, met in het achterhoofd dat de middleware niet enkel zou dienen voor een mooi console-overzicht te geven. Dit was een nice-to-have-functionaliteit voor Kayzr, maar was wel een noodzakelijke eerste stap om op verder te bouwen. Ook werd er een aangepaste *error handler* (stuk code dat fouten afhandelt die zich voortdoen tijdens een API call) geschreven, waardoor ook foutberichten konden gelogd worden.

Listing 3.3: Parameters toegevoegd

```
app.use(multilogger.log({ development: false, extended: false }));
```

Listing 3.4: MultiError.js, de custom error handler

```
const throwMultilogError = () => {
        return (err, req, res, next) => {
                if (!err) {
                        return next();
                }
                res.locals.multiError = {
                        errorMessage: err.message,
                        errorStack: err.stack
                };
                next();
        };
};

module.exports = throwMultilogError;
```

3.1.2 Grafana en InfluxDB

Vervolgens moest er een systeem gekozen worden om verzamelde data op te slaan en weer te geven. Dit kan natuurlijk gerealiseerd worden met verschillende databanken en frontend frameworks. Origineel werd er geopteerd om gebruik te maken van MySQL samen met een React.js applicatie om de data te visualiseren, maar na verder onderzoek bleek dat hier reeds handigere software voor bestaat, namelijk InfluxDB, en Grafana, dixit **Hill2015**. InfluxDB is een snelle, open source time series database, gemaakt om op een snelle manier data zoals metrieken en events op te slaan, gebonden doorheen een tijdsperiode. Grafana daarentegen, is een open platform om time series data om te zetten in prachtige grafieken. Het spreekt dan ook voor zich dat deze twee hand in hand gaan.

Er werd een server van Kayzr toegewezen om dit te testen. Hiervoor werden via Docker twee containers aangemaakt (een soort van virtuele omgeving om deze software op te laten draaien), en werden tenslotte toegewezen aan InfluxDB en Grafana. Uiteindelijk werd de server opgestart, werd Grafana geïnstalleerd en geïnitialiseerd en werd de (lege) Influx-databank gekoppeld.

3.1.3 Wegschrijven en abstraheren

Nu dat de databank was aangemaakt, werd het tijd om deze te koppelen aan de middleware. Gebaseerd op het voorgaande logsysteem, werd een object aangemaakt dat alle belangrijke info zou bevatten.

Listing 3.5: Log object

```
const object = {
        method: req.method,
        statusCode: res.statusCode,
        statusMessage: res.statusMessage,
        date: new Date().toUTCString(),
        responseTime: elapsedTimeInMs,
        contentType: req.header("Content-Type") || " ",
        hostname: req.hostname,
        url: req.url,
        path: res.statusCode !== 404 && req.route && req.route.path ? req.route.
            path : "No Path",
```

```
        body: req.method === "POST" ? realBody : " ",
        params: _.isEmpty(req.params) ? " " : JSON.stringify(req.params),
        query: _.isEmpty(req.query) ? " " : JSON.stringify(req.query),
        cookies: _.isEmpty(req.cookies) ? " " : JSON.stringify(req.cookies),
        auth: req.header("Authorization") || req.header("x-access-token") || " ",
        ip: req.connection.remoteAddress,
        location: location,
        clientInfo: req.header("User-Agent") || " ",
        memoryUsage: memoryUsage,
        cpuUsage: cpuUsage,
        errorMessage: res.locals.multiError || " "
};
```

Via een reeds bestaande npm package, namelijk *node influx*, werd dit snel opgelost. Er werd een schema ontwikkeld waaruit optimaal de data zou kunnen geformatteerd worden naar Grafana, en niet veel later vloeiden de eerste metrieken Grafana binnen. Tenslotte werd er een buffer geïmplementeerd, dat om een gegeven interval de opgeslagen data wegschrijft en zichzelf weer leegt voor het volgende interval.

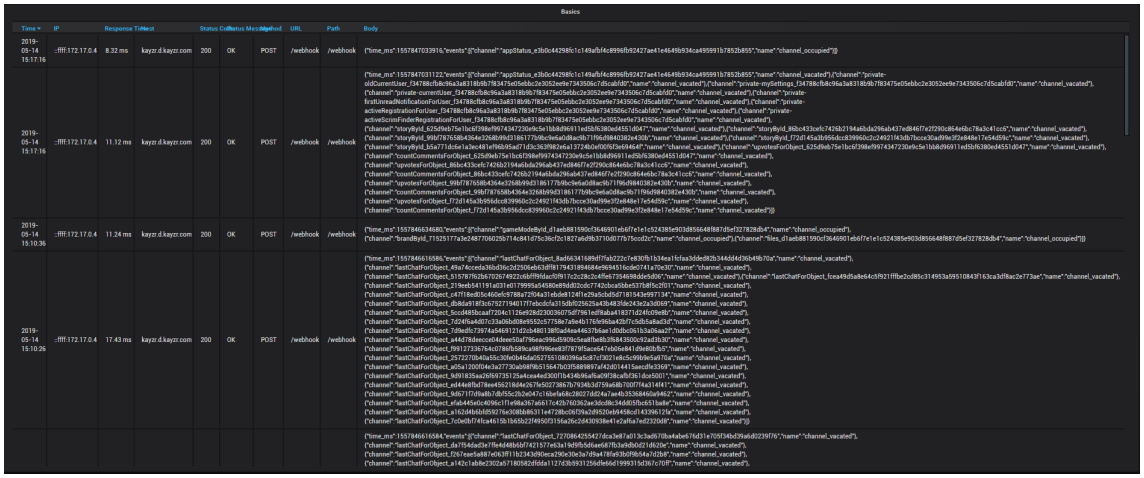
De volgende stap in het ontwikkelen van een middleware oplossing voor Kayzr, was het abstraheren van de code. Zo kan niet enkel Kayzr hiervan gebruik maken, maar echter iedereen met een Node.js en Express configuratie. Er kwamen wat moeilijkheden opduiken bij het loskoppelen van de code, maar uiteindelijk werd er een abstracte, logische en schaalbare middleware geschreven volgens de richtlijnen van npm. Via parameters kan een gebruiker de functionaliteiten van de middleware aanpassen, en er werd ook ruimte voorzien om later extra database management systemen toe te voegen. De middleware werd omgedoopt tot express-influx-multilogger, werd open source gemaakt en werd uiteindelijk op npm gepubliceerd. Als laatste werd deze gloednieuwe package dan binnengetrokken op de backend en frontend van Kayzr's webplatform. Zo geraakte influx op een snellere manier gevuld met relevante data zodat er begonnen kon worden met het ontwikkelen van de volgende stap.

3.1.4 Grafana's grafieken

Als allerlaatste, werd Grafana opgebouwd om mooie relevante grafieken weer te geven aan de eindgebruiker. De verzamelde data bleek niet altijd even relevant of goed genoeg te zijn om er een grafiek mee op te bouwen, dus was er regelmatig een noodzaak om weer naar de vorige stap terug te gaan en het schema van InfluxDB aan te herzien. Hierdoor kwamen ook nieuwe ideeën tot stand, zoals het toevoegen van een gebruiker zijn geolocatie en het monitoren van de snelheid van elk databank verzoek per API call. Op het moment van schrijven zijn er in Grafana 7 grafieken te bezichtigen.

Basics

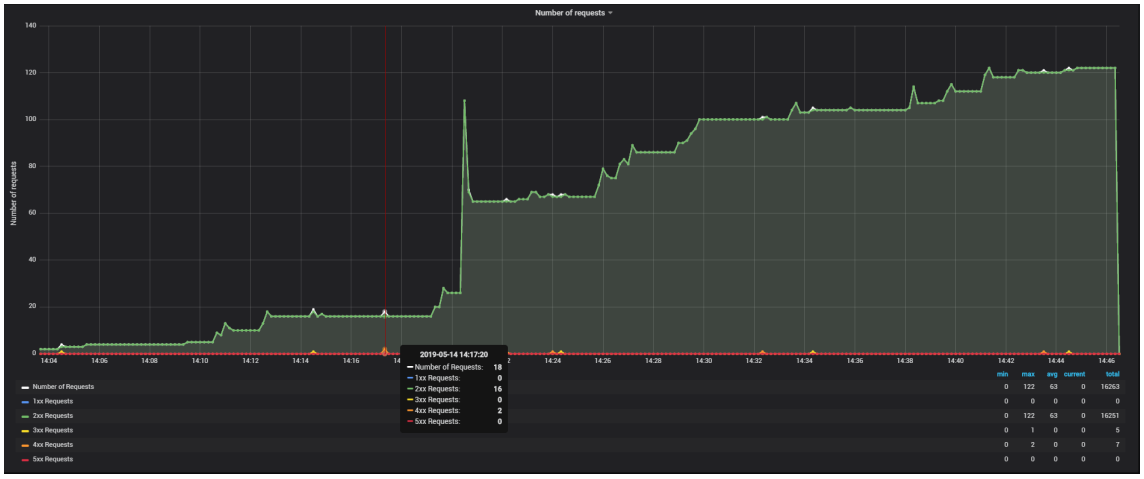
Hier worden de basis headers van elke API call getoond. De data bestaat uit een timestamp, ip-adres, response time, host, statuscode, statusbericht, methode, url, pad, en body.



Figuur 3.1: Basics-paneel

Number of requests

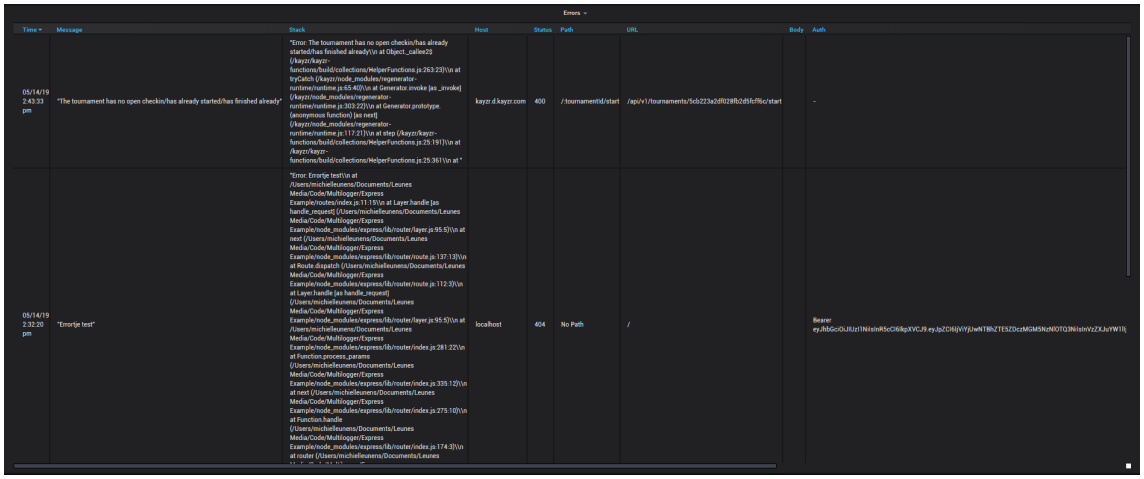
Het aantal verzoeken in een gegeven tijdsframe. Ook kunnen deze opgesplitst worden per statuscode, zodat men bijvoorbeeld kan observeren hoeveel API calls een fout gaven.



Figuur 3.2: Aantal requests-paneel (wit voor totaal, blauw voor requests dat starten met 1xx, groen voor 2xx, geel voor 3xx, oranje voor 4xx en rood voor 5xx.)

Errors

Hier worden de foutmeldingen getoond, gegroepeerd per unieke foutmelding. Naast een timestamp, foutmelding en de error stack, worden ook de host, status, url en path weergeven. Aangezien Kayzr dit ook wou gebruiken voor debug-doeleinden, worden ook de body en authenticatietokens weergeven.



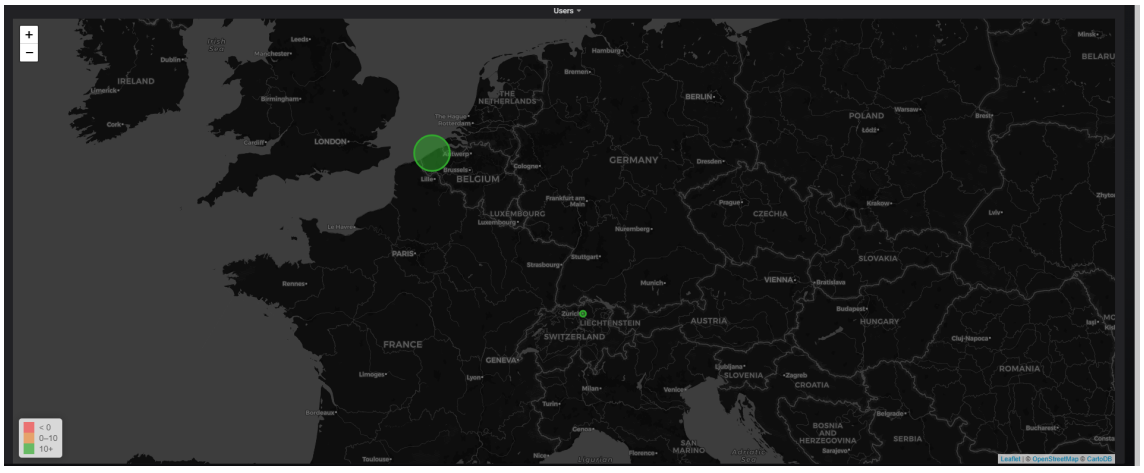
Figuur 3.3: Errors-paneel

Database timings

Hier wordt per API call de tijden gelogd van een bepaald verzoek naar de databank. Zo kan men per call zien hoeveel tijd het vraag om specifieke data op te halen.

Users

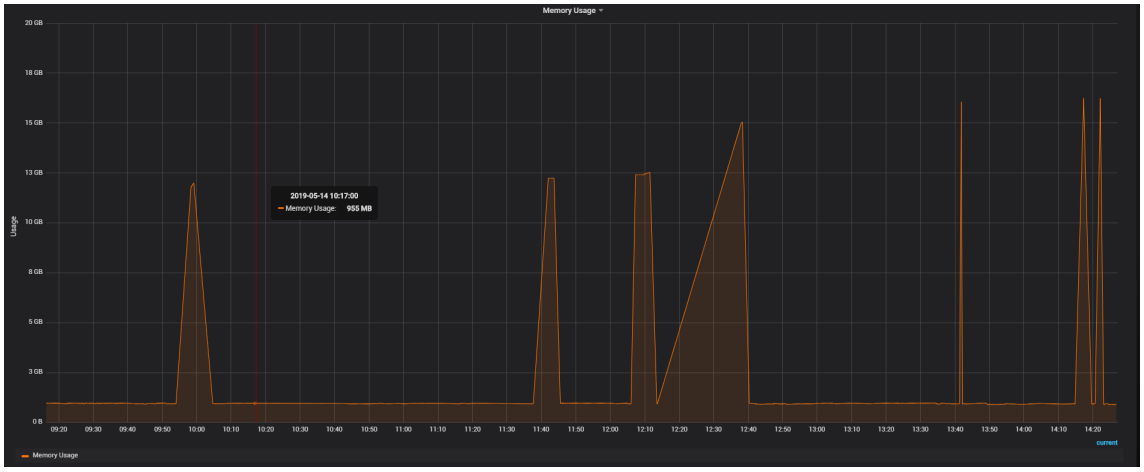
Op een wereldkaart wordt het aantal gebruikers per land getoond. De geolocatie wordt afgeleid van het ip-adres van de gebruiker, en specifieke locaties worden dus niet opgevraagd.



Figuur 3.4: Users-paneel

Memory Usage

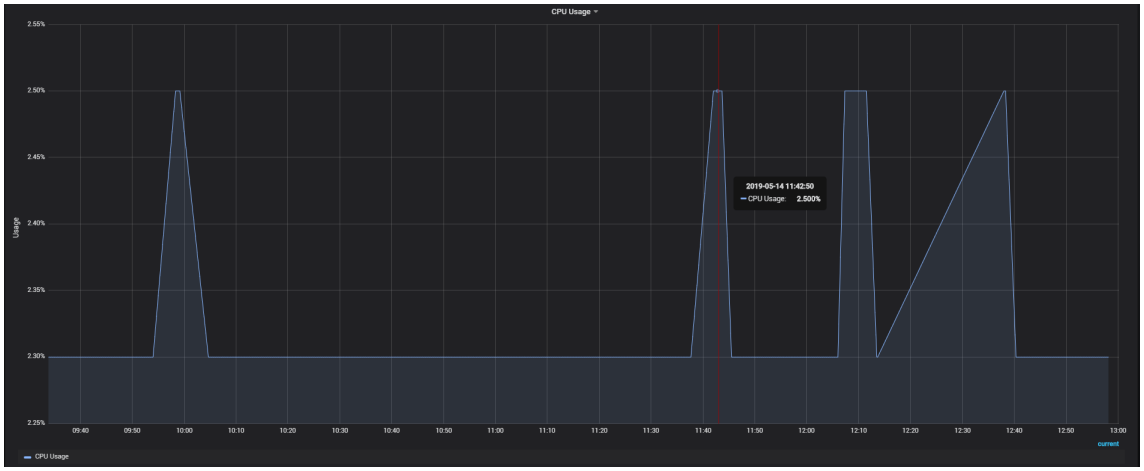
Het geheugenverbruik per gigabyte van de machine waarop de server draait.



Figuur 3.5: Geheugenverbruik-paneel

CPU Usage

Het processorverbruik in percentages van de machine waarop de server draait.



Figuur 3.6: Processorverbruik-paneel

3.2 Publicatie

De package is ondertussen gepubliceerd op npm onder de naam Express-influx-multilogger , alsook op github. Installatie en instructies kunnen gevonden worden in de README.md.

4. Conclusie

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introductie

Node.js is een backend Javascript framework wiens populariteit in de afgelopen jaren hard is toegenomen. Ontwikkelaars genieten van verschillende voordelen. Het werkt asynchroon, het is makkelijk schaalbaar en het is zeer functioneel. Doordat het Javascript is, kan elk besturingssysteem gebruik maken van de krachtige functies die Node.js te bieden heeft. Dit maakt het een uitstekend framework om webapplicaties te ontwikkelen. Node.js is echter niet makkelijk om te debuggen doordat het asynchroon is opgebouwd. Het toepassen van de juiste monitortechnieken kan de slaagkansen van een project echter goed verhogen, net als de levensduur van de applicatie. Op welke manieren kunnen we het monitoren van zulke applicaties aanpakken? Welke software en tools worden hiervoor gebruikt? Welke technieken worden het best toegepast? En kan er ook intern in het proces van Node.js gekeken worden om daaruit nuttige informatie te halen? En zijn al deze technieken drastisch veranderd sinds het framework werd uitgebracht in maart 2009?

A.2 State-of-the-art

Node.js is reeds een matuur framework. Zoals gezegd wordt in (**Runtime2017**): Node.js is sterk afhankelijk van asynchrone en continue programmeerstijl. I/O-bewerkingen worden uitgevoerd door middel van oproepen naar asynchrone functies waarbij een callback

moet worden doorgegeven om aan te geven hoe de berekening wordt voortgezet zodra de genoemde I/O-bewerking asynchroon is voltooid. Het Node.js executiemodel bestaat uit een hoofdgebeurtenislus die wordt uitgevoerd op een single-threaded proces. Het is daarom niet makkelijk om deze soort frameworks te debuggen, en wordt uiteindelijk een uitdagende taak. Gelukkig zijn hiervoor dan ook weer verschillende hulpmiddelen en tools uitgebracht om dit te vereenvoudigen. Zoals hierboven vermeld in (**Runtime2017**), kan asynchrone code subtiele bugs opleveren die niet meteen zichtbaar zijn. Hier is nog niet echt onderzoek over gedaan. Waarin er honderden vergelijkende studies bestaan over de frameworks zelf en of Node.js een goede optie is, zijn er geen of amper studies over de beste manier om dit te debuggen en hoe men dit het best aanpakt.. (**Runtime2017**) vertelt ons meer over het identificeren van schaalbaarheidsproblemen en het aanrijken van mogelijke oplossingen. Ze maken gebruik van parametrische uitdrukkingen voor runtime monitoring van Node.js toepassingen, maar ze geven toe dat dit nog maar de eerste stap is en dat hier nog meer onderzoek naar kan gedaan worden. Doordat hier nog niet veel over staat neergepend, leek me dit een zeer interessant onderwerp dat hand in hand gaat met mijn stageopdracht. Kayzr, mijn stagebedrijf, zoekt namelijk zelf goede manieren om hun Node.js applicaties goed te kunnen debuggen, en deze studie zou zeker een goede bijdrage kunnen zijn aan deze soort doelgroep. Mijn onderzoek zal zich onderscheiden door een goed overzicht te behouden in die ongedocumenteerde zee van beschikbare frameworks, tools en software, waardoor deze studie kan gebruikt worden als een guideline voor best-practices.

A.3 Methodologie

We starten door literatuuronderzoek te doen naar de verschillende mogelijkheden van software en tools. Ook kunnen we literatuuronderzoek doen naar Javascript van ES1 naar ES6, NodeJS en zijn asynchrone processen, monitoring en het monitoren van async processen. Na de literatuurstudie maken we een steekproef van een aantal node-developers door hen te contacteren en via een enquete hen te vragen welke tools zij gebruiken om hun Node.js applicaties te monitoren, uitgebracht tussen 2009 en 2019. We kunnen erna kijken welke tools performanter zijn dan anderen dankzij het gebruik van de servers van Kayzr. We kunnen er uitgebreider onderzoeken door:

- Monitoren van api calls volgens het aantal keren opgeroepen
- Monitoren van api calls volgens duratie tot een response gestuurd wordt
- Het gemak om errors op te slaan en later te debuggen/analyseren
- Monitoren van deze node process en hun taxatie op de server waar ze op draaien (CPU, RAM, netwerk. . .)

De tools voor de bovenstaande metingen te testen gaan uiteraard de te onderzoeken tools, en hulpprogramma's als taakbeheer zijn. We maken daarna gebruik van R Studio om deze metingen en enquetes om te zetten tot mooie data waaruit we hopelijk een conclusie kunnen trekken.

A.4 Verwachte resultaten

Ik vermoed dat de resultaten uiteen zullen lopen, en dat verschillende developers zich liever vasthouden aan hun eigen methodes. Ook zal er een breuk zijn tussen ontwikkelaars die liever oudere maar matuurdere tools zullen blijven gebruiken, en developers die liever het nieuwste van het nieuwste wensen te gebruiken. Dit kunnen we dan visualiseren a.d.h.v. een staafdiagram met op de x-as deaww tool en op de y-as het aantal developers.

Qua effectieve data van de overwogen tools zal elk waarschijnlijk zijn voor en nadelen hebben. Toch vermoed ik dat we er een effectieve winnaar gaan uit kunnen halen die over het algemeen goed scoort. Dit zullen we dan toetsen aan de literatuurstudie om zo best practices te bekomen.

A.5 Verwachte conclusies

De wereld van Javascript is nog in volle groei, maar is toch al een pak matuurder dan vroeger. We zien dat de tools beter zijn geworden. De concurrentie is enorm groot aangezien Node.js een enorm populair ontwikkelaarsplatform is. We verwachten dat nieuwere tools meer functionaliteiten gaan bieden maar minder stabiel gaan zijn dan de reeds bestaande. De manier van aanpak zal variëren, maar uit de steekproef kunnen we dat toch veralgemenen.