HO
GENT

Faculteit Bedrijf en Organisatie

Monitoringmogelijkheden 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

Monitoringmogelijkheden 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 een project in Node.js 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 zijn altijd paraat staan, snel mijn vragen te willen 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 mijn vaardigheden als developer *en* als fotograaf enorm hebben kunnen groeien. Zonder hen had ik nooit deze bachelorproef kunnen verwezenlijken. De vrijheid die ik hier kreeg, apprecieerde ik dan ook ten zeerste. Maar, dat zou vooral niet gelukt zijn zonder de hulp van mijn copromotor, Michiel Cuvelier, aan wie ik honderden technische vragen kon stellen en die hij stuk voor stuk uitvoerig beantwoordde.

Het voelt enorm overweldigend aan om een eerste open-source tool gepubliceerd te hebben op npm en GIthub. Ik ben dan ook zeer trots op het resultaat, en hoop dat ik met de hulp van andere ontwikkelaars deze tool verder kan blijven ontwikkelen waardoor meerdere gebruikers hier van genieten.

Als laatste wil ik ook mijn vader bedanken voor zijn taal-en communicatieadvies.

Samenvatting

Dit onderzoek dient om na te gaan of er in 2019 goede methodes en/of alternatieven zijn om processen van Node.js en Express applicaties te debuggen. Dit werk is in opdracht van het bedrijf Kayzr, dat nood heeft aan een verbeterd debug-proces. Met een beperkt budget zijn bestaande tools ofwel onbetaalbaar, ofwel veel te primitief, en zou er onderzoek moeten gedaan worden naar een goed alternatief dat het beste van beide werelden combineert. Namelijk een betaalbaar pakket dat minimum moet voldoen aan een specifiek aantal functionaliteiten.

Deze proef doet onderzoek naar het ecosysteem van de huidige Javascript scène, en verschaft uitleg over diens meest populaire framework *Node.js*. De vereisten van Kayzr worden geanalyseerd en er wordt gekeken hoe deze ingevuld kunnen worden met de huidige hulpmiddelen en software. Vervolgens wordt er een stuk software geschreven dat de problematiek van Kayzr moet oplossen, waarna deze zal getoetst worden aan de praktijk.

Het resultaat kreeg een positieve ontvangst bij Kayzr en werd meteen in de productieomgeving gezet. De software kon ook worden geabstraheerd waardoor het mogelijk werd deze tool te delen met het wereldwijde web. Hierdoor kunnen andere bedrijven en ontwikkelaars hier ook van gebruik maken. De zelfgeschreven tool voldeed aan alle vereisten en er werden zelfs extra functionaliteiten toegevoegd.

Er kan worden geconcludeerd dat de productiesnelheid van Kayzr toenam, de efficiëntie van het debuggen verbeterde en de kostprijs niet steeg. Mede dankzij de open source natuur van de tool, kan deze tool verder onderzocht worden op zijn uitbreidingsmogelijkheden met nieuwe functionaliteiten, of de huidige versie verbeterd en geoptimaliseerd kan worden. Ook kan er nog gekeken worden of de hoge kostprijs van betalende oplossingen

verantwoord is door ze te vergelijken met deze gratis tool, en ook of de tool kan gebruikt worden in bedrijven met een ander businessplan dan Kayzr. Tenslotte kan er altijd gekeken worden of er nog betere debug-procesmiddelen bestaan aangezien het ecosysteem van het internet verandert met de dag.

Inhoudsopgave

Lijst van figuren

Lijst van codevoorbeelden

1. Inleiding

1.1 Probleemstelling

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

1.2 Onderzoeksvraag

De onderzoeksvraag is dus als volgt: bestaat er een mogelijkheid om de monitoringtechnieken van Kayzr, en mogelijk nog andere bedrijven, op een goedkope manier te stroomlijnen en te verbeteren binnen een Node.js omgeving, zodat hun debugproces geoptimaliseerd kan worden?

1.3 Onderzoeksdoelstelling

Het onderzoek is geslaagd wanneer Kayzr tevreden is met de toepassing en er in zijn project gebruik van gaat maken. De gemiddelde tijd om een probleem op te lossen moet dalen en het proces moet efficiënter verlopen. De data die per proces worden verzameld moeten op een goede manier kunnen gevisualiseerd worden zodat deze ook voor andere doeleinden kunnen worden gehanteerd. Er moeten meer data ontleed kunnen worden dan ervoor, en de software moet voldoen aan alle opgesomde requirements.

1.4 Opzet van deze bachelorproef

Verder in dit document bevinden zich de volgende onderdelen:

In Hoofdstuk 2 volgt een stand van zaken over JavaScript, Node.js, Express en Express middleware. Ook wordt er onderzoek gedaan naar verschillende monitoringsoftware en middleware die relevant zijn in 2019.

Erna wordt het huidige debug proces bestudeerd in hoofdstuk 3. Hoe verloopt dit? Wat kan er beter? Wat kan er sneller? Welke functies moet de nieuwe toepassing bevatten?

Hoofdstuk 4 omvat de strategie van dit onderzoek. Hier wordt er kort uitgelegd dat, op basis van de stand van zaken, tools zijn bestudeerd en samen met de probleemstelling een proof-of-concept zal uitgewerkt worden om te voldoen aan de vraag van Kayzr.

In Hoofdstuk 5 wordt de software geschreven volgens de requirements van Kayzr. Vervolgens wordt er nagegaan of die software daaraan voldoet, en kan er worden vergeleken met de oude processen.

In Hoofdstuk 6 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 het eigendom van dit softwareproject.

2. Stand van zaken

Dit hoofdstuk bestaat uit een 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, nadien bekijken we het belang van testen en monitoren van software en breiden we uit naar een analyse van dergelijke monitoringsoftware. 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 de 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 volgt. („What is JavaScript?",

2019)

2.1.1 Tijdlijn van JavaScript

JavaScript heeft een hele evolutie achter de rug. Ontstaan in mei 1995, wordt de taal na vele updates nog steeds dagelijks 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 (Netscape Navigator), om in hun browser een taal te implementeren 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 van een scripttaal uit in nog geen 10 dagen en overtuigde Netscape om die lichte, schaalbare en Java-complementerende taal te adopteren (Rangpariya, 2019). Javascript, toen onder de naam Mocha en vervolgens LiveScript, was geboren en zette de wereld van webontwikkelaars op zijn kop.

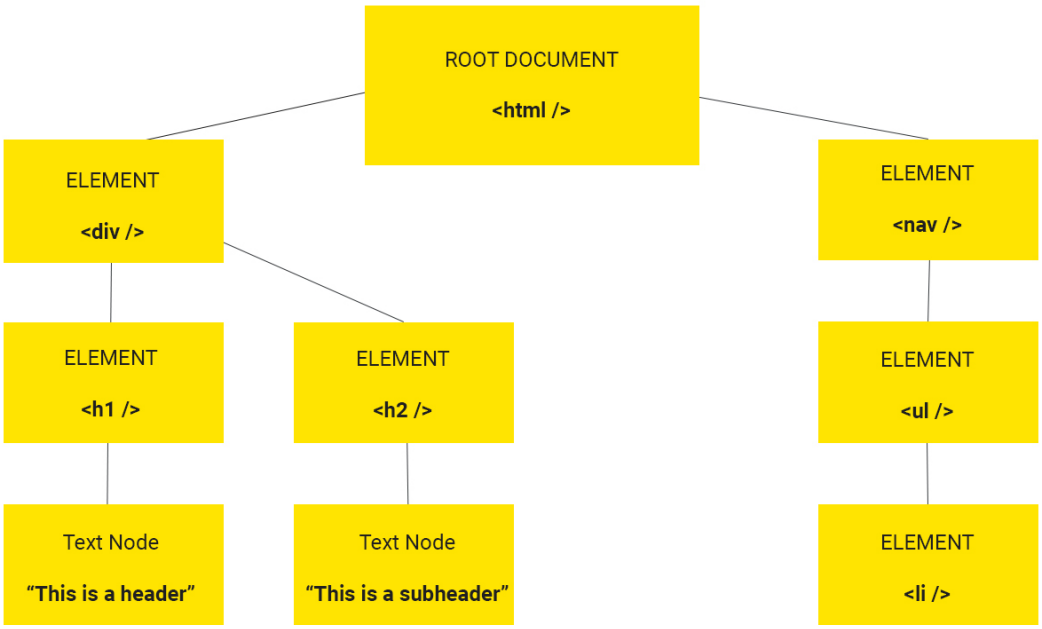
Netscape bracht achteraf nieuwere versies van JavaScript uit, maar Microsoft dreigde die te onttronen. Niet zoveel later immers, bracht Microsoft Internet Explorer 3 uit, met een eigen variant op JavaScript, namelijk JScript. Dit was een zware slag voor Netscape, aangezien Microsoft hen hierdoor zou voorbijsteken. Tegelijk was het een grote stap in de evolutie van JavaScript zoals wij het nu kennen. Dat bracht echter problemen met zich mee, want bedrijven zouden telkens eigen versies van JavaScript uitbrengen wat voor veel compatibiliteitsproblemen zou zorgen. Uiteindelijk, in 1997, werd JavaScript 1.1 gestandaardiseerd dankzij het European Computer Manufacturers Association en werd omgedoopt tot ECMAScript. (Wiley, 2016) 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 niet 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 op het feit dat het type van waarden kan veranderen tijdens uitvoertijd. Dat 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 omdat 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, zoals beschreven in 2.1. 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. Kantor (2017)

Prototype-based

In JavaScript kan een object eigenlijk aanzien worden als een Array. Hierdoor kunnen ze heel gemakkelijk aan de eigenschappen van een object en kunnen deze ook makkelijk tijdens uitvoertijd aangepast worden. Ook kunnen er op deze manier makkelijk eigenschappen aan bestaande objecten worden toegevoegd. In JavaScript wordt gebruik gemaakt 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 heen 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. Door 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 die wordt gebruikt voor server-side toepassingen. Node.js maakt het mogelijk (of eerder *makkelijker*) om JavaScript nu ook te gebruiken buiten het maken van websites. Aangezien JavaScript zo'n krachtige taal is, hadden de ontwikkelaars van Node.js één ding in gedachten: JavaScript niet enkel voor browsers, maar ook voor alleenstaande applicaties. Sindsdien evenaart JavaScript andere scripttalen, zoals Python. Patel (2018)

2.2.1 Tijdlijn van Node.js

Node.js' verhaal start in 2009, toen de ontwikkelaar Ryan Dahl een probleem ondervond met Apache Http Servers.

Zoals beschreven in (Chaniotis, Kyriakou & Tselikas, 2015), 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 ontwikkeld geweest om hedendaagse, complexe webapplicaties te schrijven. Ten tweede was Apache niet in staat om te schalen naar meerdere processorkernen, waardoor de performantie enorm daalde. PHP in combinatie met Apache was gewoonweg

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

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

Applicaties worden in toenemende mate gemaakt met cross-platform compatibiliteit in het achterhoofd, aangezien de toestroom van verschillende nieuwe besturingssystemen en apparaten het meer en meer onmogelijk maakt 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 die 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 hebben gehad op ons dagelijks leven, is het haast ondenkbaar dat real-time communicatie niet zou kunnen bestaan.

Al deze voorgaande elementen hebben bijgedragen tot de ontwikkeling van Node.js. Het project van Dahl werd met open armen ontvangen. Na enkele struikelblokken zag Node.js in 2011 voor het eerst het licht en wordt het nu gebruikt in vele moderne applicaties. Ook vandaag is het steeds één van de meest gegeerde vaardigheden van een programmeur. (Patel, 2018)

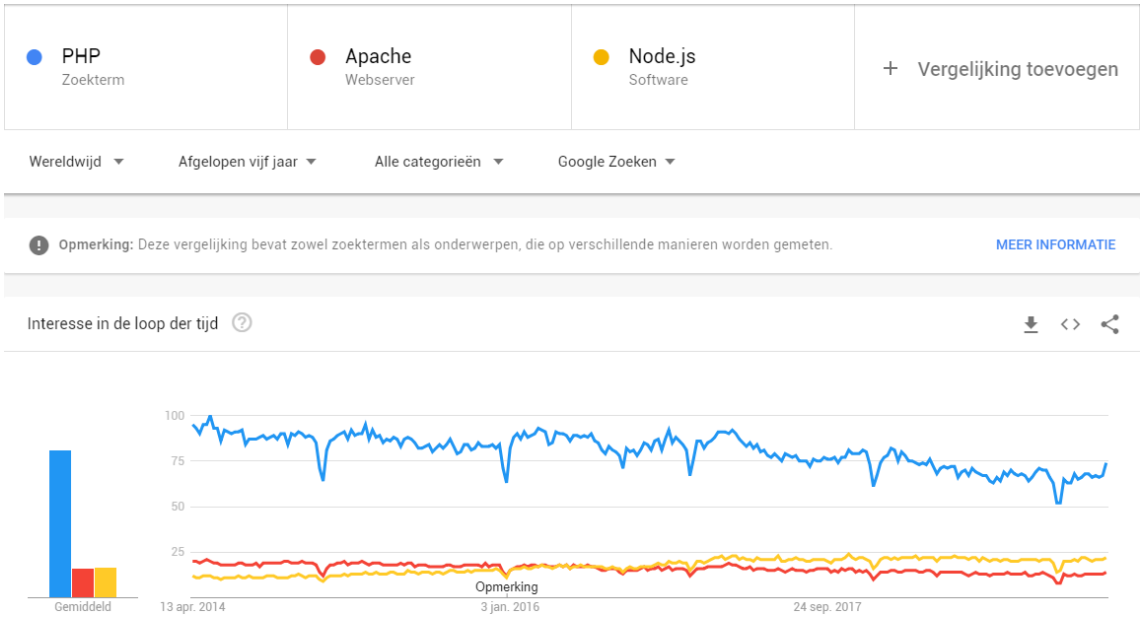
Hoewel Apache en PHP nog steeds overduidelijk op plaats één staan, hebben ze hun succes enkel nog te danken aan bedrijven die op verouderde software werken, en aan hun populariteit bij senior ontwikkelaars. PHP blijft een geliefde taal bij velen, maar in 2.2 zien we dat Node.js's populariteit duidelijk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

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 BAT (2016) toont duidelijk de voordelen van Node.js aan. LinkedIn kon zijn 15 servers reduceren naar slechts 4 en ondertussen de verkeerscapaciteit nog eens verdubbelen. Walmart kon al zijn client-side JavaScript processing naar zijn servers verplaatsen wat voor een enorme performantieboost zorgde, en eBay kon zijn verkeerscapaciteit enorm verhogen en het verbruik substantieel verlagen. De voordelen en functionaliteiten worden hieronder nog eens opgesomd zoals aangegeven door Chandrayan (2017).

[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.

Nieuw

JavaScript is relatief gezien een nieuwe taal die voortdurend geüpdatet wordt, in tegenstelling tot de traditionele server-talen zoals Python, PHP, enzovoort. Vele dialecten van JavaScript (zoals TypeScript, CoffeeScript, ClojureScript,...) compileren gewoonweg in JavaScript, waardoor overschakelen heel eenvoudig wordt. Doordat Node.js ook in JavaScript geschreven is, maakt dit het zowel voor de frontend-ontwikkelaar als voor de backend-ontwikkelaar makkelijker om client-en server-side dichter bij elkaar te brengen. („Express/Node introduction”, g.d.)

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 - 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 dat 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 (Ancona e.a., 2017), kan asynchrone code subtiele bugs opleveren die niet meteen zichtbaar zijn. Hier is nog niet echt onderzoek naar 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. (Ancona e.a., 2017) vertelt ons meer over het identificeren van schaalbaarheidsproblemen en het aanreiken 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. Hierop wordt later in dit onderzoek dieper ingegaan.

Performant

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

npm

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

2.3 Express

2.3.1 Waarom Express?

„Express/Node introduction" (g.d.) maakt ons echter duidelijk dat niet alles rechtstreeks door Node.js wordt ondersteund. Als men bijvoorbeeld aan een webapplicatie routing 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 Express. Het is veruit *het* populairste Node framework en is zelfs de grondlegger van duizenden andere pakketjes. Express wordt daarom door bedrijven, waaronder Kayzr, dan ook vaak in combinatie gebruikt met Node.js.

Request Handlers

Dankzij Express wordt het zeer gemakkelijk om routering in een 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 elk soort request en elk type 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 naar wens ingesteld worden; van poorten en routers, het instellen van een databank tot het toevoegen van Express middleware.

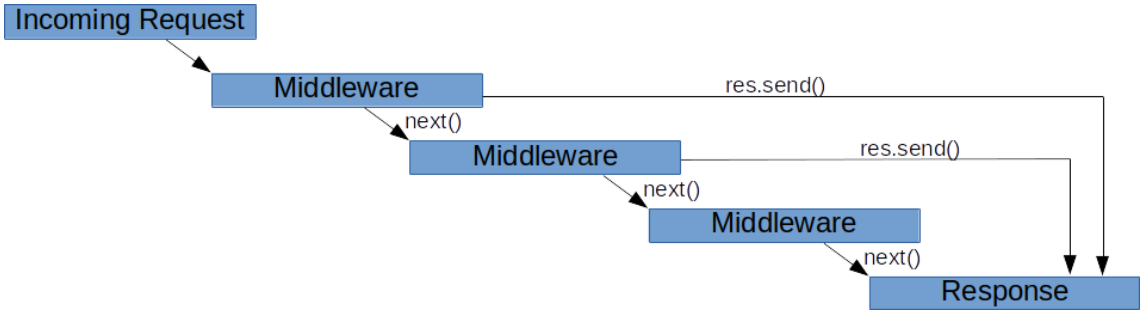
Middleware

Wellicht de krachtigste eigenschap van Express is het gebruik kunnen 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 eveneens helemaal verkrijgbaar is 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. Monitoringtoepassingen in Node.js maken vaak gebruik van middleware. Hierover volgt meer info in het volgende hoofdstuk.

Listing 2.1: Express code flow voorbeeld.

```
const express = require('express');
const app = express();

// An example middleware function
const a_middleware_function = function(req, res, next) {
// ... perform some operations
// Call next() so Express will call the next middleware function
 in the chain.
next();
}

// Function added with use() for all routes and verbs
app.use(a_middleware_function);

// Function added with use() for a specific route
app.use('/someroute', a_middleware_function);

// A middleware function added for a specific HTTP verb and route
app.get('/', a_middleware_function);

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 de router. Deze methodologie is gevisualiseerd in figuur 2.3.



Figuur 2.3: Schema van de werking van middleware, gebruikt van Brandt, 2018

2.4 Testen en Monitoring

Testen schrijven. Of men nu alleen aan software werkt, of in team in een bedrijf, elke programmeur met gezond verstand zal kunnen vertellen over het belang van testen schrijven. Iets waar vaak wordt tegenop gezien, maar nodig is om 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 die dicht bij elkaar horen. Door testen te implementeren kan men kwaliteitsvollere code schrijven, fouten sneller opsporen en is de software een pak robuuster. Het toevoegen van nieuwe functionaliteiten kan de code niet meer breken en daardoor kan de software langer meegaan.

2.4.1 Agile Testing

Agile testing is een methodologie, ontstaan rond 1990, waarin software incrementeel en iteratief wordt uitgewerkt en die op de dag van vandaag in vele bedrijven wordt gehanteerd. Vooraleer er effectieve code wordt geschreven, zullen er eerst testen uitgewerkt 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 en oude testen worden aangepast. Zo moet men telkens maar kleine gedeeltes code aanpassen of schrijven. Een goede handhaving van agile testing doet niet alleen de slaagkansen van het project vergroten, het kan het project ook goedkoper maken 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. (Chakravorty, Chakraborty & Jigeesh, 2014)

Dit kan echter ook problemen met zich meebrengen. Agile vergt voorbereiding. Chakravorty e.a. (2014) vertelt ons dat de kost en tijd van het project uit balans kunnen geraken telkens wanneer de klant iets wil veranderen. Daardoor komt er veel werk op de programmeurs te liggen. In („Adopting Agile Testing, a Borland Agile Testing White Paper”, 2012) wordt dit ook nog eens samengevat. Veel bedrijven passen de agile methodologie toe, maar niet de agile testing. Dit komt omdat in agile de verantwoordelijkheid om beslissingen te nemen naar het team van ontwikkelaars wordt verschoven. Dat breekt met de traditionele watervalmethode waar de belangrijkste keuzes door management en

niet door het team worden genomen. Dit breekt ook met de traditionele hiërarchische structuur waarin bedrijven werken. Het management 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 wijzigt maar ook de mentaliteit van het bedrijf. Veranderingen in een bedrijf gaan traag vooruit, zeker wanneer het een complete mentaliteitsverandering vergt. Daarom dat er veel bedrijven zijn die ervoor opteren om de agile methodiek slechts 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 effectief implementeren. Een andere grote hindernis bij agile testing is dat er niet veel tools bestaan die geavanceerde testingmethodes ondersteunen.

Wat men ook in de afgelopen tien jaar meer en meer terugziet, is de problematiek van het continu releasen van software. Software-reuzen zoals Google, Facebook, Mozilla, enzovoort... kozen ervoor om software telkens opnieuw uit te brengen in zeer korte periodes, ook wel *rapid releases* genoemd. Mozilla kende bijvoorbeeld oorspronkelijk 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. Daardoor brengen ze nieuwe, maar kleinere updates om de 6 weken uit. Dit kent zijn voordelen, maar echter ook zijn nadelen in het test-proces. Aangezien er aan een steeds hoger tempo moet uitgebracht worden, moeten testen meer geautomatiseerd worden, wordt er minder getest en is de software minder robuust. Hierdoor treden er sneller fouten op (Mäntylä, Khomh, Adams, Engström & Petersen, 2013). Kayzr kampt met dezelfde problemen. Daarom is er steeds meer nood aan monitoringtoepassingen 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: unit testing, integratietesting en systeemtesting. Er zijn er nog andere: sanity testing, smoke testing, interface testing, regression testing, end to end testing, performantietesting,... zijn allerlei soorten opgesomd door Pittet (2019). Dit zijn slechts de functionele testen. Men heeft ook niet-functionele testen, die dan in verband staan met 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 de info en metriek te analyseren van een lopend proces. Zo kan monitoringsoftware er op letten dat alles verloopt volgens de afspraken, en indien dit

niet het geval is, actie ondernemen (Rouse, 2018).

2.5.1 Hoe werkt monitoring?

Real-time monitoring is een monitoringtechniek 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, geheugenverbruik, 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 de 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. Monitoringsoftware 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 monitoringmiddlewares voor Express, maar geen software die aan de vereisten van Kayzr voldoet. Hier 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 niet relevant zijn voor Kayzr, niet meer ondersteund werden 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 weer te geven voor 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
- Geheugenverbruik
- Response-time
- Requests per seconde
- Load average
- Status codes

Prometheus

Prometheus is een open-source, gratis monitor voor Node.js die uitblinkt in datacompressie 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 monitoringsoftware gemaakt door New Relic, Inc. Het ondersteunt Node.js servers maar kan ook gebruikt worden voor Ruby, Java, PHP, .NET, Python en Go. Ook de ondersteuning met andere frameworks is zeer uitgebreid, onder meer de

databanken en platformen zoals mongoDB, Redis, MySQL, Express, Http, KrakenJs en nog veel meer.

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

- 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. Momenteel betaalt men per server maandelijks tussen de **€17.00** en **€600.00.** Aangezien Kayzr ongeveer 50 verschillende servers heeft, is dit budgetmatig onverantwoord. Ook bevat het veel functionaliteiten die 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 monitoringoplossing van Stackify, probeert zich te onderscheiden van de andere oplossingen door meer features aan te bieden in één strakke applicatie die een heel stuk goedkoper is dan de concurrentie. Naast Node.js ondersteunen ze ook .NET, PHP, Ruby en Java. Hun top-features zijn:

- App Performance Monitoring, een krachtige tool die 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 alle code, die wordt uitgevoerd, op te zoeken valt.
- 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 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 op code level, rekening houdend met gemonitorde heap-en geheugenmetingen en tal van andere middelen. Het kan zelfs fouten opsporen die 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 en het kan gratis gebruikt 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 ontwikkeld 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 wel 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 maken het meteen een enorm duur opstapje.

Samengevat

Men kan hieruit concluderen dat de gratis middleware uitstekend werken om eenvoudige zaken te monitoren. Het moment dat er wordt overgeschakeld naar een betalende oplossing, stijgt het aantal functionaliteiten enorm. Elke betalende monitoringsoftware heeft ook minstens één eigenschap waarmee die zich onderscheidt van de concurrentie.

Er kan hieruit besloten worden dat er geen software bestaat dat het beste brengt van beide werelden, namelijk de betalende en niet-betalende oplossingen.

3. Het probleem van Kayzr

3.1 Probleemstelling

Het probleem van Kayzr is dat hun data over verschillende servers verspreid staan. Kayzr wenst deze data te centraliseren. Zo hoeven ze niet langer voor sommige zaken naar Google App Engine te gaan, andere voor andere naar Google Cloud Service, nog andere zaken in Morgan, of simpelweg de Node console, enzovoort. Dit zou het best kunnen opgelost worden via een Express middleware die 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, zoals besproken in hoofdstuk 2.5.3.

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

3.2 Effectieve requirements

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

- 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

op de backend server. Maar daar 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 de taxatie van deze Node.js processen 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 eerder opgesomde betalende monitoringsoftwares bevatten te veel functies die Kayzr niet meteen zou gebruiken, maar wel handig *zouden kunnen* zijn. Dit verantwoordt echter hun grote kostprijs niet, en Kayzr wenst dus dat budget er niet aan te spenderen. Bestaat er geen goede middenweg?
- Afhankelijk zijn van betalende services betekent ook dat de data van Kayzr terecht komen bij (dure) third party oplossingen. Zijn die data veilig? Volgen ze de GDPR regels? Ze houden liever hun data in eigen beheer.
- Men wil niet enkel realtime zaken kunnen bekijken, maar ook data opslaan zodat die op een later moment nog eens bekeken kunnen worden.

Kayzr wenst dus een middleware oplossing, gratis te downloaden via npm, die 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 vormt voor het ontwikkelingsteam, er meer data is uit te halen dan vóór de verkregen middleware, en aan alle voorgaande requirements is voldaan.

De opbouw van die middleware wordt omschreven in hoofdstuk 5.

4. Methodologie

In *Stand van zaken* werd er onderzoek gedaan naar verschillende middleware en tools voor het monitoren van een Node.js applicatie die gebruikt maakt van Express. In *Het probleem van Kayzr* werd dit document aangevuld met functionaliteiten en vereisten die Kayzr in de software graag wenst te zien.

Nu dat al de nodige informatie is verzameld, kan het ontwerpen van de middleware toepassing van start gaan. Daarom zal volgens een bepaalde werkwijze een *proof-of-concept* worden uitgewerkt, waarover zo dadelijk in hoofdstuk 5 meer uitleg zal worden gegeven. Daarna volgt er een uitleg van de verschillende functionaliteiten die de software bevat.

Tijdens het ontwerpen van de middleware zal echter niet enkel met de requirements van Kayzr rekening gehouden worden. Het is de bedoeling dat dit geen *hardcoded* stuk software in hun backend wordt (software die niet aanpasbaar en distribueerbaar is), maar een abstracte, universele middleware die gedownloaded en geïnstalleerd kan worden via Node.js package manager. Daarom zal een verwijzing naar Github en npm alsook de *README.md*, een bestand dat informatie bevat over alle andere bestanden in de software, worden toegevoegd als bijlage om te bewijzen dat de software effectief openbaar staat om in een eigen project te verwerken.

Tenslotte zal er in hoofdstuk 6 worden geconcludeerd of de software effectief in de productie-backend van Kayzr wordt ingehaakt, hoe Kayzr de toekomst ziet, en of deze studie kan uitgebreid worden d.m.v. een verder onderzoek.

5. Proof of concept

5.1 Werkwijze

Eerst werd een middleware ontwikkeld die op een mooie manier elke request kon loggen in de console. Daarna werd een server aangemaakt op één van de servers van Kayzr, en daar werden 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 de backend van Kayzr. Op het einde werden er dan grafieken aangemaakt in Grafana die aan de requirements van Kayzr zouden voldoen.

5.1.1 Multilogger

De te ontwikkelen middleware werd omgedoopt tot Multilogger (en later officieel express-influx-multilogger). De eerste stap bestond uit het aanmaken van een basis express applicatie, waarin de eerste stapjes middleware software werden geschreven. Daarna werd per keer dat een API call werd gemaakt, de methode van deze call naar de console gelogd.

In app.js

Listing 5.1: app.js eerste stap

```
const 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 5.2: 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 verder uitgebreid, zodat er een mooi breed overzicht naar de console

werd gegenereerd. Natuurlijk wensen niet alle ontwikkelaars dat er per API call naar de console zou worden gelogd. Dit kan immers oplopen van (uiteraard afhankelijk van de software) honderden tot duizenden calls per enkele seconden met dus zeer veel overbodige informatie tot gevolg. Daarom werd er een parameter toegevoegd om deze functionaliteit uit te schakelen, met als achterliggende redenering dat de middleware niet enkel mag 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 5.3: Parameters toegevoegd

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

Listing 5.4: 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;
```

5.1.2 Grafana en InfluxDB

Vervolgens moest er een systeem gekozen worden om verzamelde data op te slaan en weer te geven. Dat kan natuurlijk gerealiseerd worden met verschillende databanken en frontend frameworks. Oorspronkelijk 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 Hill (2015). 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.

5.1.3 Wegschrijven en abstraheren

Eens dat de databank was aangemaakt, werd het tijd om deze te koppelen aan de middleware. Op basis van het voorgaande logsysteem, werd een object aangemaakt dat alle belangrijke info moest bevatten.

Listing 5.5: Log object

```
const log = (extended, development) => {
  return async (req, res, next) => {
    const startHrTime = process.hrtime();

    res.on("finish", async () => {
      const elapsedHrTime = process.hrtime(startHrTime);
      const elapsedTimeInMs = elapsedHrTime[0] * 1000 +
    elapsedHrTime[1] / 1e6;

      const realBody = _.isEmpty(req.body) ? " " :
    JSON.stringify(req.body);
      const cpuUsage = await getCpuInfo();
      const memoryUsage = await getMemInfo();
      const hostInfo = await getHostInfo();

      const location = await iplocation(req.connection.remoteAddress)
      .then(result => {
      result.geohash = geohash.encode(result.latitude,
    result.longitude);
      return result;
      })
      .catch(err => {});

      if (extended) {
        getBasic(req, res, hostInfo, elapsedTimeInMs);
        getParameters(req, realBody);
        getAuth(req);
        getPerformance(cpuUsage, memoryUsage);
```

```
26      }
27
28      logObject.method = req.method || " ";
29      logObject.statusCode = res.statusCode || " ";
30      logObject.statusMessage = res.statusMessage || " ";
31      logObject.date = new Date().toUTCString();
32      logObject.responseTime = elapsedTimeInMs;
33      logObject.contentType = req.header("Content-Type") || " ";
34      logObject.hostname = req.hostname || " ";
35      logObject.osHost = hostInfo.hostname || " ";
36      logObject.url = req.originalUrl || req.url || " ";
37      logObject.path =
38      res.statusCode !== 404 && req.route && req.route.path
39      ? req.route.path
40      : " ";
41      logObject.body = req.method === "POST" ? realBody : " ";
42      logObject.params = _.isEmpty(req.params)
43      ? " "
44      : JSON.stringify(req.params);
45      logObject.query = _.isEmpty(req.query) ? " " :
    JSON.stringify(req.query);
46      logObject.cookies = _.isEmpty(req.cookies)
47      ? " "
48      : JSON.stringify(req.cookies);
49      logObject.auth =
50      req.header("Authorization") || req.header("x-access-token") ||
    " ";
51      logObject.ip = req.connection.remoteAddress || req.ip || " ";
52      logObject.location = location || " ";
53      logObject.clientInfo = req.header("User-Agent") || " ";
54      logObject.memoryUsage = memoryUsage;
55      logObject.cpuUsage = cpuUsage;
56      logObject.errorMessage = res.locals.multiError || " ";
57
58      if (development) {
59        console.log(logObject);
60      }
61
62      multilog.pushToData(logObject);
63      logObject = {};
64      });
65     next();
66   };
67 };
68
```

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

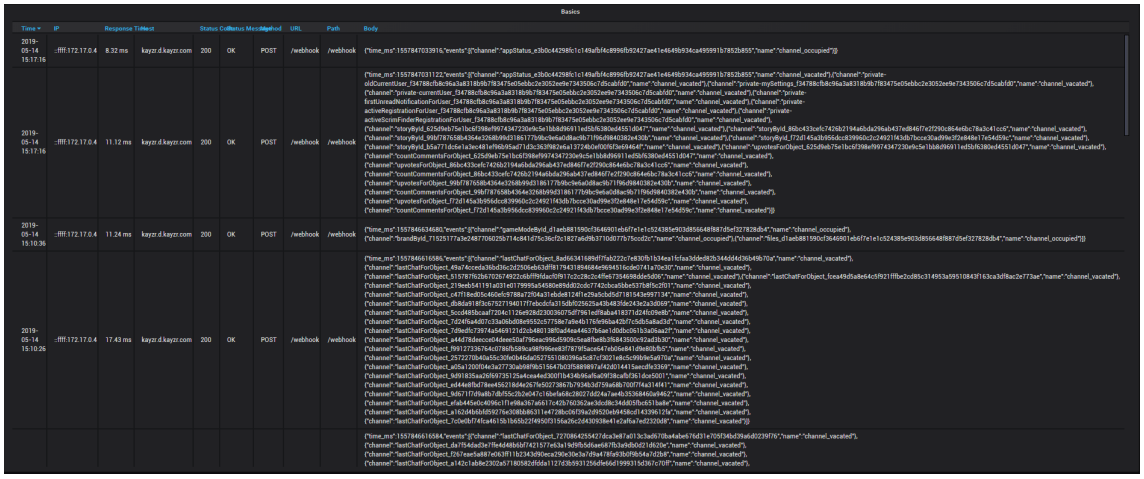
De volgende stap in het ontwikkelen van een middleware oplossing voor Kayzr was het abstraheren van de code. Zo kan niet enkel Kayzr hiervan gebruik maken, maar iedereen met een Node.js en Express configuratie. Er kwamen wat moeilijkheden opduiken bij het loskoppelen van de code, maar uiteindelijk werd 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 managementsystemen 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 het webplatform van Kayzr. Zo geraakte influx op een snellere manier gevuld met relevante data zodat er begonnen kon worden met het ontwikkelen van de volgende stap.

5.1.4 Grafana's grafieken

Tenslotte werd Grafana opgebouwd om mooie, relevante grafieken weer te geven voor de eindgebruiker. De verzamelde data bleken 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 keren en het schema van InfluxDB te herzien. Hierdoor kwamen ook nieuwe ideeën tot stand, zoals het toevoegen de geolocatie van een gebruiker en het monitoren van de snelheid van elk databankverzoek per API call. Op dit ogenblik zijn er in Grafana 7 grafieken beschikbaar.

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.



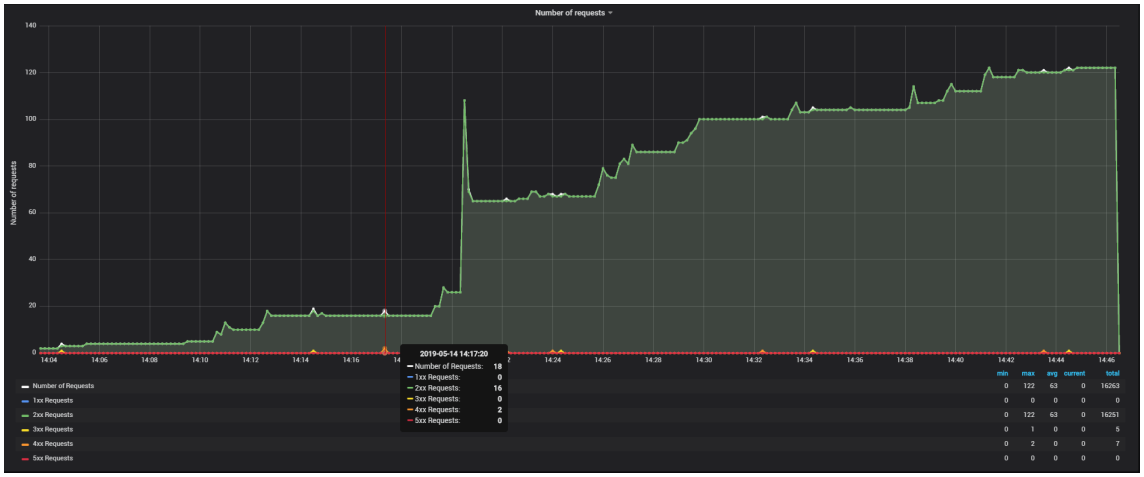
Figuur 5.1: Basics-paneel

Number of requests

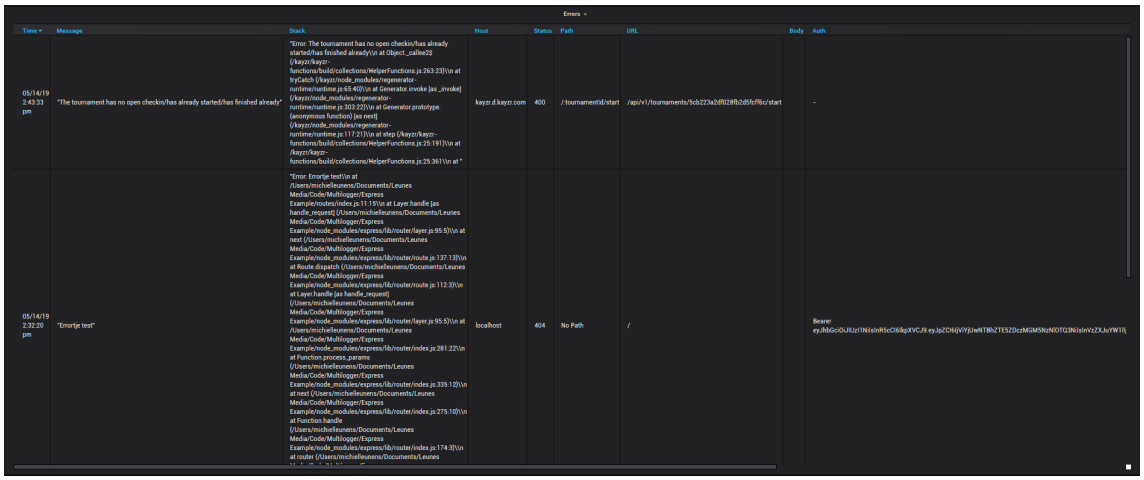
Het aantal verzoeken in een gegeven tijdsframe. Ook kunnen deze opgesplitst worden per statuscode, zodat men bijvoorbeeld kan observeren hoeveel API calls een fout gaven.

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 weergegeven. Aangezien Kayzr dit ook wou gebruiken voor debug-doeleinden, worden ook de body en authenticatietokens weergegeven.



Figuur 5.2: Aantal requests-paneel (wit voor totaal, blauw voor requests die starten met 1xx, groen voor 2xx, geel voor 3xx, oranje voor 4xx en rood voor 5xx.)



Figuur 5.3: Errors-paneel

Database timings

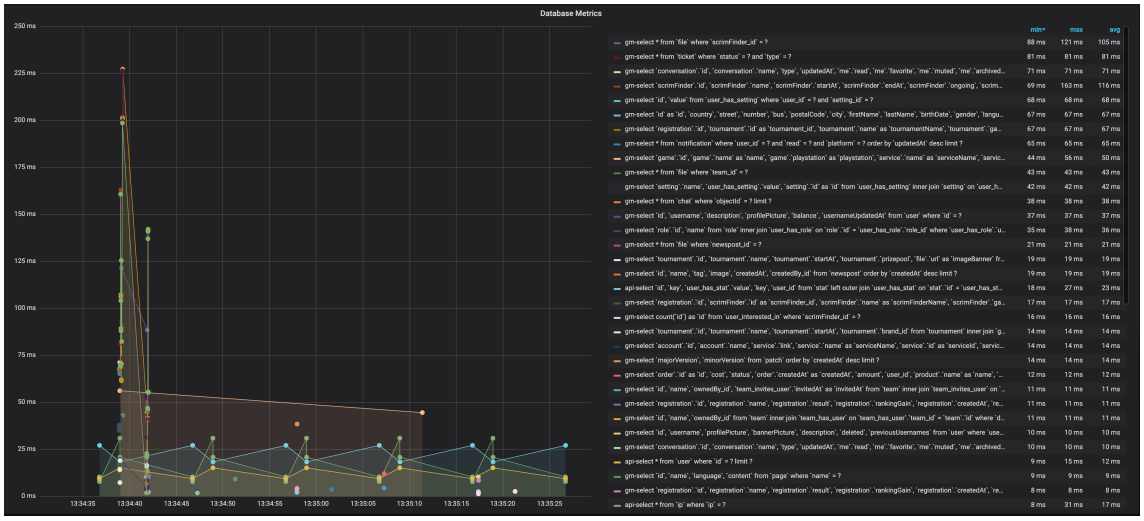
Hier wordt per API call de tijden gelogd van een bepaald verzoek naar de databank. Zo kan men per call zien hoeveel tijd het vraag om specifieke data op te halen.

Users

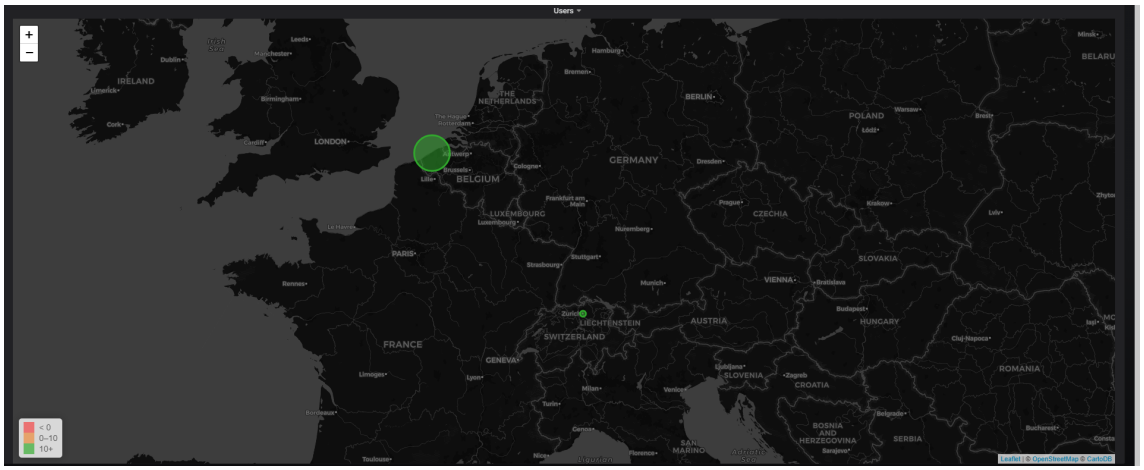
Op een wereldkaart wordt het aantal gebruikers per land getoond. De geolocatie wordt afgeleid van het ip-adres van de gebruiker, en specifieke locaties worden dus niet opgevraagd.

Memory Usage

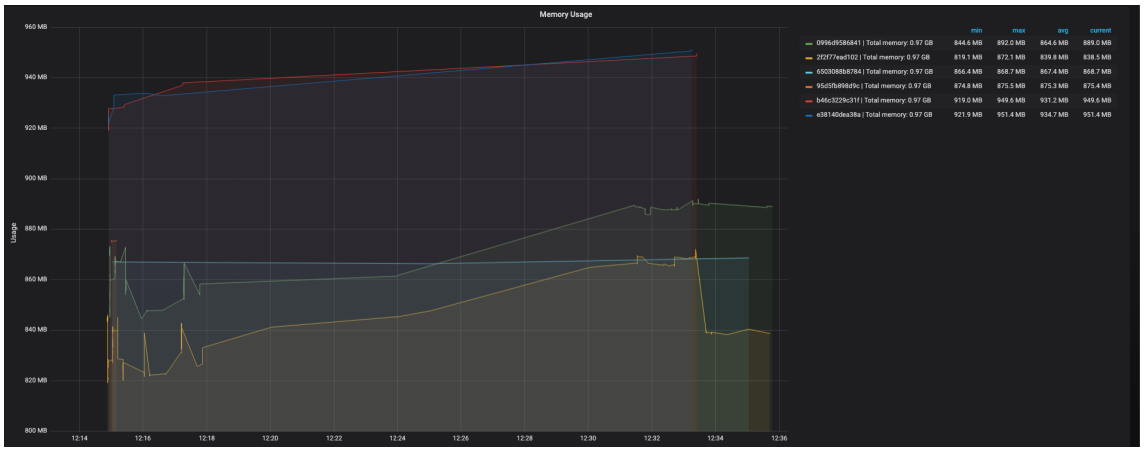
Het geheugenverbruik in gigabytes van de machine waarop de server draait.



Figuur 5.4: Database-metrics-paneel



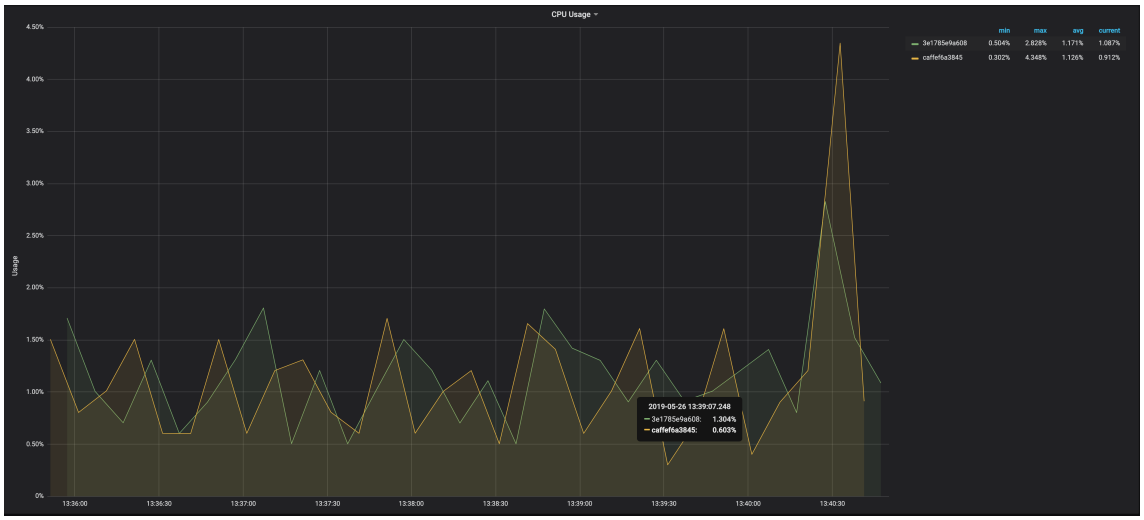
Figuur 5.5: Users-paneel



Figuur 5.6: Geheugenverbruik-paneel

CPU Usage

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



Figuur 5.7: Processorverbruik-paneel

5.2 Publicatie

De package is ondertussen gepubliceerd op npm onder de naam Express-influx-multilogger, en ook op Github, waar de source code kan geraadpleegd en bestudeerd worden. Installatie en instructies kunnen gevonden worden in de onderstaande README.md. Ook staan hier code-voorbeelden, uitleg van de parameters en tal van andere zaken in. De middleware is open-source, met de bedoeling om krachtiger en efficiënter te worden in de toekomst. Op dit ogenblik is de laatste versie **1.0.12**.

Express-influx-multilogger

An Express middleware for better monitoring of your Node.js apps. Parse important req, res and header objects to Influx and Grafana. Get an easier insight of your API without any costs.

Note: this is in ***active*** development, and could contain bugs. Please make an issue if you find some.

Getting started

Installation

1. Install package
    - Npm

```
npm install express-influx-multilogger
```

    - Yarn

```
yarn add express-influx-multilogger
```

2. Require

```
const multilogger = require('express-influx-multilogger');
```

3. (Required if you want to write to influx) Initialize database in your app.js

```
multilogger.init({
 database: {
     server: "127.0.0.1",
     name: "myMultilogDb",
     port: 8086
      },
 interval: 10000,
 performance: true,
});
```

4. Add multilogger log function right before your router of choice

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

5. Add multierror function before catching 404 and after your router

```
app.use(multilogger.error());
```

6. In case of using geolocation, make sure your server can make use of req.connection.remoteAddress
7. You can also add extra custom metrics for monitoring other things, like your database calls

   Add this line in your code (make sure you require multilogger). Name and timing parameters are required. Custom is optional.

```
multilogger.insertDatabaseCallSpeed({
  name: "Random name", // name of your metric (like a database call)
  timing: 0.2443, //in ms
  custom: {
    customOption1: "foo",
    foo: foo.bar,
  }
});
```

Example

```
const express = require("express");
const bodyParser = require("body-parser");
const cookieParser = require("cookie-parser");
const multilogger = require('express-influx-multilogger'); // Add this to
imports

const indexRouter = require("./routes/index");

//  Add this to write data to influx (not required)
multilogger.init({
    database: {
        server: "127.0.0.1",
        name: "myMultilogDb",
        port: 8086
         },
    interval: 10000,  // Write to Influx every 10 seconds,
    performance: true // Write host performance to Influx (mem, cpu,..)
});
multilogger.startPerformanceMetrics();

const app = express();

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.use(cookieParser());
```

```
// Add this before your router of choice
app.use(multilogger.log({ development: false, extended: false }));
app.use("/", indexRouter); // Your router

// Add this before 404 if you want to capture your errors as well
app.use(multilogger.error());

module.exports = app;
```

Loggable headers

The data being sent to Influx consists of ...

Fields

...logs of the amount of calls per statuscode per given interval

```
    amountOf1xx: Influx.FieldType.INTEGER,
    amountOf2xx: Influx.FieldType.INTEGER,
    amountOf3xx: Influx.FieldType.INTEGER,
    amountOf4xx: Influx.FieldType.INTEGER,
    amountOf5xx: Influx.FieldType.INTEGER
```

...logs of Response Time, CPU and memory Usage of the OS, host and ip;

```
    responseTime: Influx.FieldType.FLOAT,
    cpuUsage: Influx.FieldType.FLOAT,
    memoryUsage: Influx.FieldType.FLOAT,
    requests: Influx.FieldType.INTEGER,
    host: Influx.FieldType.STRING,
    ip: Influx.FieldType.STRING
```

Tags

...logs of basic headers to Influx

```
    "statusCode",
    "statusMessage",
    "method",
    "path",
    "url",
    "ip",
    "country",
    "geohash", // Used to get the geolocation of your api call from a
given IP
    "client",
```

```
    "body",
    "query",
    "params",
    "auth", // In case you wanna know wich user that triggered the api
call
    "errorMessage", // In case of an error: sends message and stack
    "errorStack"
```

Parameters

1. Extended: Logs a pretty view of req, res and headers (defaults false)
2. Development: Logs the object that Influx will recieve
3. Database:
    - server: The address of your Influx database. *(defaults: 127.0.0.1)*
    - name: Name of your Influx database. *(defaults: myMultilogDb)*
    - port: Port of your Influx database. *(defaults: 3000)*
    - username: Login credentials of your Influx database. *(defaults: '')*
    - password: Password of your Influx database. *(defaults: '')*
4. Interval: Defines the rate in ms of the interval you want to write your data to your Influx database

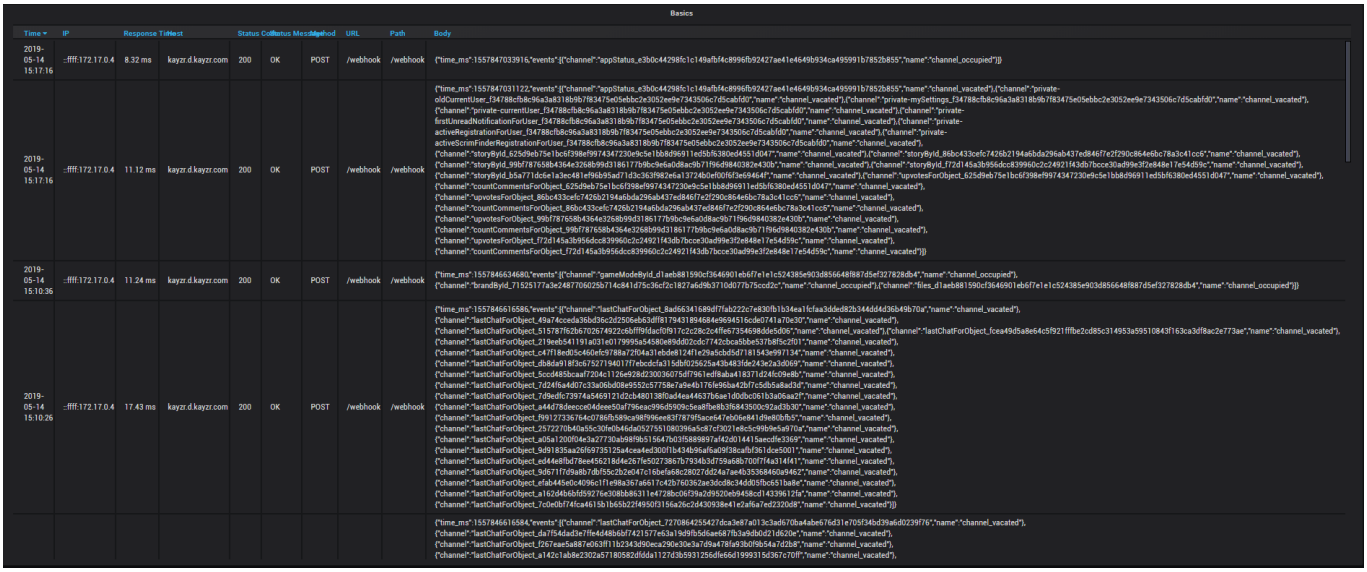
Reset Table

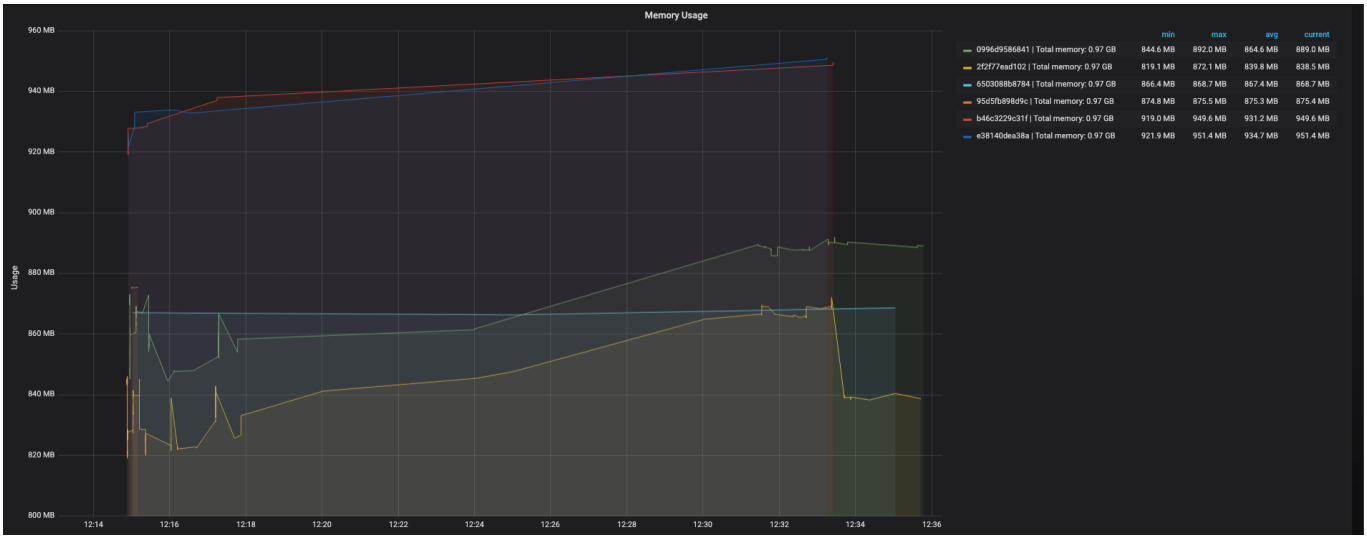
What if you have sent wrong info to Influx? You can make use of the Influx-cli to drop your database or a certain metric or series (table)

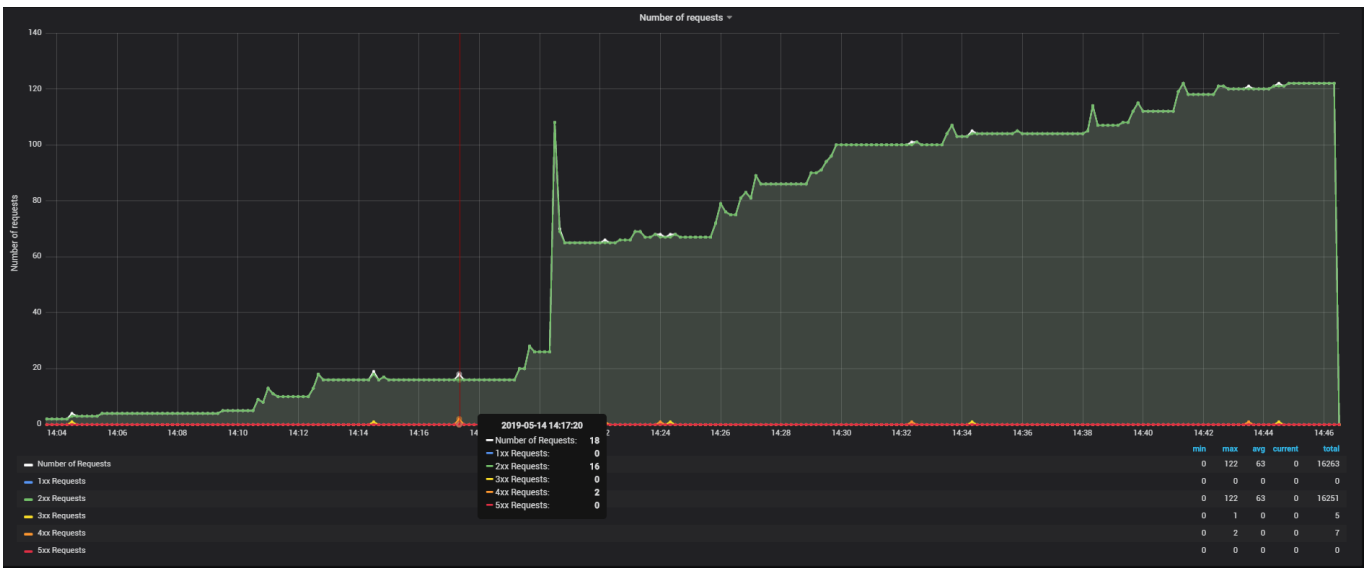
```
> DROP DATABASE <db_name>
> DROP MEASUREMENT <measurement_name>
> DROP SERIES FROM <db_series>
```

For more info, you can check the docs.

Grafana







If you want to use Grafana and don't want to write your dashboard, you can import the dashboard JSON. You can certainly write your own dashboard, as I'm not that skilled with Grafana.

By using this dashboard, make sure you install the worldmap plugin.

License

Distributed under the MIT License. See `LICENSE` for more information.

Dependencies

- systeminformation
- lodash
- node-influx
- ngeohash
- iplocation

Contributors

- Michiel Cuvelier

6. Conclusie

Nu de middleware gepubliceerd is en binnengetrokken op de servers van Kayzr, kan er worden bekeken of hij voldoet aan de vereisten, of er voordelen en al dan niet extra nice-to-haves uit te halen zijn, en of er verbeteringen kunnen plaatsvinden. Dit hoofdstuk is gewijd aan het toetsen van de geschreven software aan de praktijk.

6.1 Meting aan de vereisten

Vooraleer er een definitieve conclusie wordt gemaakt, moet er eerst gekeken worden of de middleware natuurlijk aan alle opgesomde vereisten voldoet, zoals beschreven in hoofdstuk 3.2. We starten met de eerste vereiste:

6.1.1 Toetsing

> *"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?"*

Dankzij de *Number of Requests*-tabel, kan dit nu volledig bekeken worden. Men kan kijken hoeveel API calls doorkomen per 10 seconden, of per uur, of per dag... maar men kan ze ook nog eens groeperen per soort response. Hoeveel 4xx of 5xx calls (calls die foutmeldingen geven) komen door? Ook via de *basics*-tabel kan er veel extra informatie opgevraagd worden. Zo kan men bijvoorbeeld kijken of er tussen 2 pm en 4 pm enkele

requests voorkwamen die een grotere uitvoertijd dan twintig milliseconden hadden en dus een zware last zouden vormen voor de servers.

> *In Google Cloud Stack Driver kan men mooi de errors terugvinden die voorkwamen op 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?*

Dit is volledig opgelost. Door de *errors*-tabel te observeren kan men namelijk

1. kijken op welke URL en pad de fout zich voordeed
2. lezen welke foutmeldingen er werden geregistreerd, samen met de error stack van de desbetreffende fout.
3. Ook nuttige parameters aflezen zodat men context kan plaatsen bij de fout. Parameters zoals de request body, authentication token, queries enzovoort.

Fouten kunnen ook opgezocht worden met behulp van een zoekfunctionaliteit, en worden ook geordend volgend error stack en vervolgens volgens error message. Zo worden duplicaten weggefilterd en wordt er een zeer praktisch en gebruikersvriendelijk overzicht weergegeven.

> *Kan men de (gemiddelde) tijd monitoren tussen het versturen van een bepaalde API call en een verkregen antwoord?*

De gemiddelde tijd wordt gemonitord en kan in de meeste tabellen teruggevonden worden. Ook is het mogelijk om de gemiddelde tijd van database-calls te monitoren. Hierover wordt later nog meer uitleg gegeven.

> *Monitoren van deze Node.js processen hun taxatie op de server waar ze op draaien (CPU, RAM, netwerk,. . .)*

Geheugenverbruik en processorverbruik worden per server weergeven in de desbetreffende *performantie*-tabellen.

> *Middleware als Morgan toont veel te weinig, Google Cloud Stack Driver toont veel maar geeft geen context mee. De eerder opgesomde betalende monitoringsoftwares bevatten te veel functies die Kayzr niet meteen zou gebruiken, maar wel handig zouden kunnen zijn. Dit verantwoordt echter hun grote kostprijs niet, en Kayzr wenst dus dat budget er niet aan te spenderen. Bestaat er geen goede middenweg?*

Deze tool heeft al bewezen dat hij zeer veel zaken aankan, en schaalbaar is om er meerdere uitbreidingen aan toe te voegen. In de eerste release kan deze hij al meer dan Morgan, neemt alle delen die Kayzr handig vindt van over Google Cloud Stack Driver en zet ze om in een eigen Grafana werkomgeving, een werkomgeving die geïmporteerd kan worden in een eigen configuratie waardoor het niet verplicht is om eigen grafieken beginnend van niets op te stellen.

De kost van een Grafana-server voor Kayzr bedraagt ongeveer vijftien euro per maand. Dit is puur voor hosting van een server, en is helemaal geen grote kost. De meeste functionaliteiten van de betalende oplossingen (waarvan prijs veel hoger lag, zoals beschreven in 2.5.3), waren overbodig. Functionaliteiten zoals error stack tracing, konden makkelijk zelf geïmplementeerd worden. En aangezien de middleware open-source en in actieve ontwikkeling is, spreekt het ook voor zich dat in de toekomst meerdere functionaliteiten gaan kunnen toegevoegd worden. Vijftien euro per maand om één server te laten draaien die informatie van verschillende servers kan verzamelen, tegenover gemiddeld zeventig euro per maand *per server* voor een veel te uitgebreide tool, kan natuurlijk als een groot pluspunt van express-influx-multilogger worden beschouwd.

> *Afhankelijk zijn van betalende services betekent ook dat de data van Kayzr terecht komen bij (dure) third party oplossingen. Zijn die data veilig? Volgen ze de GDPR regels? Ze houden liever hun data in eigen beheer.*

De data van Kayzr worden veilig opgeslagen op hun eigen servers. Niemand buiten Kayzr kan aan die data, wat een groot pluspunt is in een tijd waarin privacywetgeving nog nooit zo belangrijk werd geacht.

> *Men wil niet enkel realtime zaken kunnen bekijken, maar ook data opslaan zodat die op een later moment nog eens bekeken kunnen worden.*

InfluxDB is overal beschikbaar op de servers van Kayzr. Indien de data nog ergens anders moeten gebruikt worden, kan dat zonder enige moeite.

6.1.2 Nice-to-have

Verder werden er nog enkele optionele functionaliteiten toegevoegd. Dezen worden hieronder opgesomd. Meerdere extraatjes kunnen mogelijks later nog worden toegevoegd.

Geolocatie

Kayzr kan nu op een interactieve wereldkaart bekijken waar het merendeel van de verzoeken vandaan komt. Er werd geopteerd om de locatie te traceren via een binnenkomend ip-adres. Dit is allesbehalve de meest precieze methode, maar ze voldoet om het land van herkomst te detecteren. Ook wordt er op deze manier geen privacy geschonden van de eindgebruiker. Eindgebruikers die zich via een vpn-service zouden verbinden kunnen worden verwaarloosd, aangezien deze meetpunten relatief weinig voorkomen.

Deze nieuwe functionaliteit is handig om te kijken of een toernooi populairder is in België of Nederland, en of Kayzr misschien kan uitbreiden naar het buitenland.

Databank metrieken

Sinds versie 1.0.9 is het ook mogelijk om op een willekeurige plek een multilogger-functie op te roepen in de eigen backend. Deze functie biedt de mogelijkheid om een naam, timings en extra parameters toe te voegen. Zo kan men bijvoorbeeld een specifieke call naar een databank timen (bijvoorbeeld: hoe lang duurt het om een nieuwe gebruiker weg te schrijven naar MySQL). De naam van deze databank en de gemeten tijd stuurt men dan mee door, samen met extra optionele parameters. Al deze informatie wordt dan weggewerkt naar InfluxDB en kan dan ook in Grafana worden bekeken.

Listing 6.1: Extra informatie wordt doorgestuurd naar InfluxDB

```
const addToObject = ({ name, timing, ...custom } = {}) => {
  let databaseMetrics = {
    name,
    timing
  };
  _.flatMap(custom, param => {
    return _.map(param, (value, key) => {
      return (databaseMetrics[key] = value);
    });
  });
  object.databaseMetrics.push(databaseMetrics);
};
```

Listing 6.2: Deze functie kan overal in de API worden opgeroepen

```
router.get("/", function(req, res, next) {
  multilogger.insertDatabaseCallSpeed({
    name: "Testje",
    timing: 5.203,
    custom: {
      optionalInfo: "1",
      2: "3",
      foo: "fighters",
      objectAverageExample: object.example
    }
  });
  res.send("Got a GET request");
  next();
});
```

6.2 Hoe ziet Kayzr de toekomst?

Uit de voorgaande opsomming blijkt dat de onderzoeksdoelstelling werd bereikt, en de onderzoeksvraag werd beantwoord. Kayzr zelf liet weten dat ze zeer tevreden zijn met de nieuwe beschikbare tool en willen deze meteen op al hun servers laten draaien. Zo kan er al meteen informatieve data worden opgehaald van de gloednieuwe CSGO (Counter Strike: Global Offensive) servers, zodat productiesnelheid omhoog kan.

Ook gaan ze dankzij de extra database-metriekfunctionaliteit de database processen stresstesten van hun zelfgeschreven data-abstractielaag, genaamd *gnewmine* en kijken waar deze laatste geoptimaliseerd zal kunnen worden.

6.3 Uitbreiding en verder onderzoek

Dit onderzoek heeft aangetoond dat een debugoplossing in Node.js en Express niet duur hoeft te zijn. Met slechts 1 enkele extra server kan er een middleware geschreven worden die al het werk voor je doet.

Het enige nadeel is dat er een bijkomende leercurve achter zit. Niet enkel voor het maken van de middleware, maar ook voor het instellen van InfluxDB en het opstellen van grafieken in Grafana. Dit onderzoek stelt echter ook alle middelen open naar andere ontwikkelaars, zodat er veel werk bespaard kan worden. Zo kan de middleware gedownload worden via npm, en kan eenmaal de server is opgesteld, het meegegeven configuratiebestand in

de README.md geïmporteerd worden in Grafana waardoor het dashboard automatisch wordt ingesteld. Het hele pakket is open-source, dus de middleware staat open voor extra uitbreidingen.

Dit onderzoek kan vervolledigd worden door requirements van andere bedrijven, die in dezelfde situatie zitten, toe te voegen aan de software. Ook kan het verder worden uitgebreid door sommige elementen te optimaliseren, zoals het wegschrijven naar InfluxDB, meer met asynchrone processen te werken en betere ontwerppatronen te volgen. De software zou meer testen moeten bevatten, en fouten zouden nog sneller moeten worden opgevangen om crashmogelijkheden te beperken. Grafieken van Grafana staan open voor uitbreiding zodat uit dezelfde data nog meer informatie kan gehaald worden. Ondersteuning voor meerdere database-management systemen is zeker en vast een must om deze middleware oplossing bij meer ontwikkelaars populair te maken.

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 (Ancona e.a., 2017): Node.js is sterk afhankelijk van asynchrone en continue programmeerstijl. I/O-bewerkingen

worden uitgevoerd door middel van oproepen naar asynchrone functies waarbij een call-back 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 (Ancona e.a., 2017), 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.. (Ancona e.a., 2017) 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.

Bibliografie

Adopting Agile Testing, a Borland Agile Testing White Paper. (2012). Micro Focus Limited. Verkregen van https://cs.anu.edu.au/courses/comp3120/public_docs/Borland_Whitepaper_v7_AgileTesting.pdf

Ancona, D., Franceschini, L., Delzanno, G., Leotta, M., Ribaudo, M. & Ricca, F. (2017). Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things. In *Proceedings First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017, Turin, Italy, September 18, 2017.* (pp. 27–42). doi:10.4204/EPTCS.264.4

BAT, M. (2016). Infographic on Node.js. Verkregen van https://coderwall.com/p/_ukoeg/infographic-on-node-js

Brandt, L. (2018). Working of middleware.

Chakravorty, T., Chakraborty, S. & Jigeesh, N. (2014). Analysis of Agile Testing Attributes for Faster Time to Market: Context of Manufacturing Sector Related IT Projects. *Procedia Economics and Finance*, *11*, 536–552. Shaping the Future of Business and Society. doi:https://doi.org/10.1016/S2212-5671(14)00219-6

Chandrayan, P. (2017). All About Node.js you wanted to know. Verkregen van https://codeburst.io/all-about-node-js-you-wanted-to-know-25f3374e0be7

Chaniotis, I. K., Kyriakou, K.-I. D. & Tselikas, N. D. (2015). Is Node.js a viable option for building modern web applications? A performance evaluation study. *Computing*, *97*(10), 1023–1044. doi:10.1007/s00607-014-0394-9

Express/Node introduction. (g.d.). Mozilla. Verkregen van https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

Hill, L. (2015). Using InfluxDB + Grafana to Display Network Statistics. *Automation, networking, product management*.

Kantor, I. (2017). Dom tree. Verkregen van https://javascript.info/dom-nodes

Mäntylä, M. V., Khomh, F., Adams, B., Engström, E. & Petersen, K. (2013). On Rapid Releases and Software Testing. In *2013 IEEE International Conference on Software Maintenance* (pp. 20–29). doi:10.1109/ICSM.2013.13

Patel, P. (2018). What exactly is Node.js? Verkregen van https://medium.freecodecamp.org/what-exactly-is-node-js-ae36e97449f5

Pittet, S. (2019). The different types of software testing. *Software testing*. Verkregen van https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

Rangpariya, N. (2019). Evolution of JavaScript: Revolution of Web Development. Verkregen van https://medium.com/swlh/evolution-of-javascript-revolution-of-web-development-5df234a617e6

Rouse, M. (2018). IT monitoring. *How to achieve application performance in ops*. Verkregen van https://searchitoperations.techtarget.com/definition/IT-monitoring

SimilarTech. (g.d.). NodeJs Vs PHP. Verkregen van https://www.similartech.com/compare/nodejs-vs-php

What is JavaScript? (2019). Mozilla. Verkregen van https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction#What_is_JavaScript

Wiley. (2016). What is JavaScript? Verkregen van http://media.wiley.com/product_data/excerpt/88/07645790/0764579088.pdf