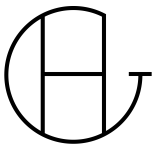


HoGent

Faculteit Bedrijf en Organisatie

De overstap van de Monolithic Application Architecture naar de Microservice Architecture: Waarom, en hoe, wordt dit in de praktijk gedaan?

Pieter Willockx

Scriptie voorgedragen tot het bekomen van de graad van
Bachelor in de toegepaste informatica

Promotor:
Johan Decorte
Co-promotor:
Sander Van Schoote

Instelling: —

Academiejaar: 2015-2016

Tweede examenperiode

Faculteit Bedrijf en Organisatie

De overstap van de Monolithic Application Architecture naar de Microservice Architecture: Waarom, en hoe, wordt dit in de praktijk gedaan?

Pieter Willockx

Scriptie voorgedragen tot het bekomen van de graad van
Bachelor in de toegepaste informatica

Promotor:
Johan Decorte
Co-promotor:
Sander Van Schoote

Instelling: —

Academiejaar: 2015-2016

Tweede examenperiode

Samenvatting

De informatica-business staat niet stil. Dat weet men vanaf men de eerste stappen in deze wereld zet. Een welbekende quote uit de IT-wereld, *The one certainty is change.* Newman (2015), is hier het levende bewijs van. Een ontwikkelaar moet daarom vrede nemen met het feit dat hij in deze branche levenslang zal moeten bijleren.
De IT-business evolueert constant. Gebruikers passen zich aan aan nieuwe systemen die op de markt komen en verwachten van de ontwikkelaar dat hij zich hier ook aan aanpast, en de tijdlijn van applicatieontwikkeling is hier het beste voorbeeld van.
Vroeger was het aanleveren van oplossingen aan de klant een gemakkelijkere zaak. Applicaties werden geschreven in één enkele codetaal en konden op één soort client draaien, in bijna elk geval een desktop-machine. De gehele applicatie werd gebundeld in één consistent pakket: Een monolithische applicatie die alle logica en functionaliteit bevatte. Vandaag de dag hebben we echter te maken met recente ontwikkelingen waar dit soort architectuur niet tegen opgewassen is.
Tegenwoordig bestaan er een groot aantal verschillende soorten clients die elk op een zeer specifieke manier met informatie omgaan. Aan de ene kant heb je desktop-clients die beschikken over relatief grote processoren en een krachtige internetverbinding. Langs de andere kan zijn er de mobiele clients zoals smartphones, tablets, smartwatches en degelijke. Deze soorten toestellen zijn ontwikkeld om draagbaar te zijn, en beschikken dus over relatief gelimiteerde rekenkracht en een potentieel instabiele internetverbinding, afhankelijk van de locatie waar de gebruiker zich bevind.
Hoewel de gebruikers nu verspreid zijn over verschillende soorten clients wil men toch nog steeds een applicatie kunnen ontwikkelen die door zoveel mogelijk van deze gebruikers kan worden geconsumeerd. In dit scenario is de monolithische manier van ontwikkelen verouderd. Er is nood aan een nieuwe architectuur die toestaat om één applicatie te schrijven die voor elk soort toestel dezelfde kwaliteit van gebruik kan bieden.
In dit schrijven wordt een architectuur geanalyseerd die een oplossing biedt voor deze problemen. Een architectuur die in de laatste jaren enorm aan populariteit heeft gewonnen en wordt toegepast in enkele van de grootste bedrijven die vandaag competitief zijn op de markt. De "Microservices Architecture". De grootste voordelen en nadelen worden besproken en vergeleken met de monolithische architectuur, en er wordt getracht om deze stellingen te bewijzen aan de hand van simulaties waarbij parameters zoals latency, geheugengebruik en andere gemeten en vergeleken worden. Verder wordt er ook gekeken naar de mogelijke barrières die momenteel bestaan voor bedrijven die het moeilijk maken om de overstap naar deze nieuwere architectuur te maken.

Voorwoord

Dit schrijven was niet mogelijk geweest zonder de hulp van mijn co-promotor, Sander Van Schoote, werknemer bij Cisco Systems Australia Pty., Ltd. Cisco zet zich actief in voor de ontwikkeling van de Microservices Architectuur. Dit is te zien aan het feit dat ze een volledige "Platform as a Service"(PaaS), Mantl genaamd, ontwikkelden die verder volledig open-source is. Net daarom is het zo voordelig geweest dat ik de hulp had van iemand die te werk gesteld is in dit bedrijf.
Verder wil ik graag mijn promotor, Johan Decorte, bedanken voor zijn nuttige feedback die zonder twijfel gezorgd heeft dat dit paper van hogere kwaliteit is.
Ook bedank ik graag alle docenten die mij in de laatste drie jaar de nodige kennis aanleverden die ik nodig had om dit werk te kunnen leveren, en die ik ook ongetwijfeld later zal nodig hebben in de business.

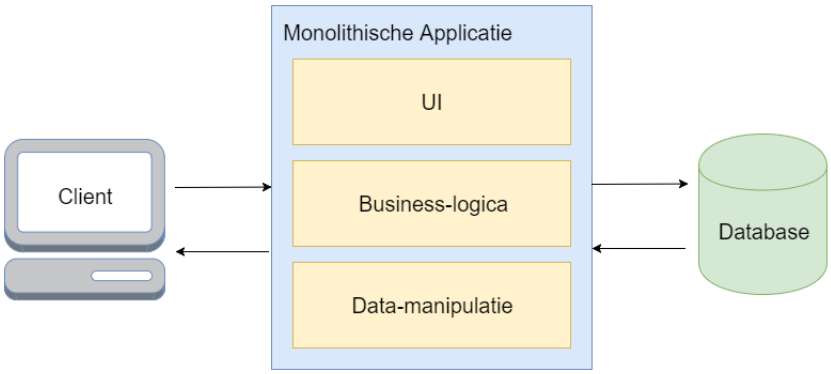
Inhoudsopgave

Hoofdstuk 1

Inleiding

1.1 Wat is een monolithische applicatie?

Definitie 1. *In het vakgebied van de "Software Engineering"beschrijft een monolithische applicatie een single-tier applicatie die zowel de code voor de gebruikersinterface als de code voor data manipulatie omvat in één programma, op één platform. Press (1998)*



Figuur 1.1: Structuur van een monolithische applicatie

Een monolithische applicatie is dus het "standaardmodel"van applicatie. De architectuur hier achter bestaat reeds een lange tijd, en is bij de meeste ontwikkelaars een vertrouwd model. Het is een applicatie die volledig samengebundeld is om te functioneren als één geheel. Dit zorgt voor enkele basiskenmerken die zowel in het voordeel als het nadeel van de ontwikkelaar kunnen spelen.

1.1.1 Samengebundeld in één programma

Eerst en vooral zorgt de samenbundeling ervoor dat de applicatie gemakkelijk te omvatten is. De complexiteit gaat een stuk naar beneden, waardoor het gemakkelijker is voor ontwikkelaars om de structuur te begrijpen. Op dezelfde manier zorgt de samenbundeling er uiteraard voor dat we te maken krijgen met een applicatie die na verloop van tijd enkel groter en groter zal worden. Dit kan er dan weer voor zorgen dat de gehele applicatie moeilijker te omvatten is. Om dit tegen te gaan wordt meestal geopteerd om een duidelijke structuur op te bouwen door klassen en objecten die gelijkaardige functionaliteiten hebben samen te groeperen.
Een monolithische applicatie is zéér gekoppeld, in het opzicht dat logica die van belang is voor een bepaalde functionaliteit kan verspreid zijn over de gehele applicatie. Wanneer wijzigingen gebeuren aan stukken code in de applicatie kan dit dus een invloed hebben op andere delen. Wijzigingen maken aan een monolithische applicatie gaat daarom meestal gepaard aan een uitvoerige testfase om te controleren of alles wel nog naar behoren werkt. Wanneer een team van ontwikkelaars aan een applicatie werkt moet er dus veel gecommuniceerd worden om ervoor te zorgen dat teamleden elkaar niet negatief beïnvloeden met de aanpassingen die ze maken. Uiteraard zijn hier versiebeheer-systemen voor die dat een stuk makkelijker maken.
Over het algemeen is een monolithische applicatie een robuust, betrouwbaar systeem. Services en klassen die binnen het systeem communiceren kennen elkaars definitie en weten welke methodes er beschikbaar zijn. Mits de ontwikkelaar goed werk levert kan er op gebied van data-uitwisseling weinig fout lopen. Dit komt ook de performantie ten goede. De snelheid hangt puur af van de processorkracht van het systeem waarop de applicatie draait, er moet dus geen rekening gehouden worden met andere factoren zoals bijvoorbeeld bandbreedte.

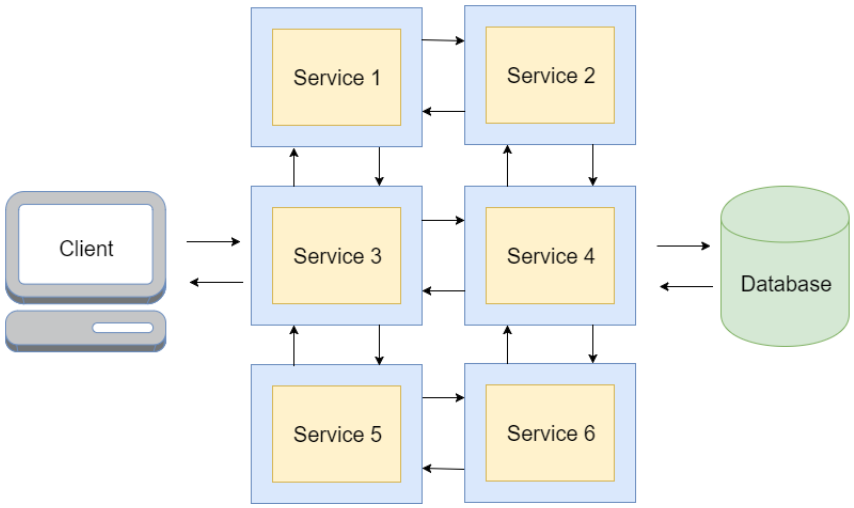
1.1.2 Gebaseerd op één technologie

Omdat het gehele systeem als één programma draait moet het dus van nature uit gebaseerd zijn op één technologie, of beter gezegd, een set van technologieën. Wanneer een team van ontwikkelaars aan een nieuwe monolithische applicatie begint, moet er dus een keuze gemaakt worden over welke technologie zal gebruikt worden. Na analyse van de functionaliteiten die de applicatie moet hebben zal gekozen worden voor een technologie die all-round het meeste geschikt is voor de implementatie hiervan. Dit heeft zowel voordelen als nadelen. Een groot voordeel is dat ontwikkelaars waarschijnlijk vertrouwd zullen zijn met de gekozen technologie, en dus snel efficiënt kunnen werken aan de applicatie. Een nadeel is dat de technologie voor de implementatie van sommige functionaliteiten niet de meest performante oplossing zal zijn. De keuze staat echter vast, en hier kan tijdens de ontwikkeling van de applicatie niet meer van afgeweken worden.

Een monolithische applicatie is dus een groot maar robuust systeem, gebaseerd op één technologie, die voor ontwikkelaars gemakkelijk begrijpbaar is. Door zijn grootte kan het voor een buitenstaander echter wel moeilijk zijn om de structuur te begrijpen. Door de afhankelijkheid van de gekozen technologie kan het zijn dat voor sommige delen van de applicatie niet voor de meest geschikte implementatie kan gezorgd worden.

1.2 Wat is een microservice?

Definitie 2. *De Microservice Architectuur is een methode voor het ontwikkelen van software-applicaties als een verzameling van onafhankelijke, kleine, modulaire services waarin elke service in functie van een specifieke business-doelstelling staat, en communiceert via een simpele, goed gedefinieerde interface. Rouse (2015)*



Figuur 1.2: Structuur van een microservice-applicatie

In het licht van de probleemstelling die in de samenvatting aan bod kwam was er dus een nood aan een nieuwe soort architectuur. Een architectuur die ontwikkelaars in staat stelde om applicaties aan te leveren die door meerdere verschillende soorten gebruikers konden geconsumeerd worden. De Microservices Architecture biedt hiervoor een oplossing.
In de praktijk is een microservice dus een kleine, autonome service die volledig individueel draait op een systeem, en samenwerkt met andere microservices om een geheel van functionaliteit te vormen. U merkt 2 kernwoorden op: klein en autonoom. Hier gaan we verder op in.

1.2.1 Klein, en met één duidelijke verantwoordelijkheid

De grootte van een service is moeilijk te meten. Wanneer bijvoorbeeld het aantal lijnen code wordt opgeteld, kan men meestal nog steeds niet zeggen of het gaat over een grote of een kleine service. Bepaalde codetalen zijn namelijk expressiever dan anderen en krijgen dezelfde functionaliteit uitgeschreven in slechts een fractie van de lijnen code. Toch moet een microservice klein zijn. De "grootte"van een microservice wordt vooral bepaald door de functionaliteit die het bevat. Er wordt gestreefd naar een service die één duidelijke verantwoordelijkheid heeft en in die zin mag het slechts functionaliteit bevatten die rechtstreeks inspeelt op deze verantwoordelijkheid. Maar waar liggen de grenzen van deze verantwoordelijkheid? Welke code is acceptabel om op te nemen in een service en welke code treedt over de grenzen hiervan?
Hier is geen eenduidig antwoord voor. De ontwikkelaar of het team van ontwikkelaars bepaalt zelf deze grenzen, maar deze zijn echter meestal intuïtief. We nemen als voorbeeld een inlog-systeem van een applicatie voor het quoteren van films. Een service voor het inloggen en uitloggen van gebruikers op het systeem zal in dit geval verantwoordelijk zijn voor het ophalen van de gebruikersgegevens aan de hand van de gebruikersnaam en het wachtwoord, het creëren van een sessie-token voor de gebruiker, de gebruiker op ingelogd/uitgelogd zetten in de database en dergelijke. Moet er echter een functionaliteit zijn die de onlangs gequoteerde films van de gebruiker moet ophalen nadat hij is ingelogd, dan laten we deze functionaliteit beter over aan een andere service. Op deze manier creëren we verschillende loshangende services die compact zijn en een duidelijke verantwoordelijkheid hebben.
Een andere manier om te bepalen of een service klein genoeg is, is door te kijken hoeveel ontwikkelaars nodig zijn om ze te onderhouden. Wanneer een team van ontwikkelaars werkt via de microservice architectuur is het logisch dat het hele team opgesplitst wordt in kleinere teams die elk aan hun eigen service werken. Wanneer men ziet dat een service te groot wordt en niet meer efficiënt kan beheerd worden door één team, dan is dit een reden om na te denken over een opsplitsing van deze service in kleinere en eenvoudigere delen.

1.2.2 Autonoom

Een microservice moet autonoom zijn in enkele opzichten. Eerst en vooral moet ze gezien worden als een aparte entiteit. Een stuk code dat op een aparte "machine"draait, of een apart systeemproces is. Een microservice kan dus in dit opzicht gezien worden als een aparte, kleine applicatie die deel uitmaakt van een groter geheel.
Dat een microservice autonoom is, wil niet zeggen dat deze volledig afgesloten van de andere services zal functioneren. Microservices communiceren namelijk met elkaar voor het verkrijgen van data die buiten hun verantwoordelijkheidsveld ligt, maar die wel van

belang is voor het uitvoeren van hun eigen taak. Zo zal een login-service bijvoorbeeld contact zoeken met een encryptie-service wanneer een ingegeven wachtwoord moet vergeleken worden met het geëncrypteerde wachtwoord dat uit de databank komt.
Wanneer een microservice crasht door een systeemfout, moeten er systemen zijn ingebouwd die ervoor zorgen dat de andere microservices kunnen blijven verdergaan, eventueel met mindere functionaliteit. In dit opzicht kan er ook over autonomie gesproken worden.

1.3 De microservice architectuur als een geheel

Nu een duidelijk beeld gevormd is van wat een microservice op zichzelf is, kan besproken worden hoe deze zich gedraagt in een volledig systeem. Hoe krijgt men hetzelfde resultaat dat men zou hebben met een monolithische applicatie, maar dan met een microservice structuur. Welke systemen zijn er die de services koppelen, aanspreken, laten samenwerken met elkaar en die de data samen verpakken om een coherent resultaat terug te geven aan de gebruiker?
In een monolithische applicatie is communicatie tussen verschillende services, of in dit geval methodes, vanzelfsprekend. Er wordt een instantie van een klasse aangemaakt en vervolgens kunnen zijn methoden aangeroepen worden. Communicatie tussen microservices daarentegen is niet zo vanzelfsprekend. Aangezien ze elk als een aparte instantie draaien kan er niet zomaar binnen het proces gecommuniceerd worden.
De communicatie tussen microservices gebeurd via het netwerk. Deze services maken gebruik van een mechanisme dat "Inter-process Communication"(IPC) genoemd wordt. Wanneer een bepaalde service informatie nodig heeft van een andere service, zal hij een "request"sturen en in de meeste gevallen ook een "response"terug verwachten. De communicatie binnen een microservice architectuur is dus te vergelijken met een reeks calls naar een "Application Programming Interface"(API). Hier wordt verder iets dieper op ingegaan.
Communicatie tussen de services gebeurd via IPC, maar hoe communiceert een gebruiker met de applicatie?
Hier zijn 2 technieken voor. Een ontwikkelaar kan er voor kiezen om de gebruikers rechtstreeks met de verschillende microservices te laten communiceren. Requests die de gebruiker verstuurt naar de services komen dus op dezelfde manier aan als die verstuurd door andere services, en worden ook op dezelfde manier verwerkt. Deze manier van werken is in principe mogelijk, maar wordt afgeraden. Dit omdat er geen onderscheid gemaakt wordt tussen communicatie met een gebruiker en communicatie met een service, terwijl we eerder bespraken dat er wel degelijk nood is aan onderscheid tussen verschillende clients.
Een betere implementatie is daarom het gebruik van een API-gateway. Kort beschreven is dit een centraal punt waar alle requests van de client binnenkomen. Er kunnen meer-

dere API-gateways geïmplementeerd worden voor verschillende soorten clients, die de opgevraagde data elk op een specifieke manier behandelen om de data te optimaliseren alvorens ze terug naar de client gestuurd wordt. Ook hier wordt verder iets dieper op ingegaan.

1.4 Samengevat

Een microservice is dus een klein, autonoom proces dat als een aparte instantie draait in een groter geheel. Er bestaat een duidelijke scheiding van verantwoordelijkheden, waarbij iedere service enkel en alleen zorgt voor zijn eigen taken en communiceert met andere services wanneer hij data nodig heeft dat buiten zijn verantwoordelijkheidsveld valt. Communicatie gebeurd via requests en responses over het netwerk, soortgelijk aan communicatie met een API.

1.5 Probleemstelling en Onderzoeksvragen

De microservice architectuur wordt tot op heden vooral gebruikt bij grotere bedrijven. Dit komt omdat de extra complexiteit die de architectuur met zich meebrengt kleinere bedrijven afschrikt om de overstap te maken. Teams moeten dynamischer te werk gaan en moeten opgesplitst worden in kleinere teams om efficiënt in een microservice omgeving te werken. De meeste van deze bedrijven houden zich dus nog steeds vast aan het monolithische model wanneer ze hun applicaties ontwikkelen. In een wereld die meer en meer zal bestaan uit clients van verschillende soorten en formaten, elk met hun eigen manier van omgang met data, kan dit ervoor zorgen dat software bedrijven oplossingen ontwikkelen die niet efficiënt kunnen geconsumeerd worden door elk soort client.
Kan er een gulden middenweg gevonden worden die het gemakkelijker maakt voor bedrijven om deze overstap te wagen, en zo niet, **zijn er maatregelen die kunnen getroffen worden die de overstap van monolithic naar microservices in de toekomst gemakkelijker maakt?**
Verder, is de microservice architecture wel de perfecte oplossing voor het maken van applicaties die door een grote variëteit aan clients kunnen geconsumeerd worden? Met andere woorden, **zijn microservices wel een vooruitgang op alle vlakken, vergeleken met het monolithische model?** Het is duidelijk dat er enkele nadelen verbonden zijn aan het werken volgens deze architectuur. **Zijn er oplossingen te vinden die deze nadelen kunnen doen verdwijnen?**

Hoofdstuk 2

Methodologie

In eerste instantie wordt de vergelijking tussen het microservice model en het monolithisch model gemaakt. De voordelen en nadelen worden onderzocht en vergeleken met elkaar.
Om deze voordelen en nadelen wel degelijk te bewijzen zal een simulatie uitgevoerd worden die beide architecturen in een vergelijkbaar scenario test. Op deze simulatie worden tests uitgevoerd die bepalen hoe goed de architecturen omgaan met dataverkeer van verschillende groottes. Hieruit kunnen de knelpunten van beide architecturen gehaald worden. De knelpunten worden geanalyseerd en er wordt gezocht naar mogelijke oplossingen hiervoor.
De uiteindelijke uitkomst: Is het voordelig voor kleinere bedrijven om de overstap naar een microservice architecture nu al te maken? Zijn er maatregelen die nu al kunnen getroffen worden om de overstap later gemakkelijker te maken? Een antwoord op deze vragen.

Hoofdstuk 3

De Microservice Architectuur geanalyseerd

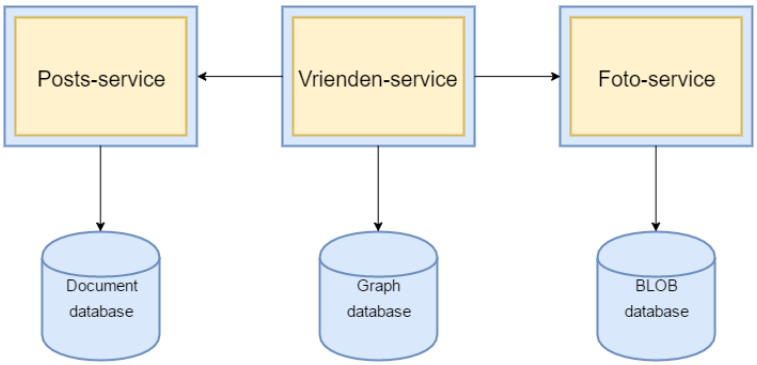
De microservice architectuur werd ontwikkeld om een antwoord te bieden op enkele belangrijke ontwikkelingen in de informatica-wereld. Meer en meer toestellen zijn tegenwoordig verbonden met het internet, wat enkel zorgt voor extra dataverkeer. Applicaties moeten performant blijven, ongeacht het aantal gebruikers die tezelfdertijd verbinding willen leggen en ongeacht de grootte van de data die wordt opgevraagd.
Er moet rekening gehouden worden met het feit dat niet elke client dezelfde data wilt. Het kan bijvoorbeeld nuttig zijn voor een webbrowser om data terug te krijgen als ruwe HTML-tekst, maar daarmee kan een mobiel toestel die een native app draait niets aanvangen.
De extra complexiteit die een architectuur, toegespitst op deze problemen, met zich meebrengt mag zijn tol niet eisen op de ontwikkelaars. Het ontwikkelingsproces moet soepel verlopen en ontwikkelaars moeten in staat zijn om snel te reageren op problemen die zich ongetwijfeld zullen voordoen tijdens dit proces.
De microservice architectuur is gebaseerd op enkele pijlers die beschrijven hoe deze problemen kunnen aangepakt worden.

3.1 Belangrijkste pijlers

3.1.1 Technologie-heterogeniteit

De dag van vandaag bestaat er een groot aantal codetalen. Elk daarvan heeft zijn voordelen en nadelen, en er zijn verschillende scenario's waarin een ontwikkelaar een voorkeur heeft voor een bepaalde taal om een bepaalde functionaliteit te implementeren. Monolithische applicaties zijn bijna altijd geschreven in één enkele codetaal waardoor ontwikkelaars geen keuze krijgen anders dan in die taal te programmeren. Enerzijds is

dit een voordeel, want een applicatie die slechts in één codetaal geschreven is laag in complexiteit. Anderzijds is dit een nadeel, want voor sommige functionaliteiten in de applicatie zijn er misschien andere codetalen die een veel performantere en misschien wel eenvoudigere implementatie kunnen garanderen. Omdat een monolithische applicatie één geheel is, is dit echter onmogelijk.
Bij microservices is dit anders. Een microservice applicatie bestaat uit meerdere individuele services. Deze kunnen elk in een andere codetaal geschreven worden en het geheel zal nog steeds zonder problemen functioneren. Dit biedt een zekere vorm van flexibiliteit, want voor elke functionaliteit kan nu de meest performante oplossing gezocht worden. We nemen als voorbeeld een sociaal netwerk-platform zoals Facebook of Twitter. Een dergelijke applicatie heeft te maken met miljoenen gebruikersaccounts, posts, foto's, volgers of vrienden en andere. Al deze informatie bijhouden moet op een zo performant mogelijke manier gebeuren. Het is bijvoorbeeld voordelig om alle gebruikersaccounts in een graph-database te bewaren gezien de hoge graad van interconnectie, maar misschien kunnen de posts van deze gebruikers op een performantere manier in een document-database bewaard worden. De structuur van een microservices systeem laat ons toe om deze keuze te maken.



Figuur 3.1: Het gebruik van meerdere soorten databanken in een microservice-architectuur

Een ander voordeel is dat nieuwe technologieën veel gemakkelijker kunnen geïntegreerd worden in de applicatie. Wanneer men een nieuwe technologie wil uittesten op een monolithische applicatie zal dit impact hebben op een groot deel van het systeem, wat ontwikkelaars kan afschrikken. Als gevolg stellen we vast dat een monolithische applicatie meestal voor altijd zal blijven draaien op de technologieën die werden gekozen aan het begin van zijn ontwikkeling. Zulke applicaties zijn snel verouderd.
Aangezien een microservice applicatie bestaat uit verschillende services kan een nieuwe technologie uitgetest worden op één enkele service zonder dat dit een impact heeft op de rest van de applicatie. De mogelijkheid om snel nieuwe technologieën op te pikken en te integreren in een applicatie kan een belangrijk voordeel zijn voor bedrijven die

competitief willen blijven op de markt.

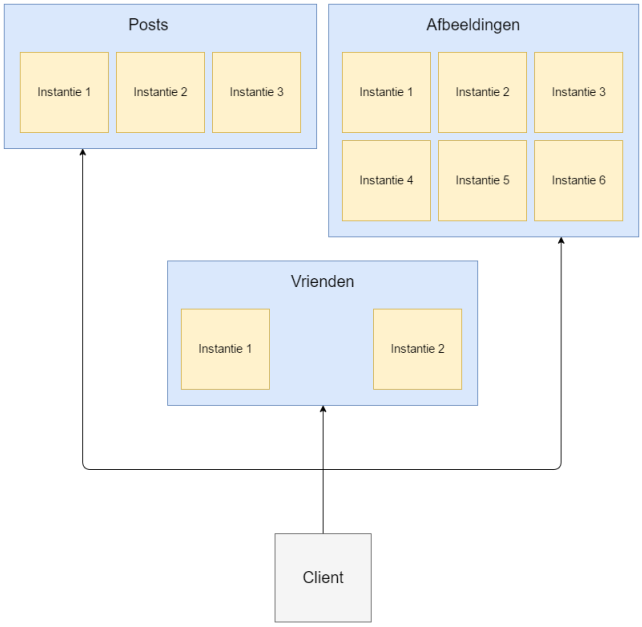
3.1.2 Afhandeling van systeemfouten

Een belangrijk onderdeel bij het ontwikkelen van een applicatie is om ervoor te zorgen dat de kans op systeemfouten zo klein mogelijk wordt gemaakt. Wanneer een systeemfout voorkomt, valt het hele systeem in elkaar en in de meeste gevallen moet een manuele heropstart gedaan worden. Wanneer dit zich voordoet bij een applicatie die live draait voor de gebruikers kan men spreken van een klein rampscenario. Uiteraard kunnen niet alle fouten vermeden worden. Wanneer zich bijvoorbeeld een stroompanne of kortsluiting voordoet kan de infrastructuur falen en zal een applicatie hoe dan ook stoppen met functioneren. Toch zijn er maatregelen die kunnen genomen worden om een applicatie zo "failure-proof"mogelijk te maken.
Een monolithische applicatie is in deze gevallen veel kwetsbaarder dan een microservices systeem. Wanneer zich in een monolithische applicatie een kritische systeemfout voordoet zal het hele systeem falen, ongeacht waar de fout gebeurde. Dit komt uiteraard omdat de hele applicatie samengebundeld is en op één instantie draait. Om zulke gebeurtenissen tegen te gaan kan geopteerd worden om meerdere instanties van dezelfde applicatie te draaien. Hierdoor kan overgeschakeld worden op een andere instantie bij fouten in het systeem. Dit is echter een vrij inefficiënte oplossing.
Wanneer we microservices bekijken - we weten dat deze bestaat uit verschillende apart draaiende services - zien we dat deze voorvallen vanzelf al veel minder kritiek zijn voor het gehele systeem. Doet er zich een fout voor in een bepaalde service, dan sluit deze zich automatisch af. Andere services kunnen blijven voortgaan omdat deze op een aparte instantie draaien. Wanneer we nu gebruik maken van een soort ërror-handler"die de wacht houdt over het gehele systeem, kunnen fouten in services geïsoleerd worden en kunnen deze gecontroleerd afgesloten worden. Andere services worden dan gewaarschuwd dat bepaalde functionaliteiten ontoegankelijk zijn, maar deze kunnen wel blijven doorgaan, hetzij met mindere functionaliteit. Services die cruciaal zijn voor het functioneren van de applicatie kunnen gedupliceerd worden zoals dat zou gedaan worden met een monolithische applicatie. We zien nu dat een microservice applicatie op een even performante manier met fouten kan omgaan vergeleken met een monolithische applicatie, door slechts een fractie van de infrastructuur te gebruiken.

3.1.3 Schaalbaarheid

Een monolithische applicatie schalen is vanzelfsprekend. Men dupliceert de gehele applicatie op een nieuwe instantie. Hierdoor heeft het geheel twee keer zoveel capaciteit. Dit is echter een inefficiënte oplossing. In vele gevallen zijn er slechts enkele functionaliteiten in de applicatie die extra vermogen nodig hebben. Toch moet het gehele systeem geschaald worden om dit mogelijk te maken.

Dankzij de losgekoppelde natuur van microservices wordt dit uiteraard een stuk simpeler. Enkel die service die meer vermogen nodig heeft moet geschaald worden terwijl de rest van het systeem op dezelfde grootte kan blijven verder draaien. Het voordeel hiervan kan eenvoudig aangetoond worden.
We nemen opnieuw het voorbeeld van een sociaal netwerk. Om het simpel te houden bestaat deze uit drie services die elk hun eigen verantwoordelijkheid hebben. Eén service staat in voor het ophalen van de vrienden van een gebruiker. Een tweede service staat in voor het inladen van de posts die de gebruiker en zijn vrienden gemaakt hebben. De laatste service houdt zich bezig met het ophalen van afbeeldingen die de gebruiker en zijn vrienden op hun profiel hebben. De "vrienden-service"zal simpelweg URL's naar de verschillende profielen doorsturen naar de client, wat weinig bandbreedte zal innemen. De "posts-service"moet elke post als een string doorsturen naar de client. Gezien een gebruiker typisch meer dan één post op zijn profiel heeft zal deze service meer capaciteit nodig hebben dan de vorige. Als laatste de "foto-service"die een groot aantal afbeeldingen zal moeten doorsturen in zijn responses. Afbeeldingen pakken uiteraard meer geheugen in dan strings, dus we zien hier dat deze service nog meer capaciteit zal nodig hebben om op dezelfde kracht te kunnen functioneren. Uit deze stellingen kunnen we afleiden dat bepaalde services meer geschaald zullen moeten worden dan anderen. Op onderstaande afbeelding is een mogelijke schaalbaarheids-strategie te zien die toepasbaar is op dit voorbeeld.



Figuur 3.2: Voorbeeld van een mogelijke strategie voor het schalen van microservices

3.1.4 Uitrolbaarheid

Als ontwikkelaar heb je een belangrijk voordeel in de business wanneer je updates en bugfixes sneller kan laten uitrollen dan de competitie. Dit is echter niet altijd gemakkelijk te verwezenlijken. Wanneer een deel van de code van een monolithische applicatie een update moet krijgen, is het niet zeker of deze aanpassingen of toevoegingen gevolgen zullen hebben op de rest van het systeem. Er moet een uitgebreide testfase voorgaan aan het uitrollen van updates.
Een ander knelpunt is het feit dat wanneer er updates moeten uitgebracht worden, de hele applicatie die live op de servers draait offline moet worden gehaald zodat de aanpassingen geïntegreerd kunnen worden. Dit is een riskante onderneming, omdat je als uitgever van software niet wil dat je klanten voor langere tijd geen gebruik kunnen maken van de applicatie. Daarom is er gemakkelijk de neiging om veel updates samen te bundelen, en in één groot pakket uit te brengen. Dit zorgt ervoor dat de applicatie slechts voor korte tijd onbruikbaar is. Op zich is dit een goede strategie, maar ook hier zijn nadelen aan verbonden. Stel dat er per toeval toch een bug in de code van de update geslopen is. Omdat de update in een groot pakket uitgebracht is, is het niet altijd evident om de oorzaak van de fout op te sporen. Hierdoor kunnen live applicaties een langere tijd te kampen hebben met een fout in het systeem, terwijl de ontwikkelaars deze proberen op te sporen en op te lossen.
Microservices zijn losgekoppeld van elkaar, en zijn elk relatief kleine stukken code. Dit maakt updaten gemakkelijk. Een service kan geüpdatet worden zonder dat het hele systeem opnieuw moet gecompileerd worden. Het risico van het uitbrengen van de update is een stuk lager dan bij het monolithische systeem, en in veel gevallen kan de gebruiker zelfs gebruik blijven maken van de meeste functies van de applicatie terwijl een update voor een bepaalde service wordt uitgerold. Ontwikkelaars kunnen nu veel frequenter verbeteringen uitbrengen, en als gevolg daarvan zal elke update van kleinere schaal zijn.
Rolt er toch een bug uit met een bepaalde update, zal deze veel gemakkelijker op te sporen zijn. Wanneer men weet tijdens welke update de bug live kwam in het systeem, heeft men de oorzaak van de fout eigenlijk al gevonden. Zo kunnen fouten ook veel sneller uit het systeem gehaald worden.
Over het algemeen is dus één van de belangrijkste troeven van de microservice architectuur. Hoe sneller updates kunnen uitrollen, hoe actueler een applicatie zal zijn. Dit kan voor een belangrijke voorsprong op de concurrentie zorgen.

3.1.5 Flexibele teamstructuur

Met een groot team aan één applicatie werken kan de productiviteit verlagen. In een team van ontwikkelaars moet veel communicatie zijn, vooral wanneer deze ontwikkelaars tegelijkertijd aan dezelfde functionaliteiten moeten werken. Hoe groter het team,

hoe meer dat er moet gecommuniceerd worden. Dit is niet altijd doenbaar.
Het is voordelig om teams op te splitsen in kleinere sub-teams die elk aan aparte stukken code werken. Nog beter is het wanneer deze teams functionaliteiten kunnen implementeren zonder dat deze een invloed hebben op het werk van andere teams.
De structuur van een microservice systeem leent zich uitermate tot het werken in sub-teams. Elk team werkt aan zijn eigen service en omdat de services onafhankelijk van elkaar zijn en bovendien losgekoppeld zijn van elkaar, zullen updates voor een bepaalde service geen invloed hebben op de al bestaande functionaliteit van andere services.

3.1.6 Combineerbaarheid

Een ander belangrijk streefdoel bij het efficiënt ontwikkelen van applicaties is het hergebruik van code. Dit lijkt misschien vanzelfsprekend, een ontwikkelaar kan code "hergebruiken"door deze te kopiëren en te plakken van andere gelijkaardige stukken code, maar dit is geen efficiënte oplossing. Het wordt zelfs afgeraden om veel aan "copy-pasting"te doen in een project omdat dit onnodig veel extra lijnen toevoegt aan een applicatie die, als ze monolithisch is, al van grote omvang is. Er bestaan verschillende mogelijkheden die hergebruik van code toestaan zoals bijvoorbeeld het gebruik van abstractie en het samenkoppelen van klassen in interfaces. Binnen een monolithische applicatie zijn dit zowat de voornaamste manieren, maar niet de enigen.
Wanneer men een bepaalde functionaliteit heeft die kan worden gebruikt in meerdere applicaties, kan gekozen worden om deze te compileren tot een "library". Libraries geven ontwikkelaars de mogelijkheid om bestaande functionaliteit te integreren in hun werk in plaats van het allemaal zelf te moeten herschrijven. Een goede manier om code te hergebruiken, maar ook deze komt niet zonder nadelen of restricties op hun gebruik.
Eerst en vooral zullen de libraries die kunnen geïntegreerd worden ook afhangen van de gekozen technologie waar de applicatie op gebaseerd is. Libraries draaien in principe op dezelfde instantie als de gehele applicatie, en worden door dezelfde compiler omgezet. Een applicatie zal dus enkel kunnen profiteren van een library die dezelfde technologie ondersteunt.
Echte flexibiliteit haal je ook niet echt uit het gebruiken van libraries. Hergebruik van code is één ding, maar een ander probleem dat we willen oplossen is het feit dat code apart moet kunnen gewijzigd en uitgerold worden. Wanneer aanpassingen gemaakt worden in de library, en men wilt dat deze aanpassingen kunnen gebruikt worden in de applicatie, moet deze nog steeds helemaal opnieuw gecompileerd en uitgerold worden. Een library is sterk gekoppeld aan de applicatie die ze gebruikt, en andersom.
Is er echter geen betere manier om code te hergebruiken, zonder de nadelen van deze sterke koppeling?
Alweer toont de losgekoppelde structuur van een microservice applicatie zijn voordelen. Aangezien services op zichzelf al een werkend geheel zijn, kunnen ze hergebruikt

worden zonder dat daar enig extra werk voor nodig is. Eén service kan bijvoorbeeld geconsumeerd worden door meerdere applicaties, aangezien het aanroepen van een service slechts een kwestie van een request te sturen is. Dit geeft ontwikkelaars de mogelijkheid om applicaties als het ware op te bouwen uit verschillende al bestaande services, wat de ontwikkelingstijd aanzienlijk naar omlaag kan drijven voor een nieuw project.
Het kan zelfs nog een stap verder gaan door services rechtstreeks publiek toegankelijk te maken. Dit wordt tegenwoordig meer en meer gedaan. Wanneer een andere ontwikkelaar gebruik maakt van een service die over het internet beschikbaar is, noemen we dit het gebruik van "third-party services".
Services kunnen verder voor elk soort client op andere manieren gekoppeld worden, om de data die aan deze clients aangeleverd wordt te optimaliseren voor hun gebruik. We nemen opnieuw het voorbeeld van Facebook. Wanneer een gebruiker via een webbrowser naar deze website surft, krijgt hij de gebruikelijke activiteitenfeed te zien. Naast deze activiteitenfeed staan tegenwoordig suggesties voor groepen, en gebruikers waar men mogelijks mee bevriend wil worden. Het is logisch dat dit zal geregeld worden door een aparte service, die rekening houdt met enkele algoritmen voor het bepalen van deze suggesties. Deze service zal dus aangeroepen worden wanneer de applicatie merkt dat een gebruiker via de webbrowser werkt.
Op een smartphone daarentegen, is er niet genoeg plaats op het scherm om deze suggesties te tonen. Wanneer een gebruiker dus via een smartphone communiceert met de applicatie, zal er geen gebruik gemaakt worden van de verzoek-service. Services worden op deze manier samengebundeld om de optimale data voor een bepaalde client terug te geven aan de gebruiker.

3.1.7 Vervangbaarheid

Applicaties of systemen upgraden naar nieuwere versies is niet altijd evident. Stel bijvoorbeeld dat een organisatie gebruik maakt van een server-applicatie om een bepaald proces in het bedrijf te automatiseren. Deze applicatie zal waarschijnlijk cruciaal zijn om efficiënt te kunnen werken binnen dit bedrijf, waardoor er integraal gebruik van zal worden gemaakt. Het is echter zo dat deze applicatie na verloop van tijd zal verouderen. Niemand wil het echter op zich nemen om de applicatie te vernieuwen. Ze is namelijk erg groot, en monolithisch, en zal dus veel tijd nodig hebben om geüpdatet te worden. Verder is het een risicovolle operatie omdat ze zo cruciaal is voor het bedrijf, waardoor mogelijke bugs van een nieuwe versie van dit systeem voor grote problemen zouden kunnen zorgen. Ontwikkelaars zitten nu vast met een systeem dat steeds ouder en ouder wordt.
Het vervangen of zelfs compleet verwijderen van een microservice is echter een veel minder risicovolle onderneming. Omdat ze zo klein van omvang zijn kunnen ze in de meeste gevallen op enkele weken compleet herschreven worden. Verder zijn bugs in

het systeem niet zo een groot probleem, omdat er veel minder afhangt van deze ene kleine service.
Deze flexibiliteit zorgt ervoor dat microservice applicaties constant actueel kunnen gehouden worden. Nieuwere versies van technologieën zijn vaak performanter. De vlotte integratie van deze technologieën zorgt in zijn geheel dus ook voor een meer performant systeem.

3.2 Nadelen van de Microservice Architectuur

De kracht van microservices kan worden afgeleid uit de opsomming van de pijlers uit het vorige hoofdstuk. Ze werd ontwikkeld als reactie op het al maar groeiende aantal verschillende clients en de nood aan schaalbare applicaties, en blinkt daar ook in uit.
Het feit dat ze met een speciaal doel in het achterhoofd werd ontwikkeld, wilt echter zeggen dat er onvermijdbaar ook scenario's zijn waar de architectuur het niet zo goed zou doen. De architectuur heeft dus uiteraard ook nadelen en struikelpunten,en een team van ontwikkelaars dat niet voorbereid is op deze struikelpunten zal in bijna elk geval minder efficiënt werken dan een team dat met de monolithische architectuur werkt.

3.2.1 Extra werklast voor de ontwikkelaar

Een monolithische applicatie beheren is vrij simpel, zelfs als ze substantieel is. Ze draait op een cluster van servers, in sommige gevallen verbonden met enkele monitoring tools, en kan bestaan uit verschillende instanties wanneer nodig. Diezelfde applicatie, herwerkt tot een microservice applicatie, kan al snel heel erg complex worden. Het kan hier al snel gaan over tientallen verschillende services die elk apart uitgerold, getest, gestopt en gestart moeten worden. Als we schaalbaarheid in acht nemen, zal het aantal eigenlijke instanties nog groter zijn. In die applicatie zitten ook nog tal van services die het hele systeem draaiende houden zoals "load-balancers"die het binnenkomend verkeer verdelen over de verschillende instanties, ërror-handlers"die instanties afsluiten bij fouten, services die de activiteiten van de applicatie wegschrijven in logbestanden, enzovoort.
Dit hele systeem beheren is een grote taak. Voor elk van deze services moet gezorgd worden dat ze actief blijven of heropgestart worden bij fouten, voldoende ruimte hebben voor informatie op te slagen, niet in een "deadlock"komen te zitten bij concurrente transacties op de databank. Om deze hele opdracht draagbaar te maken voor één ontwikkelaar of een klein team van ontwikkelaars worden er meestal programma's voorzien die een deel van deze taken automatiseren.
Omdat de microservice architectuur echter nog niet zo oud is, zijn er nog niet veel

kant-en-klare programmapakketten die al deze functionaliteiten bieden. Zo zijn ontwikkelaars meestal gedwongen om zelf een deel van deze programma's te schrijven. Dit is zonder twijfel een extra last die het productieproces kan vertragen.

3.2.2 Ervaren ontwikkelaars zijn een must

Het onderhouden van een microservice applicatie brengt dus veel extra taken met zich mee. Om dit vlot te laten verlopen moet het team van ontwikkelaars zeer vertrouwd zijn met de architectuur. Daarbovenop moet de communicatie binnen het team ook voldoende zijn. Hoewel er aan een service kan gewerkt worden zonder dat dit invloed heeft op andere services, moeten de communicatiepunten tussen de services dezelfde blijven om het systeem draaiende te houden. Wanneer deze communicatiepunten toch moeten aangepast worden, moet dit gecommuniceerd worden naar alle teamleden die hier belang bij hebben. Een ontwikkelaar moet zo dus ook het hele systeem kennen om te weten op welke services deze verandering een impact zal hebben.
Verder moet het team bestaan uit ontwikkelaars die ervaring hebben met verschillende programmeertalen aangezien de microservice architectuur technologie-heterogeen is. Het is uiteraard voordelig om als ontwikkelaar in zo een team ervaring te hebben met verschillende technologieën, zodat kan bijgesprongen worden bij de ontwikkeling van een service wanneer nodig.
Een ander voorbeeld waarbij ervaren ontwikkelaars belangrijk zijn is bij de rol van database-administrator. Bij een monolithische applicatie die bijvoorbeeld gebruik maakt van een MySQL-database, is het voldoende om een administrator aan te stellen die bekend is met het MySQL-systeem. Bij een microservice applicatie daarentegen kan het goed mogelijk zijn dat er gebruik gemaakt word van verschillende soorten databases die geoptimaliseerd zijn voor hun corresponderende services. De database-administrator moet in dit geval een uitgebreidere kennis hebben van de verschillende soorten databases om deze efficiënt te kunnen beheren. In dit geval wordt deze taak dus een stuk moeilijker.
Ontwikkelaars met dit profiel zijn niet altijd even gemakkelijk te vinden. Als teamleider of werkgever kan dit dus een groot struikelblok zijn.

3.2.3 Afhankelijkheid bij uitrollen van updates

Uitrolbaarheid wordt gezien als een van de belangrijke pijlers van de microservice architectuur. Updates kunnen uitgegeven worden zonder dat het hele systeem opnieuw moet opgestart worden. In de meeste gevallen is dit zo, maar soms is een update die meerdere services beïnvloed noodzakelijk. Wanneer bijvoorbeeld een communicatiepunt van een service moet worden aangepast, moet deze aanpassing doorgevoerd worden naar alle services die communiceren met deze service. In dit geval is het apart uitrollen niet mogelijk. Wanneer de service met het nieuwe communicatiepunt apart

zou worden uitgebracht, wordt deze onbruikbaar omdat de andere componenten van de applicatie nog steeds gebruik maken van het oude communicatiepunt. Het uitrollen zal dus gecoördineerd moeten gebeuren, zodat het systeem op geen enkel moment fout communiceert.
Het gecoördineerd uitrollen is niet altijd even vanzelfsprekend en voegt extra risico's toe aan het proces. Een team dat hier niet op voorbereid is kan zijn eigen applicatie op deze manier laten stuk lopen, en op dit moment is het niet altijd even gemakkelijk om de oorzaak van de fout terug te vinden.

3.2.4 Genoodzaakt hergebruik van code

In sommige gevallen is het zo dat een bepaald stuk code moet geïmplementeerd worden in verschillende componenten van de applicatie. Voor een monolithische applicatie is dit vanzelfsprekend. De code wordt in een publieke methode gegoten en kan dan via verwijzing gebruikt worden op de verschillende punten waar deze functionaliteit nodig is.
Bij een microservice applicatie bestaan er enkele mogelijkheden om dit te bereiken. Een eerste mogelijkheid is het aanmaken van een nieuwe service die dan wordt aangeroepen door de services waar de functionaliteit nodig is. Een andere mogelijkheid is het primitief kopiëren en plakken van de code in de verschillende services. Ook zou er gebruik kunnen gemaakt worden van een gedeelde library die deze functionaliteit bevat.
Geen van deze mogelijkheden is echter een goede oplossing. Je kan niet voor elke gedeelde functionaliteit een nieuwe service aanmaken want dat zorgt op de lange termijn voor een systeem dat bestaat uit onnodig veel services. Ook krijgt de applicatie te maken met extra "latencyömdat de functionaliteit nu over het netwerk moet worden aangeroepen. Het kopiëren en plakken van de code maakt komaf met het "latency-probleem, maar ook dit is maar een gedeeltelijke oplossing. Wanneer de toegevoegde functionaliteit moet getest worden, moet dit namelijk op verschillende plaatsen gebeuren. Verder zal de code dus ook op verschillende plaatsen moeten worden aangepast wanneer de functionaliteit moet bijgewerkt worden. Het gebruik van libraries wordt al helemaal afgeraden. Aangezien ze technologie-gebonden zijn kunnen ze waarschijnlijk niet in elke service gebruikt worden. Ook voegt het gebruik van libraries koppeling toe tussen de verschillende services die ze gebruiken, wat één van de belangrijkste voordelen van de microservice architectuur teniet doet.
In de praktijk zal de code meestal gekopieerd en geplakt worden, aangezien dit het minste invloed heeft op de performantie van de applicatie. Dit is echter geen goede oplossing.

3.2.5 Performantieproblemen op kleine schaal

Microservices brengen een groot aantal extra factoren mee waar rekening mee moeten gehouden worden. Zo zorgt de gedistribueerde structuur voor extra "latencyömdat alle communicatie tussen de services via het netwerk moeten gaan. Ook de vertaling en ontcijfering van uitgewisselde informatie, wat moet om de services met elkaar te laten communiceren, neemt extra tijd in beslag die niet zou gebruikt worden met een monolithische applicatie.
Deze factoren, en nog enkele anderen, vertragen de snelheid van het systeem aanzienlijk, en zorgen ervoor dat een monolithische implementatie van dezelfde applicatie op kleine schaal bijna altijd efficiënter zal zijn.
Deze mindere performantie maakt dat een van de grootste voordelen van microservices, schaalbaarheid, geheel onbelangrijk wordt. Het gebruik van microservices op kleine schaal kan echter nog steeds voordelig zijn. Een applicatie waar snelheid niet van het grootste belang is, maar uitrolbaarheid of afhandeling van systeemfouten wel belangrijke succesfactoren zijn, kan zelfs in zo een scenario over het algemeen beter presteren dan dezelfde applicatie in monolithische vorm. Ook kan het als ontwikkelaar voordelig zijn om naar de toekomst te kijken. Verwacht je dat de applicatie in de toekomst met een grote influx aan gebruikers te maken zal krijgen, dan is het misschien geen slecht idee aan het begin van de ontwikkeling al microservices te gebruiken.

3.2.6 Asynchrone werking

Gezien het feit dat een microservice applicatie bestaat uit services die elkaar niet in acht houden, zullen de meeste functionaliteiten asynchroon werken. Dit wil zeggen dat twee services tegelijk kunnen worden aangeroepen, en de aanroep tegelijkertijd zullen afhandelen. In de meeste gevallen zorgt dit voor een verhoging van de performantie. Wanneer de opgehaalde informatie van de ene service echter van belang is voor de aanroep van de andere service, moet dit proces synchroon verlopen. Het synchroon laten verlopen van dit proces zal extra werk met zich meebrengen wat de complexiteit verhoogt. Er zullen bijvoorbeeld identificatiecodes moeten meegestuurd worden met de aanroepen en antwoorden zodat het systeem weet wanneer de gevraagde informatie binnenkomt, en er kan verdergegaan worden met andere aanroepen.

Hoofdstuk 4

Simulatie

Om de voorgenoemde voordelen en nadelen te staven, werd een simulatie van de twee architecturen in actie opgezet.
De simulatie bestaat er uit om voor elke architectuur een applicatie te schrijven met exact dezelfde functionaliteit, en deze beide bloot te stellen aan verschillende gradaties van dataverkeer. Zo wordt er bepaald welke architectuur de beste is voor welke scenario's.

4.1 De applicatie

De functionaliteit van de applicatie bestaat uit 2 modules. Als eerste is er een äccounts-module die uit een database accounts kan ophalen, als tweede een "web-module die deze accounts binnenkrijgt en teruggeeft aan de gebruiker via HTML-code die kan gelezen en ingeladen worden door een web-browser.

4.1.1 Monolithische versie

In de monolithische versie is de implementatie erg simpel. Alle functionaliteit, van het ophalen van de gegevens tot het terugsturen ervan, zit samen in één enkele controller-klasse. Verder is er nog 1 object-klasse die de accounts definieert. De functionaliteit van de applicatie wordt aangeroepen door een API-aanroep waarbij bijvoorbeeld een numerieke identificatie van de gebruiker, of de naam van de gebruiker wordt meegegeven.

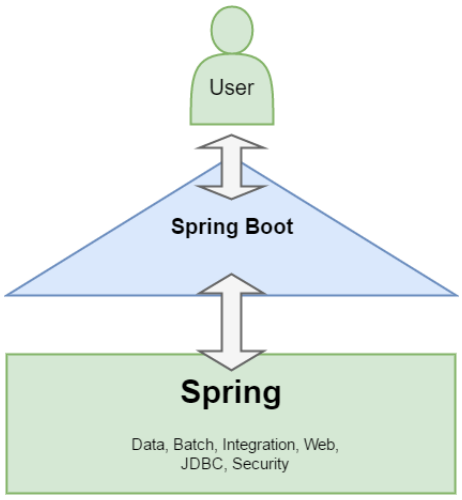
4.1.2 Microservice versie

In de microservice applicatie worden alle logische verantwoordelijkheden dus verdeeld in verschillende services. Er zal dus een "web-service zijn wiens functie het vertalen van

informatie in HTML-code is, en er zal ook een äccounts-service zijn die de gegevens uit de databank haalt. De functionaliteit wordt op dezelfde manier aangeroepen als met de monolithische applicatie.

4.2 Werkwijze

Beide applicaties werden geschreven in het Spring Boot framework. Spring Boot is een technologie die gebruik maakt van het al bestaande Spring framework. Het biedt de mogelijkheid om op een gemakkelijke manier productie-waardige applicaties of services te ontwikkelen die gemakkelijk uit te rollen zijn. Webb (2013)



Figuur 4.1: Structuur van het Spring Boot framework

De keuze om de twee applicaties te implementeren met gebruik van dezelfde technologie is gemaakt omdat het tijdens de simulatie gewenst is dat beide applicaties relatief zo snel mogelijk hun functionaliteiten kunnen uitvoeren. Op deze manier zijn de testresultaten zo accuraat mogelijk in de vergelijking van de twee architecturen.
Om de applicaties en services beschikbaar te stellen voor een client, wordt gebruik gemaakt van een bekende "Platform-as-a-Service", Amazon Web Services, of AWS. Amazon Web Services biedt een ontwikkelaar een betrouwbare, schaalbare en goedkope manier om services en applicaties uit te rollen op de "Cloud". Varia (2014) Het wordt momenteel al gebruikt door duizenden bedrijven in ongeveer 190 landen van de wereld. De service is normaal gezien betalend, maar voor deze simulatie was het voldoende om gebruik te maken van de gratis proefversie. Toch worden de kosten die zouden aangerekend worden voor het online zetten van de applicaties meegerekend in het uiteindelijke besluit van dit paper.

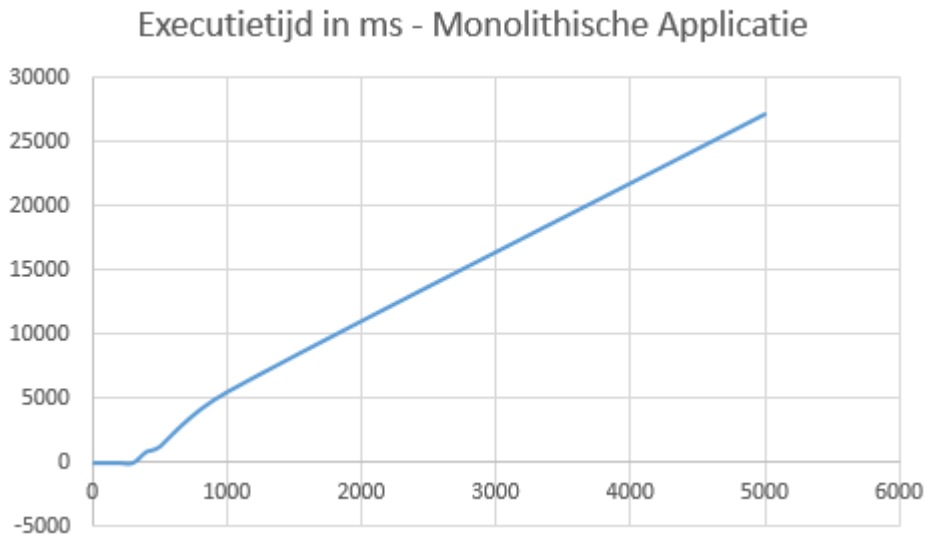
Om de applicaties op de servers van de Amazon Web Service te krijgen wordt gebruik gemaakt van nog een andere tool, Boxfuse. Dit is een tool die een Java-applicatie kan analyseren, kan omzetten naar het kleinst mogelijke systeembestand, en kan uitrollen naar services zoals AWS. Östling (2015)
Om de simulaties op grote schaal mogelijk te maken zal werd gebruik gemaakt van Oracle JMeter. Dit is een standalone applicatie die pakketten zoals bijvoorbeeld HTTP-verzoeken in grote getale kan versturen, en terzelfder tijd statistieken bijhoudt zoals de executietijd en de bandbreedte van het netwerk.
De simulaties zullen bestaan uit reeksen van HTTP-verzoeken naar de applicaties, die gradueel stijgen in dichtheid. Een eerste reeks zal bijvoorbeeld één verzoek per seconde versturen, terwijl een latere reeks één verzoek per 100 milliseconden verstuurd. Voorts zal voor elk verzoek gemeten worden hoe lang het duurt om een antwoord van de applicatie terug te krijgen.

4.3 Resultaten

Uit de simulaties konden een paar belangrijke bevindingen gemaakt worden. De belangrijkste resultaten worden hier opgesomd. De ruwe resultaten, in de vorm van tabellen en grafieken, zijn te vinden aan het einde van dit document.

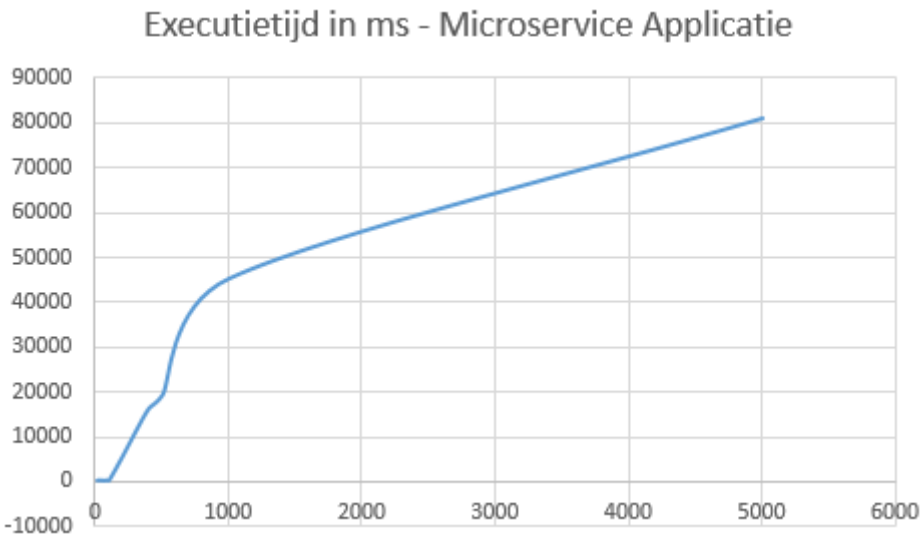
4.3.1 Uitvoeringssnelheid

Eerst en vooral werd getest wat de uitvoeringstijd van elke applicatie is wanneer slechts één verzoek gestuurd wordt zonder dat er daarvoor al andere gestuurd werden. Dit kunnen we gebruiken als een maatstaf van snelheid wanneer we meerdere verzoeken tegelijk doorsturen.
De optimale uitvoeringssnelheid van de monolithische applicatie bedraagt gemiddeld 22 milliseconden. Voor de monolithische applicatie is dit 141 milliseconden. Er kan dus meteen afgeleid worden dat de monolithische applicatie veel performanter is in scenario's met weinig dataverkeer.
Bij tests met meerdere verzoeken bleef de monolithische applicatie dicht tegen zijn optimale uitvoeringssnelheid tot het punt waarbij meer dan 35 verzoeken per seconde werden verstuurd, ofwel ongeveer één verzoek per 28,5 milliseconden. Vanaf dit punt begonnen verzoeken zich op te stapelen binnen de applicatie en ging de uitvoeringstijd exponentieel omhoog. Zo is de uitvoeringstijd voor de monolithische applicatie gemiddeld 5548 milliseconden wanneer er 100 verzoeken per seconde, of één verzoek per 10 milliseconden wordt verstuurd. Bij een test waarbij 500 verzoeken per seconde werden verstuurd, kwam de uitvoeringstijd op gemiddeld 27171 milliseconden.



Figuur 4.2: Uitvoeringstijd voor de monolithische applicatie tegenover het aantal verzoeken

De microservice applicatie bereikte het punt waar het zijn optimale uitvoeringssnelheid verloor wanneer er meer dan 8 verzoeken per seconde werden verstuurd. Dit is een stuk slechter in vergelijking met de testresultaten van de monolithische applicatie. Ook voor deze architectuur ging de uitvoeringstijd exponentieel omhoog voorbij dit punt. Een test waarbij 100 verzoeken per seconde werden verstuurd gaf als resultaat een gemiddelde uitvoeringstijd van 45223 milliseconden. Bij een test van 500 verzoeken per seconden kwam het resultaat op een extreme uitvoeringstijd van gemiddeld 80978 milliseconden.



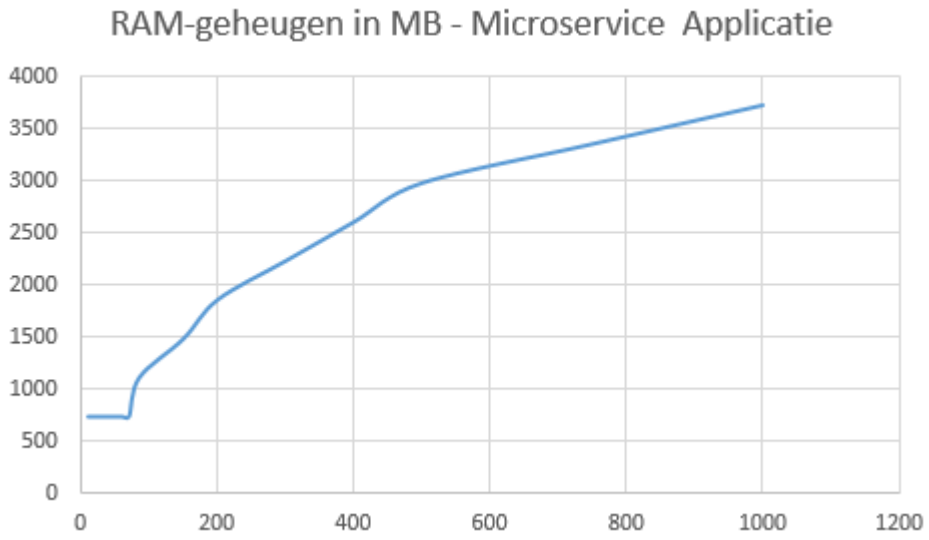
Figuur 4.3: Uitvoeringstijd voor de microservice-applicatie tegenover het aantal verzoeken

Een opmerkelijk besluit dat we echter kunnen afleiden is dat de stijging van de executietijd exponentieel groter is voor de monolithische applicatie dan voor de microservice applicatie. Bij het monolithisch model is de executietijd bij 500 verzoeken per seconde namelijk 1294 keer zo groot als zijn optimale executiesnelheid. Bij het microservice-model is de executietijd voor die test slechts 574 keer zo groot als zijn optimale uitvoeringstijd. Hieruit kunnen we afleiden dat, hoewel de microservices het op kleine schaal enorm veel slechter doen, ze op een bepaald punt de uitvoeringssnelheid van de monolithische applicatie zullen evenaren en er onder gaan.
Uiteraard zijn deze uitvoeringstijden niet realistisch. Geen enkele gebruiker gaat 80 seconden wachten op de informatie van zijn account. Deze testen zijn echter uitgevoerd zonder rekening te houden met schaalbaarheid. Er was dus telkens slechts één instantie van de applicatie draaiende.

4.3.2 Schaalbaarheid

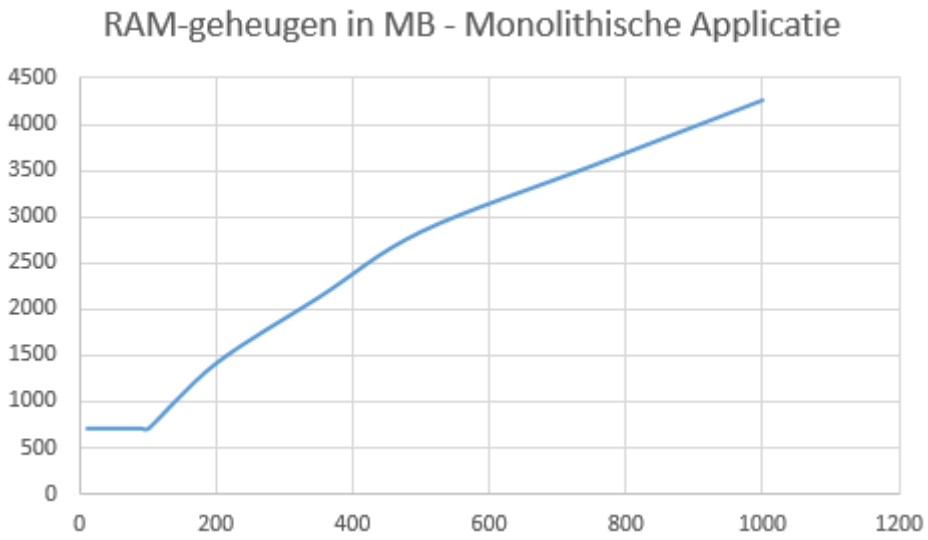
Om tot iets meer realistische resultaten te komen, werden de volgende tests uitgevoerd met meerdere draaiende instanties van elke applicatie. Hier werd gepoogd voor elk testscenario de optimale uitvoeringstijd te benaderen, en werd vooral gekeken naar hoeveel geheugenruimte de applicaties in totaal in beslag namen.
Voordat de tests uitgevoerd werden kon al afgeleid worden dat de microservice-applicatie met telkens één instantie van elke service meer werkgeheugen in beslag nam dan de ene instantie van de monolithische applicatie. Wanneer de microservices aangeroepen

werden namen ze in totaal 736,9 megabytes aan werkgeheugen in beslag, terwijl de monolithische applicatie maar 710,2 megabytes aan werkgeheugen innam.
Naarmate de verzoeken per seconde groter werden, moesten er uiteraard meerdere instanties aangemaakt worden om de uitvoeringstijd te drukken. Aangezien de uitvoeringstijd bij de microservice-applicatie veel sneller steeg, moesten er dus ook sneller nieuwe instanties bijkomen. Dit zorgde voor een snelle stijging van het ingenomen werkgeheugen. Echter moesten er slechts van één service, de äccounts-service, instanties worden aangemaakt om de extra verzoeken te verwerken. Dit zorgde ervoor dat er telkens maar ongeveer 370 megabytes aan werkgeheugen extra in beslag werden genomen.



Figuur 4.4: Geheugengebruik voor de microservice-applicatie tegenover het aantal verzoeken

Bij de monolithische applicatie moesten er niet zo snel nieuwe instanties bijkomen, maar als dat dan wel het geval was moest meteen de hele applicatie gedupliceerd worden. Dit zorgde telkens voor een verhoging van het ingenomen werkgeheugen van 710,2 megabytes.



Figuur 4.5: Geheugengebruik voor de monolithische applicatie tegenover het aantal verzoeken

Uit de grafieken blijkt dat de exponentiële verhoging van het ingenomen werkgeheugen bij de monolithische applicatie hoger ligt dan bij de microservices. Er moet dus een bepaald moment zijn tijdens de simulaties waar de monolithische applicatie meer werkgeheugen zal beginnen gebruiken dan de microservices, om dezelfde grootte aan dataverkeer af te handelen. Dit punt werd bereikt wanneer er 70 verzoeken per seconde werden verstuurd. Op dat moment had de microservice-applicatie een geheugengebruik van 3356,3 megabytes, en de monolithische applicatie een geheugengebruik van 3554,7 megabytes.
Hoe hoger de schaal, hoe efficiënter een microservice-applicatie dus zal omgaan met zijn geheugengebruik. Er moet rekening gehouden worden met het feit dat voor services zoals AWS meestal moet betaald worden voor het geheugen dat gebruikt werd. Dit zorgt er dus voor dat een microservice-applicatie rechtstreeks goedkoper zal zijn om op dezelfde schaal te laten draaien in vergelijking met een monolithische applicatie.

4.3.3 Foutafhandeling

Als laatste werd er getest wat er zou gebeuren moest een bepaalde instantie van een applicatie stopgezet worden.
Uit de testen bleek dat, bij het stopzetten van een instantie van een monolithische applicatie, alle verkeer naar die applicatie niet meer beantwoord werd. In het geteste scenario bestonden er twee instanties van de applicatie, waarbij het binnenkomend verkeer gelijkmatig verdeeld werd over de instanties. Bij het wegvallen van de eerste

instantie viel al het verkeer dat nog naar die instantie gestuurd moest worden weg, waardoor ongeveer de helft van de verzoeken onbeantwoord bleven.
Ditzelfde scenario werd bij de microservices ook getest. Hieruit bleek dat, wanneer één van de instanties wegviel, alle verkeer werd omgeleid naar de andere overblijvende instantie. Hoewel dit voor een veel tragere uitvoeringstijd zorgde, werden alle verzoeken wel beantwoord. Dit is te wijten aan het feit dat de service die alle microservices beheerd te weten krijgt wanneer een instantie wegvalt. Deze kan dan op een gepaste manier reageren door alle verzoeken om te leiden naar andere beschikbare instanties.

Hoofdstuk 5

Conclusie

5.1 De beste keuze voor kleine en middelgrote ondernemingen

Uit de simulaties en de opsomming van de voordelen van de microservice architectuur kan afgeleid worden dat deze niet even performant is als de monolithische architectuur wanneer het dataverkeer van en naar de applicatie beperkt blijft. Het is dus vanzelfsprekend om te stellen dat de microservice architectuur geen vooruitgang biedt voor bedrijven die geen groot klantenbestand hebben. Over het algemeen zullen kleinere bedrijven dus beter af zijn met een monolithische applicatie.
Het probleem ligt hier vooral bij het feit dat er in een microservice-structuur veel meer communicatie over het netwerk moet gebeuren. Het aantal oplossingen die dit probleem zouden kunnen verhelpen is ook beperkt. Eventueel kan overwogen worden om de communicatie tussen de services te laten gebeuren via optische glasvezelbekabeling. Dit zal zeker een positieve invloed hebben op de uitvoeringssnelheid, maar ook hier zijn enkele nadelen aan verbonden. De kostprijs van deze bekabeling is zeker niet goedkoop en zal daarom in vele gevallen niet de moeite waard zijn voor kleinere bedrijven.
Over het algemeen kan dus tot de conclusie gekomen worden dat kleine en middelgrote ondernemingen meer voordelen hebben bij het gebruik van de monolithische architectuur. Wanneer er toch voor de microservice-architectuur zou gekozen worden verliest het bedrijf al één van de grootste voordelen van het gebruik hiervan: de flexibele schaalbaarheid. Het is financieel en technisch gezien gewoon minder voordelig om in dit soort situaties over te stappen op microservices.

5.2 Moet de microservice architectuur daarom voor deze bedrijven afgeschreven worden?

Hoewel er best niet overgestapt word op een puur microservice-gerichte structuur, zijn er zeker zaken die van deze architectuur kunnen overgenomen worden om de performantie van de gebruikte architecturen te bevorderen. Het kan bijvoorbeeld al voordelen bieden wanneer de mechanismen voor de afhandeling van systeemfouten geïmplementeerd worden in een monolithische applicatie. Ook kan het helpen om al een systeem te integreren dat het binnenkomend verkeer van de applicatie dynamisch verdeeld over de verschillende instanties van de applicatie. Dit systeem zal grotendeels hetzelfde functioneren als bij een microservice-applicatie.
Het is over het algemeen een goed voornemen om als ontwikkelaar de beslissing te nemen om al enkele van de systemen van een monolithische applicatie te integreren in de al bestaande applicaties. Op deze manier wordt de overstap naar deze architectuur in de toekomst alleen maar gemakkelijker. Het kan namelijk zo zijn dat een klein of middelgroot bedrijf in de toekomst enorm veel extra klanten zal bij krijgen en met een veel groter dataverkeer zal te maken krijgen. Op dit moment zou het een enorme opdracht zijn om alle bestaande applicaties te herschrijven als microservices, die nu ineens wel performanter zouden gaan werken. Het op voorhand integreren van een aantal van de features van microservices kan ervoor zorgen dat dit probleem vermeden wordt.

5.3 Kunnen alle nadelen van de microservice-architectuur verholpen worden?

Over het algemeen is dit niet zo vanzelfsprekend. De meeste nadelen zijn eerder business-gerelateerd. De extra werklast voor de ontwikkelaar komt van nature mee met de extra complexiteit die de architectuur met zich meebrengt. Ook de nood aan ervaren ontwikkelaars is op zich een nadeel dat niet echt verholpen kan worden.
De performantieproblemen op kleine schaal kunnen deels verholpen worden, maar ook hier zullen deze oplossingen vooral voordelig zijn voor grotere bedrijven.

5.4 Algemene conclusie

De microservice-architectuur toont zijn sterkte op verschillende vlakken. Ze kan zijn performantie op peil houden, zelfs wanneer er veel dataverkeer is, en kan enorm goed overweg met onverwachte fouten in het systeem. Voor grote bedrijven is het bijna niet meer weg te denken om nieuwe applicaties te ontwikkelen volgens deze architectuur, of minstens enkele van de belangrijkste pijlers op te nemen in het ontwikkelingsproces. Voor kleinere bedrijven is het echter niet voordelig om meteen volledig op dit model over te schakelen. Een betere oplossing is het in acht houden van enkele van de pijlers van de architectuur die wel een onmiddellijk voordeel bieden voor de performantie van hun applicaties. Misschien veranderd het internet, en alle clients die hiervan gebruik maken, in de toekomst zo erg dat er genoodzaakt zal moeten overgeschakeld worden op deze architectuur. Op dit moment, echter, is dit nog niet nodig.

Bibliografie

Newman, S. (2015). *Building Microservices: Designing fine-grained systems*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Press, W. (1998). Three-tier application model. `https://msdn.microsoft.com/en-us/library/aa480455.aspx`.

Rouse, M. (2015). Microservices. `http://searchitoperations.techtarget.com/definition/microservices`.

Östling, L. (2015). Immutable infrastructure with boxfuse. `http://www.slideshare.net/Larsstling/immutable-infrastructure-52179433`.

Varia, J. (2014). Overview of amazon web services.

Webb, P. (2013). Spring boot – simplifying spring for everyone. `https://spring.io/blog/2013/08/06/spring-boot-simplifying-spring-for-everyone`.

Lijst van figuren