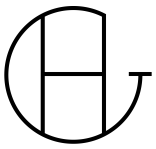


HoGent

Faculteit Bedrijf en Organisatie

De overstap van Monolithic Application Architectures en Microservice Architectures: 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 Monolithic Application Architectures en Microservice Architectures: 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 van het moment dat men er de eerste stappen in zet. Een welbekende quote uit de IT-wereld, *The one certainty is change.*, 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. De dag van vandaag hebben we echter te maken met een ontwikkeling in de business waar dit soort architectuur niet tegen opgewassen is.
Tegenwoordig bestaat er een groot aantal verschillende soorten clients die elk op een zeer specifieke manier met data 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 minder rekenkracht en een internetverbinding die soms instabiel kan zijn, 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 microservice?

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 een kleine, autonome service die volledig individueel draait op een systeem, en samenwerkt met andere microservices om een geheel van functionaliteit te vormen. Meteen merkt u 2 kernwoorden op: klein en autonoom. Hier gaan we verder op in.

1.1.1 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.

1.1.2 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.

1.2 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 verstuurd 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 meerdere 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.

1.3 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.

1.4 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 Architecture 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 Technology Heterogeneity

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 data bijhouden moet op een zo performant mogelijke manier gebeuren aangezien er met zoveel data wordt omgegaan. 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.
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.

3.2 Resilience

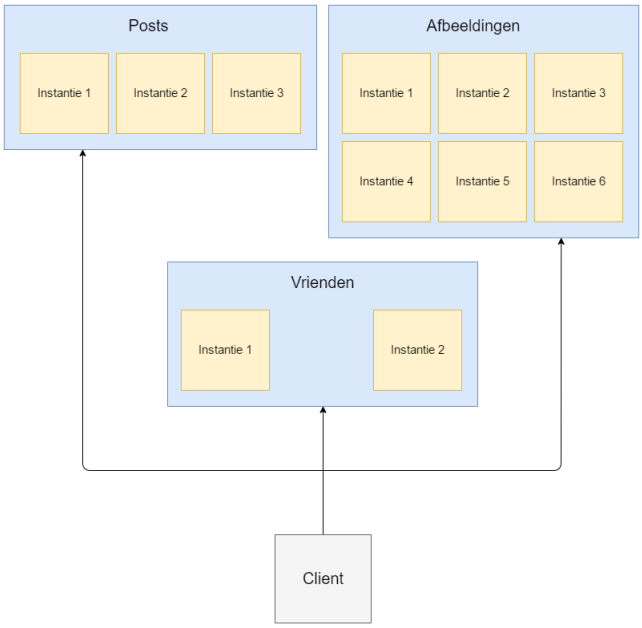
Een belangrijk onderdeel bij het ontwikkelen van een applicatie is om ervoor te zorgen dat de kans op systeemfouten zo klein mogelijk wordt gemaakt. Wanneer een systeemfout voorkomt, valt het hele systeem in elkaar en in de meeste gevallen moet een manuele heropstart gedaan worden. Wanneer dit zich voordoet 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 microservi-

ces 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.

3.3 Scaling

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 capa-

citeit 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.



3.4 Ease of Deployment

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.

3.5 Organizational Alignment

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 subteams. 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.

3.6 Composability

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. Voor een monolithische applicatie zijn dit zowat de voornaamste manieren. Is er echter geen betere manier om code te hergebruiken?
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.

3.7 Optimizing for Replaceability

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.

Hoofdstuk 4

Conclusie

Lijst van figuren

Lijst van tabellen