HO
GENT

Faculteit Bedrijf en Organisatie

Microservice integration patterns on SAP order-to-cash process.

Lyva Van Damme

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

Promotor:
Karin Samyn
Co-promotor:
Nicolas Pauwelyn

Instelling: HoGent

Academiejaar: 2018-2019

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Microservice integration patterns on SAP order-to-cash process.

Lyva Van Damme

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

Promotor:
Karin Samyn
Co-promotor:
Nicolas Pauwelyn

Instelling: HoGent

Academiejaar: 2018-2019

Tweede examenperiode

Woord vooraf

Deze bachelorproef werd geschreven voor het behalen van het bachelordiploma Toegepaste Informatica. Eerst was het niet gemakkelijk om een onderwerp te vinden. Ik wist niet goed waar ik het over wou hebben. Ik zou daarvoor graag mijn stagebedrijf, Delaware, voor het voorstellen van een onderwerp. Dankzij hun kon ik mijn bachelorproef doen over een onderwerp dat mij echt interesseerde. Ik zou ook graag mijn ouders bedanken voor de steun die zij mij gaven. Voor hun geduld bij de stressvolle situaties. Ook wil ik mijn broer en zus bedanken voor hun hulp. Ik zou graag mijn mede-studenten bedanken voor alle hulp die ze mij gaven.

Samenvatting

Dit onderwerp werd gegeven door delaware. Om te onderzoeken hoe microservices een invloed kan hebben op het order-to-cash proces binnen SAP.

Eerst wordt er diep ingegaan op wat microservices zijn. Er wordt meer uitleg gegeven wat microservices precies zijn. Ook worden er vergelijkingen gedaan tussen verschillende onderdelen van microservices zoals de authenticatie en authorisatie, de link met andere opkomende ideologieën en de bescherming van microservices.

Er zullen zeker vragen en aanvullingen komen in de toekomst. Dit is een evoluerende technologie. En er zullen betere oplossingen komen voor onderdelen.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

1.1 Probleemstelling

Dit onderwerp werd voorgesteld door Delaware. De doelgroep is dus Delaware.

Uit je probleemstelling moet duidelijk zijn dat je onderzoek een meerwaarde heeft voor een concrete doelgroep. De doelgroep moet goed gedefinieerd en afgelijnd zijn. Doelgroepen als “bedrijven,” “KMO’s,” systeembeheerders, enz. zijn nog te vaag. Als je een lijstje kan maken van de personen/organisaties die een meerwaarde zullen vinden in deze bachelorproef (dit is eigenlijk je steekproefkader), dan is dat een indicatie dat de doelgroep goed gedefinieerd is. Dit kan een enkel bedrijf zijn of zelfs één persoon (je co-promotor/opdrachtgever).

1.2 Onderzoeksvraag

De algmene onderzoeksvraag is: "Hoe microserivce integration patterns een order-to-cash proces in SAP kan beïnvloeden?". De algemene vraag delen we op in volgende puntjes:

- Wat zijn microservices?
- Hoe kan er overgeschakeld worden naar een microservice architectuur?
- Welke aanpassingen kunnen of moeten er gebeuren aan de architectuur om de microservices te laten werken?
- Hoe zal de communicatie tussen de verschillende microservices werken?
- Hoe zit een order-to-cash (OTC) proces er uit?
- Welke business requirements heeft een order-to-cash proces?

- Welke invloed heeft de microservice architectuur op de performance van een order-to-cash proces?

Dit zal een theoretische studie zijn.

1.3 Onderzoeksdoelstelling

Het doel van deze paper is het onderzoeken van de effecten van een microservices architectuur op de architectuur van een OTC in SAP. Het doel houdt ook in dat we aan de hand van een zes-stappen plan, theoretisch, gaan kijken hoe een OTC verandert door microservices.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie. Hierin zal er meer uitleg gegeven worden over microservices en het order-to-cash proces binnen SAP.

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen. Hier zal het onderzoek worden uitgevoerd over hoe microservices het order-to-cash proces zullen beïnvloeden.

In Hoofdstuk 4, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Stand van zaken

Dit hoofdstuk bevat de literatuurstudie omtrend het onderwerp. Hier wordt de architectuur en termen uitgelegd. Ook zal hier een vergelijking gemaakt worden tussen onderdelen van microservices. Een duidelijk beeld van een order-to-cash proces zal hier ook gemaakt worden.

2.1 Microservices

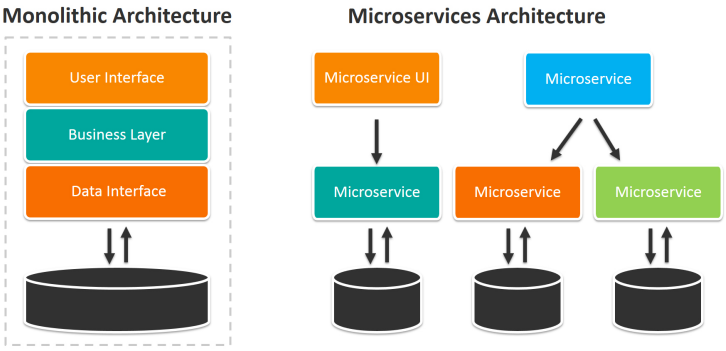
2.1.1 Definitie

Een term die vaak zal terug keren in deze paper is 'monolithic'. "Monolithic software is designed to be self-contained; components of the program are interconnected and interdependent rather than loosely coupled as is the case with modular software programs."Wigmore (2016). Om te begrijpen waarom een overschakeling naar microservices een goed idee is, kaarten we de moeilijkheden bij monolithic architectuur aan. Bij een verandering binnen een monolithic architectuur, wordt er een heel nieuwe versie van de architectuur uitgebracht. Een verandering brengt een hoop extra werk mee Mauersberger (2017). Dit omvat

- De volledige architectuur moet opnieuw getest worden.
- Deze architectuur kan heel complex worden bij het toevoegen van functionaliteiten.
- De complete architectuur moet opnieuw gedeployed worden bij elke update.
- De impact van een verandering kan verkeerd ingeschat worden.
- Bij een fout in een proces, kan de volledige architectuur falen.

De definitie die te vinden is in dit artikel, gaat als volgt: "A method of developing software

applications as a suite of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goals.". Als je de definitie leest, zie je drie onderdelen. Het eerste onderdeel vertelt hoe een microservice in elkaar zit. Het is een onafhankelijke, kleine, modulaire services. Modulaire services zijn services waarbij veel delen uitwisselbaar zijn met diverse services. De services staan los van elkaar. Ze hebben geen invloed op elkaar. Daarom zijn ze ook onafhankelijk. Wordt er info gestuurd of gevraagd van services A dan zal dit geen invloed hebben op de andere services. De eenvoudige communicatie is een tweede eigenschap van microservices. Er is nood aan communicatie omdat sommige services wel data moeten uitwisselen om hun 'job' te kunnen doen. De communicatie kan gebeuren op verschillende manier. De manier die gekender is, is 'Messaging via a Message Broker'. Dit wil zeggen dat microservice A een bericht plaats op de wachtrij bij microservice B wanneer die data wil doorsturen. Dan kan microservices B aan die data wanneer hij die nodig heeft. Ze zullen soms moeten wachten maar ze zijn zo goed als onafhankelijk van elkaar. De derde eigenschap omvat dat een microservice wordt gemaakt in functie van een requirement uit de business. Elk product in de business heeft een doel dat moet voldoen aan eisen. Het unieke aan microservices is dat we ze gaan bekijken vanuit de eisen binnen de business. Het doel van microservices is, de problemen die te vinden zijn bij een monolithic, verhelpen. De vorige definite legde uit wat microservices zijn. Dit artikel zegt waar men microservices kan plaatsen. Er is dus één groot framework. Daar zitten meerdere onafhankelijke services in.



Figuur 2.1: Een monolithic architectuur naast een microservice architectuur. Watts (2018)

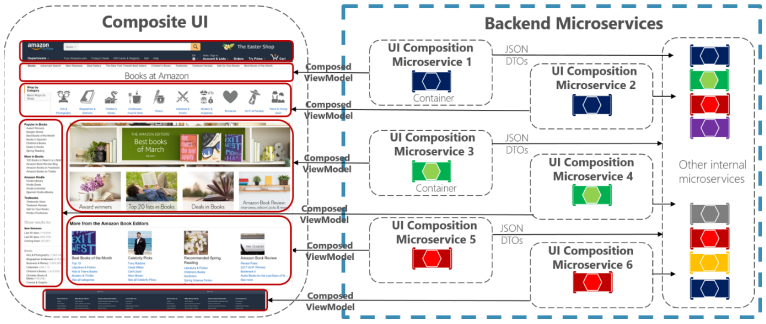
Zoals te zien in figuur 2.1, Watts (2018), is er een groot verschil tussen een monolithic architectuur en die van een microservice. De monolithic wordt weergegeven in de linkerkant van de foto. Aan de rechterkant van de foto is een voorbeeld te zien van een microservice architectuur. Daar is duidelijk te zien dat elke microservice een eigen databank/datastore heeft. Voor elke functionaliteit wordt een microserives aangemaakt, die dan nog eens apart een databank voor zich krijgt. Als bij microservice A een probleem is dan heeft dit niet meteen impact op de andere services. De communicatie tussen microservice A en de anderen zal wel hinder ondervinden. Maar de andere microservices kunnen wel nog steeds onafhankelijk verder. Een andere definitie van microservices is: "A software architecting pattern that allows software to be developed into relatively small, distinct components. Each of the components is abstracted by an API(s) and provides a distinct subset of the functionality of the entire application". Ook hier zien we weer het puntje passeren dat

een microservice een klein componentje is van een groter geheel. Die eigenschap wordt heel hard benadrukt. Na het uitleggen van microservices, schrijven ze ook over hoe microservices zo scalable zijn. Met scalable bedoelen ze schaalbaarheid. De mogelijkheid van software om mee te groeien als het aantal gebruikers vermeerderd. Dus eigenlijk dat de software nog steeds even goed presteerd bij 10 gebruikers als bij 2 000 gebruikers. Ook lichten ze toe hoe belangrijk API's zijn binnen een microservice architectuur. API's zijn een set van definities die ervoor zorgen dat deeltjes in een programma met elkaar kunnen communcieren. Een voordeel van API's is dat je niet moet weten hoe de andere code werkt. Verder wordt er ook gepraat over de verschillen tussen een monolithic architectuur en een microservice architectuur **series2018**.

Zoals te zien is in figuur 2.2. We zien dat de monolithic alle puntjes in een kader heeft. Dit staat symbolisch voor het grote geheel dat eigen is aan een monolithic. Alles zit samen in één grote doos. Maar bij microservices is dit niet zo, daar zit elk deeltje/requirement in een aparte doos Benetis (2016b) .

Figuur 2.2: Een monolithic vergeleken met een microservice. Benetis (2016b)

In figuur X is duidelijk te zien hoe microservices een website pagina opuvllen. De site

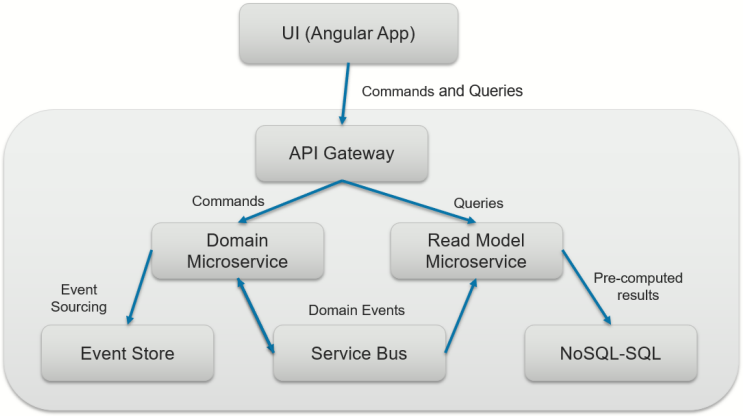


Figuur 2.3: Microservices die de inhoud op een site invullen. Koukia (2018)

bestaat uit zes onderdelen. Die onderdelen zijn allemaal dynamisch en ze halen hun info uit databanken. Bij het laden van de pagina gaat elk onderdeeltje zijn info gaan opvragen. Omdat dit allemaal met microservices gebeurt, wordt de info gelijktijdig opgehaald. Ook zullen ze niet moeten wachten op elkaar, tot de een info heeft ontvangen om dan naar de volgende zijn benodigde info te sturen. Elk spreekt zijn eigen databank aan. De microservices die de website opvullen, die kunnen onderliggend ook nog andere microservices aanspreken. Bijvoorbeeld: Een klant is aangemeld op de website van

Amazon. De authenticatie microservices zorgt ervoor dat alle microservices weten dat die bepaalde persoon is aangemeld. Dan bij het inladen van voorkeuren of 'soort gelijke producten', kan de microservices die die sectie moet opvullen, vragen aan een andere microservice om de voorkeuren te laten genereren. Koukia (2018).

Een voorbeeld van een algemene architectuur is te zien op figuur X. Als eerste wordt er



Figuur 2.4: Een algemene architectuur voor microservices. Koukia (2018)

een request gemaakt. Die wordt naar de gateway gestuurd. Dan wordt er gekeken, moet er een commando of een query uitgevoerd worden. Bij een commando wordt er naar een domein microservice gestuurd en anders naar een real-model microservice. Daarna worden er gegevens opgeslaan.

2.1.2 Het belang van microservices

Bij een monolithic kan het aanpassen van een deeltje, veel werk vragen. Dit werd meer uitgelegd bij de definite van microservices. Microservices spelen gemakkelijker in op het periodieke opleveren van delen software. Deze technologie legt niet heel het framework plat als er deeltjes moeten bij gecodeerd worden. Microservices kunnen sneller inspelen op de Agile analyse/ontwikkel methode. Dit is ook een reden waarom microservices zo een opkomst kent. De analyse methode Agile werkt met periodieke opleveringen die kunnen gaan van twee weken tot een maand. In de periode wordt er gewerkt aan een functionaliteit of een eis van de klant. Er voor zorgen dat software schaalbaar is, is een belangrijk punt en daar spelen microservices goed op in, **series2018**.

Troisi (2019) geeft acht best practices over de bescherming van microservices. De best practices:

- Het gebruik van OAuth voor gebruikers identificatie en wat de gebruiker kan. OAuth/OAuth2 is een protocol voor authorisatie. Het is een gemak om gebruik te maken van een protocol. Een protocol zijn een aantal regels om te communiceren tussen computers.
- Gebruik bescherming in de diepte om een prioriteit toe te kennen aan service keys. Dit kan verwoord worden als bescherming steken in verschillende lagen van een

systeem. Er moet worden nagegaan welke deeltjes het kwetsbaarst zijn en daar dan op verschillende lagen van beveiliging op toepassen. Microservices maken het toepassen van deze methode, eenvoudiger. Doordat er gefocust kan worden op beveiliging. Het framework maakt het gemakkelijker om de verschillende lagen vast te stellen. Als ze binnen zijn bij een van de microservices zijn ze niet binnen in het volledige systeem.
- Schrijf zelf geen krypto code. Er zijn genoeg open source alternatieven. Enkel bij heel uitzonderlijke redenen wordt een eigen crypot code geschreven.
- Update je bescherming tijdig. Als er updates komen in de software van beveiliging, moeten die ook uitgevoerd worden. Het automatiseren van die updates, kan veel werk besparen achteraf. Dit wordt dan ook best gedaan bij het begin van microservices. Bescherming binnen software is niet meer een nice to have maar een must have.
- Maak gebruik van een firewaal met gecentraliseerde controle. Het biedt ook meer controle aan aan de gebruiker.
- Zorg dat je 'containers' niet in een publiek netwerk te vinden is. Dit is eigenlijk zorgen dat gebruikers je achterliggende architectuuur niet kunnen zien. Ervoor zorgen dat erna maat één toegangspoort is. Hier kan een vorige manier ook bijgestoken worden. Microservices kunnen achter een firewall gestoken worden als bescherming. Als er met containers wordt gewerkt, moet er ook beschemring aanwezig zijn. Een container is een plek waar kleine deeltjes code kunnen op gedeployed worden.
- Maak gebruik van software om virussen te vinden.
- Monitor alles.

Nog andere redenen om microservices te gebruiken, Koukia (2018), zijn volgende:

- Het is gemakkelijker om kleine services te onderhouden. Bij een monolithic is alles één groot geheel, als daar een deeltje van moet worden uitgelegd, kan je verdwalen in het grote geheel. Dit is niet zo bij microservices. Want elke services is afgebakend met een functionaliteit. Bij het uitleggen van een services kan er duidelijk aangetoond worden waar een services begint en eindigt.
- Een microservices kan onafhankelijk gedeployed worden. Eens een microservices klaar is om gebruikt te worden, moet er naar niks anders gekeken worden. Bij het deeltje definitie werd hier dieper op ingegaan.
- Gemakkelijker aan te passen aan nieuwe technologie. Komt er een nieuwe technologie uit die kan toegepast worden op een paar microservices, dan moeten enkel die microservices herschrijven worden. Dit ligt in contrast met een monolithic. Als er een nieuwe technologie is die kan toegepast worden op onderdelen van het geheel, moet de gehele architectuur herschreven worden.
- Het is eenvoudiger om te scalen. Zoals al eerder aangehaald, kan schalen van de architectuur gebeuren door de microservices te dupliceren. Niet het gehele systeem moet geschaald worden, enkel de nodige microservices moeten gedupliceerd worden.
- No single point of failure. Faalt er een microservices in het uitvoeren van zijn functionaliteit, dan heeft dit geen invloed op de andere services. Hier is dieper op ingegaan tijdens de definitie.
- Freedom of technology stack choices. Dit omvat dat elk team kan kiezen in welke programmeertaal ze de microservices schrijven. Een team is verantwoordelijk voor één microservices. Ze zijn dus gespecialiseerd in die ene service.

- De evolutie en de oplevering van business features is sneller. Dit komt door de onafhankelijkheid van de services. Er kan op het zelfde moment aan verschillende services gewerkt worden, zonder elkaar te beïnvloeden.

Onder volgende deeltjes wordt er dieper ingegaan op de authenticatie en authorisatie, het verband met Agile en Devops, het debuggen binnen microservices en de bescherming van microservices.

De verschillende manieren van bescherming

Microservices moeten een doel in de business vervullen. Naast dit, zorgen microservices er ook voor dat de bescherming eenvoudiger wordt, RDX (2016).

Enkele tips om de bescherming van microservices aan te pakken, Matteson (2017), da Silva (2017):

- Zorg bij het ontwikkelen van microservices voor coderingsstandaarden die kunnen herbruikt worden. Door eenmaal een goede code te voorzien wordt de kans op kwetsbaarheden en gaten in de bescherming kleiner.
- Ga na welke schade er kan toegebracht worden aan een microservice als die zonder bescherming zou worden geupload.
- Maak gebruik van toegangscontroles. Zorg ervoor dat er gewerkt wordt met leesrechten. Een microservice dat de aankooporders ophaalt moet niet aan de verkooporders kunnen.
- Ga geen beveiligingsprincipes gebruiken van externen, implementeer die in de code van de microservice.
- Zorg voor goede documentatie van elke microservice. Dit kan handig zijn bij het ontdekken van een zwak punt in de bescherming. De documentatie kan mogelijke problemen verduidelijken.
- Maak een API gateway. Wat het precies allemaal doet wordt verder nog uitgelegd.
- Zorg ervoor dat enkel de API gateway zichtbaar is en dat alle data onleesbaar moet worden verzonden. Dit kan gebueren aan de hand van SSL of TSL. Wat SSL is wordt verder in deze thesis uitgelegd.
- Zorg voor garantie op data privacy. In Europa is de GDPR een wetgeving die zegt wat er wel en niet mag gebeuren met mensen hun data. Daarom is het belangrijk dat er gebruik wordt gemaakt HTTPS. Op elk level moet er gezorgd worden voor een correcte beveiliging van de gebruikers hun data.
- Voor het encrypteren van data, wordt er best gebruik gemaakt van al bestaande technologies.
- Zorg ervoor dat er geen denial of service kan gebeuren. Denail of service komt voor wanneer er heel veel requests naar de applicatie worden gestuurd, waardoor de applicatie faalt. Maak gebruik van throtteling. De term wordt na deze opsomming verder uitgelegd.
- Maak gebruik van CSRF en CORS filters. Cross-site request forgery is een poging tot hacken waarbij de eindgebruiker gedwongen wordt om acties te doen terwijl hij geauthenticeerd is. Cross-origin resource sharing laat toe dat er requests van een

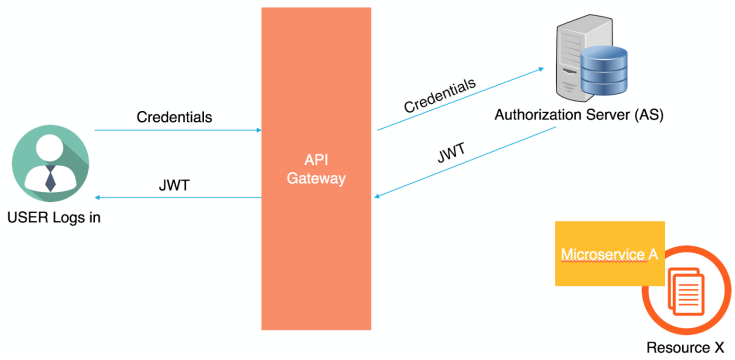
ander domein kunnen worden gemaakt.
- De API gateway mag nog zo goed beschermd zijn als iets, zorg ook voor goede bescherming aan de code. Zorg ervoor dat er niks moet worden gerund als administrator, geef duidelijke namen. Zorg ervoor dat enkel de nodige personen de juiste permissies krijgen.

Throtteling, Cavalcanti (2018), is een manier van bescherming die het volgende inhoud: "Throttling is a process used to control the usage of APIs by consumers during a given period.". Het kan zijn dat de gebruiker heel veel requests stuurt naar de API gateway of een bug kan zorgen voor een oneidig aantal request. Om dit te voorkomen kan er een limiet aantal request binnen een bepaalde periode opgelegd worden. Bijvoorbeeld als je je toegangscode drie maal fout hebt op je gsm, dan blokeert die voor een bepaalde tijd. Het systeem zo ontwerpen dat het bestand is tegen fouten en falen. Met bestand zijn tegen, wordt bedoelt om ervoor te zorgen dat bij een bug of een fout, deze goed wordt opgevangen zodat de gebruiker er geen last van heeft.

Eén van de meer bekendere manieren is, API gateway. De verantwoordelijkheden van een API gateway zijn volgende, Siraj (2017):

- Het ontvangen van request van gebruikers.
- De requests doorsturen naar de correcte microservice.
- Het 'antwoord' van de microservice in ontvangst nemen en doorsturen naar de gebruiker.

Zoals te zien is op figuur X, is een API gateway het toegangspunt. API gateway wordt gebruikt voor authenticatie en authorisatie. Meer uitleg hierover in de sectie over authenticatie en authorisatie.

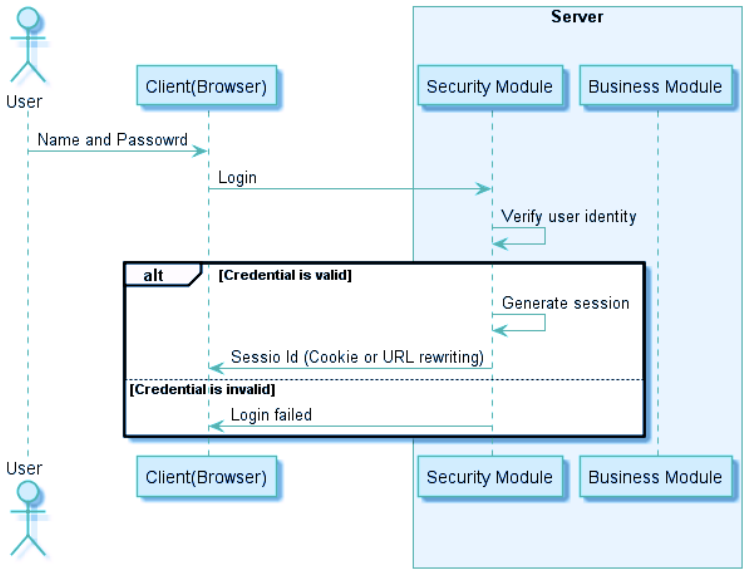


Figuur 2.5: Een voorstelling van API gateway. Siraj (2017)

Authenticatie en authorisatie

Een belangrijk aspect van microservices is de authenticatie en de authorisatie. Het is een moeilijkheid om op een uniforme manier veiligheid, bescherming, authorisatie en authenticatie toe te passen op microservices, Ayoub (2018). Authenticatie is bevestigen dat jij het bent. Dit wordt gedaan via een gebruikersnaam en een wachtwoord. Authorisatie

is wat je kan doen met een programma. Bijvoorbeeld een beheerder van een site kan meer dan een bezoeker. Zoals te zien is op bovenstaande afbeelding, figuur 2.3, wordt



Figuur 2.6: Een diagram van authenticatie bij een monolithic. Ayoub (2018)

de authenticatie afgehandeld binnen het monlithic proces. Wanneer de gebruiker inlogt, wordt de beveiligings module aangesproken. Deze kijkt of de gebruiker een bekende is, of er al gegevens in de databank zitten. Als het aanmelden gelukt is, wordt er een sessie gecreëerd. Een sessie wordt opgeslaan aan de hand van cookies. Cookies worden op je computer geplaatst door je browser. Het zijn kleine tekstbestanden. De sessie onthoudt wie je bent aan de hand van een ID. Dit zorgt ervoor dat je je niet elke keer opnieuw moet aanmelden als je van pagina verandert. Elke keer dat de bezoeker van de site iets doet, wordt de sessie ID samen gestuurd met de request. Een request is een aanvraag. Als het ID correct is, weet de site dat de gebruiker ingelogd is. Bij elke aanvraag wordt de ID meegestuurd zodat er kan gecontroleerd worden of die zo wordt de authenticatie bij een monolithic afgehandeld. Wanneer er gekeken wordt om authenticatie toe te passen bij microservices, komen volgende puntjes veel voor:

- In elke microcservice moet er authenticatie en authorisatie afgehandeld worden. Het beste wordt dit toegepast op een uniforme manier. Dan wordt er van uit gegaan dat er in elke microservices een stukje code gaat komen dat herbruikt wordt. Maar dit zorgt ervoor dat elke microservice toch afhankelijk is. Bij het uitkomen van een nieuwe versie, moet dit deeltje dan weer geüpdate worden. Dit heeft invloed op de flexibiliteit van het framework.
- "Single responsibility"zijn twee woorden die microservices mooi omschrijven. Een microservice omvat een stukje business logica. De algemene logica van authenticatie en authorisatie mag niet in een microservices gegoten worden.

In het algemeen zijn authenticatie en authorisatie een complex onderdeel van microservices. Er zijn vijf oplossingen volgens dit artikel.

- Distributed session manangement,

Eigenschappen	Sidecar proxy	API gateway	Single SignON (SSO)
Wat?	"A sidecar proxy, is a proxy running in the same host where your microservice is deployed", Cavalcanti (2018).	"An API gateway is a piece of software running on or near the periphery of the network hosting your system services", Everard (2017).	"SSO simply means login, just once to a suite of independent applications.", Everard (2017).
Heeft volgende opties:	De ingebouwde functies zijn volgende: • Authenticatie/authorizatie • Retry policies • Routing • Error afhandeling • circuit-breakers: Een automatische machine voor het stoppen van de flow voor veiligheidsredenen.	/	Bij het inloggen wordt een cookie aangemaakt die dan wordt opgeslaan in de API gateway.
Voordelen	De niet-functionele onderdelen van de microservice(s) kunnen hieronder gebracht worden.	Dit zorgt ervoor dat de microservice zich enkel moet bezig houden met de business logica.	
Nadelen	Er komen meerdere componenten bij om te monitoren, te onderhouden en te deployen.	Omdat dit een single point of entry kan dit een bottleneck, een plek van opstopping, zijn in de architectuur.	

Tabel 2.1: De verschillende manieren voor authenticatie en authorisatie toe te passen. Cavalcanti (2018), Everard (2017)

	HTTP Basic Authentication	API keys	OAuth
Definitie	"A HTTP user agent simply provides a username and password to prove their authentication".	"An unique generated value is assigned to each first time user, signifying that the user is known. When the user attempts to re-enter the system, their unique key is used to prove that they're the same user as before.".	"The user logs into a system. That system will then request authentication, usually in the form of a token. The user will then forward this request to an authentication server, which will either reject or allow this authentication. From here, the token is provided to the user, and then to the requester. Such a token can then be checked at any time independently of the user by the requester for validation, and can be used over time with strictly limited scope and age of validity".
Voordelen	Er is geen nood aan cookies, session ID's, login pagina's, ook een handshake bevestigen is niet nodig.	Dit is een manier van authenticatie die snel gebeurt. Het is een niet zo complex proces om de sleutels te genereren.	Het is de beste manier van authenticatie en authorisatie uit deze tabel.
Nadelen	Is er geen SSL (Secure Sockets Layer) aanwezig dan is het eenvoudig om de gegevens op te halen en ligt alles open en bloot op het internet. Ook bij het gebruik van SSL is er een nadeel. De tijd dat moet gewacht worden op een antwoord wordt vertraagd.	De API key is niet geschikt voor authorisatie. Bij het fout gebruik van API keys kan dit een grote impact hebben op de bescherming van de applicatie.	Als er enkel gebruik moet gemaakt worden van één van de twee, authenticatie of authorisatie, dan is OAuth overbodig. Want OAuth heeft veel meer te bieden.

Tabel 2.2: De verschillende methoden in API authenticatie, Sandoval (2018)

- client token,
- single sing-on,
- client token with API gateway.

Distributed Session management is de eerste oplossing. Het managen van een sessie over microservices. Dit kan op verschillende manieren. Aan de hand van Sticky sessions. Dit houdt in dat alle requests van één gebruiker naar dezelfde server worden gestuurd. Dan kan men ervan uitgaan dat de gebruikte data van die specifieke gebruiker is. Of men kan dit toepassen via session replication. Dit houdt in dat alle instanties de sessie data synchronizeren. Deze manier van toepassen heeft als nadeel dat er veel overhead zal zijn op het netwerk. Een andere methode is centralized session storage. Dit omvat dat bij het aanspreken van een microservice, deze de gebruikers data gaat ophalen van op een gedeelde plaats. Een andere manier om authenticatie en authorizatie toe te passen is via een client token. Een token wordt gebruikt om aan te tonen dat je ook echt de gebruiker bent. Een token wordt bijna altijd onleesbaar gemaakt. Het klinkt bijna hetzelfde als een sessie. Het verschil ligt hem in het feit dat een sessie op de server centraal wordt bijgehouden. Een token wordt bijgehouden door de user zelf. Naast de Distributed session management en client token is er ook nog single sign-on. Na een enkele aanmelding, kan de gebruiker alle microservices gebruiken binnen de applicatie. Een andere manier is een client token wiht API gateway. Deze manier is gebaseerd op de client token. Maar nu is er een API gateway toegevoegd aan het begin van een externe request. Dit zorgt ervoor dat het framework niet zichtbaar is aan de buitenwereld.

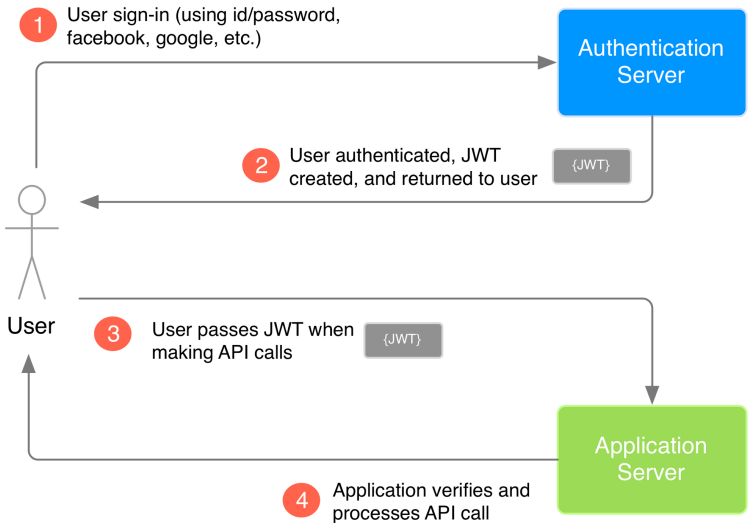
De meest voorkomende methodes in API authenticatie terug te vinden in tabel X, Sandoval (2018).

Het proces van een API gateway dat authenticatie en authorisatie op zich neemt, Siraj (2017). Door de API Gateway to combineren met JSON Web Tokens is er meer mogelijkheid om te schalen.

1. De gebruiker meldt zich aan.
2. De API gateway stuurt de request door naar de server die instaat voor de auhtenticatie.
3. is de authenticatie goedgekeurd, wordt er een JWT token terug gestuurd naar de gebruiker. Meer uitleg over de JWT na dit proces.
4. Bij volgende request wordt de token automatisch meegestuurd met de request.
5. Een andere server kijkt bij elke request naar de token, om na te gaan of de gebruiker de juiste authorisatie heeft.

De token vervalt na een bepaalde tijd. Er kan gekozen worden om deze automatisch te laten verlengen. Hier wordt niet verder op ingegaan.

De definitie van JSON Web Token, Stecky-Efantis (2016): "A JSON Web Token (JWT) is a JSON object that is defined in RFC 7519 as a safe way to represent a set of information between two parties. The token is composed of a header, a payload, and a signature.". Het diagram te zien in figuur X, is een schematische voorstelling van hoe een JWT token wordt gegeven aan een gebruiker. Het proces gaat als volgt:



Figuur 2.7: Een schematische voorstelling van het geven van een JSON Web Token. Stecky-Efantis (2016)

1. Aanmelden op een platform. Bijvoorbeeld aanmelden op facebook, twitter, instagram of google.
2. De authenticatie server stuurt een JWT terug.
3. Bij het maken van requests wordt de token meegestuurd.
4. De microservice kijkt dan of de gebruiker het recht heeft om deze microserivce/functie te gebruiken.

Verband met Agile en DevOps

Microservices komen uit dezelfde ideologie als Agile en DevOps. DevOps is een samenvoegsel van development en operations. Bij deze methode ligt de nadruk op de samenwerking en communicatie tussen verschillende partijen. Hier zijn de partijen de software engineers en andere IT specialisten. Deze ideologie omvat het volgende: het afbreken van kleine, traag evoluerende architectuur of monolithic en deze in microservices steken.

DevOps heeft volgende definitie: "DevOps is a methodology that enables developers and IT Ops to work closer together so they can deliver better quality software faster", Morgan (2019). Met DevOps probeert de productie zo goed mogelijk na te bootsen. Net zoals bij Agile, zorgt DevOps ervoor dat de software in kleinere delen wordt opgesplitst om ook weer op kortere periodes kleinere deeltjes software op te leveren. Dit is iets waar ook microservices in terug te vinden is. Verschillende DevOp teams kunnen dus tegelijkertijd werken aan microservices werken.

Enkele voordelen van de combinatie microservices en DevOps:

- Meer opleveringen van software op kortere periodes.
- Betere kwaliteit van de code.
- Software kan hergebruikt worden.
- Een hoger level van automatisatie.

Microservices en DevOps vullen elkaar aan op volgende vlakken, Mulesoft (2019):

- Deployability: Microservices bemoedigen het gebruik van Agile omdat het eenvoudiger is om periodiek op te leveren.
- Reliability: Een fout binnen een microservice, heeft enkel effect op die microservice.
- Availability: Het opleveren van nieuwe deeltjes, neemt niet veel tijd in beslag. De gehele applicatie zal niet lang offline zijn.

Het monitoren van microservices

Logs zijn records binnen een databank waar naar weggeschreven wordt terwijl de applicatie draait. Metrics zijn numerieke waarden die kunnen geanalyseerd worden. Metrics zijn terug te vinden op volgende niveau's van een applicatie, Wasson (2018):

- Node-level: Dit houdt in de gegevens van de CPU, het geheugen, netwerk en de harde schijf.
- Container: Draait de service binnen een container dan moeten er ook metrics bijgehouden worden van die container.
- Applicatie: Hier kunnen metrics bijgehouden worden om het gedrag van de service te begrijpen. Gegevens die kunnen bijgehouden worden zijn het aantal HTTP requests, de vertraging en de lengte van een bericht.
- Dependent service: Bij interactie met een externe service, hoelang deze duurt voordat de externe service reageert.

Het monitoren of loggen van microservices houdt in dat er wordt bijgehouden hoe een microservice zich gedraagt. Er wordt bijgehouden hoe snel de data wordt opgehaald uit de databank. Ook voor het vinden van problemen is monitoren een belangrijk onderdeel. Dankzij monitoring kan een fout sneller gevonden worden. Er kan worden nagegaan hoelang het duurde om een bepaalde request te maken. Van die gegevens kunnen er dan conclusies getrokken worden, Ananthasubramanian (2018).

Loggen is belangrijk omdat bij meedere services het moeilijk kan zijn om het traject binnen de services te volgen, Saldanha (2016).

De verschillende tools om te debuggen, Swersky (2019):

- Logging frameworks: Het is een open-source oplossing. Er zijn verschillende opties, bij het kiezen wordt er best rekening gehouden met volgende puntjes:
  - De netheid van de code. Is de logging code gemakkelijk te lezen?
  - Wordt de performance beeïnvloedt?
  - Is het framework voor logging al gekend onder het team?
- Logging databases: Log data wordt gebruikt voor het captueren van events. Logs worden nooit aangepast. Logs worden ook gesorteerd op datum en tijd.

Bij microservices wordt elke microservice gelogd. Bij een fout moet er gekeken worden naar alle betrokken microservices. Er moet gekeken worden naar alle logs van de services. Om logging toe te passen wordt er aangeraden om libraries te gebruiken. Enkele best practices, Melendez (2018), Eyee (2018), Timms (2018):

- Probeer te vermijden dat logs in bestanden worden opgeslaan. Logs zijn streams van een flow. Het geeft weer wat er juist gebeurt is in een flow.
- Microservices moeten niet weten waar de logs naartoe gaan. Zo kan de bestemming van het wegschrijven van de logs veranderd worden zonder dat elke microservice ervoor moet aangepast worden.
- De logging zou moeten werken voor alle verschillende codeertalen. Er zou niets moeten worden aangepast bij de configuratie files van het logging systeem.
- Geef elke request een uniek ID. Zo kan de request snel teruggevonden worden bij falen. Of bij het zien van een fout kan er snel achterhaald worden welke request er in fout is gegaan.
- Laat het antwoord ook een uniek ID meesturen. Als de gebruiker dan een fout krijgt, kan er achterhaald worden vanwaar de teruggestuurde fout komt. De administrators kunnen dan de details van de fout bekijken.
- Een oplossing is om alle logs weg te schrijven naar een centrale databank. Zo kan het hele pad van de fout snel en eenvoudig teruggevonden worden. Het duurt langer om verschillende fouten aan elkaar te linken als de logs in de datastore van de microservice worden opgeslaan. Bij het opslaan op één plaats, worden fouten sneller aan elkaar gelinkt. Het wegschrijven naar een plaats is tegen het principe van microservices. Elke microservice op zichzelf. Binnen die enkele database met alle logs kan er gezocht worden op fout, microservice, tijdstip, ... Ook het volgen van een gebruiker zijn traject binnen de applicatie is eenvoudiger. Bij een fout kan er gekeken worden naar de acties die er op voorhand zijn gemaakt.

- Zorg voor structuur in de log data. Een algemene format zoals JSON of XML om een structuur te steken in je logs.
- Geef elke request een context. Weten wat gezorgt heeft voor de fout, is belangrijk om ervoor te zorgen dat de fout niet nog eens voorkomt. Volgende velden zouden zeker in de log terug te moeten vinden zijn:
  - Dag en tijd.
  - Stack errors.
  - De naam van de service, om de logs te linken aan microservices.
  - In welke functie de fout is ontstaan.
  - De naam van de externe service waar er interactie mee is geweest.
  - Het IP adres van de server en van de gebruiker zijn requests.
  - De browser waaruit de gebruiker de request stuurde.
  - De HTTP code om later alerts te creëren.
- Overweeg om de logs naar een lokale databank weg te schrijven. Elke oplossing heeft zijn voordelen en nadelen. Het wegschrijven over HTTP naar de cloud kan zorgen voor meer verkeer op het netwerk. Dus er kan bandbreedte weggenomen worden van belangrijkere microservices.
- Kijk na wat er gelogd wordt, is het niet nodig om iets te loggen, laat het achterwege. Maar bij de start van loggen, wordt er best te veel gelogd. Zodat er eerst teveel info is en dan kan er gesneden worden in de inhoud van het loggen.

2.1.3 Algemene aanpak om microservices te implementeren

Voordat er wordt begonnen aan het overschakelen naar microservices, moet er research gedaan worden, Koukia (2018). De logische eerste stap is het lezen van artikels en hoe andere bedrijven zijn overgeschakeld naar microservices. Wat hun problemen en moeilijkheden waren. Door artikels en ervaringen van anderen te lezen, kan je zelf mogelijke problemen voorkomen. Na de verdieping in microservices, moet er een plan opgemaakt worden. Dit plan wordt best met meerdere mensen samen gemaakt. Zodat er meer mensen kunnen nadenken en om ervoor te zorgen dat iedereen op dezelfde lijn zit.

Benetis (2016a) schreef een 6-stappen plan om microservices te implementeren. Over de grote lijn wordt er daarop gebaseerd, maar per stap gaat er dieper op worden ingegaan. Een paar woorden die meer verklaring nodig hebben voordat we verder gaan. Een gateway is een netwerkpunt dat dient als toegang tot een ander netwerk. Een gateway is een soort toegangspoort. Implementatie is een procesmatige invoering van een verandering of vernieuwing. Iets implementeren of vernieuwen.

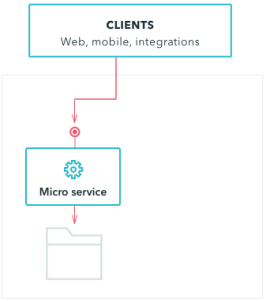
In grote lijnen is dit het zes-stappen plan. Als eerste komt aan bod "serve a business purpose". Hierna komt "protect your stuff". Eens dat gebeurt is, zegt het artikel "see no evil, hear no evil". Dan komt "find your stuffäan bod. Hierna wordt de volgende stap "create a gatewayäangehaald. Als laatste komt "construct eventsäan de beurt.

De eerste stap is "Serve a business purpose". De titel zegt al veel van wat er verwacht wordt. Een microservices is gebaseerd op een business requirement. En niet het doel dat het IT-team voor ogen heeft. Een voorbeeld van een business requirement is het ophalen

van data om die dan te analyseren om daar later dan conclusies uit te trekken. Dit kan in een microservices gegoten worden. Eens het doel voor ogen is bij een microservices, moet er ook gekeken worden naar wat de microservices moet kunnen. Zo lang het bij enkele microservices blijft, is automated deployement etc. niet zo belangrijk. Maar eens we gaan scalen en meerdere microservices in één systeem steken zouden volgende puntjes toch self-sufficient moeten zijn:

- Geautomatiseerde implementatie
- Blootstelling aan andere systemen, toegankelijk eindpunt
- Opslag van data
- Schaalbaarheid en belasting

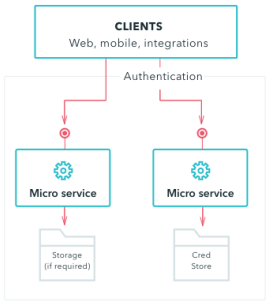
Zoals te zien is op figuur 2.2, is een microservices één klein deeltje in een groot geheel. Later in dit stappenplan zal de figuur ook uitgebreidt worden en zal het geheel duidelijk worden.



Figuur 2.8: Een microservice dat voldoet aan een business requirement. Benetis (2016a)

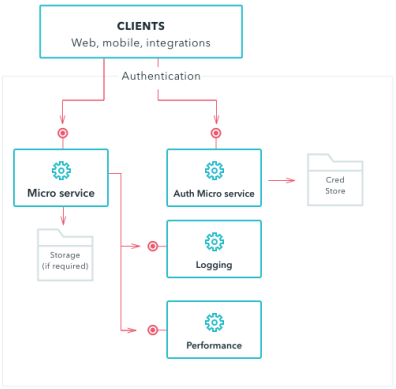
Na "Serve a business purpose"komt "Protect your stuff". Dit gaat over de bescherming van een microservices. Als het gaat over bescherming moet dit op elk moment gebeuren. Ookal heb je maar één à twee microservices, of honderden, bescherming is belangrijk. Het is belangrijk op over al de microservices een uniforme manier te vinden om ze te beschermen. De bescherming kan een requirement op zich zijn, dus ook dit kan in een microservices worden gestoken. Bescherming is een vage term daarom een korte uitleg van hoe een bescherming er zou kunnen uitzien. De meest bekende manier is natuurlijk authorisatie en authenticatie. Authorisatie is het verkrijgen van rechten om bijvoorbeeld een product toe te voegen op een site. Authenticatie is het aanmelden op facebook bijvoorbeeld. De controle of jij het wel echt bent. Een manier op de microservices te beschermen is gecentraliseerde session opslag. Hier zal verder in de thesis nog dieper op worden ingegaan. Kort uitgelegd betekend gecnetraliseerde session opslag dat de data van de user centraal opgeslagen staat. Zodat alle microservices de zelfde session data lezen en gebruiken. In figuur 2.3 is de authenticatie in een microservices gestoken.

Na de eerste twee stappen komt "See no evil, hear no evil". Eens de microservice is opgezet en gedeployed, moet er gemonitord worden. Daarmee wordt bedoeld dat hoe de microserivces zich gedraagd goed moet bijgehouden worden. Alles zou goed moeten gelogd worden, zodat bij een probleem het geen moeite is om te vinden waar het probleem



Figuur 2.9: Een microservice waar bescherming aan is toegevoegd. Benetis (2016a)

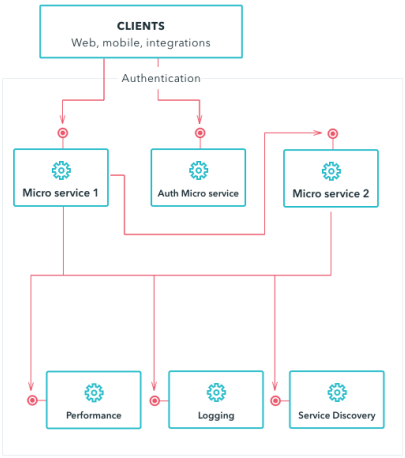
zich voordeed. Ook hier is het aangeraden om doorheen het hele systeem een uniforme manier van loggen aan te houden. Ook kan men hiervan een microservice maken. Dit wordt ook afgebeeld in figuur 2.4.



Figuur 2.10: Een microservice waar er monitoring is aan toegevoegd om chaos te voorkomen. Benetis (2016a)

Als vierde stap komt er "Find your stuff". In deze stap wordt er gezocht naar een manier om de microservices met elkaar te communiceren. Hiermee wordt bedoelt hoe dat microservices A gegevens vraag aan microservices B. Die dat dan ook vraagt van een andere microservice. Een veel gebruikte techniek hiervoor is "service registry". Een service registry is een databank waar alle services met hun instanties en locatie woren opgeslaan. Daar worden dan ook connecties in opgeslaan. Ook hier wordt er aangeraden om dat in een microservices te gieten. Zodat ook hiervan het gedrag kan gemonitord worden. In figuur 2.5 zie je hoe de service registry kan toegevoegd worden. In de figuur is die terug te vinden onder de naam "service discorvery".

Nu is er een service, bestaande uit microservices, dat beschermd is en kan doen wat het zou moeten doen. Maar niet de volledige service moet open en bloot gelegd worden. En daar zorgt stap 5, "Create a gateway", voor. Een API gateway kan een scherm zijn waar je gegevens op invult en mogelijke acties op doet en die spreken dan de correcte microservices aan. De taak van een gateway is voornamelijk zorgen dat request/aanvragen

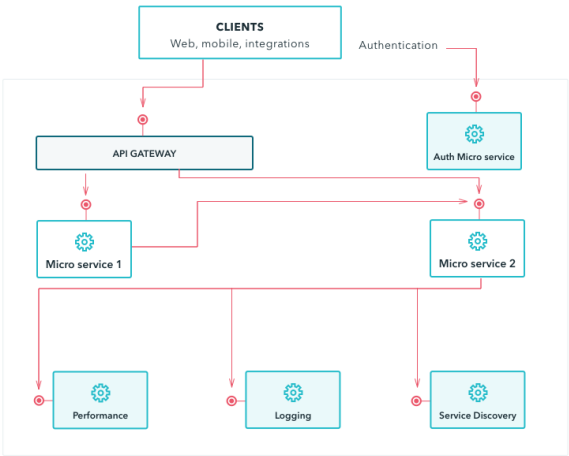


Figuur 2.11: Een microservice dat authenticatie als bescherming toepast. Benetis (2016a)

naar de juiste microservice worden doorgestuurd. Andere taken van een API gateway kunnen volgende zijn

- Beveiliging: een API gateway kan de binnenkomende aanvragen valideren.
- Prestatiegegevens kunnen geregistreerd worden.
- Omzetten van aanvragen in enkele of meerdere microservices.
- Abstractie van de clientinterface. Wanneer er van microservice verandert wordt, moet er niet van interface/scherm verandert worden.

In figuur 2.6 is te zien hoe zo een API gateway kan worden toegepast. Zo is ok te zien dat de authenticatie microservice er niet inzit. Die wordt apart gehouden. De request naar de authenticatie mogen niet langs de API gateway gaan. Omdat ze dan zo meteen 'binnen' zitten. Pas na authenticatie mag men request sturen naar de API gateway.



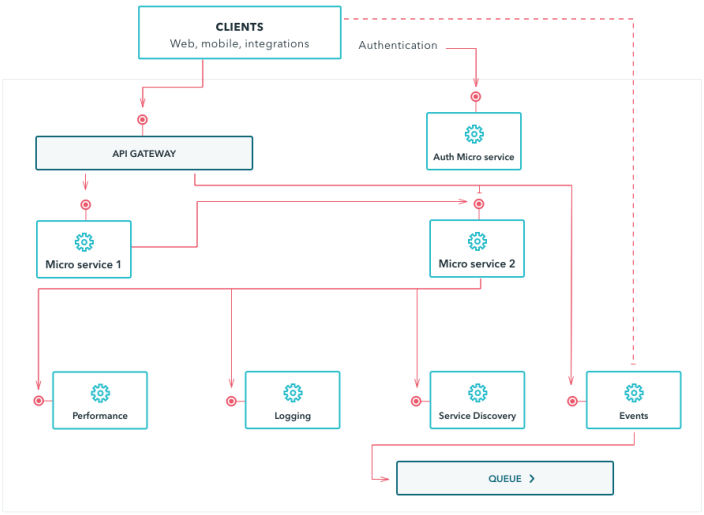
Figuur 2.12: Een microservice dat een gateway gebruikt. Benetis (2016a)

Nu is er al een deftige architectuur aanwezig. Stap 6, "construct events", zal het plaatje

dan ook compleet maken. De meeste microservices vragen aan asynchrone oplossing. Een niet-gelijktijdige verwerking van aanvragen. Een manier om asynchroon te werken, is werken met een queue of wachtrij. Een bekende manier om asynchroniteit toe te passen is publish/subscribe pattroon. Dit wil zeggen dat microservice A zijn berichten of data op een wachtrij gaat zetten. De microservices die data of berichten van microservice A moeten ontvangen, gaan zich abonneren op die wachtrij. Dus vanaf het moment dat microservice A iets op die wachtrij plaatst, krijgen de geabonneerden een melding en kunnen ze het bericht of de data gaan ophalen. Enkele voordelen van dit als asynchrone oplossing te gebruiken:

- Taken inplannen. Dit kan door deze gewoon op de wachtrij te plaatsen met een timestamp van wanneer deze moet gebeuren of door een wachtrij te maken voor geplande events.
- Abonneren op bepaalde events.
- Het asynchrone systeem laten bloot leggen zodat externe klanten verschillende notificaties kunnen handelen.

In figuur 2.7 wordt de volledige architectuur weergegeven. Daar wordt er ook mooi afgebeeld hoe men events pland. Als event A voor event B moet gebeuren dan zetten ze die zo op de wachtrij event A voor event B. Want een wachtrij werkt volgens het FIFO (first in first out) principe.



Figuur 2.13: Een microservice met asynchronisatie. Benetis (2016a)

Benetis (2016b) beschrijft dat over volgende puntjes goed moet worden nagedacht voordat men de overschakeling maakt naar microservices:

- Heeft de organisatie de microservice architectuur wel nodig?
- Zijn de juiste competenties aanwezig? Microservices zijn in het algemeen complexer dan een monolithic. Zeker omdat dit iets nieuws is.
- Staat iedereen achter deze verandering?

Eens de beslissing gevallen is om over te schakelen, moet er beslist worden hoeveel de infrastructurele vernadringen van de scope inpalmen. Het verloop om microservices te creëeren gaat als volgt in dit artikel. Eerst het uit elkaar trekken van het bestaande systeem, om het dan in microservices te steken, is een goed begin. Zo moet er constant nagedacht worden over de algemene infrastructuur. Een groot valluik in het begin van het proces is een proof of concept maken. Dit eindigt meestal in een infrastructuur dat niet overeenkomt met de waarden van microservices. Het probleem ligt dan meestal bij een onduidelijke scope. De business requirements zijn meestal wel duidelijk, dit geldt dan meestal niet voor niet-functionele requirements. Daarnaast moeten ook nog volgende stappen gerealiseerd worden:

- Bescherming.
- Deployment automatisatie.
- Loggen en monitoren van microservices hun gedrag.

Velen komen niet tot deze stap, door de onderschatting van de overschakeling naar microservices.

2.1.4 De voordelen en nadelen van microserivces

In het artikel van **series2018** wordt er veel lofzang gedaan over microservices. Het gebruik van microservices zou ervoor zorgen dat de architectuur flexibeler wordt. Met flexibeler wordt bedoelt dat de architectuur zich kan 'aanpassen' of kan inspelen op verschillende situaties. Er kunnen microservices hergebruikt worden. Dankzij microservices is het hermodeleren, implementeren van nieuwe technologieën, ... Kleinere deeltjes zijn gemakkelijker te documenteren. De snelheid van microservices zijn een groot pluspunt. Hiermee wordt er geprobeert om aan te halen dat microservices sneller reageren omdat zo kleine, onafhankelijke services zijn. Ze moeten geen 'onnodige' stappen maken om de wens van de klant te vervullen. Watts (2018) geeft enkele voordelen van een microservice. Een developer is onafhankelijk. Ze hebben vrijheid. Ook het scalen van een microservice is veel eenvoudiger. Dit komt door dat microservices minder resources nodig hebben dan een volledige monolithic. Resources zijn hulpbronnen. Zoals al vaak aangehaald in deze bachelorproef, zijn microservices onafhankelijk en zouden ze daarom ook geen hulpbronnen nodig mogen hebben. Binnen een monolithic zijn deeltjes afhankelijk van elkaar en hebben elkaar dus nodig om goed te kunnen functioneren. De deeltjes binnen de monolithic hebben elkaar dus nodig en mogelijk als hulpbron. Een ander voordeel is bij het falen van een microservices, de andere microservices er geen last van zullen hebben. Dit komt door hun onafhankelijkheid. Benetis (2016a) geeft aan dat volgende puntjes voordelen zijn van microservices:

- Sneller en gemakkelijker developen.
- Het refactoren van deeltjes is eenvoudiger door de onafhankelijkheid van de services.
- De schaalbaarheid is eenvoudiger dan bij een monolithic. We kunnen microservices gewoon 'klonen' of kopieëren.
- Het deployen van een onderdeel gaat sneller omdat het team gespecialiseerd is in die bepaalde service.

- Als er iets faalt dan is de impact veel kleiner dan bij een monolithic. Dit komt ook door de onafhankelijkheid van de services.

Maar om ervoor te zorgen dat dit allemaal vlot verloopt moeten er ook aanpassingen binnen in het bedrijf/ de organisatie gebeuren.

- Een project zal ingedeeld moeten worden in kleine requirements. De scope zal gedetailleerder moeten zijn.
- De teams zullen kleiner moeten worden gemaakt. Zodat er meer op de Agile methode kan gewerkt worden.
- Er zal een sterke band komen met DevOps. Dit komt omdat veel services volledige automatische deployment vragen.
- Ook de communicatie tussen de services zal beter moeten worden uitgedacht.
- Documenteren is belangrijk. Dit is niet enkel het geval voor microservices maar ook voor elk project.

Enkele nadelen en moeilijkheden, Koukia (2018):

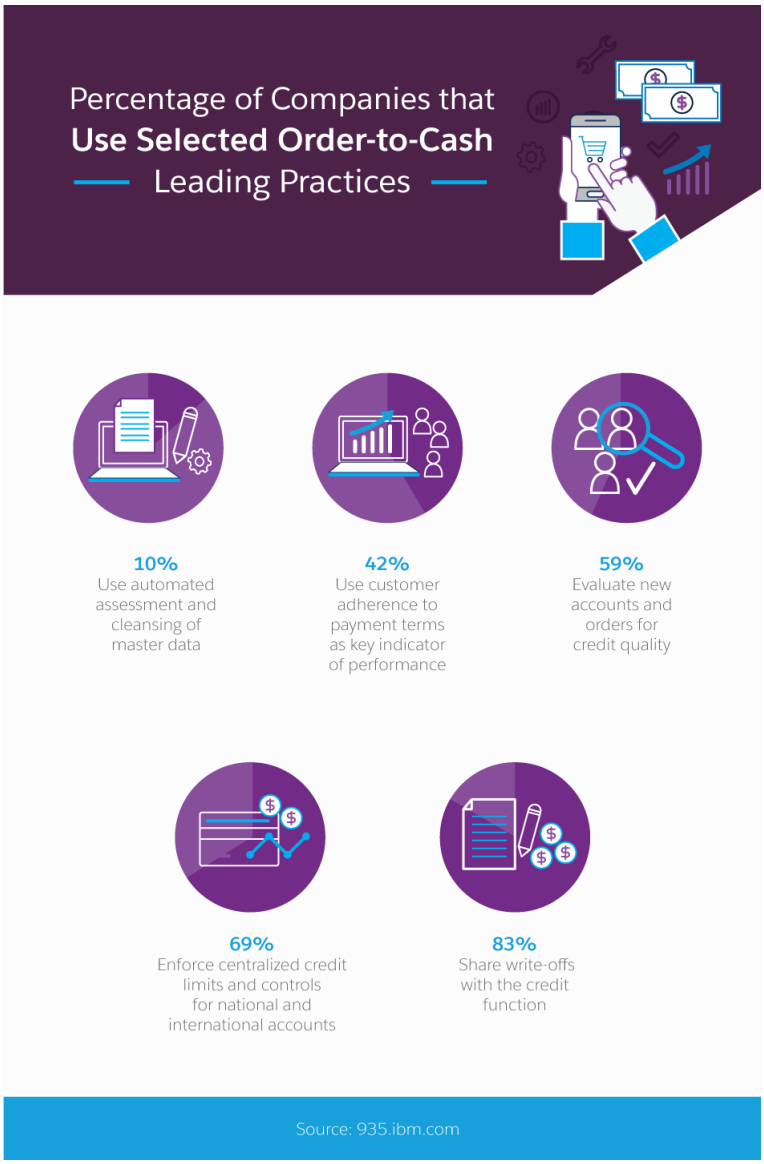
- Distributed systems are complex. Microservices vragen meer werk. Dit komt omdat het een nieuwere technology is en het concept is niet altijd meteen duidelijk.
- Complexiteit is er overal. Een monolithic bevat ook complexiteit, maar wel een bekende complexiteit. Als er al een tijdje gewerkt wordt met monolithic, dan is die complexiteit een bekende eigenschap van de architectuur. Omdat microservices iets nieuws is, komt er complexiteit die minder bekend is onder de developers.
- Debuggen en fouten vinden is moeilijker. Bij een monolithic is het eenvoudig, bij aanpassingen wordt de volledige architectuur gerund. Als de gehele architectuur bij een microservices moet getest worden. Hier werd in het vorige deel meer uitleg over gegeven.
- Bij aanpassingen binnen een monolithic moest er maar één request gedaan worden om code samen te voegen. Bij microservices is dit verschillend. Er moet bijvoorbeeld in vier microservices iets aangepast worden voor een nieuwe feature. Dan moeten er vier requests verstuurd worden om die code te mergen. Want de services binnen de microservice architectuur zijn onafhankelijk van elkaar.

2.2 Order-to-cash proces in SAP

2.2.1 Definite

Wong (2018) legt uit wat een order-to-cash proces inhoudt. Dit proces heeft veel invloed op het succes van een bedrijf. Ook het het veel invloed op de relatie met de klant. Een voordeel met de huidige technologie is dat het mogelijk is om het proces volledig te automatiseren. Dit heeft veel zorgt voor een minimaal aan fouten en vertragingen. Ook de data en gegevens die worden opgehaald en geanalyseerd is correcter. Het proces begint bij het plaatsen van een order door de klant. Alles wat ervoor zorgt dat de klant een order plaatst behoort tot branding, marketing of sales. De hoofdactiviteit van deze afdelingen ligt

bij customer relationship, iets wat plaatsvindt voor het OTC proces. Het maakt er geen deel van uit. Velen denken dat een OTC is afgerond wanneer de inning heeft plaatsgevonden. Maar er zijn nog belangrijke stappen die gebeuren na het innen van het geld. Onder andere de data die verzameld is tijdens het proces moet geanalyseerd worden om zo het proces te optimaliseren. Het OTC proces heeft invloed op volgende delen van het bedrijf:



Figuur 2.14: Het percentage van van bedrijven dat gebruik maken van order-to-cash proces. Wong (2018)

- Supply chaing management
- Voorraadbeheer
- Human resources
- Financiële afdeling

Zijn er problemen in één van die afdelingen, kan dit voor vertraging zorgen in de andere

afdelingen. Bij vertragingen van betalingen of inningen kan zorgen voor een minder goede cash flow binnen het bedrijf. Een goed OTC proces maakt indruk op de buitenwereld. Doordat het OTC goed gemanaged wordt, wordt er een beeld gecreëerd dat het bedrijf stabiel is. Ook bij een OTC is technologie cruciaal. Elk deeltje van het proces kan beter worden door de nieuwe technologie. Er zijn verschillende onderdelen nodig om het proces te optimaliseren om accurate en real-time informatie te verkrijgen. Zoals:

- Gegevens die onderling verbonden zijn.
- Automatisering
- digitale facturatie
- Digitale verzending van het management.

Volgende acht stappen komen voor in een OTC proces:

- Order management
- Credit management
- Order fulfillment
- Order shipping
- Facturatie
- Accounts receivable
- Inning van het geld
- Rapportering en data management

In figuur X wordt het afgebeeld in een schema. Order management is de eerste stap in het proces. Dit begint wanneer de klant een order plaatst. De manier waarop is niet zo belangrijk. Dit deeltje van het proces moet geautomatiseerd zijn. Bij het plaatsen van een order, moet er een ander onderdeel van het order proces getriggerd worden. Dit moet ervoor zorgen dat het order niet uit het oog verloren wordt. Door de automatisatie worden de andere deelnemende partijen direct ingelicht over het nieuwe order en dit heeft een voordeel als het op tijdig leveren aankomt. Hierna komt credit management. Dit moet ervoor zorgen dat er minder problemen zijn op het einde van het proces. Credit management houdt in dat men kijkt naar hoe het betalingsgedrag van de klant is geweest. Zijn er nog openstaande facturen, betaald de klant altijd maar na enkele aanmaningen? Door dit gedeelte te automatiseren, kan er bespaard worden op menskracht en geld. Als er toch dieper moet gekeken worden naar het betaalgedrag van een klant, kan dit doorgestuurd worden naar een werknemer die er dan naar kijkt. Maar dat zouden dan enkele klanten hun betaalgedrag moeten nagekeken worden, i.d.p.v. alle klanten die een order plaatsen. De klanten die een goed betaalgedrag vertonen, worden doorgestuurd naar de volgende stap: Order fulfillment. In deze stap wordt het order ook echt samengesteld en uit de 'rekken' gehaald. Het is een groot voordeel als ook dit gedeelte geautomatiseerd is. Bij het verkopen van een product moet de voorraad automatisch aangepast worden. Eens de producten van het order samengebracht zijn gaan we over naar de volgende stap, order shipping. De verzending van de goederen. De verzending moet goed opgevolgd worden om mogelijke vertragingen te minimaliseren. De gegevens die vrijkomen bij een verzending, moeten zo snel mogelijk in het systeem worden ignegeven. Zodat ook dit deeltje van het proces geoptimaliseerd kan worden. Na het verzenden van de goederen komt de facturatie. Op dit deeltje heeft credit management veel invloed. Doordat de wanbetalers er in stap

twee al zijn uitgehaald, zouden er hier minder problemen voorkomen. Als er hier fouten worden gemaakt, kan dit een sneeuwbal effect veroorzaken. Dit betekend dat één fout een andere fout kan triggeren en zo voorts. Het systeem moet de juiste info verkrijgen van de werknemers. De info bevat meestal volgende puntjes:

- Order specificaties
- De kosten
- Credit terms
- Order datum
- Verzendingsdatum

Die puntjes moeten ingevoerd worden zodat het facturatiesysteem geautomatiseerd kan worden. Zodat ook hier vertragingen en fouten kunnen geminimaliseerd worden. Eens de factuur is uitgestuurd, wordt er een betaling verwacht binnen een bepaalde periode. Het systeem zou dit ook moeten bijhouden en ervoor zorgen dat er een melding gestuurd wordt nog voor de betalingsperiode is afgelopen. Dit valt onder de volgende stap accoutns receivable. Deze stap probeert om te voorkomen dat mensen vergeten te betalen. De volgende stap is payment collections. Wordt een factuur niet betaald binnen de gevraagde periode dan wordt er een aanmaning gestuurd en wordt dit ook in het systeem bijgehouden. De werknemers moeten ook de klanten contacteren zodat ze een reden kunnen geven voor het mogelijks vergeten van de betaling. Als laatste stap komt reporting en data management aan bod. Er bestaan programma's om ervoor te zorgen dat performance data over elk deeltje van het proces wordt opgehaald. Door achteraf deze data te gaan analyseren, kan er veel duidelijkheid komen van waar het verkeerd loopt.

PEARSON (2017) beschrijft waarom het zo belangrijk is om een goed OTC proces te hebben. Vroeger waren de eisen van de klanten minder hoog. Als klanten iets bestellen, willen ze het de volgende dag al in huis hebben. Er zijn ook voor- en nadelen aan een OTC proces. Als het proces goed opgezet is, kan dit zorgen voor blijere klanten, minder wanbetalers, ...

Maar is het order management proces niet goed opgezet dan kan het heel snel slecht gaan. Klanten zullen niet tevreden zijn. Veel voorkomende redenen van ontevreden klanten:

- Orders zitten er dubbel in.
- Er zit vertraging tussen de verschillende onderdelen van het proces.
- De levering klopt niet met wat er gefactureerd wordt.
- Een slecht voorraadbeheer waardoor niet aan de beloften zoals leveringstijd kan voldaan worden.

Om zulke problemen op te lossen moet men eerst gaan zoeken van waar die problemen komen. Automatisering is hier een goede oplossing voor. Dordaat verschillende processen aan elkaar gelinkt zijn, is er minder ruimte voor vertragingen. Enkele best practices omvatten:

- Minimaliseren van werknemers die handmatig gegevens moeten invullen.
- Klanten de mogelijkheid geven om hun bestellingen online te maken.
- Integratie van de informatie over het gehele proces.

- Elimineren van onnodige moeilijkheden binnen in het proces.

Ook het tweede subproces, bill-to-cash proces, heeft pitfalls en best practices. In volgende opsomming komen verschillende signalen aan bod waaraan te zien is dat het bill-to-cash proces niet goed in elkaar zit.

- Het aantal verkoopsovereenkomsten met speciale eisen van de klant is groot. bivjoorbeeld meer dan de helft heeft een uniek verkoopsovereenkomst.
- Herhalend aanbiedingen en toegevingen moeten doen door fouten in het proces.
- Wat op dee oorspronkelijke offerte stond, werd niet gefactureerd, er werd meer aangerekend.
- Een groot aantal creditnota's uitgegeven.
- Tussen de verkoop, verzending en het facturerings proces zitten informatie gaten.

Om dit alles weg te werken moet er goed gekeken worden naar waar de problemen nu zitten. Enkele best practices:

- Integreer klantenprofielen in de betalingssoftware.
- Integreer de facturatie en betalingsgegevens met elkaar zodat op de factuur de juiste betalingsmethode staat.

2.2.2 Technologie

Onderdelen van een order-to-cash proces

Er zijn vier grote onderdelen, namelijk:

- Voor-verkoopsactiviteiten.
- Het order proces.
- Order afwerking.
- Betaling.

Bij de voor-verkoopsactiviteiten verstaan we het contact dat moet worden gemaakt worden met klant. De klant moet overtuigd worden van het product. Na contact komt er al dan niet een offerte. Soms kan er ook van contact rechtstreeks naar een order gaan. Het orderproces bevat maar één onderdeel namelijk: de sales order. Binnen de order afwerking valt het leveren van goederen, het verzenden van goederen. Kumaran (2015) geeft een mooi overzicht van de stappen die een order to cash proces kan bevatten. Zo begint het artikel met een toelichting dat dit proces een core proces is van de business. Dit wil zeggen dat een bedrijf, zonder dit proces, geen winst kan maken. Het is een essentieel onderdeel van zo wat elk bedrijf. Hoe deze vorm krijgt binnen een bedrijf, is heel verschillend. Een OTC proces start met het ontvangen van orders. Daarna kan, dit gebeurt niet overal, er gekeken worden naar het krediet van de klanten. Mag deze klant wel nog een bestelling plaatsen? Hierna wordt het order opgenomen in het systeem. Het product wordt verzonden en geleverd. En als laatste wordt de factuur betaald. Dit is welliswaar het 'perfecte' verloop van een order to cash proces. Er zijn ook enkele uitdagingen verbonden met dit proces. In de volgende opsomming vindt je er enkele:

Delivery document	Dit is het leveringsdocument dat gekoppeld is aan een order. Dit wordt uitgestuurd om te bewijzen dat het order geleverd is.
Picking	Het nagaan of de goederen in de voorraad zitten en ze ook gaan ophalen uit het magazijn.
Packing	Het inpakken van goederen in dozen, containers, palletten.
Loading	De goederen inladen in de vrachtwagen of de container.
Shipment document	Dit document wordt mee afgeleverd met de goederen. Het kan als een soort checklist dienen om na te gaan of alles wel in het afgeleverd pakket zit.
Shipment cost document	Hierop is de kost terug te vinden van het transport.
Customer billing document	Dit is een verkoopsdocument, dus dit moet de klant niet betalen. Dit document refereerd naar het delivery document.
Customer invoice	Dit is de factuur die de klant opgestuurd krijgt.

Tabel 2.3: Tabel met uitleg over termen. concept hub (2019)

- Hou de orders goed bij, anders loopt het fout vanaf het begin.
- Zonder automatisering verlies je veel tijd. Kostbare tijd.
- Zorgen dat de logistiek 'on point' is, is heel belangrijk. Dit is een belangrijk onderdeel binnen het proces.
- Bij wanbetalingen, moet de customer service ingrijpen. Ook zij zijn een belangrijk onderdeel van het proces.

Het managen van een OTC proces is even belangrijk als de andere aspecten. Een slecht gemanaged proces kan op lange termijn duurder uitkomen dan een onderdeel dat nog niet geautomatiseerd is.

PEARSON (2017) vindt dat een OTC proces bestaat uit twee subprocessen. Het order managmenet proces en het bill-to-cash proces. Het laatst vernoemde proces gaat over de facturatie en de inning van het geld.

concept hub (2019) geeft meer uitleg bij enkele termen die in SAP voorkomen. Bij het proces is er een sales team aanwezig. Die spreken met de klant en maken afspraken. Zij zorgen ook voor een deal met de klant. Zij hebben toegang tot informatie over de klant, producten, kwaliteit, prijzen, etc. Om ervoor te zorgen dat een order kan verzonden worden, moeten de goederen opgehaald worden. Die goederen worden opgeslaan in een magazijn. Daar wordt aan order picking en packing gedaan. Bij picking gaat men gaan kijken naar de voorraad om de order te kunnen opvullen. Packing gaat over het inpakken van de goederen tegen beschadigingen. Dit kan door allemaal dozen op een pallet te zetten. Eens de goederen door de fase picking en packing zijn geweest, komt het laden voor transport. De goederen worden dan in containers geladen om dan te verzenden via vrachtwagen of per boot. Hierna komt de facturatie. Dan moet er gewacht worden op de betaling van de klant. Bij laattijdige betaling worden er aanmaningen gestuurd.

Wat biedt SAP zelf aan voor microservices

Kyma (2019) verteld hoe Kyma in elkaar zit. Dit is een recent project van SAP om toenadering te geven toto microserivces. Kyma is een open-source project gemaakt om Kubernetes. Kubernetes wordt gebruikt om applicaties op verschillende machines te managen. Je kan hiermee cloud-based applicaties en on-premise applicaties omzetten naar een microservice architectuur of naar serverless computing. Cloud-based applicaties zijn applicaties die hun data gaan ophalen over het internet in de plaats van op de harde schijf van de computer. On-premise applicaties zijn de tegenhangers van cloud-based. On-premise is lokaal op de computer, op de harde schijf. Serverless computing gebeurt in de cloud. Dit zorgt ervoor dat er op een dynamische manier de resources van een machine kan aangepast worden. Kyma zorgt voor betere end-to-end ervarings scenario's. Die volgen een best-practices voor performance, schaalbaarheid, efficentie en beveiliging.

Semerdzhiev (2018) geeft een introductie in het project Kyma van SAP. Binnen SAP wordt er omgegaan met software van verschillende leveranciers. SAP probeert om hun software te customizen naar de wensen van de klant. Dit vraagt meer openheid en een modernere architectuur. Het idee achter Kyma is het creëeren van serverless applicaties, mashups en microservices. Ook kan het gebruikt worden om snel kleine, gecustomizede modules te developen. Die modules zijn dan verworven met business logica. Er werd iets zoals Knative gemaakt. Knative is een platform dat developers ondersteund om serverless applicaties te maken op Kubernetes. Dit zorgde voor groot enthousiasme bij SAP omdat hun Kyma project een soort van bevesteging keer. Al snel werd Kyma gerefactored om samen te kunnen werken met Knative. Er werden overlappende componenten weggelaten, wat Kyma slanker en gestroomlijnder maakt. Dankzij Knative kan Kyma zich richten op higer-level enterprise applications en service consumption scenario's. En gebruik maken van Knative voor de infrastructuur en development scenario's.

Hofmann (2018) geeft het update over de integratie tussen Kyma en Knative. Een toch redelijk belangrijke samenwerking. Beide bieden een complete set van bouwblokken aan. Beiden zijn ook een sterk framework. Bij het gebruik van beide kunnen er could-native oplossingen gebouwd worden op Kubernetes met een sterk framework. Op figuur X is te wat Kyma en wat Knative aanbiedt. De laatste maanden werden de repositories gereconstrueerd en daarom zijn er geen tot weinig referenties naar Knative. Naast die grote verandering werd er gewerkt aan een proof-of-concept cloud-native oplossing door het gebruik van Knative en Kyma te samen.

Kyma heeft hard gewerkt om deze problemen/uitdagingen te proberen wegwerken. Kyma heeft ook geprobeerd om bedrijven te helpen met de transformatie naar digitalisatie. Kyma geeft bedrijven de mogelijkheid tot:

- Be open and extendable: Kyma maakt gebruik van Open Service Broker API specificaties. Dit is een "plug and play"die de mogelijkheid geeft om code te hergebruiken. Ook code van andere partijen te gebruiken.
- Be seamlessly connected: Een eenvoudige manier om een beveiligde connectie te hebben tussen systemen. Deze kunnen gemanaged worden binnen de huidige applicatie om oplossingen te maken in hetrogene landschappen. Eén connectie geeft

veel mogelijkheden.
- Use any programming language: Developers kunnen in hun geweste programmeertaal coderen.
- Bring speed and agility: Er moet niet maanden gewacht worden om use cases of functionaliteiten op te leveren.
- Accelerate innovation: Meestal start dit als een test of een trail. Bij zulke scenario's zijn de kost en de snelheid van zeer groot belang. Dankzij Kyma kunnen bedrijven meteen beginnen werken aan een oplossingen en moet er geen tijd besteed worden aan het zoeken van de best mogelijke oplossing.

2.3 Requirements van de business

Biedron (2018) legt uit wat een order to cash proces is. Daaruit kunnen we volgende requirements uit afleiden:

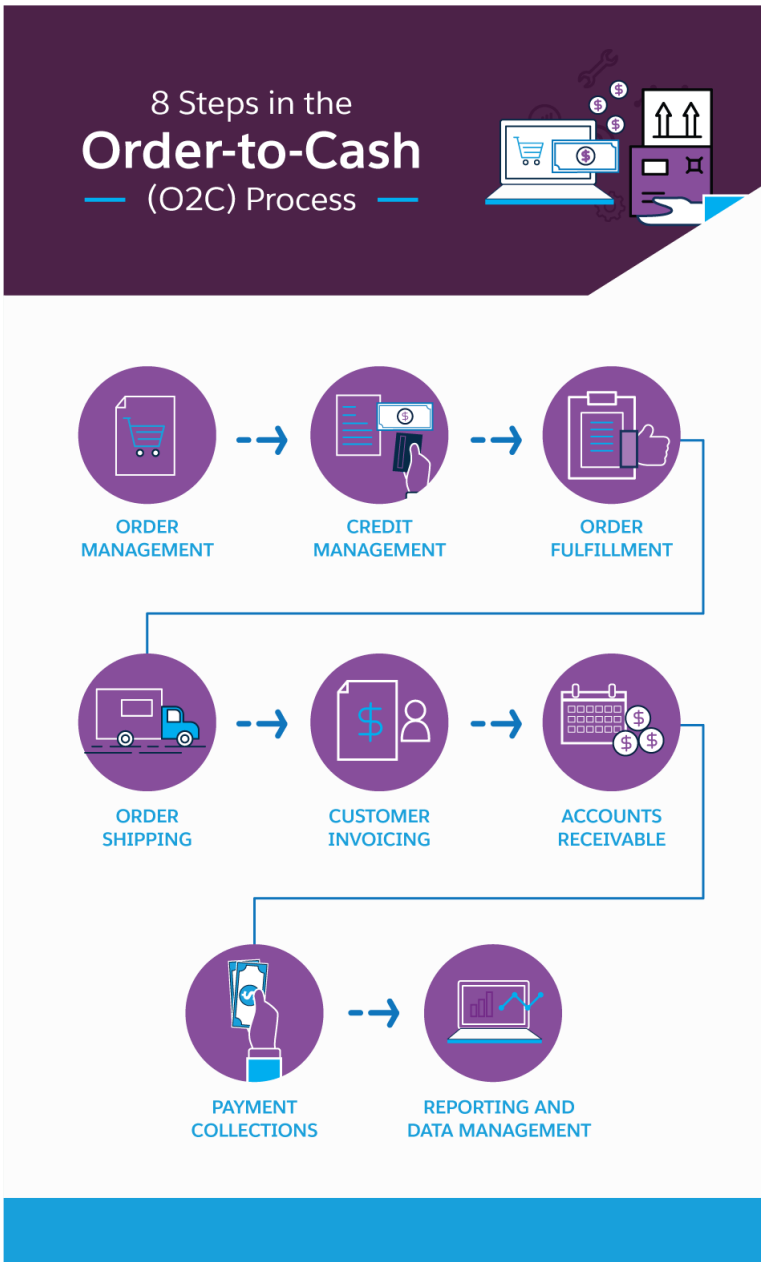
- De klant moet een order of meerdere orders kunnen plaatsen. Een goed werkend order-systeem is een must.
- De order ophalen uit de voorraad. Er moet een goed voorraad beheer aanwezig zijn. Dit moet ervoor zorgen dat het aantal laattijdige leveringen geminimaliseerd wordt. Samen met het goede voorraadbeheer wordt ook best bijgehouden waar je je goederen geplaatst hebt in je magazijn.
- De levering moet goed gepland worden. Dit zou voor het grootste deel al geautomatiseerd moeten zijn. Een email met informatie over de levering is een must.
- Aanmaken van een factuur op basis van het geplaatst order met de juiste klantgegevens zou een geautmatiseerd onderdeel moeten zijn.
- De betaling van de factuur komt toe in de financiele afdeling. Automatische afhandeling is een must.

Bij een order plaatsen komen volgende elementen aanbod:

- Er moet een lijst met producten beschikbaar zijn.
- De klant zijn gegevens moeten gekend zijn bij het bedrijf.
- Het systeem bij het bedrijf moet beschikbaar zijn.

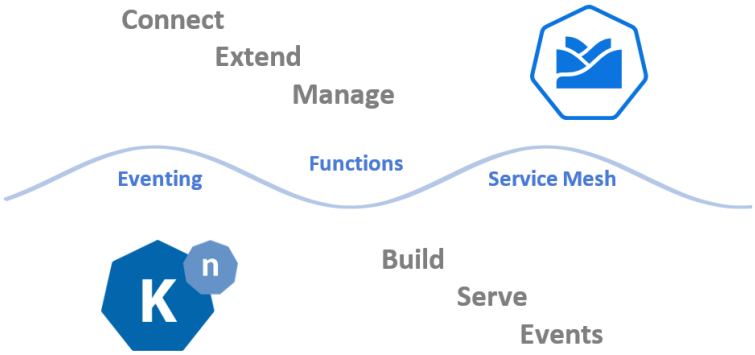
Na het plaatsen van de order moet die ook klaargezet worden om te leveren. Daar zijn volgende elementen van belang:

- Een goed voorraadbeheer is de eerste must.
- Een overzicht van waar alles staat, geeft ook een meerwaarde.
- Bij het ophalen van een product om bij een order te plaatsen, moet de hoeveelheid in de voorraad verminderen.



Figuur 2.15: Het order-to-cash proces. Wong (2018)

Figuur 2.16: Order-to-cash proces volgens Kumaran (2015).



Figuur 2.17: Wat Kyma, boven de lijn, aanbiedt en wat Knative, onder de lijn, aanbiedt Hofmann (2018).

3. Methodologie

3.1 Uitleg termen

In tabel 3.1 zijn termen terug te vinden die in dit hoofdstuk regelmatig zullen voorkomen.

3.2 Communicatie methode

Microservices moeten met elkaar kunnen communiceren. Bijvoorbeeld: Er wordt een order geplaatst door klant 66. Om na te gaan of die klant wel een order mag plaatsen, wordt de klant nummer doorgestuurd naar credit management. Om toch onafhankelijk van elkaar te blijven, gaan de microservices niet controleren of de data is aangekomen bij de andere microservices. De berichten worden op de queue van de microservice geplaatst. Dan is de microservice zelf verantwoordelijk voor het ophalen van hun data. Het ophalen van de data gebeurt via consumption en acknowledgement. Elke keer er een bericht geplaatst wordt

Queue	Een wachtrij waar berichten op geplaatst worden. Het bericht op de queue kan maar één keer gelezen worden.
Consumption	De term om een bericht van de queue te lezen.
Acknowledgement	Het verwijderen van een bericht op de queue.
Overhead	Als er teveel data aanwezig is op de queue dan spreekt men van overhead. Dit kan er voor zorgen dat de queue geen berichten meer ontvangt.

Tabel 3.1: Termen die vaker voorkomen in dit hoofdstuk.

op de queue, wordt de consumption en acknowledgement getriggerd om te gebeuren. Zo kunnen er meerdere credit controles gelijklopend gebeuren. Dus volgende queue's zouden moeten bestaan:

- QorderMan is een queue voor order management: Het plaatsen van een order.
- QcreditMan is een queue voor credit management: Om te controleren of een klant wel een order mag plaatsen.
- QorderFul is een queue voor order fulfilment: Het ophalen van de order in het magazijn.
- QorderShip is een queue voor order shipment: Het plannen van de route en welke goederen op welke vrachtwagen moeten geladen worden.
- Qfact is een queue voor de facturatie.
- QaccountsRec is een queue voor accounts receivable: De betaling van de factuur nagaan en tijdig aanmaningen sturen.

Niet alle data wordt volledig naar de queue gestuurd enkel de belangrijke data. Zoals bijvoorbeeld de klant die een order plaatste, die zijn klantnummer zal doorgestuurd worden naar credit management. Bij credit manangement wordt er dan aan de hand van het klantnummer de gegevens opgehaald en dan zo nagegaan of die klant wel een order mag plaatsen. Zo blijft de overhead op de queue minimaal. Om meer gegevens op te halen, moet de databank aangesproken worden. De gehele structuur van de databank wordt beschreven in het volgende gedeelte.

3.3 De databank structuur

Onderliggend is één grote databank waar alle masterdata in terug te vinden is. Hier is de enige plaats waar een single point-of-failure terug te vinden is. Volgende databanken zullen voorkomen in deze uitwerking:

- Klant gegevens:
  - Klant nummer,
  - naam (voornaam, achternaam),
  - adres,
  - geboortedatum,
- Order gegevens:
  - Ordernummer,
  - klantnummer: om na te gaan aan welke klant dit order gelinkt is,
  - betalingswijze,
  - totaal bedrag,
  - timestamp van aanmaak
- Orderlijnen:
  - Orderlijnnummer,
  - ordernummer,
  - productid,
  - hoeveelheid,

  - kostprijs per product,
  - totale kostprijs orderlijn.
- Product gegevens:
  - Product id,
  - productnaam,
  - beschrijving,
  - in voorraad
- Facturatie:
  - Factuurnummer,
  - ordernummer,
  - klantnummer,
  - betaald,
  - timestamp,
  - totaal bedrag BTW,
  - totaal bedrag factuur.

3.4 De verschillende microservices

Welke heb ik uit de huidige architectuur kunnen halen.

- Wat kunnen ze
- Hoe de db eruitziet
- welk deeltje van de databank spreken ze aan
- Waar kunnen ze gebruikt worden
- Waarom werd hiervan een microservice gemaakt.

De tot nu toe gevonden microservices

- Klant gegevens ophalen
  - Databank met alle klantgegevens
  - Order management
  - Credit management
  - Klant management
  - Facturatie
  - Accounts receivables
- Orders plaatsen, ophalen, verwijderen
  - Databank met alle orders
  - elk order heeft klantnummer
  - Timestamp
  - Databank voor orderlijnen om order samen te kunnen stellen
  - Order management
  - Order fullfilment
  - Facturatie
- Producten ophalen en aantal veranderen
  - nog niks

  - Order management
  - order fullfilment
- Facturatie maken, ophalen
  - Klantnr.
  - OrderNr.
  - timestamp
  - vlag betaald
  - Aantal dagen overdue
- Shipment doc opstellen
  - Order ophalen
  - order shipment
- Aanmaning opmaken, verwijderen
  - Accounts receivables

3.5 De complete architectuur opbouwen

Stap per stap in het proces alles aanhalen

- welke microservices aanspreken
- waarom?

4. Conclusie

Curabitur nunc magna, posuere eget, venenatis eu, vehicula ac, velit. Aenean ornare, massa a accumsan pulvinar, quam lorem laoreet purus, eu sodales magna risus molestie lorem. Nunc erat velit, hendrerit quis, malesuada ut, aliquam vitae, wisi. Sed posuere. Suspendisse ipsum arcu, scelerisque nec, aliquam eu, molestie tincidunt, justo. Phasellus iaculis. Sed posuere lorem non ipsum. Pellentesque dapibus. Suspendisse quam libero, laoreet a, tincidunt eget, consequat at, est. Nullam ut lectus non enim consequat facilisis. Mauris leo. Quisque pede ligula, auctor vel, pellentesque vel, posuere id, turpis. Cras ipsum sem, cursus et, facilisis ut, tempus euismod, quam. Suspendisse tristique dolor eu orci. Mauris mattis. Aenean semper. Vivamus tortor magna, facilisis id, varius mattis, hendrerit in, justo. Integer purus.

Vivamus adipiscing. Curabitur imperdiet tempus turpis. Vivamus sapien dolor, congue venenatis, euismod eget, porta rhoncus, magna. Proin condimentum pretium enim. Fusce fringilla, libero et venenatis facilisis, eros enim cursus arcu, vitae facilisis odio augue vitae orci. Aliquam varius nibh ut odio. Sed condimentum condimentum nunc. Pellentesque eget massa. Pellentesque quis mauris. Donec ut ligula ac pede pulvinar lobortis. Pellentesque euismod. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent elit. Ut laoreet ornare est. Phasellus gravida vulputate nulla. Donec sit amet arcu ut sem tempor malesuada. Praesent hendrerit augue in urna. Proin enim ante, ornare vel, consequat ut, blandit in, justo. Donec felis elit, dignissim sed, sagittis ut, ullamcorper a, nulla. Aenean pharetra vulputate odio.

Quisque enim. Proin velit neque, tristique eu, eleifend eget, vestibulum nec, lacus. Vivamus odio. Duis odio urna, vehicula in, elementum aliquam, aliquet laoreet, tellus. Sed velit. Sed vel mi ac elit aliquet interdum. Etiam sapien neque, convallis et, aliquet vel, auctor non, arcu. Aliquam suscipit aliquam lectus. Proin tincidunt magna sed wisi. Integer blandit

lacus ut lorem. Sed luctus justo sed enim.

Morbi malesuada hendrerit dui. Nunc mauris leo, dapibus sit amet, vestibulum et, commodo id, est. Pellentesque purus. Pellentesque tristique, nunc ac pulvinar adipiscing, justo eros consequat lectus, sit amet posuere lectus neque vel augue. Cras consectetuer libero ac eros. Ut eget massa. Fusce sit amet enim eleifend sem dictum auctor. In eget risus luctus wisi convallis pulvinar. Vivamus sapien risus, tempor in, viverra in, aliquet pellentesque, eros. Aliquam euismod libero a sem.

Nunc velit augue, scelerisque dignissim, lobortis et, aliquam in, risus. In eu eros. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Curabitur vulputate elit viverra augue. Mauris fringilla, tortor sit amet malesuada mollis, sapien mi dapibus odio, ac imperdiet ligula enim eget nisl. Quisque vitae pede a pede aliquet suscipit. Phasellus tellus pede, viverra vestibulum, gravida id, laoreet in, justo. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Integer commodo luctus lectus. Mauris justo. Duis varius eros. Sed quam. Cras lacus eros, rutrum eget, varius quis, convallis iaculis, velit. Mauris imperdiet, metus at tristique venenatis, purus neque pellentesque mauris, a ultrices elit lacus nec tortor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent malesuada. Nam lacus lectus, auctor sit amet, malesuada vel, elementum eget, metus. Duis neque pede, facilisis eget, egestas elementum, nonummy id, neque.

A. Onderzoeksvoorstel

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

A.1 Introductie

Hoe Microservice integration patterns een order to cash proces in SAP beinvloedt. Dit onderwerp werd gekozen omdat deze nieuwe technology een interessante invloed kan hebben op de order-to-cash proces. Dit is een manier om een proces robuuster te maken. In de meeste software wordt gebruik gemaakt van één grote databank of meerdere databanken die in staan zijn om meerde services te voorzien van data. Bij microservice integration patterns wordt voor elke service een aparte databank opgesteld. Dit is maar een klein deeltje van een microservice. De microservices moeten voldoen aan business requirements. SAP zelf heeft ook al veel ondernomen omtrend microservices. Eén van hun oplossingen is Kyma. Maar de belangrijkste vraag is namelijk: Hoe microservice integration patterns een order-to-cash in SAP beïnvloedt. Deze bachelorproef zal voor het grootste deel een theoretische vergelijking zijn. Omdat deze studie als bachelorproef dient, is er maar beperkte tijd en resources om een onderzoek te doen.

A.2 Literatuurstudie

Over het onderwerp "Microservice Integration Patterns on Order-to-Cash proces in SAP", zijn er nauwelijks thesissen te vinden. De meeste informatie komt uit artikels die meer uitleg geven over microservices en artikels met een uitgebreide beschrijving over wat

order-to-cash inhoudt. Voor wie denkt dat microservices iets nieuws is, zit er een beetje naast. Grote bedrijven zoals Netflix, Twitter, Amazon en facebook maken al gebruik van deze technologie. Ciber, 2018

A.2.1 Wat zijn microservices?

Het bouwen van aparte functies/modules met hun eigen interface en methoden. Deze manier van werken is in het voordeel van Agile. Bij Agile wordt er gewerkt met deeltjes software opleveren en opgeleverde software, daar wordt er zo goed als niks meer aan veranderd. Microservices worden onderverdeeld aan de hand van business requirements. Deels zorgen microservices ervoor dat er beter moet worden samengewerkt met de business.

A.2.2 Waarom microservices gebruiken

In het artikel van Gunaratne, 2018 werd besproken hoe je een microservice werkt. En waarom deze gebruikt worden. Volgens dit artikel zijn microservices een goede, nieuwe techniek die op lange termijn huidige SOA's kan vervangen.

Atrash, 2018 beschrijft waarom je deze techniek kunt gebruiken, in de plaats van de kleinere SOA-services. Ook hier wordt verwezen naar de belangrijkheid van de requirements van de business. Door de grootte van deze services, is er de mogelijkheid om te caching.

Devoteam, 2018 legt uit waarom microservices een kleine-SOA is. Een microservice omvat bepaalde, aanvullende, concepten omliggend deze 'kleinere services' en dit is waar ze beginnen met aantonen van de verschillen.

Is de integratie van microservices wel mogelijk? Deze vraag beantwoordt door Van Bart, 2018. Software wordt meestal nog geimplementeerd in de 3-tiers manier. Ook wel monolithic genoemd. De applicatie is uit een alleenstaande unit gemaakt. Eén verandering heeft een impact op de volledige applicatie. Is dit dan een geldige reden om voor microservices te kiezen? Dat hangt af van wat je applicatie percies nodig heeft. Niet alle applicaties worden er beter van om een microservice te implementeren. Een microservice bestaat er uit om op zichzelf te werken. Dit wordt uitgelegd in het volgende deel.

A.2.3 Principes voor Microservices Integration

De principes van microservices integration werden uitgelegd in volgend artikel Aradheye, 2018 In het artikel wordt op een duidelijke manier uitgelegd hoe microservices worden gebruikt. Microservices worden het best opgesteld aan de hand van business units. Een microservice wordt benaderd vanuit de business requirements. Deze hebben, volgens dit artikel, een betere performantie dan de huidig gebruikte techniek. Microservices worden opgedeeld in verschillende klasses. Bijvoorbeeld:

- één voor klantendata

- één voor bestellingen
- één voor "wil-ik"lijsten

De belangrijkste eigenschappen van microservices zijn:

- Microservice bestaat uit meerdere componenten
- Gemaakt voor de business
- Microservice maakt gebruik van simpele routing
- Een microservice is gedecentraliseerd
- Een microservice werkt zelfstandig

A.2.4 Order-to-cash in SAP

Order-to-cash is een van de vele processen in SAP die vast gelegd zijn. Dit proces legt uit hoe men van een bestelling naar de inning van het geld gaat. Er zijn verschillende versies van hoe het proces gaat. Volgens Akthar, 2018 verloopt het proces als volgt: er wordt een order geplaatst dan wordt die bestelling geleverd, daarna wordt er een factuur opgesteld en als laatste wordt het geld geind. Het proces op zich is niet ingewikkeld. OpenSAP, 2018 geeft veel meer uitleg over wat er achter de schermen gebeurd. Bij dit proces wordt de financiële kant van SAP aangesproken, ook de verkoop en distributie alsook de stock worden aangesproken. Een gedetailleerder proces houdt in dat er een sales order gemaakt wordt. Dan wordt de stock bekeken. Afhankelijk van de beschikbaarheid van de goederen kan er een levering gepland worden. Zijn de goederen beschikbaar, dan kan er na de planning van de levering, effectief geleverd worden. Als volgt wordt er een factuur opgemaakt. Als laatste komt dan het betalingsproces.

A.2.5 Kyma

Kyma is een open-source project van SAP. Het is vooral gebaseerd op Kubernetes. Op deze manier kan je oplossingen in de Cloud maken. Kyma is special omdat zij zo goed als alle oplossingen op één plek hebben. Zij hebben een application connector. Kyma is serverless. Ze maken service management eenvoudiger. Kyma, 2019 Volgens het artikel van Semerdzhiev, 2018 is er meer nood aan openheid en een modernere architectuur. Dat is ook de reden waarom Kyma een open source project is. Kyma ondersteund container-based werken (zoals docker) alsook cloud-native apps.

A.3 Methodologie

In dit werk gaan we onderzoeken op welke manier microservices een invloed zou kunnen hebben op een order-to-cash proces in SAP. De services die ze nu gebruiken vergelijken met microservices. Kunnen microservices de werking van SAP versnellen en performanter maken bij fouten? Welke messaging manier zou het beste kunnen zijn. Eerst willen we het volledige order-to-cash proces verstaan. Dan gaan we gaan onderzoeken welke

verschillende mogelijkheden er zijn in verband met microservices. Kyma, PaaS.io of zijn er nog andere die een grote rol kunnen spelen. Ook moet er gekeken worden welke manier van communiceren tussen de microservices het beste is. Voor dit onderzoek zal er veel literatuur studie gedaan worden.

A.4 Verwachte resultaten

Naar de gelezen literatuur kijkende, zou Kyma eigenlijk de beste oplossing moeten zijn. Deze is namelijk zelf afkomstig van SAP. Dit zou een goede oplossing moeten zijn. Maar zijn microservices wel haalbaar in een order-to-cash proces in SAP. Eerst zal dit duidelijk moeten worden vooraleer we gaan kijken naar welke microservice de best mogelijk noden opvult.

A.5 Verwachte conclusies

De conclusie die we uit dit onderzoek kunnen trekken: microservices integration patterns zijn voordeliger en gebruiksvriendelijker dan het huidige systeem is voor de mensen die de software gaan gebruiken. Bij fouten aan een service, zal het platform nog beschikbaar zijn. Het risico is wel dat door de relatief nieuwe techniek, er enkele dingen niet zullen lopen zoals we zouden willen. Het is mogelijk dat we maar tot een gedeeltelijke conclusie komen.

Bibliografie

Akthar, J. (2018). What order-to-cash cycle controls in SAP ensure compliance? *SAP*.

Ananthasubramanian, H. (2018). Challenges in implemeting microservices. *https://dzone.com/articles/challen in-implementing-microservices*.

Aradheye, Y. (2018). Principles for Microservices Integration. *DZone*.

Atrash, M. A. (2018). Why microservice architecture? *Develoteam*.

Ayoub, M. (2018, april 24). Microservices authentication and authorization solutions. *https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a*.

Benetis, R. (2016a, november 11). A 6-point plan for implementing a scalable microservices architecture. *https://www.devbridge.com/articles/a-6-point-plan-for-implementing-a-scalable-microservices-architecture/*.

Benetis, R. (2016b, januari 1). Path to microservices: Moving away from monolithic architecture in financial services. *https://www.devbridge.com/articles/path-to-microservices-in-financial-services/*.

Biedron, R. (2018). A walk through the order to cash cycle. *https://www.purchasecontrol.com/uk/blog/order-to-cash-process/*.

Cavalcanti, M. (2018). Stateless authentication for microservices. *https://medium.com/@marcus.cavalcanti/st authentication-for-microservices-9914c3529663*.

Ciber. (2018). Microservices: het einde van de SAP spaghetti? *https://www.ciber.nl/blog/deel-2-microservices-het-einde-van-de-sap-spaghetti*.

concept hub, S. (2019). Understanding order to cash cycle in sap. *http://sapconcepthub.com/order-to-cash-cycle/*.

da Silva, R. C. (2017). Best practices to protect your microservices architecture. *https://medium.com/@rcandi practices-to-protect-your-microservices-architecture-541e7cf7637f*.

Devoteam. (2018). Microservices: just another word fo 'tiny-SOA'? *Devoteam*.

Everard, D. (2017). API Gateway definition. *https://www.quora.com/What-is-an-API-gateway*.

Eyee, U. (2018). Monitoring containerized microservices with a centrlaized logging architecture. *https://hackernoon.com/monitoring-containerized-microservices-with-a-centralized-logging-architecture-ba6771c1971a*.

Gunaratne, I. (2018). Wiring Microservices, Integration Microservices & APIs. *ContainerMind*.

Hofmann, R. (2018, augustus 10). Kyma and Knative Integration - Progress Update. *https://kyma-project.io/blog/2018/8/10/kyma-knative-progress-report*.

Kannappan, G. (2018, oktober 18). Accelerate innovation with Kyma. *https://kyma-project.io/blog/2018/10/18/accelerate-innovation-with-kyma*.

Koukia, A. (2018). A microservices implementation journey - Part 1. *https://koukia.ca/a-microservices-implementation-journey-part-1-9f6471fe917*.

Kumaran, S. (2015). What is Order to Cash Cycle. *https://www.invensis.net/blog/finance-and-accounting/what-is-order-to-cash-cycle/*.

Kyma. (2019). What is Kyma? *https://kyma-project.io/*.

Matteson, S. (2017). 10 tips for securing microservice architecture. *https://www.techrepublic.com/article/10-tips-for-securing-microservice-architecture/*.

Mauersberger, L. (2017, juli 18). Why Netflix, Amazon, and Apple Care about microservices. *https://blog.leanix.net/en/why-netflix-amazon-and-apple-care-about-microservices*.

Melendez, C. (2018). Microservices Logging best practices. *https://dzone.com/articles/microservices-logging-best-practices*.

Morgan, L. (2019). DevOps and Microservices. *https://www.datamation.com/applications/devops-and-microservices.html*.

Mulesoft. (2019). Microservices and DevOps: better together. *https://www.mulesoft.com/resources/api/micros devops-better-together*.

OpenSAP. (2018). Order-to-cash overview. *Order-to-cash overview*.

PEARSON, S. (2017). The Order to Cash (O2C) Process: Definition and Best-Practices. *https://tallyfy.com/order-to-cash/*.

RDX. (2016, december 13). REST in Peace: Microservices vs monoliths in real-life examples. *https://medium.freecodecamp.org/rest-in-peace-to-microservices-or-not-6d097b6c8279*.

Saldanha, L. (2016). Tips on logging microservices. *https://logz.io/blog/logging-microservices/*.

Sandoval, K. (2018). 3 common methods of API Authentication explained. *https://nordicapis.com/3-common-methods-api-authentication-explained/*.

Semerdzhiev, K. (2018, juni 24). Introducing project Kyma. *https://kyma-project.io/blog/2018/7/24/introducti project-kyma/*.

Siraj, M. (2017). Securing microservices: The API gateway, authentication and authorization. *https://sdtimes.com/apis/securing-microservices-the-api-gateway-authentication-and-authorization/*.

Stecky-Efantis, M. (2016). 5 Esay steps to understanding JSON Web Tokens. *https://medium.com/vandium-software/5-easy-steps-to-understanding-json-web-tokens-jwt-1164c0adfcec*.

Swersky, D. (2019). Best practices for tracing and debugging microservices. *https://raygun.com/blog/best-practices-microservices/*.

Timms, S. (2018). Microservice Logging: challenges, advantages, and handling failures. *https://stackify.com/microservice-logging/*.

Troisi, M. (2019). 8 best practices for microservices app sec. *https://techbeacon.com/app-dev-testing/8-best-practices-microservices-app-sec*.

Van Bart, A. (2018). Microservices and integration: Doing IT rigth? *devoteam*.

Wasson, M. (2018). Designing microservices: logging and monitoring. *https://docs.microsoft.com/en-us/azure/architecture/microservices/logging-monitoring*.

Watts, S. (2018). Microservices: An Introduction to Monolithic vs Microservices Architecture (MSA). *https://www.bmc.com/blogs/microservices-architecture/*.

Wigmore, I. (2016). Monolithic architecture. *https://whatis.techtarget.com/definition/monolithic-architecture*.

Wong, D. (2018). Step by Step: What you Should Know About the Order-to-Cash Process. *https://www.salesforce.com/products/cpq/resources/what-to-know-about-order-to-cash-process/*.