HO
GENT

Faculteit Bedrijf en Organisatie

Microservice integration patterns op SAP order-to-cash proces.

Lyva Van Damme

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

Promotor:
Karine Samyn
Co-promotor:
Nicolas Pauwelyn

Instelling: HoGent

Academiejaar: 2018-2019

Derde examenperiode

Faculteit Bedrijf en Organisatie

Microservice integration patterns op SAP order-to-cash proces.

Lyva Van Damme

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

Promotor:
Karine Samyn
Co-promotor:
Nicolas Pauwelyn

Instelling: HoGent

Academiejaar: 2018-2019

Derde 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 zou daarvoor graag mijn stagebedrijf, delaware, willen bedanken voor het voorstellen van een onderwerp. Dankzij hun kon ik mijn bachelorproef doen over een onderwerp dat mij echt interesseerde. Het was een aangename ervaring om drie jaar van de opleiding te kunnen omzetten in dit eindwerk. De ervaring van andere mensen heeft mij geholpen om door deze periode te geraken. Als eerste wil ik mijn co-promotor, Nicolas Pauwelyn, willen bedanken voor zijn hulp. Ik zou 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. Als laatste zou ik graag mijn mede-studenten bedanken voor alle hulp die ze mij gaven.

Samenvatting

Dit onderwerp werd voorgesteld door delaware. Om te onderzoeken hoe microservice integration patterns een invloed kunnen hebben op het order-to-cash proces binnen SAP.

Dit onderzoek kent een verdiepende structuur. Waar in de start uitleg wordt gegeven omtrent microservices, kent het vervolg een vergelijking tussen verschillende onderdelen. Enkele voorbeelden hiervan zijn authenticatie en authorisatie. Deze scriptie sluit af met opkomende ideologieën en de bescherming van microservices.

In deze thesis is als eerste een uitgebreide literatuurstudie terug te vinden. In deze literatuurstudie wordt er dieper ingegaan op de technologie van microservice en de verschillende integration patterns omtrent de microservices. Daarnaast zal er meer uitleg gegeven worden over het order-to-cash proces. Na de literatuurstudie wordt het microservice integration pattern toegepast op het order-to-cash proces. Als afsluiter komt de conclusie.

Microservices zorgen voor een geautomatiseerd proces met weinig interactie met een werknemer. Het is een complexe architectuur. De overschakeling bevat veel onderdelen waar er in eerste instantie niet aan gedacht wordt. Het werd ook duidelijk dat er goed moet worden nagedacht over alle kleine dingen, zoals authenticatie, de manier van bescherming, hoe logging wordt opgevangen.

De overschakeling kan gemakkelijk onderschat worden. Een belangrijke vraag die men kan hebben is: hebben microservices wel een meerwaarde in dit proces? Of de overschakeling moet gemaakt worden, is afhankelijk van wat de noden de applicatie precies zijn.

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

De promotor en doelgroep van dit onderzoek is delaware.

1.2 Onderzoeksvraag

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

- 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?
- Welke integration patterns zijn er?
- Hoe zit een order-to-cash (OTC) proces er uit?
- Welke business requirements heeft een order-to-cash proces?
- Welke invloed heeft een microservice integration pattern op het 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 integration patterns op de architectuur van een OTC-proces in SAP. De ambitie houdt in dat er verschillende microservice integration patterns worden onderzocht. Bij de methodologie wordt er één integration pattern theoretisch benadert op het order-to-cash proces. Het integration pattern zal gekozen worden aan de hand van een vergelijking met verschillende patterns.

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, verschillende integration patterns 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 een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Stand van zaken

Dit hoofdstuk bevat de literatuurstudie omtrent het onderwerp. Hier wordt de architectuur en de termen uitgelegd. Daarnaast zal hier een vergelijking gemaakt worden tussen onderdelen van microservices. Tenslotte zal hier een beeld van een order-to-cash proces gemaakt worden.

2.1 Microservices

2.1.1 Definitie

Aangezien de term monolithic veelvuldig zal gebruikt worden, zal er bij aanvang reeds een beeld scheppen omtrent de term: '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 ' software programs.', Wigmore (2016). Om te begrijpen waarom een overschakeling naar microservices een goed idee is, worden de de moeilijkheden bij monolithic architectuur aangehaald. Bij een verandering binnen een monolithic architectuur, wordt een geheel nieuwe versie hiervan uitgebracht. Dit creëert een aanzienlijke overhead. Met architectuur wordt de algemene manier van implementatie bedoeld. Bij het gebruik van het woord architectuur, wordt er niet verwezen naar de applicatie maar de achterliggende logica ervan. De overschakeling naar microservice omvat onder andere

- 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.

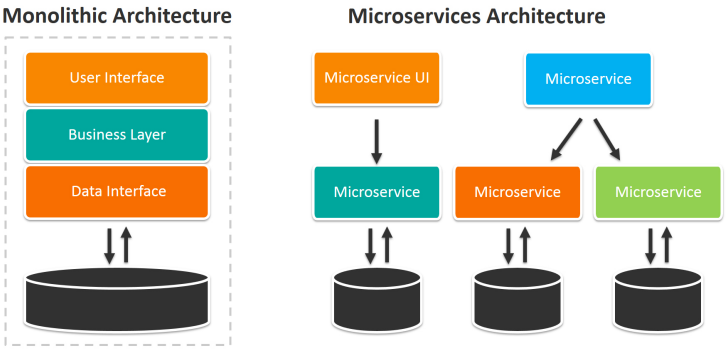
Er zijn meerdere definities terug te vinden over microservices. In volgende opsomming kunnen enkele definities gevonden worden:

- 'A method of developing software applications as a suite of independently deployable, small, services in which each service runs a unique process and communicates through a well-defined, light weight mechanism to serve a business goals.', Mauersberger (2017)
- '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.', Watts (2018)
- 'microservices stems from the same ideology as Agile and DevOps by seeking to break down slow moving, monolithic systems into multiple small, independent services that are highly decoupled and self-contained to focus on a specific function or capability.', Benetis (2016b)

Wanneer men de definities herleest, zijn er enkele terugkerende onderdelen aanwezig. Een eerste onderdeel spitst zich toe op de werking van een microservice. Het is een onafhankelijke, kleine, modulaire service. Modulaire services zijn services waarbij veel delen uitwisselbaar zijn met diverse services. Wordt er info gestuurd of gevraagd van services A dan zal dit geen invloed hebben op de andere services. Een tweede eigenschap is de eenvoudig communicatie. Er is nood aan omdat sommige services data moeten uitwisselen om hun 'job' te kunnen doen. De communicatie kan gebeuren op verschillende manieren. De manier die gekender is, is 'Messaging via a Message Broker'. Dit wil zeggen dat microservice A een bericht plaatst 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 men ze gaat bekijken vanuit de eisen binnen de business. Het doel van microservices is, de problemen die te vinden zijn bij een monolithic, verhelpen.

De verschillen tussen een monolithic en microservices kunnen het best afgebeeld worden in een afbeelding 2.1.

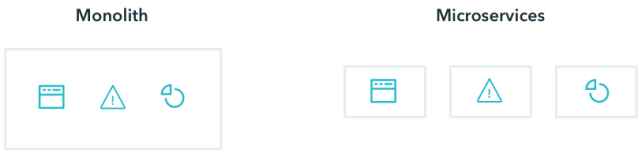
De monolithic wordt weergegeven in de linkerkant van de foto. Aan de rechterkant van de foto is een voorbeeld te zien van een microservice architectuur. Daar is duidelijk te zien dat elke microservice een eigen databank/datastore heeft. Als bij microservice A een probleem is dan heeft dit niet meteen impact op de andere services. De communicatie tussen microservice A en de anderen zullen hier echter wel hinder ondervinden. Maar de andere microservices kunnen wel nog steeds onafhankelijk verder. Hier ziet men het punt passeren dat een microservice een klein component is van een groter geheel. Er is een mogelijkheid van de software om mee te groeien als het aantal gebruikers vermeerderd. De software moet nog even goed presteren bij 10 gebruikers als bij 2 000 gebruikers. Er



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

wordt toegelicht 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.

Op figuur 2.2 ziet men 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) .

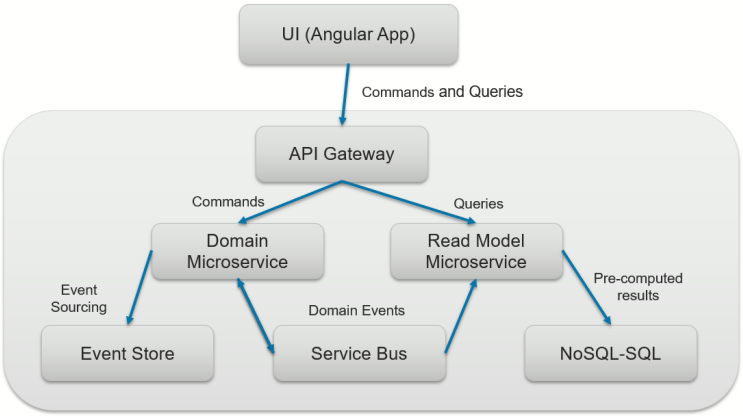


Figuur 2.2: Een monolithic vergeleken met een microservice. Benetis (2016b)

Een voorbeeld van een algemene architectuur is te zien op figuur 2.3. Deze afbeelding zal naar het einde van de stand van zaken duidelijk zijn.

2.1.2 Het belang van microservices

Microservices zijn van belang wanneer de monolithic architectuur niet meer optimaal werkt. Microservices hebben nauwelijks invloed op het framework als er deeltjes bij gecodeerd worden. Ze kunnen sneller inspelen op de Agile analyse/ontwikkelsmethode. 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.



Figuur 2.3: Een algemene architectuur voor microservices. Koukia (2018)

Enkele andere redenen om microservices te gebruiken volgens 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 het geval bij microservices. Want elke service is afgebakend met een functionaliteit. Bij het uitleggen van een service 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 niets 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 herschreven worden. Dit is anders bij 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 schalen. Schalen gebeurt door een microservice te dupliceren. Niet het gehele systeem moet geschaald worden, enkel de nodige microservices moeten gedupliceerd worden. Het dupliceren van een microservice wil zeggen, de microservice kopiëren en in een ander onderdeel gebruiken. Een microservice dat gedupliceerd kan worden, is dat van logging.
- 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 in 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 voor dat de bescherming eenvoudiger wordt.', RDX (2016).

De term bescherming omvat volgende: Er voor zorgen dat hackers de applicatie niet kapot maken. Hackers zijn mensen die inbreken op een applicatie.

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

- Zorg bij het ontwikkelen van microservices voor coderingsstandaarden die herbruikt kunnen 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 geüpload.
- 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 deze 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 gebeuren 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 van een beveiligd protocol. 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 techonologieën .
- 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 Cross-site request forgery (CSRF) en Cross-origin resource sharing (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 kan goed beschermd zijn, zorg eveneens voor goede bescherming aan de code. Zorg ervoor dat er niks moet worden gerund als administrator, geef duidelijke namen en maak duidelijke afspraken. Enkel de nodige personen mogen de juiste permissies krijgen.

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

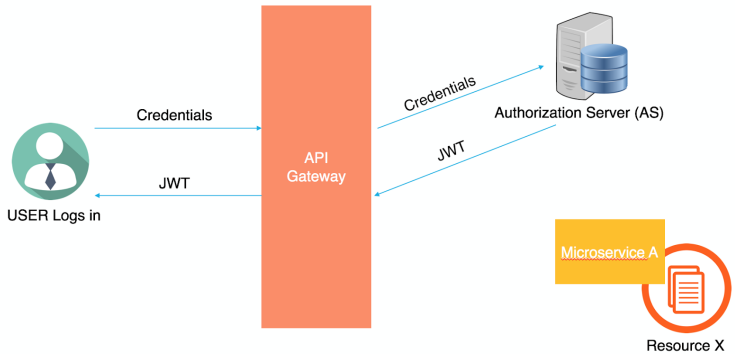
- Het gebruik van OAuth voor gebruikers identificatie en wat de gebruiker kan. OAuth/OAuth2 is een protocol voor authorisatie. Het is een gemak om gebruik te maken van een protocol. Een protocol zijn een aantal regels om te communiceren tussen computers.
- Gebruik bescherming in de diepte om een prioriteit 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 krypto code geschreven.
- Update je bescherming tijdig. Als er updates komen in de software van beveiliging, moeten die ' uitgevoerd worden. Het automatiseren van die updates, kan veel werk besparen achteraf. Dit wordt dan ' best gedaan bij het begin van microservices. Bescherming binnen software is niet langer meer een nice to have maar een must have.
- Maak gebruik van een firewall met gecentraliseerde controle. Het biedt onder andere meer controle aan de gebruiker.
- Zorg dat je 'containers' niet in een publiek net werk te vinden zijn. Dit is eigenlijk zorgen dat gebruikers je achterliggende architectuuur niet kunnen zien. Hier kan een vorige manier ' bijgestoken worden. Microservices kunnen ondergebracht worden achter een firewall als vorm van berscherming. Als er met containers wordt gewerkt, moet er ' 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.

Throtteling, Cavalcanti (2018), is een manier van bescherming die het volgende beschrijft: '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. Dit kan zorgen voor een bug door een oneidig aantal requests. Om dit te voorkomen kan er een limiet binnen een bepaalde periode opgelegd worden. Bijvoorbeeld als je je toegangscode drie maal fout hebt op je gsm, dan blokkeert 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 2.4, is een API gateway het toegangspunt. Om via de API gateway requests te kunnen maken, moet er eerst authenticatie en authorisatie toegepast worden. Meer uitleg hierover in de sectie over authenticatie en authorisatie.

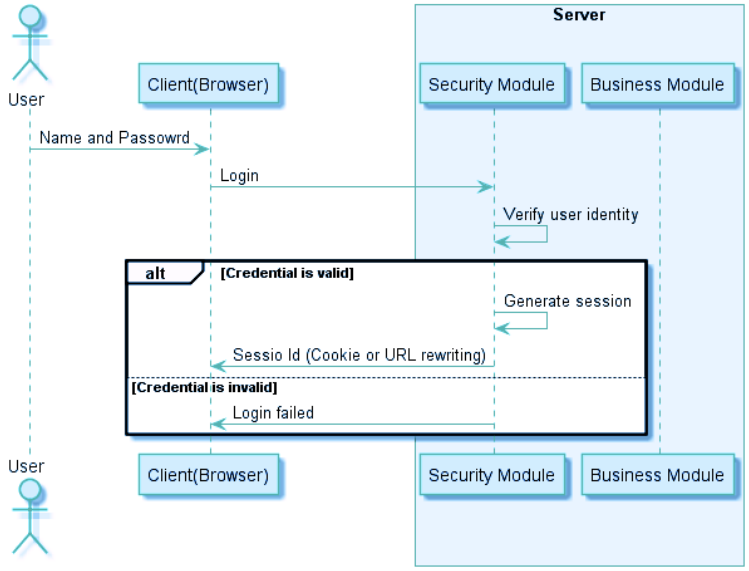


Figuur 2.4: Een voorstelling van API gateway. Siraj (2017)

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 het bevestigen van de identiteit van de gebruiker. Dit wordt doorgaans gedaan door middel van 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 2.4, wordt 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 tokens. Tokens 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.



Figuur 2.5: Een diagram van authenticatie bij een monolithic. Ayoub (2018)

Het beste wordt dit toegepast op een uniforme manier. Dan gaat men er van uit 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 vier oplossingen volgens Ayoub (2018)

- Distributed session manangement,
- 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 aanwezig zal zijn op het netwerk. Een andere methode is centralized session storage. Deze omvat dat bij het aanspreken van een microservice, deze de gebruikersdata 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 echt de gebruiker bent. Een token wordt bijna altijd onleesbaar gemaakt. Het klinkt bijna

	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 theyre 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 tokens, session ID's, login pagina's en 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. 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 2.1: De verschillende methoden in API authenticatie, Sandoval (2018)

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 nog single sign-on. Na een enkele aanmelding, kan de gebruiker alle microservices gebruiken binnen de applicatie. Een andere manier is een client token with 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 buiten wereld.

De meest voorkomende methodes in API authenticatie terug te vinden in tabel 2.2, Sandoval (2018).

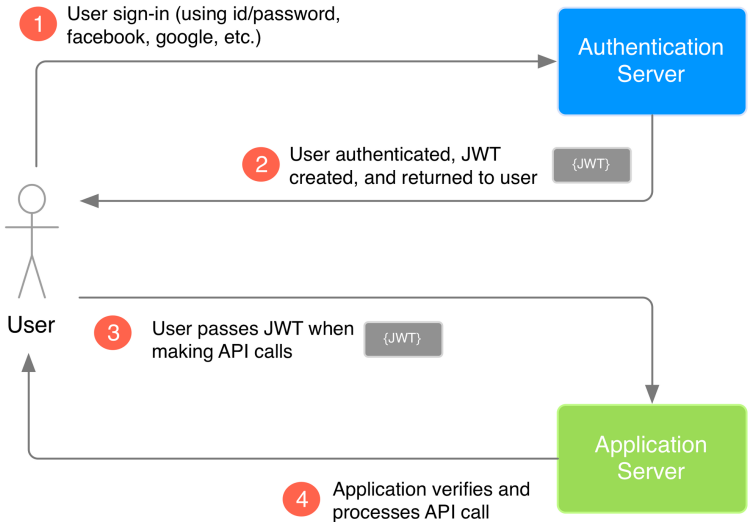
Het proces van een API gateway dat authenticatie en authorisatie op zich neemt, Siraj (2017). Door de API Gateway to combineren met JSON web Tokens is er meer mogelijkheid om te schalen.

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 2.6, is een schematische voorstelling van hoe een JWT token wordt gegeven aan een gebruiker. Het proces gaat als volgt:



Figuur 2.6: 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 contaminatie 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 men de productie zo goed mogelijk na te bootsen. Net zoals bij Agile, zorgt DevOps ervoor dat de software in kleinere delen wordt opgesplitst om weer op kortere periodes kleinere deeltjes software op te leveren. Dit is iets waar ' microservices in terug te vinden is. Verschillende DevOp teams kunnen dus tegelijkertijd 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 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 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. Bij het vinden van problemen is monitoren een belangrijk onderdeel. Dankzij monitoring kan een bug 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 meerdere 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 capturen van events. Logs worden nooit aangepast. Logs worden ' 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 ' 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. Binnen die enkele database met alle logs kan er gezocht worden op fout, microservice, tijdstip, ... ' 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 creëeren in de logs.
- Geef iedere request een context. Weten wat de oorzaak is van de fout, is belangrijk om ervoor te zorgen dat de fout zich niet zal herhalen. 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. De bandbreedte kan bij belangrijke microservices vermindert worden.
- 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. Het opstellen van dit plan gebeurt doorgaans in samenwerking met meerdere personen. Op die manier is er een bredere waaier aan opinies en om ervoor te zorgen dat iedereen op dezelfde lijn zit.

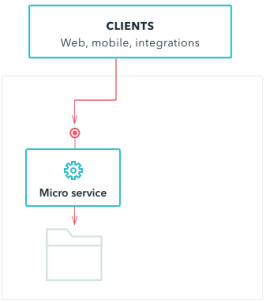
Benetis (2016a) schreef een zesstappen plan om microservices te implementeren. Over de grote lijn wordt hierop gebaseerd. Stap per stap gaat er dieper op de overschakeling worden ingegaan. Een paar woorden die meer verklaring nodig hebben voordat we verder gaan. Een gateway is een netwerkpunt dat dient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

Kort uitgelegd, is dit het zesstappen 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 stuff' aan bod. Hierna wordt de volgende stap 'create a gateway' aangehaald. Als laatste komt 'construct events' aan 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 ' gekeken worden naar wat de microservices moet kunnen. Komen enkel microservices aanbod dan is automated deployement niet van groot belang. Maar eens men gaat schalen en meerdere microservices in één systeem onder gebracht. Er wrodt verwacht dat volgende puntjes toch self-sufficient zijn:

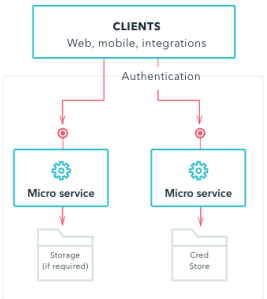
- Geautomatiseerde implementatie
- Blootstelling aan andere systemen, toegankelijk eindpunt
- Opslag van data
- Schaalbaarheid en belasting

Zoals te zien is op figuur 2.7, is een microservices één klein deeltje in een groot geheel. Later in dit stappenplan zal de figuur ' uitgebreidt worden en zal het geheel duidelijk worden.



Figuur 2.7: Een microservice dat voldoet aan een business requirement. Benetis (2016a)

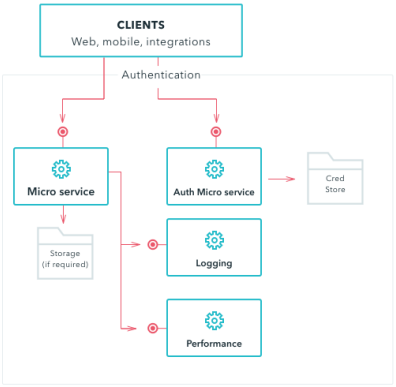
Na 'Serve a business purpose' komt 'Protect your stuff'. Dit gaat over de bescherming van een microservice. Bescherming moet op elk moment toegepast worden. Eveneens er sprake is van één à t wee microservices, of honderden, bescherming is belangrijk. Het is belangrijk om over al de microservices een uniforme manier te vinden om ze te beschermen. De bescherming kan een requirement op zich zijn, 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. De controle of jij het wel echt bent. Een manier op de microservices te beschermen is gecentraliseerde session opslag. 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 2.8 is de authenticatie in een



Figuur 2.8: Een microservice waar bescherming aan is toegevoegd. Benetis (2016a)

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 zich voordeed. Hier is het aangeraden om doorheen het hele systeem een uniforme manier van loggen aan te houden. Men kan hiervan een microservice maken. Dit wordt ' afgebeeld in figuur 2.10.

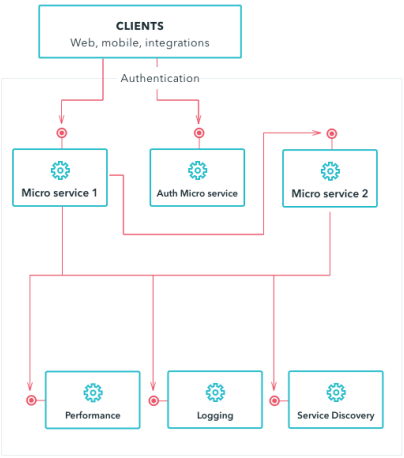


Figuur 2.9: Een microservice waar er monitoring is aan toegevoegd om chaos te voorkomen. Benetis (2016a)

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 vragen aan microservices B. Die dit op zijn beurt vraagt aan 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 ' connecties in opgeslaan. Er wordt aangeraden om dat in een microservices te gieten. Zodat hiervan het gedrag kan gemonitord worden. In figuur 2.10 zie je hoe de service registry kan toegevoegd worden. In de figuur is die terug te vinden onder de naam 'service discorvery'.

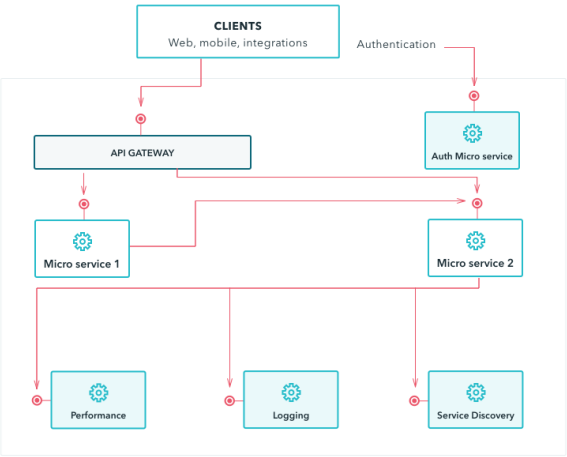
Nu is er een service, bestaande uit microservices, dat beschermd is en kan doen wat het zou moeten doen. Maar niet de volledige service moet open en bloot gelegd worden. En daar zorgt stap 5, 'Create a gateway', voor. Een API gateway kan een scherm zijn waar je gegevens op invult en mogelijke acties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



Figuur 2.10: Een microservice dat authenticatie als bescherming toepast. Benetis (2016a)

In figuur 2.11 is te zien hoe zo een API gateway kan worden toegepast. Zo is 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 een request sturen naar de API gateway.



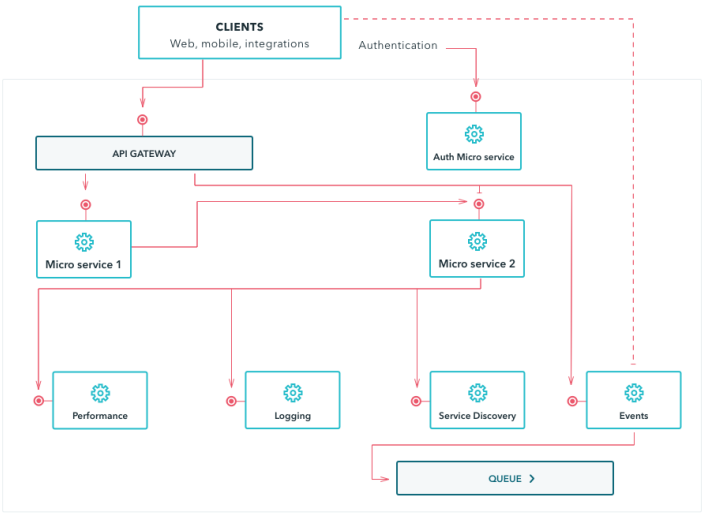
Figuur 2.11: Een microservice dat een gateway gebruikt. Benetis (2016a)

Nu is er al een deftige architectuur aanwezig. Stap 6, 'construct events', zal het plaatje dan ' 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 ophalen. Enkele voordelen van dit als asynchrone oplossing te

gebruiken:

- Taken kunnen ingepland worden. 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 2.12 wordt de volledige architectuur weergegeven. Daar wordt er ' 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 2.12: 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?

Bij de beslissing om over te schakelen, moet er beslist worden hoeveel de infrastructurele veranderingen van de scope inpalmen. Het verloop gaat als volgt. 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 ' 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

Er wordt veel goed geschreven 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. Microservices kunnen meerdere keren terugkomen in een architectuur. Dit wordt duidelijker in de methodologie. Dankzij microservices is het hermodeleren, implementeren van nieuwe technologieën, ... eenvoudiger. 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. 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 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. Men kan 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 ' door de onafhankelijkheid van de services

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

- Een project zal ingedeeld moeten worden in kleine requirements. De scope zal

gedetailleerder moeten zijn.
- Een team zal kleiner zijn. 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.
- 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 zijn ingewikkeld. Microservices vragen meer werk. Dit komt omdat het een nieuwere technologie is en het concept is niet altijd meteen duidelijk.
- Complexiteit is overal. Een monolithic bevat 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. 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 Microservices integration patterns

Hierboven werd uitgelegd wat microservices zijn. Het zijn kleine, onafhankelijke services (synoniem vinden). Maar die moeten met elkaar kunnen communiceren, taken uitvoeren, data updaten en raadplegen. Er zijn manieren om microservices gestructureerd te integreren in de applicatie. Dit kan aan de hand van Die verschillende manieren zijn onderstaande patterns:

- Anti-patterns
- Communicatie tussen de microservices
- Data integration
- Extract, transform en load (ETL)
- REST

In het derde hoofdstuk zal één van de bovenstaande patterns theoretisch toegepast worden op het order-to-cash proces. Het order-to-cash proces zal verder uitgelegd worden in deel 2.3.

2.2.1 Anti-patterns

Anti-patterns ontstaan bij een ongeplande overschakeling naar microservices. Wanneer microservices gebruikt worden, zonder grondig onderzoek, kan dat tot chaos leiden. Anti-patterns komen vaker voor dan een gestructureerde microservice architectuur. Er moet grondig onderzoek gedaan worden naar de verschillende 'microservice integration patterns'. Op welke manier kan de applicatie, gestructureerd, naar een microservice architectuur gewerkt worden. De meeste mensen hebben niet door dat ze een anti-pattern aan het maken zijn. Door ongestructureerd de microservices te gebruiken, kan er, onbewust, een anti-pattern ontstaan.

Er zullen vier anti-patterns besproken worden. Dit kan helpen bij het achterhalen van een anti-pattern. Volgende anti-patterns zullen besproken worden, Monson (2019):

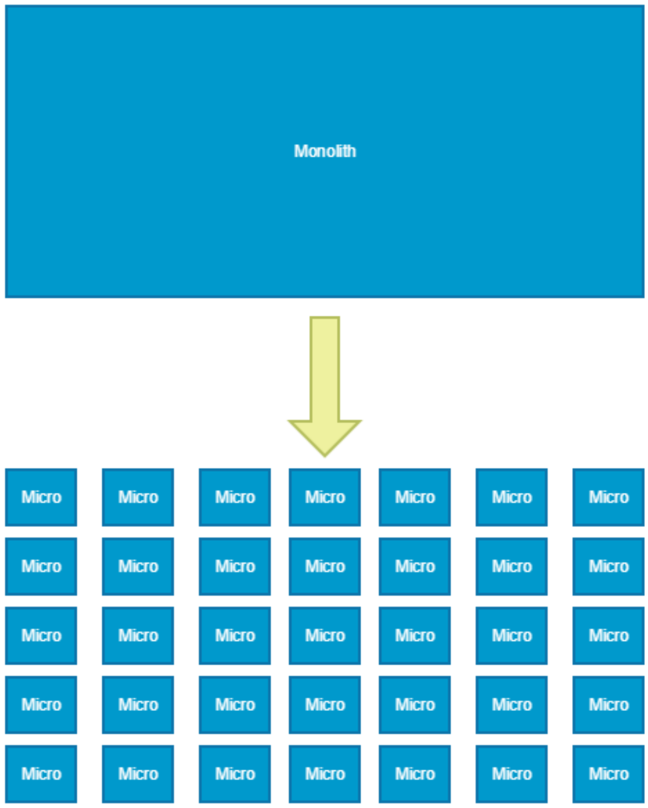
- Break the piggy bank
- Everything micro
- We are Agile: The Frankenstein

Break the piggy bank

Bij deze manier van toepassing van microservices kan de applicatie voorgesteld worden als een spaarvarken. Om van een monlithic applicatie naar een microservice applicatie te gaan, moet het spaarvarken (de monolithic applicatie) aan stukken geslaan worden. De applicaties architectuur wordt volledig uit elkaar gehaald en in kleine delen opgedeeld. De kleine delen worden omgezet naar microservices. In het begin van de verandering naar microservices, lijkt het dat de complexiteit verminderd. Maar eens verder in het proces zal het duidelijk worden dat het enkel schijn was, de complexiteit wordt enkel verhoogd. Dit komt door de ongestructureerde manier waarop men te werk is gegaan. De applicatie is opgedeeld in onderdelen en die onderdelen zijn omgezet in microservices. Bij het samenvoegen van de microservices tot een architectuur, is het mogelijk dat de applicatie een mini-monolithic wordt. Dit komt voor wanneer de services niet opgedeeld worden en willekeurig worden samengevoegd. Dit anti-pattern is het populairst wanneer een monolithic applictie niet meer houdbaar is.

Everyting micro

Alles in de applicatie wordt omgezet naar microservices, behalve de databank. Bij deze anit-pattern blijft de databank dezelfde als bij de monolithic applicatie. De verschillende microservices moeten de data uit een gemeenschappelijke databank opvragen. De databank is de 'bottleneck' in de applicatie. De term 'bottleneck' wordt gebruikt wanneer een plaats is binnen de applicatie waar alle delen vertraagt worden. Een 'bottleneck' is een soort van trechter. Alle aanvragen komen binnen maar er kunnen maar enkele verwerkt worden. Daar wordt vertraging opgelopen. Om de data van de databank op te halen moet er aan 'access control' gedaan worden. Niet alle microservices kunnen tegelijkertijd hun data opvragen en ontvangen. De databank moet op een gecontroleerde manier gebruikt worden. Wie de databank wanneer mag 'gebruiken' moet goed uitgedacht zijn. Anders is

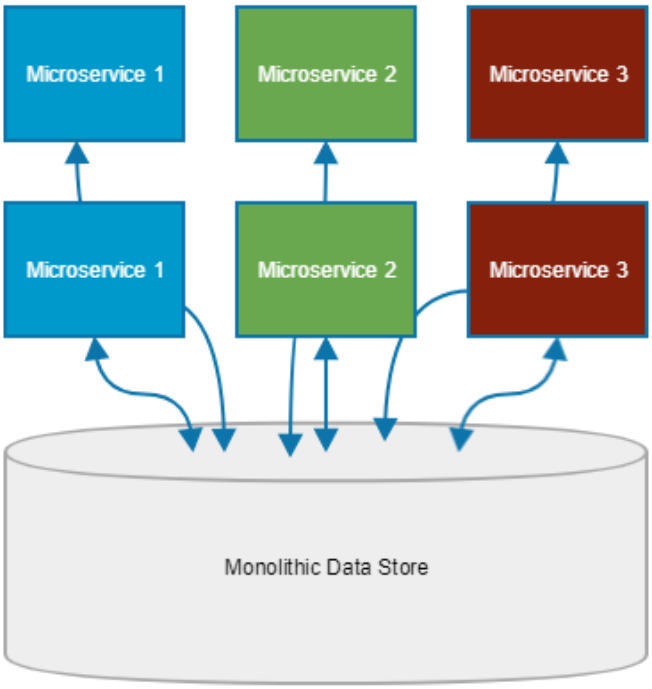


Figuur 2.13: Break the piggy bank, van monolithic naar microservices via anti-pattern Monson (2019)

er kans op ’deadlock’. Deadlock komt voor wanneer twee of meerdere services data van de databank willen halen. Wanneer er geen controle gebeurt over de toegang kan het zijn dat een services de databank voor zichzelf houdt. De onderdelen van de applicatie kunnen microservices zijn, maar als de databank nog steeds monolithic is, wordt de ’bottleneck’ verplaatst. De datastructuur blijft monolithic. In het begin lijkt dit niet zo een groot probleem. Maar de monolithic databank heeft een negatieve invloed op de applicatie. Naast de negatieve invloed is er een uitdaging: de verandering van de databank schema’s bijhouden. Wanneer er een aanpassing gemaakt wordt, moet dat bijgehouden worden. Omdat er maar één databank gebruikt wordt, moet elke verandering van elke microservices genoteerd worden. Naast die uitdaging, moet bij een verandering in de productie databank, de volledige applicatie opnieuw gedeployed worden. Een productie databank is de databank die live gebruikt wordt. Het is de databank waarmee je communiceert als je op Zalando iets besteld.

We are Agile: The Frankenstein

Dit anti-pattern ontstaat vaker door de overschakeling van waterval analyse naar de Agile methodologie. Door die overschakeling kan er gedacht worden dat er niks meer gedaan moet worden op vlak van voorbereiding. Dat alles duidelijk zal worden eens het probleem zich echt voordoet. De meeste teams gaan van de waterval methode naar een combinatie



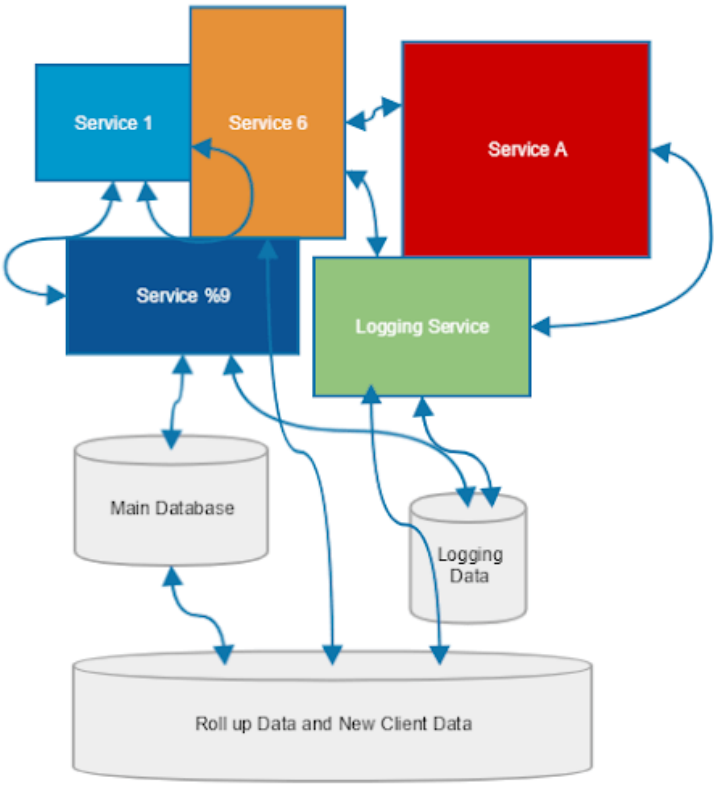
Figuur 2.14: Everyhting micro, anti-pattern met een monolithic databank. Monson (2019)

van Agile en waterval. Door een ongestructureerde planning, worden er losse delen geprogrammeerd. Die losse delen moeten samen werken. Ze worden aan elkaar 'genaaid' om samen te werken. Net zoals Frankenstein aan elkaar is genaaid met verschillende onderdelen. Al de samengevoegde onderdelen moeten de data delen. Moet er een nieuw deel functionaliteit toegevoegd worden, wordt de architectuur complexer en moeilijker om te deployen. Omdat de architectuur een samenhang is van onderdelen, kan de applicatie op lange termijn dingen doen dat niet zou mogen. Er zouden transacties kunnen gebeuren die niet mogen gebeuren binnen de applicatie. De mogelijkheid bestaat dat de applicatie gewoon uit elkaar valt door de slechte architectuur en de chaos die er is.

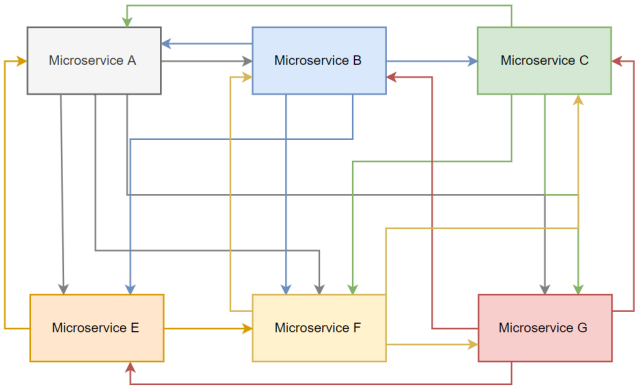
2.2.2 Communicatie tussen de microservices

Er zijn verschillende manieren om microservices te laten communiceren met elkaar. De verschillende microservices moeten met elkaar kunnen communiceren om data uit te wisselen. De microservices kunnen bijvoorbeeld elke microservice kennen die het nodig heeft. Maar dit maakt de gehele architectuur enkel maar complexer. Zoals in afbeelding X te zien is, wordt de architectuur chaotisch bij deze manier van communiceren.

Dit zorgt ervoor dat de microservices niet meer onafhankelijk zijn van elkaar. Ze worden weer afhankelijk van elkaar en vormen zo een mini-monolithic. Een soort anti-pattern zal ontstaan. Om ervoor te zorgen dat microservices onafhankelijk blijven, kan er 'messaging' toegepast worden, Solance (2018). Messaging is een manier van communiceren,

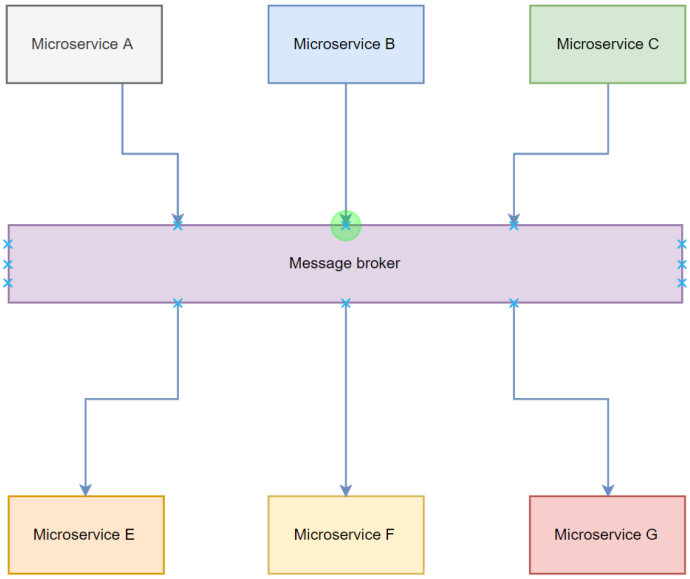


Figuur 2.15: The Frankenstein, anti-pattern waarbij de microservices bij elkaar zijn gevoegd zonder samenhang. Monson (2019)



Figuur 2.16: Een architectuur waarbij de microservices elkaar direct kennen om informatie met elkaar te delen.

om business entiteiten up te daten, dat gebeurt via een 'message broker'. Een 'message broker' is de logica die de berichten van een microservice naar een andere stuurt zonder dat ze elkaar moeten kennen. Het meest gekende patroon hiervoor is publish-subscribe pattern. Deze wordt weergegeven in afbeelding X. In de 'message broker' heeft elke microservices zijn queue. De queue is een wachtrij waar berichten worden opgeslaan. De berichten worden verwijderd van de queue eens ze gelezen en opgehaald worden. Het



Figuur 2.17: Een architectuur waarbij een message broker als communicatiemiddel wordt gebruikt.

bericht moet eerst gelezen worden voordat het verwijderd kan worden van de queue. Een microservice stuurt een bericht naar de 'message broker' en moet zich verder niks van het bericht aantrekken. De 'message broker' zorgt ervoor dat het bericht bij de juiste microservice(s) wordt afgeleverd. De uiteinden kennen elkaar niet. Ze kunnen volledig onafhankelijk van elkaar functioneren. Dit is een doel van microservices. En om dat doel te bereiken, moet de manier van communicatie goed gekozen worden.

2.2.3 Data integration

Een opsomming van enkele definites omtrend 'Data integration':

- 'Two or more services read and write data out of one central data store', Aradhye (2018).
- 'Microservices embrace independent, autonomous, specialized service components, each with the freedom to use its own data store', Kumar (2018)

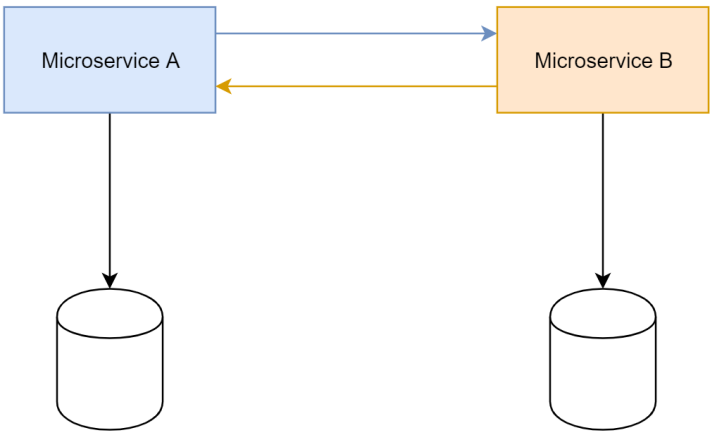
Een opvallende eigenschap aan een monolithic architectuur is de centrale databank. Bij de overgang wordt de databank regelmatig uit het beeld gelaten, omdat er gedacht wordt dat er niks aan de databank moet veranderen. De data blijft hetzelfde, dat is correct. Maar de structuur waarmee er met de data wordt gewerkt, verandert in zijn geheel. De oude databank, die bij de monolithic gebruikt werd, is niet de optimale oplossing. Als de monolithic databank nog gebruikt wordt bij een microservice architectuur, brengt dat veel problemen en obstakels met zich mee. Volgende zijn enkele voorbeelden:

- De microservices zijn afhankelijk van de databank en zijn daardoor afhankelijk van elkaar. Als ze meerdere microservices de databank nodig hebben, moeten ze wachten tot de ander gedaan heeft.
- Microservices onafhankelijk deployen, is onmogelijk. Een aanpassing heeft invloed op de databank en elke microservice kent de databank.
- Een microservice schalen wordt een moeilijke opgave omdat die vasthangt aan de centrale databank.
- Omdat alle data in één databank wordt bijgehouden, worden de tabellen gigantisch groot en op termijn onhoudbaar en slordig.

Elke microservices heeft elk zijn eigen databank met data betreffende hun functionaliteit. Hieronder kunnen enkele voorbeelden teruggevonden worden over de voordelen van een microservice met een eigen databank:

- De databank bevat enkel de relevante data.
- Elke microservices kan apart gedeployed worden. Niemand anders is verbonden met zijn databank.
- Een microservices kan niet rechtstreeks aan de databank van een andere microservices. Hoe een microservices onrechtstreeks de databank van een ander kan, zal later uitgelegd worden.

Sommige microservices hebben data van andere microservices nodig. Omdat hun databank de geweste gegevens niet kan leveren. De data wordt dan gevraagd aan de databank via een de microservice zelf.



Figuur 2.18: Hoe een microservice de databank van een andere microservice aanspreekt.

De manier waarop een microservice een andere microservice zijn databank aanspreekt, gebeurt zoals beschreven in het vorige deel over 'communicatie tussen de microservices'.

Eens de volledige communicatie is uitgedacht, moet er een databank gekozen worden, passend bij de micoservices. Het kan zijn dat voor de ene microservices een SQL-databank beter past en voor de ander een NOSQL-databank. SQL en NOSQL zijn soorten databan-

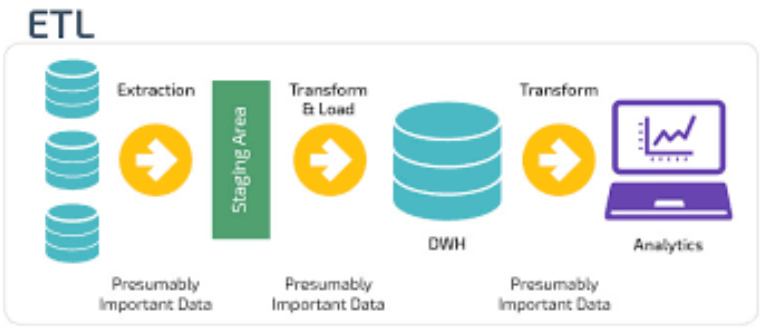
ken. De manier waarop de data wordt opgeslaan, opgevraagd en teruggestuurd is anders.

2.2.4 Extract, transform en load (ETL)

Volgende opsomming bevat verschillende definities omtrent ETL:

- 'The type of data integration that refers to the three steps used to blend data from multiple sources', Loshin (2019).
- 'Data from multiple systems is combined to a single database, data store, or warehouse for legacy storage or analytics', Alley (2018).
- 'The process by which dat ais extracted from data sources that are not optimized for analytics, moved to a central host and optimized for analytics', Stich (2019).
- "A process that extracts the data from different database source systems, then transforms data and finally loads data into datawarehouse', Guru (2019).
- 'The process of transferring data from soruce database to destination datawarehouse', Naveen (2016).

Uit bovenstaande definities kan er afgeleid worden dat ETL gebruikt wordt om data te verzamelen, te transformeren en dan in een andere databank te steken. De data wordt geanalyseerd om de business een beter beeld te geven van de real-time gebeurtenissen bij bijvoorbeeld marketing om te weten wat het meest verkocht wordt om 12uur 's middags. Het proces dat ervoor zorgt dat de verzamelde data analyseerbaar is, bevat drie stappen. De eerste stap is 'extraction', de data ophalen. De verzamelde data komt niet van één databank. Ze wordt opgehaald uit verschillende bronnen. Die bronnen zijn databanken die online informatie bevatten over het gedrag van die klant/gebruiker. Het verzamelen van die data mag de werking van de originele databank niet beïnvloeden. Eens alle data verzameld is, wordt die getransformeerd en gefilterd om enkel de nodige data te gebruiken. Er wordt meer data verzameld dan nodig. De data wordt gefilterd aan de hand van regels opgesteld door de business. Hierna blijft enkel de hoognodige data over. Die data wordt ingelanden in een datawarehouse. Een datawarehouse is een databank waar men data analyseerd en logica op los laat. De business legt op welke analyse er moet gebeuren. De uitkomst van deze analyse kan gebruikt worden binnen het bedrijf of kan gebruikt worden voor artificiële intelligentie.



Figuur 2.19: Hoe het ETL proces eruit ziet. Panolapy (2019)

2.2.5 REST

2.3 Order-to-cash proces in SAP

2.3.1 Definite

Wong (2018) legt uit wat een order-to-cash proces inhoudt. Dit proces heeft veel invloed op het succes van een bedrijf. Dit proces heeft veel invloed met de klant. Een voordeel met de huidige technologie is dat het mogelijk is om het proces volledig te automatiseren. Dit zorgt voor een minimaal aantal fouten en vertragingen. 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:

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

Zijn er problemen in één van die afdelingen, dan kan een vertraging in de andere afdelingen als gevolg hebben. Alsook een minder vlotte cashflow. Een goed OTC proces maakt indruk op de buiten wereld. Doordat het OTC goed gemanaged wordt, wordt er een beeld gecreëerd dat het bedrijf stabiel is. 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 2.14 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. Hierdoor moeten enkele klanten hun betaalgedrag gecontroleerd 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 ' echt samengesteld en uit de 'rekken' gehaald. Het is een groot voordeel als ' dit gedeelte geautomatiseerd is. Bij het verkopen van een product moet de voorraad automatisch aangepast worden. Eens de producten van het order samengebracht zijn gaat men 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 ingegeven. Zodat 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 reeds zijn uitgehaald, zouden er hier minder problemen voorkomen. Wanneer 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

Deze punten moeten ingevoerd worden zodat het facturatiesysteem geautomatiseerd kan worden. Zodat 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 moeten bijhouden en ervoor zorgen dat er een melding gestuurd wordt nog voor de betalingsperiode is afgelopen. Dit valt onder de volgende stap accounts 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 in het systeem bijgehouden. De werknemers moeten 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 voor- en nadelen aan een OTC proces. Als het proces goed opgezet is, kan dit zorgen voor blijere klanten, minder wanbetalers, ...

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.

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. Bijvoorbeeld: meer dan de helft heeft een uniek verkoopsovereenkomst.
- Herhalend aanbiedingen en toegevingen moeten doen door fouten in het proces.
- Wat op de oorspronkelijke offerte stond, werd niet gefactureerd, er werd meer aangerekend.
- Een groot aantal creditnota's uitgegeven.
- Tussen de verkoop, verzending en het factureringsproces kan er informatie ontbreken.

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.

2.3.2 Technologie: Wat biedt SAP zelf aan voor microservices

Kyma is een recent project van SAP om toenadering te geven tot 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. Kyma (2019)

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. ' 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. Semerdzhiev (2018)

De samenwerking tussen Kyma en Knative is belangrijk. Er wordt een complete set van bouwblokken aangeboden en het zijn twee sterke frameworks. Bij het gebruik van beide kunnen er could-native oplossingen gebouwd worden op Kubernetes met een sterk framework. Op figuur 2.17 is te zien wat Kyma en wat Knative aanbiedt. Hofmann (2018) 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 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. 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 gewenste 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.

2.4 Requirements van de business

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

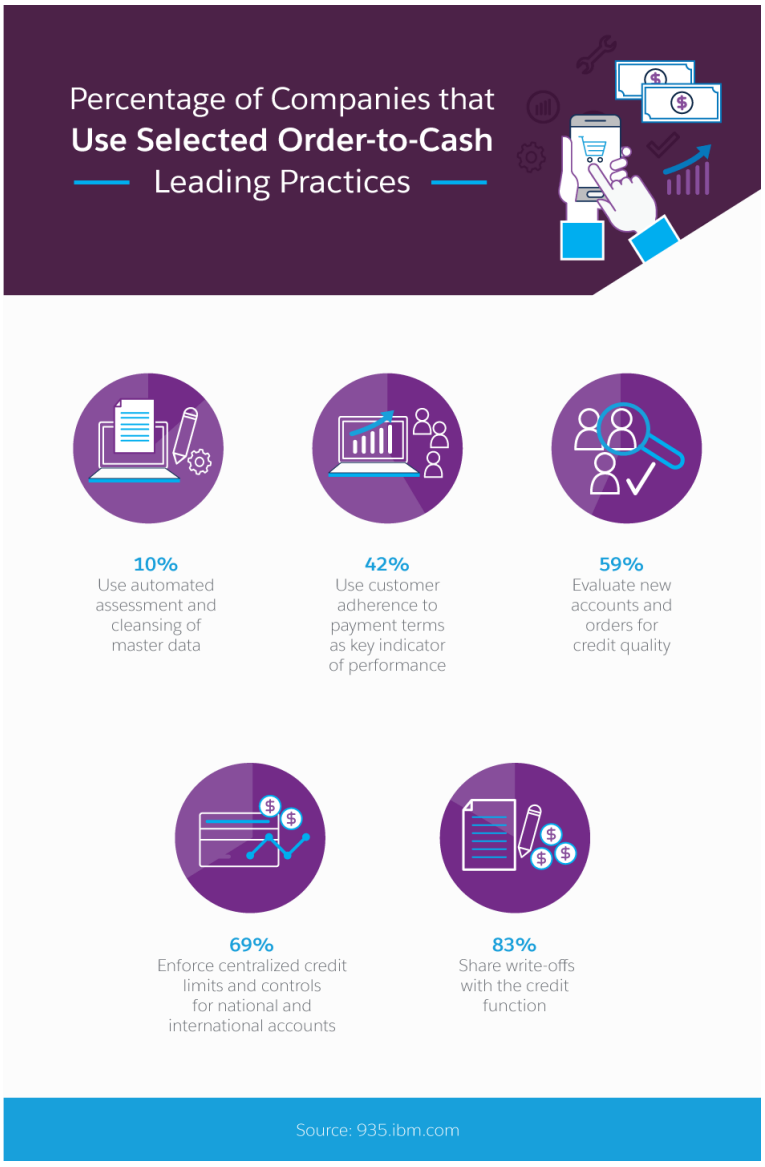
- De klant moet een order of meerdere orders kunnen plaatsen. Een goed werkend order-systeem is een must.
- Het order ophalen uit de voorraad. Er moet een goed voorraad beheer aanwezig zijn. Dit moet ervoor zorgen dat het aantal laattijdige leveringen geminimaliseerd wordt. Samen met het goede voorraadbeheer wordt ' best bijgehouden waar men goederen geplaatst heeft in het 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:

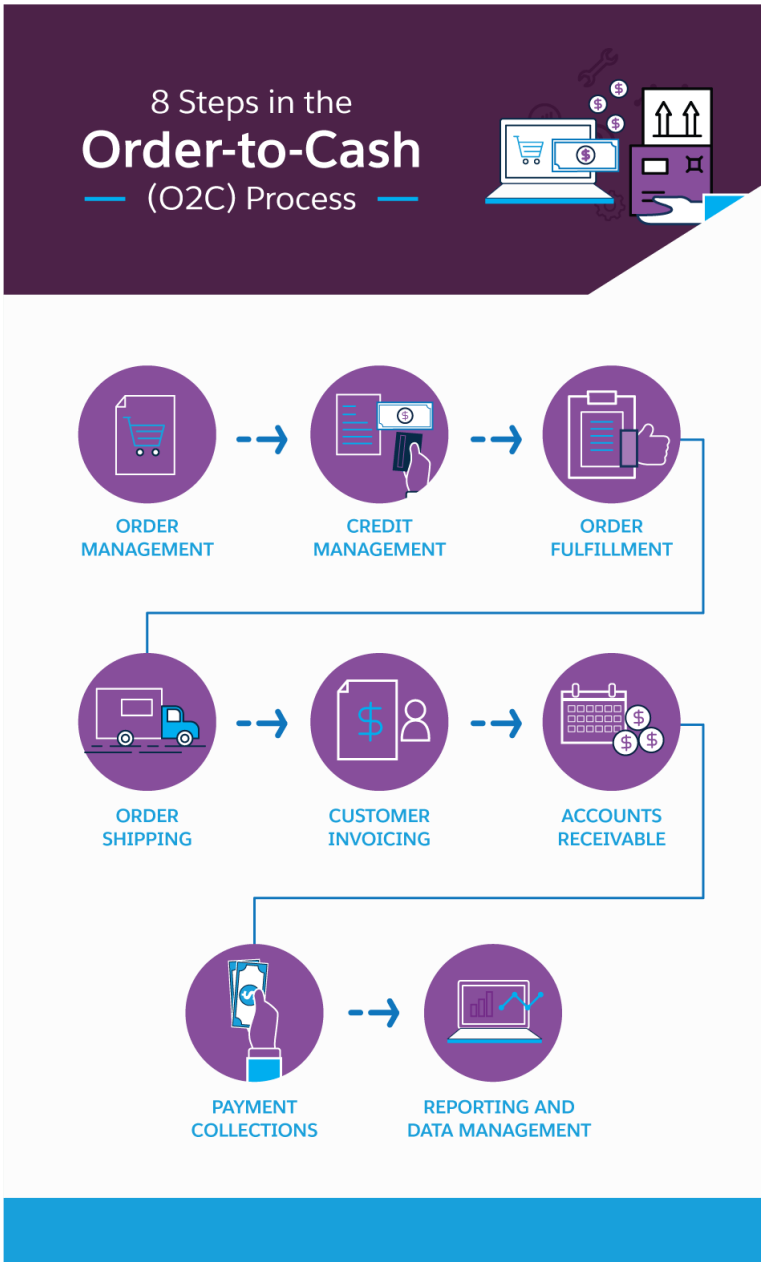
- Er moet een lijst met producten beschikbaar zijn.
- De klant zijn gegevens moeten gekend zijn bij het bedrijf.
- Het systeem bij het bedrijf moet beschikbaar zijn.

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

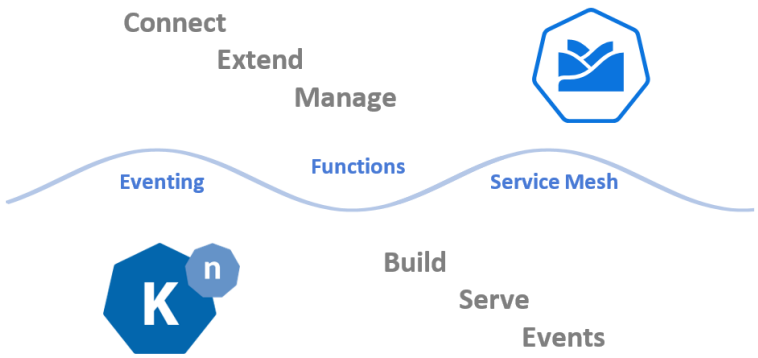
- Een goed voorraadbeheer is de eerste must.
- Een overzicht van waar alles staat, geeft een meerwaarde.
- Bij het ophalen van een product om bij een order te plaatsen, moet de hoeveelheid in de voorraad verminderen.



Figuur 2.20: Het percentage van van bedrijven dat gebruik maken van order-to-cash proces. Wong (2018)



Figuur 2.21: Het order-to-cash proces. Wong (2018)



Figuur 2.22: Wat Kyma, boven de lijn, aanbiedt en wat Knative, onder de lijn, aanbiedt Hofmann (2018).

3. Methodologie

In dit hoofdstuk zal er een schets gemaakt worden van het OTC proces met een microservice architectuur.

In de sectie 'Stand van zaken' zijn er acht stappen terug te vinden. In deze studie worden er zeven aangehaald. De laatste stap: 'reporting and data management' wordt niet behandeld.

Eerst worden er enkele termen uitgelegd. Dan worden alle microservices binnen het OTC proces uitgelgd. Wat ze precies doen en tot welk onderdeel, van het OTC proces, ze behoren. Hierna wordt de communicatiemethode tussen de verschillende microservices uitgelegd. Als voorlaatste wordt de databank structuur uitgetekend. Als laatste wordt de complete architectuur uitgelegd. Onder de complete architectuur worden volgende punten aangehaald:

- De communicatie tussen de onderdelen van het OTC proces. Hier wordt aan de hand van een schema uitgelegd hoe de communicatie gebeurt tussen de onderdelen.
- De architectuur. Dit deel bevat een schema, waar duidelijk op wordt welk onderdeel welke microservices aanspreekt. Per onderdeel wordt ook uitgelegd waarom deze microservices worden aangesproken.
- Als laatste worden de API gateway, logging, authenticatie en authorisatie toegevoegd.

3.1 Uitleg termen

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

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.
Datastore	Een datastore is een kleinere versie van een databank. Er kunnen files in opgeslaan worden.

Tabel 3.1: Termen die vaker voorkomen in dit hoofdstuk.

3.2 De microservices binnen het OTC proces

De microservices die logging, authenticatie, authorisatie en bescherming omvatten, zullen niet uitgelegd worden. De microservices die hier uitgelegd worden, liggen achter het OTC proces. Deze microservices worden aangesproken door de onderdelen van het OTC proces. De onderdelen van het OTC proces zijn ook microservices. Deze zijn microservices spreken de hieronder opgelijste microserivces aan. In dit deel wordt naar een databank verwijst. Deze gaat later aangehaald worden. Voor elk van volgende microservices gaan we volgende vragen beantwoorden:

- Waarom werd hiervan een microservice gemaakt?
- Wat kunnen ze?
- Door welk onderdeel van het OTC proces wordt deze gebruikt?

Deze microservices lopen gelijk met business requirements die terug te vinden zijn in de literatuurstudie. Eerst werd elk onderdeel van het OTC proces apart bekeken. Er werd gekeken naar de requirements van elk onderdeel. Na het bekijken van elk onderdeel, werd de conclusie getrokken dat bepaalde requirements meermaals terugkwamen. Die requirements zijn omgegoten in onderstaande onderdeel van de methodologie. Later in deze bachelorproef zal duidelijk worden welke microservices bij welk onderdeel van het OTC proces horen.

Er werd gekozen om de microservice niet nog kleiner te maken. De microservices zouden nog kleiner gemaakt kunnen worden. Maar hiervoor werd niet gekozen wegens overzichtelijkheid. Het opsplitsen in nog kleinere microservices zorgt ervoor dat men kan verloren lopen tussen de microservices. Nu weet men welke microservice men moet gebruiken als men iets met orders wil doen.

Een microservice is schaalbaar. Omdat de volgende requirement meermaals voorkomen in het proces, is het voordeliger om te dupliceren en te hergebruiken van deze microservices.

3.2.1 De microservices van het order-to-cash proces met hun onderliggende communicatie

Klantengegevens ophalen

Veel onderdelen van het OTC proces moeten aan de klantgegevens kunnen. Om te zorgen dat alles op een gelijke manier gebeurt, is hier een microservice van gemaakt. Deze microservice gaat ervoor zorgen dat de klantengegevens uit de databank worden gehaald. Het zal de databank aanspreken en vragen om de data van een specifieke klant. Deze wordt gebruikt in meerdere delen van het order-to-cash proces. Deze microservice komt voor in volgende onderdelen:

- Order management
- Credit management
- Order shipment
- Klant management
- Facturatie
- Accounts receivables

Orders plaatsen, ophalen en verwijderen

Een belangrijk onderdeel is het plaatsen, ophalen en verwijderen van orders. Er zijn een aantal onderdelen van het OTC proces dat de specificaties van een order moeten weten. Een aantal documenten zijn gebaseerd op het order, zoals een factuur en leveringspapieren. Deze microservice zal gebruikt worden in volgende onderdelen:

- Order management
- Credit management
- Order fullfilment
- Order shipment
- Facturatie
- Klant management

Producten ophalen en het aantal in voorraad veranderen

In het order-to-cash proces worden er geen producten toegevoegd aan de lijst dus moet dit niet in een microservice gestoken worden. Het aantal van de voorraad moet worden aangepast wanneer er een product uit de rekken wordt gehaald en bij een bestelling wordt geplaatst. Het ophalen van producten is vooral voor het werk achter de schermen. Het ophalen van een product omvat vooral de verkoopprijs ophalen om die dan te gebruiken in de orderlijn. Deze microservice zal gebruikt worden in volgende onderdelen:

- Order management
- order fullfilment

Facturatie maken en ophalen

Het begin van het einde in een order-to-cash proces. Facturen maken en ophalen zijn van groot belang bij een order-to-cash proces. Het zorgt er voor dat mensen geld gaan betalen voor hun order. Een factuur moet overeenstemmen met wat geleverd is. Het is van groot belang dat achter kan gekeken worden of de factuur klopt met de order. Het maken van de factuur gebeurt aan de hand van het order. Deze microservice komt voor in het onderdeel facturatie.

Shipment documentatie opstellen

Het shipment document wordt gegenereerd afhankelijk van het order. Er wordt gekeken naar het ordernummer en dan wordt er gekeken naar het klantnummer. Hierna wordt dan de microservice om klantgegevens op te halen aangesproken om de gegevens van de specifieke klant op te halen. Dit wordt enkel gebruikt binnen het onderdeel verzending.

Aanmaning opmaken en verwijderen

Het opmaken en verwijderen van aanmaningen gebeurt enkel wanneer een wanbetaler is. Het proces zou niet vaak mogen voorkomen.

Berichten plaatsen op de queue

Berichten plaatsen op de queue met de correcte gegevens. Deze microservice zorgt daarvoor. Hiervan wordt een microservice gemaakt omdat dit in elke stap voorkomt. Het is dan eenvoudiger om er een microservice van te maken, zodat elk onderdeel op een uniforme manier het bericht plaatst. Deze microservice komt voor in elk onderdeel van het proces.

Berichten ophalen van de queue

Elk onderdeel moet berichten van zijn queue kunnen halen. Door het meermaals voorkomen van deze taak, wordt er een microservice van gemaakt. Het ophalen van berichten gebeurt dan op een uniforme manier. Deze microservice komt voor in elk onderdeel van het proces.

3.2.2 Communicatie methode tussen microservices

Microservices moeten met elkaar kunnen communiceren. De manier wordt voorgesteld in figuur 3.1. Klant 66 plaats een order met vier verschillende producten. Het ordernummer is 23. De volledige bestelling komt binnen in het systeem en wordt op de queue van ordermanagement geplaatst. Het order wordt van de queue gehaald aan de hand van consumption en acknowledgement. Ordermanagement verwerkt de binnengekomen order. Dan wordt het klantnummer en ordernummer op de queue van creditmanagement

geplaatst. Creditmanagement haalt het bericht van de queue. Dan wordt er gekeken naar het betaalgedrag van de klant. Is het een wanbetaler of betaalt de klant altijd correct. Staat de klant gekend voor wanbetalen, dan wordt er geen goedkeuring gegeven om het order te plaatsen. Bij een corect betaalgedrag, krijgt het order goedkeuring om geplaatst te worden. Dan wordt het ordernummer doorgestuurd naar het volgende deel van het proces. Dus volgende queue's zouden moeten bestaan:

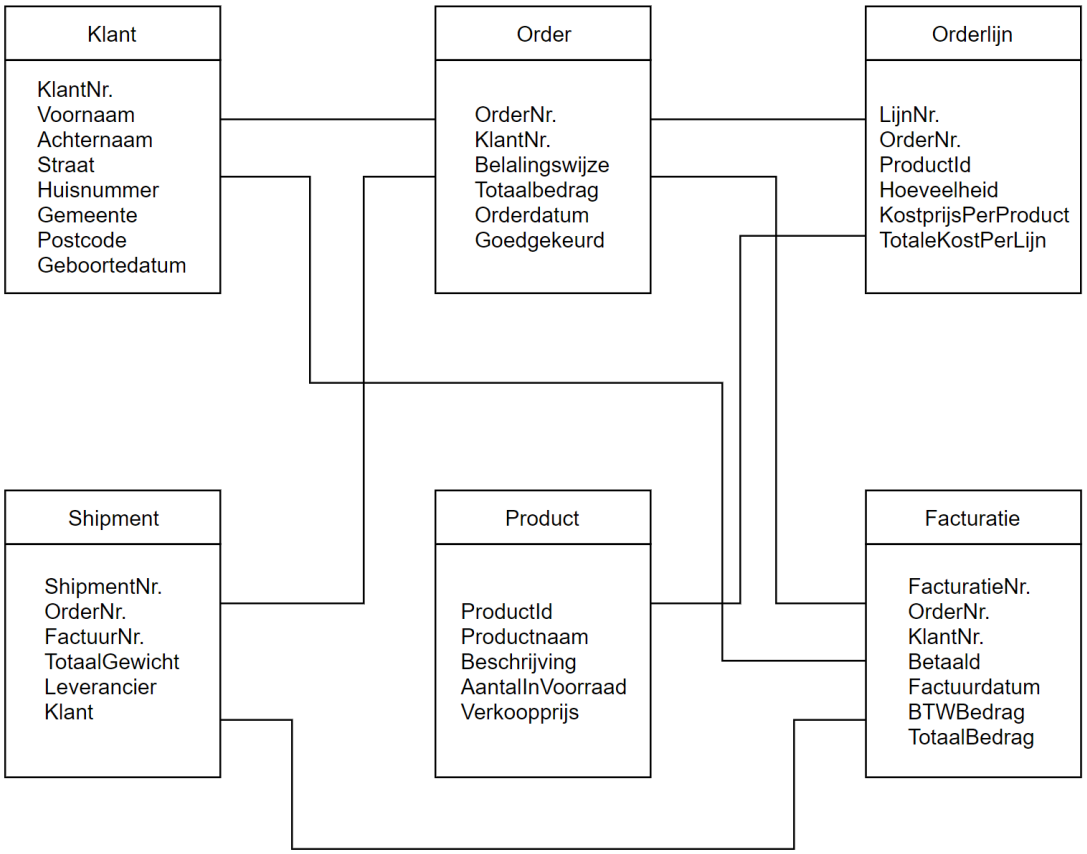
- QorderMan is een queue voor order management: Het plaatsen van een order.
- QcreditMan is een queue voor credit management: Om te controleren of een klant wel een order mag plaatsen.
- QorderFul is een queue voor order fulfilment: Het ophalen van de order in het magazijn.
- QorderShip is een queue 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 het nummer van de klant die een order plaatste. Het ordernummer en klantnummer worden naar creditmanagement doorgestuurd. Bij creditmanangement wordt er dan aan de hand van het klantnummer de gegevens opgehaald en dan zo nagegaan of die klant wel een order mag plaatsen. Het ordernummer werd meegestuurd om zeker te zijn dat de correcte order goed- of afgekeurd wordt. 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. Naast de grote databank, heeft elke microservice zijn datastore. Bij een verandering in een datastore, wordt deze aangebracht in de algemene databank. Eens de databank het record heeft toegevoegd of aangepast, stuurt hij een bericht naar elke datastore zijn queue om te melden dat er een verandering gebeurt is. Leest een microservice het bericht van verandering, dan wordt er een nieuwe kopie van de masterdata gemaakt. De master data zit in een grote databank zodat er maar één plaats is waar de correcte data zit. De verschillende datastores nemen een kopie van de data die zij nodig hebben.

Een order weet wie zijn klant is. Credit management kan de eigenschap bij het order veranderen van 'goedgekeurd' naar 'niet goedgekeurd'. Van elk order wordt een orderlijn bijgehouden. Zodat er geweten is wat er op de order staat. Op elk orderlijn staat er een product of service. Van het product moet er een beschrijving en verkoopprijs zijn. Het aantal in voorraad is gekend omdat bij het order fullfilment moet de voorraad aangepast worden van het specifieke product. Een order en de klant zijn gekend op de facturatie. Zo



Figuur 3.1: De databank structuur.

kan er nagegaan worden of de factuur overeenkomt met wat er op het order staat. De klant zijn gegevens moeten gekend zijn om naar het juiste adres te factureren.

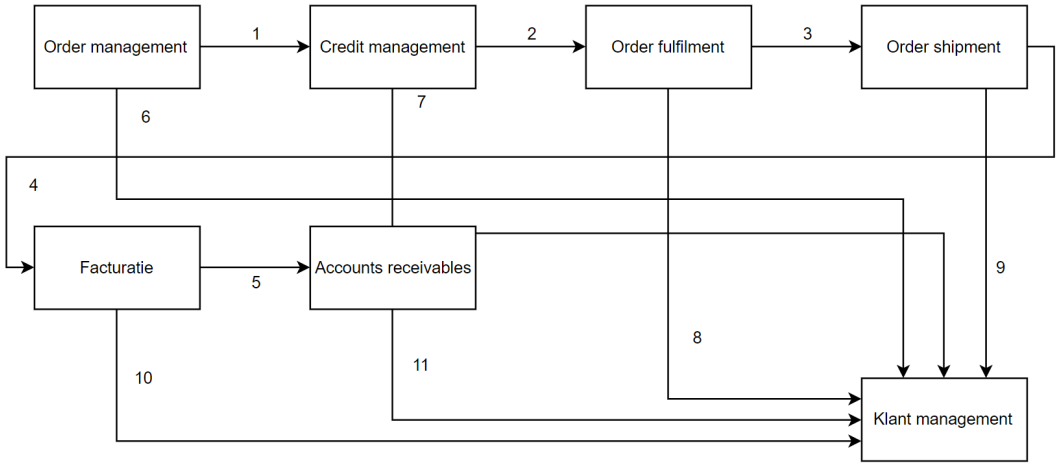
Elke datastore heeft een queue. Wanneer een microserivce B iets nodig heeft van microserivce A, kan die vraag naar de queue gepost worden.

De datastore van microservice 'klant gegevens ophalen' zal alle data van de klanten bevatten. De datastore van microservice 'order plaatsen, ophalen en verwijderen' zal alle data van de orders bevatten. De datastore van microservice 'producten ophalen en aantal in de voorraad veranderen' zal alle data van de producten bevatten. De datastore van microservice 'facturatie maken en ophalen' alle data van de facturatie bevatten. De datastore van microservice 'Shipment documentatie opstellen' bevat de data van de orders om aan de hand van die data, de documenten op te stellen en dan toe te voegen aan de data met alle shipment documenten. De datastore van microservice 'Aanmaning opmaken en verwijderen' bevat alle data van de aanmaningen. De microservices 'bericht op queue plaatsen' en 'bericht van de queue halen', hebben geen datastore. Het zijn twee microservices die geen data moeten gaan ophalen of wegschrijven. Ze staan in voor het ophalen en plaatsen van berichten.

3.4 De complete architectuur opbouwen

Op figuur 3.3 is te zien hoe het proces in elkaar ziet. Door de microservice architectuur toe te passen op het order-to-cash proces verandert er niks aan de volgorde van het proces of het doel van het proces. De manier waarop gegevens worden opgehaald, verandert wel.

3.4.1 De communicatie tussen de onderdelen van het OTC proces



Figuur 3.2: De communicatie tussen de verschillende onderdelen van het order-to-cash proces.

Zoals al eerder vermeld, elk onderdeel van het OTC proces is een microservice. Elk onderdeel laat andere microservices het werk doen. Op figuur 3.2 is het communicatie patroon zichtbaar. Voordat het proces van start kan gaan, moet een klant een order plaatsen. Een order bevat een aantal producten. Eens de klant een order geplaatst heeft dan is er een ordernummer gekend. De klant staat bekend in het systeem met een klantnummer. Dat zijn twee belangrijke gegevens die veel gebruikt zullen worden. Meer uitleg over wat elk onderdeel doet, bevindt zich in het volgende hoofdstuk. De nummers van de opsomming komen overeen met die op afbeelding 3.2.

1. De communicatie tussen ordermanagement en creditmanagement. Er is een order geplaatst door klant 66. Ordermanagement zorgt ervoor dat het order in de databank geraakt. Eens de taak van ordermanagement gedaan is, plaatst die het klantnummmer en het ordernummer op de queue van creditmanagement. Dan is zijn taak gedaan. Ordermanagement houdt zich niet bezig met wat er gebeurt nadat hij het bericht geplaatst heeft.
2. De communicatie tussen creditmanagement en order fullfilment. Creditmanagement heeft het ordernummer en klantnummer van ordermanagement gekregen. Het bericht wordt van de queue gehaald. Creditmanagement gaat dan aan de hand van de klantgegevens beslissen of de klant het order mag plaatsen of niet. Is het order goedgekeurd dan wordt het ordernummer op de queue van order fullfilment gezet.

Er moet niet worden teruggekeerd naar order management. Credit management heeft de rechten om het veld goedgekeurd aan te passen. De taak van creditmanagement is nu afgerond.

3. De communicatie tussen order fullfilment en order shipment. Order fullfilment heeft het ordernummer doorgekregen van credit management. Order fullfilment geeft het ordernummer door aan order shipment eens zijn taak gedaan is.
4. De communicatie tussen order shipment en de facturatie. Het ordernummer werd op de queue van order shipment geplaatst. Eens alles opgemaakt is bij order shipment, wordt het ordernummer doorgestuurd naar de facturatie. De taak van order shipment is afgerond.
5. De communicatie tussen facturatie en accounts receivables. De facturatie kreeg een bericht van order shipment om de factuur op te maken voor het order met des betreffende ordernummer. Is de taak van facturatie afgerond dan wordt de factuurnummer doorgestuurd naa accounts receivables. Zodat de betaling verder kan worden opgevolgd.
6. Ordermanagement plaatst het ordernummer op de queue van klant management. Klant management stuurt een bevestiging naar de klant van het goed ontvangen van zijn/haar order. En laat weten dat er eerst een controle komt op het betaalbedrag van de klant.
7. Creditmanagement plaatst het ordernummer en de status van goedgekeurd of afgekeurd op de queue. Dan wordt er een bericht verzonden naar de klant. Het bericht bevat de info of de klant zijn order is goedgekeurd of afgekeurd.
8. Order fullfilment plaatst een bericht op de queue om te laten weten dat het order gemaakt is. Dat de goederen uit het magazijn zijn opgehaald en klaar gemaakt worden voor verzending.
9. Order shipment plaatst een bericht op de queue wanneer de goederen klaar zijn voor vertrek. Klant management zorgt er dan voor dat het bericht tot bij de klant geraakt.
10. Facturatie plaatst een bericht op de queue van klant management om ervoor te zorgen dat de factuur tot bij de klant geraakt.
11. Accounts receivables plaatst een bericht op de queue van klant management als er aanmaningen moeten worden gestuurd.

3.4.2 De architectuur

In volgende tabel, zijn er letters en bijhorende microservices terug te vinden. Deze worden gebruikt bij het uitleggen van de architectuur.

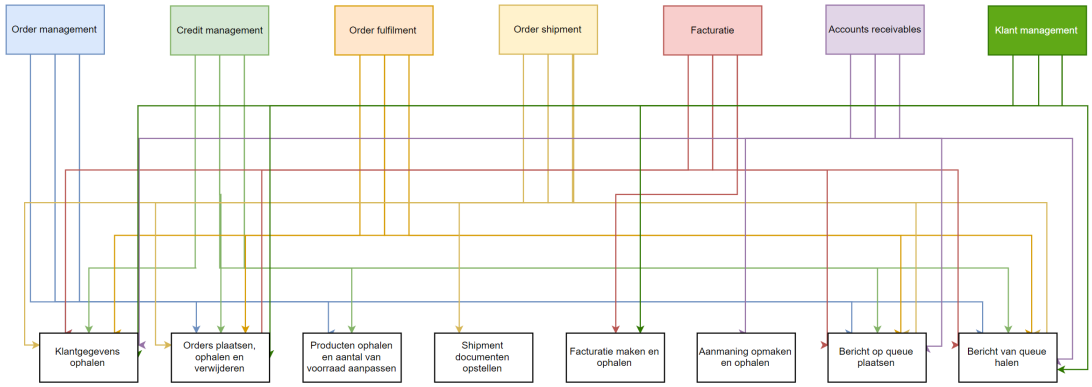
In tabel 3.2 is de legende terug te vinden die gebruikt wordt bij het uitleggen van elk onderdeel apart. In figuur 3.3 is te zien hoe het volledige proces communiceert met de verschillende microservices. Elk onderdeel van het proces is een microservice op zich. Een microservice die meerdere microservices aanspreekt om goed te functioneren.

Order management

Order management communiceert met volgende microservices:

A	Klantgegevens ophalen
B	Orders plaatsen, ophalen en verwijderen
C	Producten ophalen en het aantal van voorraad aanpassen
D	Shipment documenten opstellen
E	Facturatie maken en ophalen
F	Aanmaning opmaken en verwijderen
G	Bericht plaatsen op de queue
H	Bericht ophalen van de queue

Tabel 3.2: Legende die gebruikt wordt in de afbeeldingen.



Figuur 3.3: Welk onderdeel, welke microservices aanspreekt.

- Klantgegevens ophalen.
- Orders plaatsen, ophalen en verwijderen.
- Producten ophalen en het aantal van de voorraad aanpassen.
- Bericht plaatsen op queue.
- Bericht van queue halen.

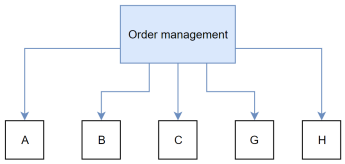
Een klant plaatst een order op het platform van het bedrijf. De gegevens van de klant worden opgehaald en gelinkt aan het nieuw gecreëerde order. Eens die twee taken zijn afgerond, wordt het bericht op de queue geplaatst van credit management.

Credit management

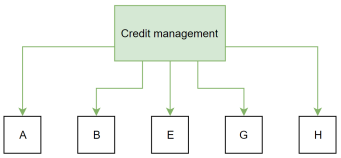
Credit management communiceert met volgende microservices:

- Klantgegevens ophalen.
- Orders ophalen, plaatsen en verwijderen.
- Facturatie maken en ophalen.
- Bericht plaatsen op queue.
- Bericht van queue halen.

Eerst wordt het bericht dat door order management op de queue geplaatst werd, opgehaald. Dan is er een klantnummer en een ordernummer. Bij credit management kijkt of de klant



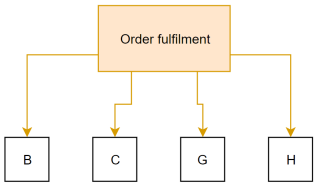
Figuur 3.4: Order management communiceert met volgende microservices.



Figuur 3.5: Credit management communiceert met volgende microservices.

een goed betaalgedrag heeft. Staat de klant gekend als wanbetaler dan naar het order gegaan om de plaatsing van het order af te keuren. In andere gevallen krijgt de order een goedkeuring en wordt het ordernummer geplaatst op de queue van order fullfilment. Dan zit de taak van credit management erop.

Order fullfilment



Figuur 3.6: Order fullfilment communiceert met volgende microservices.

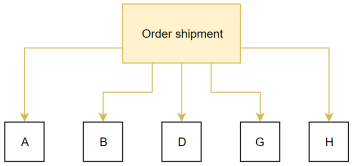
Order fullfilment communiceert met volgende microservices:

- Orders ophalen, plaatsen en verwijderen.
- Producten ophalen en het aantal in de voorraad aanpassen.
- Bericht plaatsen op queue.
- Bericht van queue halen.

Bij order fullfilment staat er een ordernummer op de queue. Die wordt opgehaald en aan de hand van het ordernummer, weet men welke goederen er uit het magazijn moeten gehaald worden. Voor alle goederen moet het aantal in de voorraad aangepast worden. Eens het order compleet is, wordt het ordernummer op de queue van order shipment geplaatst.

Order shipment

Order shipment communiceert met volgende microservices:

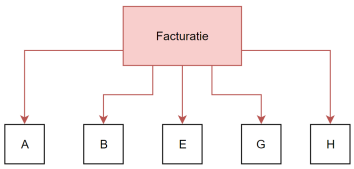


Figuur 3.7: Order shipment communiceert met volgende microservices.

- Orders ophalen, plaatsen en verwijderen.
- Klantgegevens ophalen.
- Shipment document opstellen.
- Bericht plaatsen op queue.
- Bericht van queue halen.

Order shipment krijgt het ordernummer binnen via zijn queue. Het order wordt bekeken en de klant gegevens worden opgehaald aan de hand van het gekoppelde klantnummer aan het order. Eens het order shipment document is opgesteld en alle goederen klaar zijn voor vertrek, wordt het ordernummer geplaatst op de queue van facturatie.

Facturatie



Figuur 3.8: Facturatie communiceert met volgende microservices.

Facturatie communiceert met volgende microservices:

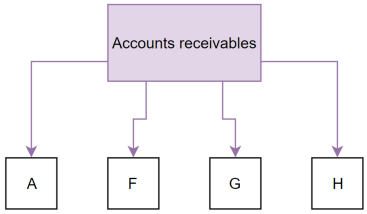
- Klantgegevens ophalen.
- Orders ophalen, plaatsen en verwijderen.
- Facturatie maken en ophalen.
- Bericht plaatsen op queue.
- Bericht van queue halen.

Eens het ordernummer opgehaald is van de queue, wordt alle nodig info opgehaald om de factuur op te maken. De nodige gegevens zijn klantgegevens en het order. Eens de factuur opgemaakt is, wordt het factuur nummer geplaatst op de queue van account receivables.

Accounts receivables

Accounts receivable gaat na of de betaling wel goed wordt afgerond. Om dit te kunnen, worden volgende microservices aangesproken:

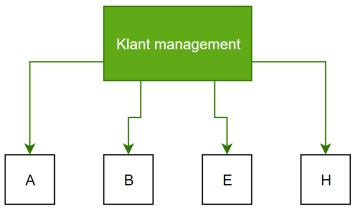
- Klantgegevens ophalen.



Figuur 3.9: Accounts receivables communiceert met volgende microservices.

- Facturatie maken en ophalen.
- Bericht plaatsen op queue.
- Bericht van queue halen.

Klant management



Figuur 3.10: Klant management communiceert met volgende microservices.

Klant management staat in voor het contact met de klant. Dit deel gebruikt volgende microservices:

- Klantgegevens ophalen.
- Orders ophalen, plaatsen en verwijderen.
- Facturatie maken en ophalen.
- Bericht van queue halen.

Als een van de vorige onderdelen een bericht plaatst op de queue, zal die verwerkt worden en de nodige info wordt opgehaald.

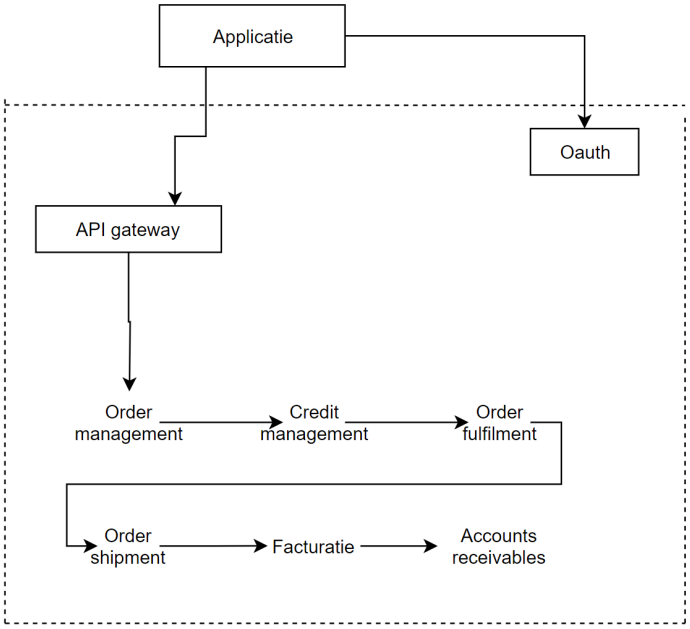
3.4.3 De volgende stappen: Het toevoegen van API, logging en authenticatie en authorisatie

Logging

Logging kan toegepast worden door nog een extra microservice te ontwikkelen. Hiernaar kunnen de andere microservices hun logs posten. Hier zal niet verder op worden ingegaan.

API gateway

Een API gateway wordt toegevoegd om ervoor te zorgen dat de architctuur niet openbaar is voor iedereen. Er is één aanspreekpunt en die regelt de rest van de communicatie. De API communiceert in dit geval enkel met order management. De API geeft alle nodige informatie door en order management schrijft deze weg naar de datastore en naar de databank.



Figuur 3.11: Vereenvoudigd schema.

Authenticatie en autorisatie

Onder het hoofdstuk 'stand van zaken', is een vergelijking te vinden over authenticatie. Uit die vergelijking, is de keuzen API gekozen als authenticatie. Binnen API zijn er verschillende opties, die worden ook overlopen in het vorige hoofdstuk. Er is gekozen voor OAuth omdat dit de meest bekende en gebruikte versie is.

De gebruiker moet gekend zijn voordat die een order kan plaatsen. De authenticatie van de gebruiker gebeurt vooraf. Er is een aparte micrservice voor authenticatie. Hier wordt niet dieper op in gegaan. Op afbeelding 3.12 is wel een mogelijke plaatsing zichtbaar.

4. Conclusie

De conclusie van deze theoretische studie zal in dit deel duidelijk worden.

Als eerste werd de vraag 'Wat zijn microservices?' gesteld. Microservices zijn kleine delen die elk een requirement omvatten. Elke microservice functioneert apart en heeft weinig tot geen communicatie met de andere microservices. Om microservices te gaan ontwikkelen, moet er veel onderzocht worden. Wat te merken is aan de literatuurstudie. De literatuurstudie is niet heel diepgaand maar probeert wel zo veel mogelijk aspecten van microservices aan te halen. Microservice kan in meerdere contexten voor komen. Binnen elke context zal er meer verkaring nodig zijn.

De volgende vraag was 'Hoe kan er overgeschakeld worden naar een microservice architectuur?'. Er kan op verschillende manieren overgeschakeld worden naar microservices. Maar de meest aangeraden manier is: systematisch requirements vaststellen en omzetten naar een microservices. In een stappenplan werken, zodat er geen chaos ontstaat bij de overschakeling. Eerst de kern opbouwen en dan alle andere zaken erbij brengen. Eens de kern van de architectuur is getekend, kan men authenticatie, authorisatie, bescherming, logging en de API gateway er in steken. Er moet eerst goed worden nagedacht over hoe de implementatie van authenticatie, authorisatie, bescherming en logging zal gebeuren. Voor de overschakeling kan er gekeken worden naar de manier waarop men nu de authenticatie, authorisatie, bescherming en logging toepast en die manier dan verwerken bij de microservices.

Daarna kwam 'Welke aanpassingen kunnen of moeten er gebeuren aan de architectuur om microservices te laten werken? De veranderingen worden hieronder opgesomd:

- Heeft deze applicatie nood aan een microservice architectuur?

- De databank structuur moet herbekeken worden.
- Het volledige proces herbekijken.
- De manier van communiceren tussen de verschillende onderdelen moet worden onderzocht.
- Welke manier van authenticatie en authorisatie gaat men gebruiken?
- Welke manier van bescherming kan er toegepast worden?
- Moeten we de teams herstructureren?

Een andere vraag die in het begin gesteld werd: Hoe zal de communicatie tussen de verschillende microservices werken? Voor de communicatie zijn er veel verschillende opties. Er moet gekeken worden naar welke manier het beste bij het bedrijf past. Is er een manier die al gekend is in het bedrijf? Heeft een andere manier enkele voordelen ten opzichte van de huidige techniek. In deze studie is er gekozen voor volgend communicatie methode: Via message queues. De data die microservice B nodig heeft, zal door microservice A naar microservice B zijn queue worden gestuurd. Bij deze manier heeft elke microservice een queue. Heeft microservice A data doorgekregen en haar taak ermee gedaan, dan plaatst ze een link naar de data op de queue van de microservice die de data nodig heeft. Dit zorgt ervoor dat als microservice B uitvalt, de nodige data niet verloren gaat. De queue slaat de gegevens op en verwijdert ze pas als ze zijn opgehaald. Als deze methode niet past binnen de verwachtingen van de architectuur dan kan men onderzoeken welke methode beter is.

Als volgt werd deze vraag gesteld: Hoe ziet een order-to-cash proces eruit? Het order-to-cash proces is een proces dat voorkomt bij het plaatsen van een order tot het betalen van de factuur. In volgende opsomming zijn de stappen terug te vinden:

1. Order management
2. Credit management
3. Order fulfilment
4. Order shipping
5. Facturatie
6. Accounts receivables

De voorlaatste vraag, klinkt als volgt: Welke business requirements heeft een order-to-cash proces? De belangrijkste requirement is dat de klant zijn bestelling krijgt. Een andere requirement is de wanbetalers minimaal te houden door ze vroeg in het proces er uit te halen. De requirements liggen in lijn met de noden van de klant. De business requirements werden opgesteld door het order-to-cash proces grondig te bestuderen. In de literatuurstudie is meer uitleg te vinden over het OTC proces.

Als laatste vraag werd 'Welke invloed heeft de microservice architectuur op de performance van een order-to-cash proces?' gesteld. Het verschil op performance vlak kan pas getest worden bij een studie met een proof of concept. Wat wel geconcludeerd kan worden, is volgende: De performance zal stabieler zijn bij pieken van orders. Omdat de microservices onafhankelijk zijn van elkaar. De onderdelen van het OTC proces kunnen onafhankelijker functioneren. Valt er een onderdeel uit of tijdelijk niet beschikbaar dan heeft dat geen grote gevolgen voor het proces in het algemeen. Het onderdeel ervoor kan

nog altijd data op de queue plaatsen. Maar de onderdelen na het onbeschikbare deel, zal wel wat hinder ondervinden. Maar niet zodat het gehele proces plat zal liggen.

Microservice is een architectuur en ideologie waarin logica en de requirements, die te vinden zijn bij een monolithic, terugkomen. Dit heeft invloed op de overschakeling naar microservices. Deze manier van implementatie moet aangepast worden bij het gebruik van microservices. Dat kan een obstakel zijn. Er wordt een andere manier van denken en organiseren gevraagd binnen een de IT-afdeling. De teams worden hervormd. Een team bevat niet meer de kennis van de volledige architectuur., enkel over de microservice waaraan zij werken.

Ten slotte kan er geconcludeerd worden dat:

- Microservices een brede term is en kan voorkomen in verschillende contexten.
- Een microservice architectuur niet altijd nodig is.
- Microservices kunnen een OTC proces automatiseren.
- Microservices zorgen ervoor dat elk onderdeel van het proces apart kan functioneren.

Zelf heb ik ondervonden dat het aanpassen van een monolithic niet eenvoudig is. De monolithic architectuur zit ingebakken in het informatica landschap. De manier van omgang met microservices, kan vergeleken worden met die van Agile. De Agile methode moest in eerste instatie zijn toegevoegde waarde op een project bewijzen. Dan pas zijn projectmanagers deze ook gaan gebruiken. Microservices zal eerst zijn voordeel moeten bewijzen ten opzichte van monolithic om een statement te kunnen maken.

A. Onderzoeksvoorstel

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

A.1 Introductie

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*.

Alley, G. (2018). What is ETL? *https://www.alooma.com/blog/what-is-etl*.

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

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

Aradhye, Y. (2018, november 14). How integration patterns impact your microservices architecture. *https://headspring.com/2018/11/14/integration-patterns-microservices-architecture/*.

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/ 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*.

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

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

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*.

Guru. (2019). ETL (Extract, Transform, and Load) Process. *https://www.guru99.com/etl-extract-load-process.html*.

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

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

Kumar, R. (2018, september 13). Selecting the Right Database for Your Microservices. *https://thenewstack.io/selecting-the-right-database-for-your-microservices/*.

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

Loshin, D. (2019). What is ETL? *SAS*. Verkregen van https://www.sas.com/en_us/insights/data-management/what-is-etl.html

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*.

Monson, D. (2019, maart 18). Microservices Anti-Patterns. *https://keyholesoftware.com/2019/03/18/microse anti-patterns/*.

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/micr devops-better-together*.

Naveen, B. (2016). What is ETL? An introduction. *https://intellipaat.com/blog/tutorial/data-warehouse-tutorial/what-is-etl/*.

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

Panolapy. (2019). 3 Ways to Build An ETL Process. *https://panoply.io/data-warehouse-guide/3-ways-to-build-an-etl-process/*.

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/introduc 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/*.

Solance. (2018). Communication Between Microservices: Why You Need Messaging. *https://solace.com/blog/messaging-between-microservices/*.

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*.

Stich. (2019). ETL Database. *https://www.stitchdata.com/etldatabase/etl-process/*.

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/*.