



HoGent

Faculteit Bedrijf en Organisatie

Welke technologie is het meest geschikt om in een IoT-applicatie data van microservices te gebruiken?

Ruben Desmet

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

Promotor:
Sonia Vandermeersch
Co-promotor:
Maarten Meersseman

Instelling: TVH

Academiejaar: 2018-2019

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Welke technologie is het meest geschikt om in een IoT-applicatie data van microservices te gebruiken?

Ruben Desmet

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

Promotor:
Sonia Vandermeersch
Co-promotor:
Maarten Meersseman

Instelling: TVH

Academiejaar: 2018-2019

Tweede examenperiode

Woord vooraf

Samenvatting

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus.

Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetur adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Inhoudsopgave

1	Inleiding	13
1.1	Probleemstelling	14
1.2	Onderzoeksvraag	14
1.3	Onderzoeksdoelstelling	14
1.4	Opzet van deze bachelorproef	14
2	Stand van zaken	15
2.1	Microservices	15
2.2	Kafka	17
2.3	RabbitMq	19
2.4	Kafka vs RabbitMq	20
2.5	Google Pub/Sub	21

2.6	IoT-applicatie	22
2.6.1	Informatie verzamelen en verzenden	24
2.6.2	Informatie verzamelen en reageren	24
2.6.3	Combinatie	24
2.7	TVH	25
3	Methodologie	27
3.1	Producer project	27
3.1.1	Kafka	29
3.1.2	RabbitMq	33
3.1.3	Google Pub/Sub	35
3.2	Consumer project	37
3.3	Onderzoek	37
4	Conclusie	39
A	Onderzoeksvoorstel	41
A.1	Introductie	41
A.2	Literatuurstudie	41
A.3	Methodologie	42
A.4	Verwachte resultaten	42
A.5	Verwachte conclusies	43
	Bibliografie	45

Lijst van figuren

2.1	Monolitische architectuur	16
2.2	Wisselwerking tussen partities en consumer-groepen, (Sookocheff, 2015)	18
2.3	Voorbeeld van een Kafka cluster, (Johansson, 2016)	19
2.4	Simpele voorstelling van de werking van een offset, (Sookocheff, 2015)	19
2.5	Werkings drie verschillende exchanges bij <i>RabbitMQ</i> , (Johansson, 2015)	20
2.6	Populariteit <i>Kafka</i> en <i>RabbitMQ</i> in het afgelopen jaar, (Trends, 2019)	21
2.7	Populariteit <i>Kafka</i> en <i>RabbitMQ</i> in de afgelopen vijf jaar, (Trends, 2019)	21
2.8	De flow die een message aflegt bij <i>Google Pub/Sub</i> , (Google, 2019)	22
2.9	De gecreëerde waarde van IoT, (i-scoop, 2019)	23
3.1	Configuratie van de topic	29
3.2	Configuratie van de topic voor de gebruikte subscription	35

Lijst van tabellen

1. Inleiding

Dit onderzoek wordt uitgevoerd ten voordele van het bedrijf TVH. Dit is een familiebedrijf die oorspronkelijk opgericht is door Paul Thermote en Paul Vanhalst. Ze begonnen in een schuur waar ze kapotte landbouwmachines en forklifts repareerden. Later begonnen ze ook met het verkopen van deze machines. Dit bestond voornamelijk uit het repareren van tweedehands machines en het verkopen ervan. Vervolgens begonnen ze snel met het verhuren van deze werktuigen. Nog enkele jaren later begonnen ze met de verkoop van vervangonderdelen voor machines. Daarna werden er nog een paar bedrijven overgenomen waardoor het bedrijf uitgroeide tot het grote bedrijf dat we vandaag de dag kennen.

Maar wat is het nut nu van dit onderzoek voor TVH? Zoals je overal ziet, is internet bijna nergens meer weg te denken. Ook voor dit bedrijf kan internet enkele voordelen bieden. Zoals later vermeld in sectie 2.6 over IoT-applicatie gebruikt TVH vooral de eerste categorie, namelijk het verzamelen van informatie en het verzenden ervan.

Al deze informatie wordt samengebracht en verwerkt. De IoT-applicatie die deze informatie daadwerkelijk zal verwerken heeft een grote verantwoordelijkheid. Het is namelijk belangrijk dat er geen data verloren gaat bij het verzenden en dat de data heel snel verwerkt kan worden. Er zijn enkele mogelijkheden van software om met deze informatie om te gaan. Dit onderzoek zal nagaan welke technologie het beste is om te gebruiken voor TVH met zijn specifieke data.

Doordat dit onderzoek zal aantonen welke technologie de beste zal zijn, ontstaan er enkele voordelen voor het bedrijf TVH. Op deze manier kan er data op de meest efficiënte wijze verzonden worden waardoor alles ook sneller verwerkt kan worden. Hierdoor kan het bedrijf meer winst maken omdat het sneller conclusies kan trekken uit de verkregen data.

1.1 Probleemstelling

Als technologie werd er vroeger binnen het bedrijf *RabbitMQ* gebruikt. Ondertussen zijn alle teams overgeschakeld naar een andere technologie namelijk *Kafka*. Oorspronkelijk was het dus de bedoeling van dit onderzoek om deze twee met elkaar te vergelijken en te concluderen dat TVH al dan niet de juiste beslissing had gemaakt om over te schakelen naar *Kafka*. Ondertussen ben ik ook al gestart met mijn stage binnen TVH waarbij ik deel uitmaak van een student-squad. Binnen ons team gebruiken wij *Kafka* noch *RabbitMQ*. Wij gebruiken een derde mogelijkheid *Google Pub/Sub*. Maar alle andere teams hebben nog steeds *Kafka* als technologie dat ze gebruiken. Er zijn dus verschillende technologieën die gebruikt worden binnen dit bedrijf. Waarbij niemand weet welke nu eigenlijk het beste is om te gebruiken.

1.2 Onderzoeksvraag

De bedoeling van dit onderzoek is deze drie technologieën vergelijken en concluderen welke ervan de beste blijkt te zijn. Zo komt het bedrijf te weten als het handig zou zijn om in elk team te blijven werken met *Kafka* of om terug over te schakelen naar een andere technologie.

1.3 Onderzoeksdoelstelling

Dit onderzoek zal geslaagd zijn indien we voor TVH met hun specifieke data kunnen concluderen welke technologie het beste kan gebruikt worden voor alle teams. Op deze manier werkt elk team met dezelfde technologie en kan er ook meer kennis gedeeld worden met elkaar onderling.

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. Enkele begrippen worden grondig uitgelegd alsook de werking van de verschillende technologieën.

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 4, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen.

2. Stand van zaken

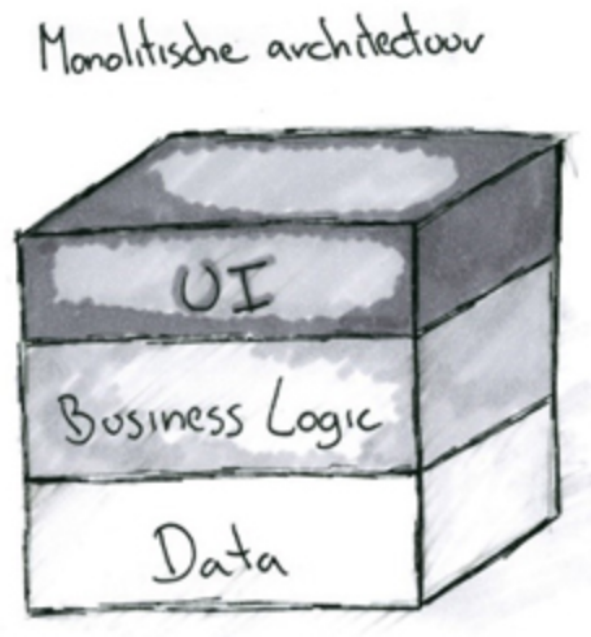
In dit hoofdstuk gaan we Kafka en RabbitMq bespreken. Want natuurlijk zijn de begrippen zoals *Kafka* en *RabbitMq* niet voor iedereen even duidelijk. Ook de term microservices zal bij sommigen de wenkbrauwen wel eens doen fronsen. In dit hoofdstuk is het de bedoeling uit te leggen wat al deze begrippen betekenen. Na dit hoofdstuk zul je dankzij deze uitleg in staat zijn om om alles goed te begrijpen.

2.1 Microservices

Microservices is een software architectuur. De applicatie bestaat uit meerdere kleinere microservices die samen één groot geheel vormen. Deze kleinere microservices zijn onafhankelijk van elkaar en hebben elk hun eigen proces. Deze software architectuur is vrij recent en is de laatste jaren een echte hype aan het worden.

Een belangrijke vraag is: waarom zijn microservices ontstaan? Server-side applicaties gebruiken meestal object-georiënteerde programmeertalen. Deze programmeertalen hebben abstracties om de complexiteit van hun programma's te behandelen in modules. Dit wordt ook wel eens 'the monolith' genoemd. Dit is één groot geheel dat meestal uit drie lagen bestaat. De presentatielaag, de businesslaag en de data laag. Deze lagen kunnen niet apart van elkaar gebruikt worden omdat ze verschillende resources met elkaar delen. Dit zorgt ervoor dat bij iedere wijziging in de applicatie, alle lagen opnieuw gereleased worden in een nieuwere versie. Je kan natuurlijk al raden dat dit in een grote applicatie niet de beste oplossing is.

Daarom is er een andere architectuur die een heel andere aanpak heeft, namelijk microservices. Dit bestaat uit verschillende kleinere componenten die indien nodig onafhankelijk



Figuur 2.1: Monolitische architectuur

van elkaar uitgevoerd kunnen worden. Bij deze aanpak is het ook belangrijk dat je uw services klein houdt. Hierdoor kunnen je services gemakkelijk hergebruikt, begrepen en opnieuw gebuild worden. Iedere microservice heeft dus maar 1 verantwoordelijkheid.

Omdat microservices op verschillende machines moeten kunnen draaien, bijvoorbeeld op verschillende besturingssystemen, is het beter om ze in te pakken samen met hun dependencies in een container. *Docker* is een voorbeeld van een technologie die deze containers aanbiedt. Door microservices in containers te plaatsen, zorg je ervoor dat de uitvoering van deze services onafhankelijk gebeurt van andere applicaties op dezelfde machine. Containers zijn dus onafhankelijk van een besturingssysteem, hierdoor kunnen microservices op verschillende locaties gedraaid worden. Een van de voordelen hiervan is dat ze door containers gemakkelijker met de cloud kunnen werken. Een ander voordeel is dat iedere microservice zijn eigen verantwoordelijkheid heeft. Omdat ze maar één verantwoordelijkheid hebben, kunnen ze beter garanderen dat dit ook goed uitgevoerd wordt. Een nadeel van microservices is dat het steeds over het netwerk loopt. Er kan hierdoor vertraging zijn, maar er kan ook iets fout lopen over het netwerk waardoor er mogelijk informatie verloren gaat. Het is nooit de bedoeling dat informatie verloren gaat natuurlijk. Je kan hiervoor maatregelen nemen zodat de kans dat dit gebeurt minimaal is. Een ander nadeel is dat als je veel microservices hebt en die moeten allemaal met elkaar communiceren, dan mag er niets mis zijn met het netwerk om elkaar te vinden. Dit probleem kan opgelost worden door service discovery. Dit is het vinden van de netwerk locatie van een ander microservice. Indien een service een dynamische ip-adres geeft, is het niet de bedoeling dat twee services niet meer met elkaar kunnen communiceren indien een ip-adres verandert. Indien je meer wilt weten over de werking van service discovery dan verwijst ik u graag door naar het artikel van Xu (2019).

De vier grootste voordelen van microservices zijn:

- Schaalbaarheid
- Beperken complexiteit
- Verkorten time-to-market
- Autonomie van ontwikkelteams: je kan met verschillende teams werken, je kan de beste technologie kiezen, afhankelijk van het probleem dat je wilt oplossen.

Kafka, *RabbitMq* en *Google Pub/Sub* zijn technologieën die gebruikt kunnen worden om informatie te verwerken in microservices. Hieronder verklaren we hoe deze verschillende technologieën werken en waar de verschillen zitten.

(Guidi, Lanese, Mazzara & Montesi, 2017), (Velthoven, 2016) en (Xu, 2019)

2.2 Kafka

Wanneer je als bedrijf veel berichten binnen krijgt op een korte tijdsspanne, dan heb je natuurlijk een goed functionerende technologie nodig die al deze berichten kan verwerken. *Kafka* is een voorbeeld van zo een technologie. Het is dus een berichtensysteem waarbij schaalbaarheid en redundantie een grote troef zijn. Bepaalde kernwoorden zijn belangrijk om de architectuur van *Kafka* te begrijpen. Deze kernwoorden zijn: topics, producers, consumers en brokers.

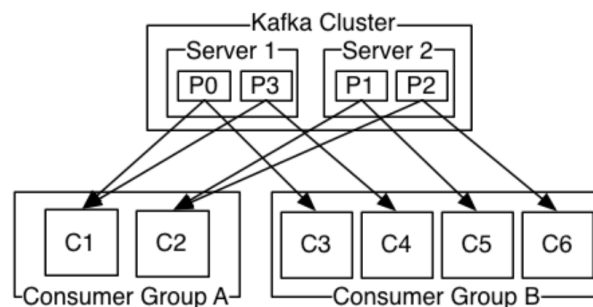
Als producer kun je een bericht verzenden naar uw gewenste topic. Topics zijn, zoals de naam al doet vermoeden, verschillende onderwerpen. Alle berichten zijn gegroepeerd in een van deze topics. Een topic kan verschillende nodes hebben, maar het kan ook zijn dat er maar één node is. Als er verschillende nodes zijn dan kan je de inkomende data repliceren. Een node kan opgedeeld zijn in verschillende partities. Als een bericht binnenkomt, dan wordt deze opgeslagen in een partitie van je topic die op zijn beurt weer op een verschillende node kan zitten.

Een topic kan je dus verdelen in verschillende partities. Dit wil zeggen dat je uw data kunt verdelen in ongeveer gelijke groepen. Iedere partitie kan dan staan voor een specifieke groep data binnen een topic zodat je niet alle berichten altijd moet overlopen. Dit principe van verschillende partities wordt ook gebruikt bij traditionele databanken. Het is dus geen garantie dat het effectief gelijke groepen zijn. De bestemming van een bericht hangt af van de message-key. Als je bijvoorbeeld woorden wilt groeperen en je verdeelt je groepen volgens de eerste letter van ieder woord. Dan staan alle woorden die beginnen met een a samen, ieder woord dat begint met een b, ... Als je dan toevallig veel woorden binnen krijgt die beginnen met een c, dan kan het gebeuren dat je groepen niet gelijk verdeeld zijn.

Een bericht heeft een bepaalde retention. Dit is de tijd dat het bericht maximum beschikbaar is. Als bijvoorbeeld de retention op zeven dagen staat, en je binnen deze zeven dagen het bericht niet gelezen hebt, dan wordt het bericht verwijderd. Dan is het niet meer mogelijk om de inhoud van het bericht nog ooit op te vragen. Het formaat van een bericht kan verschillend zijn. Het type kan een gewone tekst zijn, kan van een Json-formaat zijn, of

kan iets helemaal anders zijn. Dit waren allemaal voorbeelden van leesbare data, maar het is ook mogelijk dat het niet leesbaar is. Het formaat kan ook binair zijn, Avro is hier een voorbeeld van. Het is mogelijk om zowel naar een specifieke topic een bericht te verzenden als te ontvangen.

Dit brengt ons naadloos bij het volgende begrip: consumers. Een consumer kan zelf bepalen van welke topic hij berichten wilt ontvangen. Consumers binnen een consumer-groep worden automatisch enkele partities toegewezen. Door topics en consumers op deze manier op te delen, zorg je ervoor dat het mogelijk is dat meerdere consumers kunnen lezen van meerdere partities in een topic. Dit heeft als positief gevolg dat je meer berichten kunt verwerken binnen een bepaalde tijd. Het is al even vermeld, maar wat is nu eigenlijk een consumer-groep? Dit is een verzameling van verschillende consumers. Iedere consumer op zich leest van één specifieke partitie waardoor je het aantal verwerkte berichten binnen één tijdseenheid kunt verhogen. Alle consumers binnen één groep lezen samen alle berichten die op een topic staan. Het opdelen van je topic in verschillende partities zorgt er dus niet voor dat je een deel van je data verliest. Mochten er meer consumers zijn dan dat er partities zijn, dan zitten er sommigen zonder werk. Omgekeerd, als er meer partities zijn dan consumers, dan krijgen consumers van verschillende partities berichten binnen. De hoeveelheid consumers binnen een groep is afhankelijk van de hoeveelheid partities die er zijn.

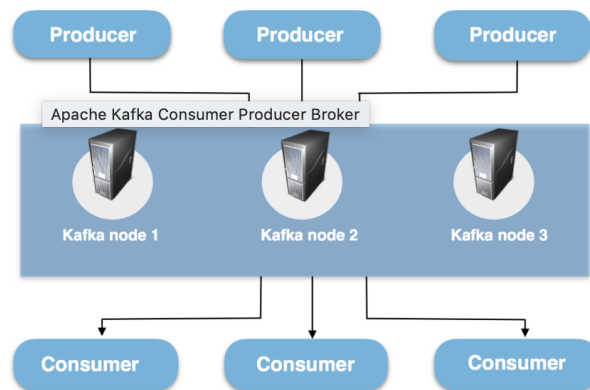


Figuur 2.2: Wisselwerking tussen partities en consumer-groepen, (Sookocheff, 2015)

Figuur 2.2 is een voorbeeld hoe een topic in verschillende servers kan opgedeeld worden en hoe consumers in consumer-groepen kunnen onderverdeeld worden. Je ziet dat de Kafka cluster in twee servers onderverdeeld is. Iedere server heeft twee partities. Er zijn 2 consumer-groepen, groep A bestaat uit twee consumers, groep B bestaat uit 4 consumers. Iedere partitie kan dus naar verschillende consumer-groepen berichten versturen, maar kan niet binnen dezelfde consumer-groep naar een andere consumer iets verzenden.

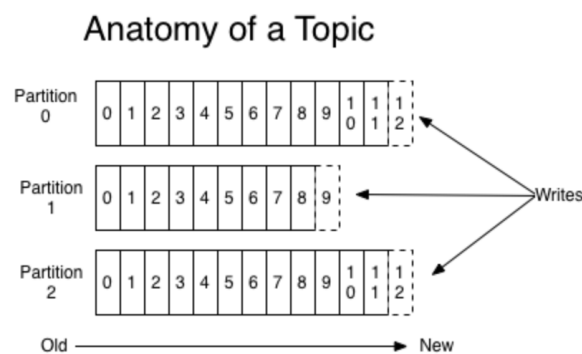
Het laatste woord dat nog moet verduidelijkt worden is een broker. *Kafka* draait op een cluster. Iedere cluster bestaat uit één of meerdere nodes(servers). Per Kafka cluster kunnen er verschillende producers en consumers zijn, zoals te zien is op figuur 2.3.

Ieder bericht binnen een partitie heeft een offset. Dit is een identifier, waardoor het mogelijk is om de berichten te ordenen. Normaal gezien als je als consumer een subscriptie maakt op een topic, dan krijg je vanaf dit moment alle nieuwe berichten die binnen komen op de partitie waarop je een subscriptie hebt. Door een offset is het mogelijk om iets oudere



Figuur 2.3: Voorbeeld van een Kafka cluster, (Johansson, 2016)

berichten die op een partitie staan dan het moment dat je een subscriptie aangemaakt hebt, ook op te vragen. Op figuur 2.4 zie je een simpele voorstelling van een producer die berichten op een partitie van een topic plaatst. De cijfertjes stellen de offset voor van een bericht.



Figuur 2.4: Simpele voorstelling van de werking van een offset, (Sookocheff, 2015)

(Sookocheff, 2015) en (Johansson, 2016)

2.3 RabbitMq

Een alternatief voor *Kafka* is *RabbitMq*. Dit is software waar men berichten kan plaatsen en ophalen op één of meerdere 'queues'. Ook hier heb je verschillende mogelijkheden van wat het formaat is van de berichten. Het kan zowel een gewone tekst zijn, als een Json-file.

Er zijn hier ook producers die data op de queue zetten, alsook consumers die een binding kunnen maken op een queue. Wanneer een bericht van een queue wordt gelezen, dan wordt deze verwijderd van de queue. Het is dus niet mogelijk om later opnieuw hetzelfde bericht op te gaan vragen.

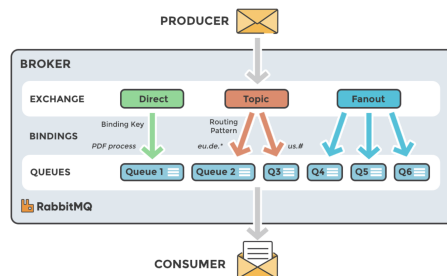
RabbitMq kan een voordeel zijn om te gebruiken in verschillende situaties. Als je bijvoor-

beeld als gebruiker de data die je opslaat in een queue wilt distribueren naar verschillende consumers, dan is ook *RabbitMQ* een ideale technologie om dit te doen. Het is dus mogelijk dat verschillende consumers de berichten lezen. Een ander voordeel is dat het mogelijk is om je berichten te verdelen over verschillende consumers. Op deze manier wordt dan de hoeveelheid mooi gebalanceerd verdeeld over de verschillende consumers.

Een broker bij *RabbitMQ* is de exchange en de queues. De exchange is verantwoordelijk voor het verzenden van de berichten naar de verschillende queues. Indien een bericht in de broker toekomt, moet deze dus eerst langs de exchange voordat hij in een queue terecht kan. De link tussen de queue en de exchange wordt ook wel eens een binding genoemd.

Er zijn vier verschillende soorten van exchanges. Dit zijn: direct, fanout, topic en headers exchanges. Drie van deze worden uitgebeeld in figuur 2.5

- De direct exchanges sturen berichten naar een queue op basis van een routing key. Deze key is hetzelfde als de binding key van de queue waarnaar het naartoe moet verzenden.
- Fanout exchanges sturen berichten naar alle queues die verbonden zijn met de exchange
- Bij Topic exchanges worden wildcard matches gebruikt. De match wordt gedaan tussen de routing key en de routing pattern die in de binding gespecificeerd wordt.
- Header exchanges gebruiken de attributen uit de header om te bepalen welke queue de bestemming is van een bericht.



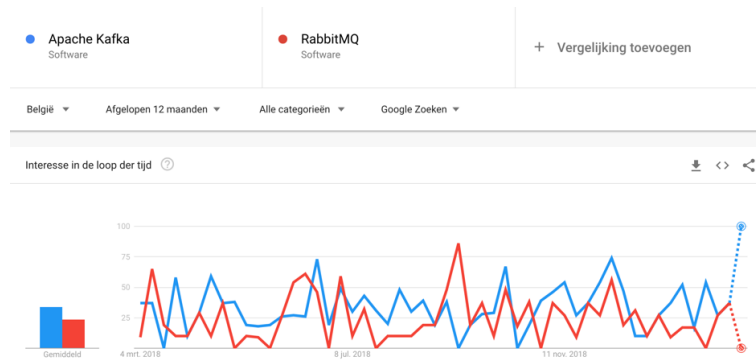
Figuur 2.5: Werking drie verschillende exchanges bij *RabbitMQ*, (Johansson, 2015)

(Johansson, 2015)

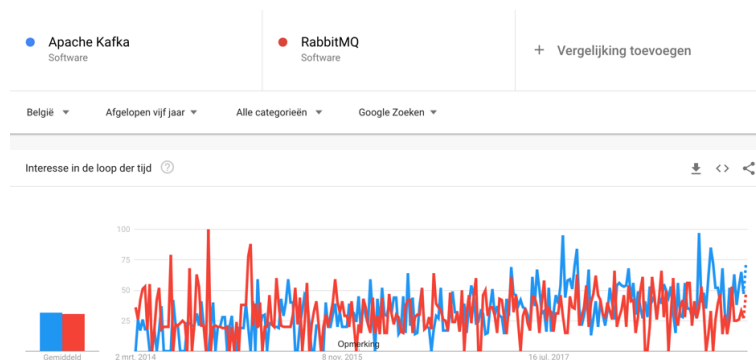
2.4 Kafka vs RabbitMQ

Als we kijken naar Google Trends om te bepalen welke van deze twee technologieën nu het populairste is, dan krijgen we in figuur 2.6 een grafiek te zien van het afgelopen jaar.

In deze grafiek is duidelijk te zien dat *Kafka* iets populairder is dan *RabbitMQ*. Er zit tussen juli en november wel een piek in waarbij *RabbitMQ* veel populairder is, maar over het algemeen gezien kunnen we concluderen dat in het afgelopen jaar *Kafka* toch iet wat populairder is.



Figuur 2.6: Populariteit *Kafka* en *RabbitMq* in het afgelopen jaar, (Trends, 2019)



Figuur 2.7: Populariteit *Kafka* en *RabbitMq* in de afgelopen vijf jaar, (Trends, 2019)

Als je figuur 2.7 bekijkt, dan zie je een algemene trend. Namelijk dat vijf jaar geleden *RabbitMq* veel populairder was, maar dat deze trend systematisch omgedraaid is. Uit deze figuur valt ook af te leiden dat in het algemeen de interesse naar technologieën voor microservices lichtjes toegenomen is in de laatste vijf jaar.

(Trends, 2019)

2.5 Google Pub/Sub

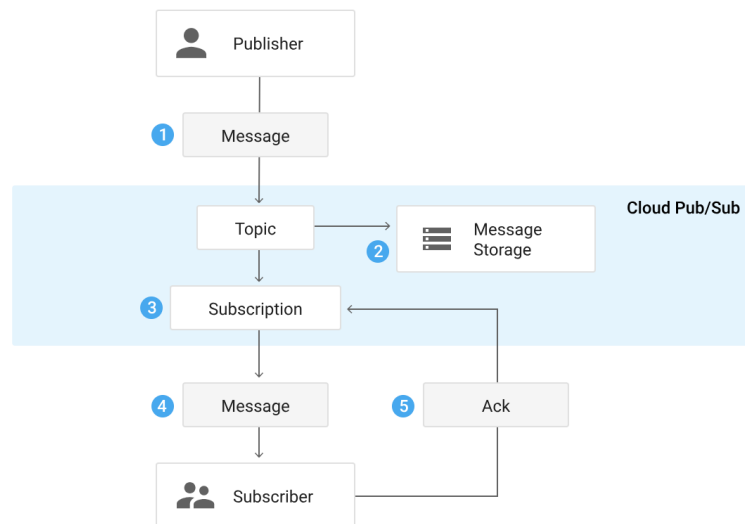
Tijdens het academiejaar dat dit onderzoek is uitgevoerd, zijn de gebruikte technologieën die TVH gebruikt ook wat veranderd. Er is momenteel een nieuwe technologie bij gekomen, namelijk *Google Pub/Sub*. Zoals de naam doet vermoeden is Google eigenaar van deze technologie.

Ook hier zijn het ongeveer dezelfde kernwoorden die belangrijk zijn:

- Topic
- Subscription
- Message

- Publisher
- Subscriber

De termen topic en message zijn hetzelfde als bij de andere technologieën. Subscription is de link tussen de subscriber (deze wordt straks uitgelegd), en de topic. Dit is dus een soort van abonnement die een subscriber aangaat met een topic. Een subscriber is iemand of iets die de verschillende messages van een specifieke topic leest. Een publisher is dan degene die nieuwe messages op een topic plaatst.



Figuur 2.8: De flow die een message aflegt bij *Google Pub/Sub*, (Google, 2019)

Op figuur 2.8 is te zien hoe een message tot bij een subscriber geraakt, iemand of iets die de message leest. Eerste stap is de publisher die een message plaatst (published) naar een topic. Een message bevat een payload, dit is de inhoud van de message. Ook kan er eventueel attributen toegevoegd worden die iets meer vertellen over de inhoud van de payload. Vervolgens wordt in de tweede stap de message opgeslagen in de Message Storage totdat een subscriber de message leest en acknowledged. Dit wil zeggen dat de subscriber een bevestiging stuurt naar de subscription om te melden dat hij de message goed ontvangen heeft. Hierna zet de Pub/Sub deze message klaar in al zijn subscriptions. Deze message wordt dan gelezen als bijvoorbeeld de subscriber deze message binnen trekt. In de vierde stap zien we dat alle berichten die nog niet gelezen zijn door de subscriber, bij de subscriber binnen komen. Als een message goed is aangekomen, dan wordt er een acknowledge gestuurd naar de subscription. Dan wordt deze message ook verwijderd en kan deze niet meer opnieuw gelezen worden. Dit is de vijfde stap.

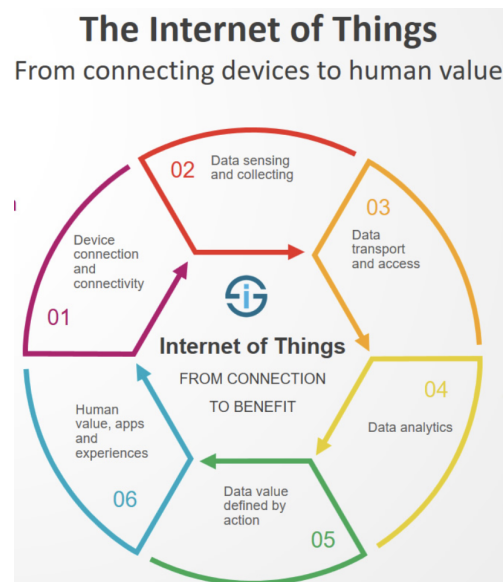
(Google, 2019)

2.6 IoT-applicatie

Microservices architectuur wordt meestal gebruikt in een IoT-applicatie. Bij microservices is het meestal zo dat er veel data is. En bij IoT wordt er veel data verzonden. Voluit staat

IoT voor Internet of Things. Onder deze noemer verstaan we machines en alledaagse objecten die verbonden staan met het internet en eventueel ook met elkaar.

Iedereen weet wat de voordelen zijn van internet op toestellen zoals een gsm, tablet, ... Laten we naar het voorbeeld van een gsm bekijken. Als je vergelijkt wat een gsm allemaal kon toen er nog geen internet mogelijk was en nu, dan zie je dat het eigenlijk een ander toestel geworden is met veel meer functionaliteiten. Dit is ook zo bij alledaagse objecten. Met het toevoegen van internet op deze objecten verhoog je de functionaliteit ervan. We zullen dit even illustreren met een voorbeeldje. Neem nu een lamp in je woonkamer. Dankzij IoT ben je niet meer verplicht om manueel een schakelaar die fysiek aanwezig is in je woonkamer om te draaien. Door de lamp te verbinden met het internet kan je die ook vanop afstand aan of uit schakelen via een app op je smartphone of iets dergelijks. Dit geeft extra voordelen aan het product. Als we ons eigen voorbeeldje nog een bekijken, kan dit handig zijn als je vergeten het licht te doven bent indien je bijvoorbeeld naar je werk vertrokken bent. Dan neem je gewoon je smartphone en doof je het licht vanop afstand. Zonder IoT was dit allemaal niet mogelijk geweest.



Figuur 2.9: De gecreëerde waarde van IoT, (i-scoop, 2019)

Op figuur 2.9 kan je de flow zien hoe Internet of Things extra waarde oplevert voor een bedrijf. In de eerste stap zorg je ervoor dat je object waar je informatie uit wilt halen verbonden is met het internet. Vervolgens worden er metingen en waarnemingen gedaan zodat er iets is dat het toestel kan verzenden. Nadien wordt de data natuurlijk effectief verzonden. Als vierde stap kan je eventueel analyses doen op je verkregen informatie zodat je nog extra zaken kunt concluderen. In de volgende stap moet je natuurlijk waarde hechten aan de informatie die binnen komt. Want natuurlijk weet het apparaat dat informatie verstuurt niet waarvoor een bepaald cijfer staat. De bedoeling in deze stap is dus bepaalde waarde voor het bedrijf toevoegen aan de verkregen getallen of waarden. Als laatste stap kan je conclusies trekken en eventueel zelf als bedrijf mogelijke acties ondernemen. Deze 6 stappen illustreren hoe je door het toevoegen van connectiviteit extra waarde voor een bedrijf kunt realiseren.

Het is nu niet zo dat alleen alles wat je van op afstand aan of uit kunt schakelen onder de noemer IoT valt. Er zijn voornamelijk drie categorieën waarin we Internet of Things objecten onderscheiden.

- Zaken die informatie verzamelen en verzenden.
- Zaken die informatie verzamelen en daaropvolgend een actie uitvoeren.
- Zaken die de twee bovenstaande combineren.

Het bedrijf TVH gebruikt voornamelijk de eerste categorie. Dit wordt in 2.7 uitgelegd.

2.6.1 Informatie verzamelen en verzenden

Bij deze categorie komen er sensoren bij kijken. Sensoren kunnen van verschillende type zijn: bewegings-, temperatuur- of lichtsensoren bijvoorbeeld. Alle mogelijke types van sensoren kunnen hiervoor gebruikt worden. Deze sensoren zijn verbonden met het internet en nemen bepaalde waarnemingen aan van de omgeving. Deze waarnemingen zijn afhankelijk van het type sensor. Bijvoorbeeld bij een temperatuursensor zal deze om een bepaalde tijd de temperatuur van zijn locatie verzenden. De informatie die sensoren verzenden kunnen van grote waarde zijn voor bedrijven. Hierdoor kunnen ze bepaalde beslissingen nemen die mogelijk tot meer winst of andere voordelen leiden.

2.6.2 Informatie verzamelen en reageren

De voorbeelden van deze categorie liggen dichterbij de hand dan je zou denken. Er zijn tal van voorwerpen die dagelijks gebruikt worden die informatie verzamelen en erop reageren. Enkele voorbeelden zijn: een printer die een document ontvangt en uitprint of een autosleutel die je auto open maakt of sluit wanneer je op het juiste knopje duwt.

Deze voorbeelden zorgen niet meteen voor een business-waarde om bijvoorbeeld de winst van een bedrijf te verhogen. Dit wilt niet zeggen dat deze zaken daarom minder interessant zijn.

2.6.3 Combinatie

Door deze twee categorieën te combineren, krijgen we nieuwe mogelijkheden. Door deze combinatie kun je een voortdurende wisselwerking creëren tussen acties die iemand neemt en de reacties die het object maakt. Laten we een fictief voorbeeldje verzinnen. Stel je wilt een cilindervormig voorwerp vullen met water. Het voorwerp kan 100 liter water bevatten. De bedoeling is dat het water altijd tussen de 80 en 90 liter moet blijven. Dan kan je een sensor installeren die detecteert of het water onder de 80 liter zit. Indien de sensor dit detecteert, moet hij automatisch het voorwerp bijvullen. De machine stopt met het object te vullen wanneer de sensor detecteert dat hij aan 89 liter zit. Hierop voert hij een actie uit om geen water meer bij te vullen. Op deze manier is er een continue werking van het systeem door het verzamelen van informatie en een actie uit te voeren aan de hand

van de ontvangen informatie. Voor veel bedrijven kan dit handig zijn om processen te optimaliseren, want er moeten in principe geen mensen meer aan de pas komen. Vroeger zou in dit voorbeeldje een mens voortdurend handmatig controleren of het water niet onder de 80 liter zat, daarna zelf het voorwerp bijvullen. Hierbij was er nog een extra risico dat de persoon te laat ontdekt dat er meer dan 90 liter water zou zijn. Dit zou erge gevolgen kunnen hebben voor een bedrijf.

(McClelland, 2019) en (i-scoop, 2019)

2.7 TVH

Aan ieder werktuig hangt een tracking-device. Met dit toestel is het mogelijk om de locatie van al hun toestellen die ze verhuren te lokaliseren. Maar deze apparaten verzenden niet alleen hun locatie, ze verzenden ook veel andere nuttige informatie. Ze houden bijvoorbeeld bij hoeveel werkuren een machine al bezig is. Of ze sturen door als de motor van de machine aanligt, de machine aan het bewegen is, ... Kort samengevat: er komt heel veel informatie binnen uit deze toestellen en daar zit heel veel handige informatie in voor het bedrijf. Zo kunnen ze bijvoorbeeld na een tijdje te weten komen wat de tijd is tot dat een onderdeel van een machine vervangen moet worden. Of ze kunnen bedrijven die machines huren controleren dat ze effectief alleen de werktuigen gebruiken tijdens de werkweek, en niet tijdens het weekend.

Laten we nog even figuur 2.9 erbij nemen. We hebben al algemeen uitgelegd wat deze afbeelding voorstelt. Nu gaan we dit even toepassen op het bedrijf TVH. In de eerste stap worden de tracking-devices aan de werktuigen gekoppeld. Nadien verzamelen deze devices informatie over de machines. Deze informatie wordt dan in stap drie verzonden. Een voorbeeld van analyse op de informatie die binnenkomt binnen het bedrijf is controleren als het tracking-device dat informatie verzendt wel gekoppeld is aan een werktuig. In stap vijf koppel je de cijfers aan specifieke waarden. Bijvoorbeeld input 2 geeft 0 of 1 terug. Dit kan betekenen dat een motor van een machine al dan niet aanligt. In de laatste stap kan het bedrijf effectief acties ondernemen uit de conclusies die getrokken kunnen worden van de verkregen informatie.

Dankzij de tracking-devices zijn alle machines gekoppeld aan het internet. Voor een IoT-applicatie die al deze informatie verzamelt heb je natuurlijk een goede technologie nodig. *Kafka*, *RabbitMQ* en *Google Pub/Sub* zijn mogelijke technologieën en zijn al reeds uitgelegd. Dit onderzoek moet uitwijzen welke technologie het beste is voor TVH om deze informatie te verzamelen.

3. Methodologie

Om te testen welke van de technologieën die in dit onderzoek aan bod komen nu eigenlijk het beste is, zou je met veel zaken rekening moeten houden. Wat het beste is hangt namelijk af van welke soort data je gebruikt. Ook de hoeveelheid transformaties die je data ondergaan voor dat het uitgelezen wordt of verzonden speelt een grote rol. Natuurlijk is het moeilijk om binnen de tijdspanne die er is voor dit onderzoek al deze factoren te gaan onderzoeken. Daarom zal dit onderzoek zich toespitsen op een bepaald scenario en daar conclusies uit trekken.

3.1 Producer project

Het opzetten van een omgeving is per technologie verschillend. Elk heeft zijn eigen manier om data te verzenden en te ontvangen. Hieronder wordt eerst eens per technologie de implementatie getoond hoe we het project opgezet hebben. Daarna wordt er uitgelegd hoe we gaan testen welke nu de beste zal zijn. De reden waarom we specifiek deze technologieën gekozen hebben om te vergelijken kan opnieuw gelezen worden in de Stand van Zaken en de Inleiding.

Natuurlijk moet er iets gelijkaardigs zijn om te kunnen vergelijken. Het data object dat we zullen verzenden en ontvangen bij zowel *Kafka*, *RabbitMQ* en *Google Pub/Sub* zal hetzelfde zijn. Op deze manier is het toekomstige resultaat niet afhankelijk van het type data. Dit onderzoek zal zich dus toespitsen op data met het type `Json`. Deze klasse (`Data.class`) hergebruiken we in elke technologie. Deze klasse is het type van het bericht dat we verzenden.

```
2 @AllArgsConstructor
3 @NoArgsConstructor
4 @ToString
5 public class Data {
6     private int id;
7     private String name;
8     private String description;
9     private Date sendedDate;
10 }
```

Deze klasse bevat vier attributen. Dit is om een Json object na te bootsen. Het is vanzelfsprekend dat deze klasse niet zo maar automatisch kan omgezet worden naar een Json. De manier waarop dit gedaan wordt is opnieuw per technologie verschillend. De uitleg zal gegeven worden in de subsectie van de technologie zelf hoe het omzetten gebeurt. De attributen die hier gebruikt worden zijn: een id van het type int, dit is om elk data object van elkaar te kunnen onderscheiden tijdens het verzenden. Als tweede zie je naam van het type String. Dit heeft als bedoeling opnieuw om verschillende objecten van elkaar te kunnen onderscheiden en om een realistische attribuut te geven aan het Data-object. Deze reden is hetzelfde voor het attribuut description van het type String. Als laatste hebben we de sendedDate van het type Date. Deze zal de datum en het tijdstip doorsturen van het moment wanneer het object verzonden is. Dit zal later nog eens aangehaald worden wanneer we deze objecten genereren.

Als u boven de klasse kijkt dan ziet u nog vier annotaties staan. Deze kunnen we gebruiken door de library van lombok. Dit werd mogelijk gemaakt door deze dependency toe te voegen aan onze pom.xml want het gebruikte project voor dit onderzoek maakt gebruik van Maven om libraries toe te voegen.

```
1 <dependency>
2 <groupId>org.projectlombok</groupId>
3 <artifactId>lombok</artifactId>
4 <optional>true</optional>
5 </dependency>
```

Deze geeft op voorhand al enkele handige implementaties. De gebruikte annotaties zullen we uitleggen, want natuurlijk bestaan er meer dan enkel deze die dit onderzoek gebruikt. Voor de andere mogelijkheden verwijs ik u graag door naar de site van Project Lombok waar u alle mogelijkheden op een rijtje ziet: <https://projectlombok.org/features/all>.

De eerste annotatie is @Getter, deze zorgt ervoor zoals de naam al doet vermoeden, dat er voor ieder attribuut een getter aanwezig is. Op deze manier staat uw code niet overvol met getters van al uw attributen. In deze klasse zijn er niet zo veel attributen dus zou het aantal getter nog meevallen. Maar u kan zich wel inbeelden dat in een groter project in een klasse met veel attributen en andere methodes dat dit een handige annotatie is. Je ziet dus niet dat de getters aanwezig zijn maar die zijn er wel effectief. Laten we de tweede en de derde samen bekijken: @AllArgsConstructor en @NoArgsConstructor. Deze zorgen ervoor dat er een constructor gegenereerd wordt met respectievelijk al de attributen en zonder de

attributen. Deze zijn vaak voorkomende constructors en zijn handig om ook hiervoor een annotatie te hebben. De laatste is de `@ToString`. Deze werd gebruikt tijdens de opmaak van het project. Dit vormt het object om in een String. Dit was handig om lokaal te testen. Het data object werd dan uitgeprint tijdens het verzenden om te controleren als het object wel de juiste waardes bevatte.

3.1.1 Kafka

In figuur 3.1 ziet u welke configuratie de topic heeft. Er wordt gebruik gemaakt van zes partities en de data wordt vier weken bijgehouden. Met andere woorden: als de data op de topic komt, en deze niet binnen de vier weken door een consumer, dan wordt de data verwijderd en kan deze data niet meer opnieuw uitgelezen worden.

MANAGEMENT > TOPICS > New topic

Availability

Settings Balanced availability/durability

Cleanup policy

Cleanup policy Delete Time to retain data 4 weeks

Max size on disk in GB 1 GB

Message size

Maximum message size in MB 1 MB

Save & create Back

TOPIC SUMMARY

name	test-batch-0
partitions	6
replication factor	3
cluster	-
min.insync.replicas	2
cleanup.policy	delete
retention.ms	2419200000
retention.bytes	1000000000
max.message.bytes	1000000
delete.retention.ms	86400000

Figuur 3.1: Configuratie van de topic

KafkaConfig

Zo ziet de implementatie van de `KafkaConfig.class` eruit.

```

1  @Configuration
2  public class KafkaConfig {
3
4      @Bean
5      @Primary
6      public KafkaProperties kafkaProperties(
7          @Value("${kafka_key}") final String kafkaKey,
8          @Value("${kafka_secret}") final String kafkaSecret
9      ) {
10
11          KafkaProperties kafkaProperties = new
12              KafkaProperties();
13
14          KafkaProperties.Producer producer =
15              kafkaProperties.getProducer();

```

```

15     producer.getProperties().put(ProducerConfig.
        KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.
        common.serialization.StringSerializer");
16     producer.getProperties().put(ProducerConfig.
        VALUE_SERIALIZER_CLASS_CONFIG, "org.
        springframework.kafka.support.serializer.
        JsonSerializer");
17
18     producer.getProperties().put(ProducerConfig.
        BOOTSTRAP_SERVERS_CONFIG, "pkc-412nj.europe-
        west1.gcp.confluent.cloud:9092");
19     producer.getProperties().put(ProducerConfig.
        RETRIES_CONFIG, "0");
20     producer.getProperties().put(ProducerConfig.
        ACKS_CONFIG, "all");
21     producer.getProperties().put(ProducerConfig.
        RETRY_BACKOFF_MS_CONFIG, "1000");
22     producer.getProperties().put(ProducerConfig.
        REQUEST_TIMEOUT_MS_CONFIG, "30000");
23     producer.getProperties().put(ProducerConfig.
        LINGER_MS_CONFIG, "200");
24     kafkaProperties.getProperties().put(SslConfigs.
        SSL_ENDPOINT_IDENTIFICATION_ALGORITHM_CONFIG, "
        https");
25     kafkaProperties.getProperties().put(SaslConfigs.
        SASL_MECHANISM, "PLAIN");
26     kafkaProperties.getProperties().put("request.
        timeout.ms", "20000");
27     kafkaProperties.getProperties().put("retry.backoff
        .ms", "500");
28     kafkaProperties.getProperties().put(SaslConfigs.
        SASL_JAAS_CONFIG, "org.apache.kafka.common.
        security.plain.PlainLoginModule required
        username=\"\"
29     + kafkaKey + "\" password=\"\" + kafkaSecret + "\"";
        ");
30     kafkaProperties.getProperties().put("security.
        protocol", "SASL_SSL");
31
32     kafkaProperties.getProperties().put(JsonSerializer
        .ADD_TYPE_INFO_HEADERS, "false");
33
34     return kafkaProperties;
35 }

```

In deze klasse ziet u dat het instellen van de nodige properties in de `kafkaProperties` methode van deze klasse gebeurt. Deze zouden ook in de `application.yml` file kunnen ingesteld worden. In dit project van het onderzoek hebben we besloten om dit in een config

klasse te doen. Tijdens het opzetten van het project kon de applicatie de properties niet uitlezen vanuit de application.yml file. Om geen tijd te verliezen hebben we besloten om deze properties in een config klasse in te stellen. Dit verandert niets van functionaliteit of aan resultaten van dit onderzoek. Maar als onderzoeker was het mogelijk om sneller aan de slag te gaan met het belangrijkere werk.

Boven de klasse ziet u de annotatie `@Configuration`. Dit is een annotatie van het Spring framework. Deze zorgt ervoor dat je aangeeft dat deze klasse een configuratie instelt. In de klasse ziet u boven de methode `kafkaProperties` twee andere annotaties staan. `@Bean` zorgt ervoor dat er een bean aangemaakt wordt voor deze methode. Dit zorgt ervoor dat door het Spring framework voor deze methode eerder een instantie aangemaakt wordt en daardoor de applicatie bij het opstarten deze properties kan instellen. Daaronder zie je een andere annotatie die hiermee te maken heeft: `@Primary`. Deze zorgt ervoor als het Spring framework meerdere beans heeft van `KafkaProperties`, dat hij deze voorneemt op al de andere. Op deze manier zijn we zeker dat deze properties gebruikt worden en niet andere indien er meerdere beans zouden zijn.

U ziet dat er tamelijk wat properties geconfigureerd zijn. Er zullen slechts enkele properties besproken worden, uiteraard zijn dit de belangrijkste voor dit onderzoek. In dit hoofdstuk werd al eens vermeld dat het object van de klasse `Data` niet zomaar omgezet wordt naar een `Json`. Hiervoor is er een serializer nodig die dit doet. Op lijn 16 van de `KafkaConfig` klasse is er te zien dat de gekozen serializer ingesteld wordt. In dit geval wordt de `JsonSerializer` gebruikt uit de package `springframework.kafka.support.serializer`. Dit is dus een specifieke `JsonSerializer` voor `Kafka`. De key wordt dan weer geserializeerd naar een `String` door de `StringSerializer` uit de package `org.apache.kafka.common.serialization`, dit is te zien op de regel er boven. Wat als het niet lukt voor de producer om de data te verzenden? In ons geval mag hij dit niet opnieuw proberen. Op deze manier kunnen we ook te weten komen hoeveel data er mislukt is om te verzenden. Dit zie je op lijn 19. Indien je meer zekerheid wilt dat je een data object zeker verzonden wordt, dan kan je deze property in plaats van 0 op bijvoorbeeld 3 zetten. Je mag zeker ook niet vergeten om de username en password mee te geven van waar je `Kafka` topic zich bevindt. Op lijn 28 en 29 zie je dat dit meegegeven wordt. De username en wachtwoord worden niet in plaintext in de code geplaatst. Het is niet de bedoeling dat iedereen weet wat het wachtwoord en username is. Deze worden ingesteld in uw `application.yml` en worden meegegeven via de parameters van de methode door het spring framework. Door de annotatie `@Value` en de naam van de key van uw waarden in de `application.yml`, weet spring perfect waar hij het wachtwoord en username moet gaan halen.

KafkaPublisher

De `KafkaPublisher.class` is de effectieve producer van `Data` objecten. Veel moet er niet ingesteld worden om deze producer te doen laten werken. Wat je zeker moet hebben is een `KafkaTemplate` en de naam van de topic waarnaar je verzend. Laat ons even deze klasse stap voor stap bekijken.

Boven de klasse staat de annotatie `@Component`. Dit is opnieuw een annotatie van het

spring framework. Deze zorgt ervoor dat je via constructor- of setterinjectie in andere klassen telkens dezelfde instantie van de klasse `KafkaPublisher` kunt gebruiken. Op die manier moet je niet telkens een nieuwe instantie maken van de klasse. Er zijn twee attributen: de `kafkaTemplate` van het type `KafkaTemplate` en een `String` `topicName`; De `KafkaTemplate` is een template die je gebruikt om je message naar een Kafka topic te verzenden.

In de constructor worden `kafkaTemplate` en `topicName` meegegeven als parameters. Spring Boot zorgt ervoor dat de `kafkaTemplate` automatisch gecreëerd wordt. De `topicName` willen we niet zo maar in onze code schrijven, dus injecteren we die vanuit de `application.yml`. Hiervoor wordt opnieuw `@Value` gebruikt, maar dit principe werd al eens uitgelegd.

Als laatste bevat deze klasse ook nog de `publish` methode. Deze verzend een message effectief naar de topic. In onze implementatie krijgt deze methode een lijst van `Data` objecten mee. In dit onderzoek is het niet van toepassing om één enkel `Data` object te verzenden, dus om praktische redenen wordt er een lijst meegegeven. In deze methode wordt deze lijst overlopen en telkens worden dezelfde operaties uitgevoerd. Er wordt een object aangemaakt van de klasse `Message`, zoals de naam al doet vermoeden wordt dit onze message. In de payload van deze message stoppen we ons `Data` object. In de header geven we twee zaken mee. Als message key geven we het id mee, omdat in deze implementatie het id toch telkens uniek zal zijn. Als tweede moeten we nog de naam van onze topic meegeven zodat de `kafkaTemplate` weet naar waar hij moet versturen. Het enigste dat deze methode dan nog moet doen is de `kafkaTemplate` gebruiken om de message te verzenden. Het omzetten van de message in een json formaat wordt automatisch gedaan omdat we in de properties een serializer opgegeven hebben. Ook de message key wordt voor dezelfde reden automatisch geserialiseerd.

```
1      @Component
2      public class KafkaPublisher {
3
4          private KafkaTemplate kafkaTemplate;
5          private String topicName;
6
7          public KafkaPublisher(KafkaTemplate kafkaTemplate,
8              @Value("${kafka.topic.name}") String topicName) {
9              this.kafkaTemplate = kafkaTemplate;
10             this.topicName = topicName;
11         }
12
13         public void publish(List<Data> dataList) {
14             for (Data data : dataList) {
15                 Message<Data> message = MessageBuilder.
16                     withPayload(data)
17                     .setHeader(KafkaHeaders.MESSAGE_KEY, String.
18                         valueOf(data.getId()))
19                     .setHeader(KafkaHeaders.TOPIC, topicName)
20                     .build();
21
22                 kafkaTemplate.send(message);
23             }
24         }
25     }
```

```
21     }  
22   }  
23 }
```

3.1.2 RabbitMq

RabbitMqConfig

In dit onderzoek zijn alle configuraties van RabbitMq gedaan in application.yml.

```
1 spring:  
2   rabbitmq:  
3     host: "hostaddress"  
4     port: 16379  
5     username: ${rabbitmq_username}  
6     password: ${rabbitmq_password}  
7     virtual-host: "rabbitmq"  
8     ssl:  
9       enabled: true  
10      algorithm: "TLSv1.2"  
11     listener:  
12       simple:  
13         default-requeue-rejected: false  
14     direct:  
15       default-requeue-rejected: false
```

Veel configuraties zijn er niet. De belangrijkste zijn host, port, username en password. De host is het adres waar uw RabbitMq exchange op draait met de juiste poort die opgegeven wordt in port. In het project is de String "hostaddress" vervangen door de effectieve host. Om privacy redenen is hier dit adres gewijzigd. Username en password worden ook nog bijgehouden in een variabele maar deze staan natuurlijk niet op het codevoorbeeld. Hiernaast wordt ook nog de naam van de exchange bijgehouden die we later in de implementatie nodig zullen hebben. Uiteraard zit deze property ook niet in het codevoorbeeld.

RabbitMqPublisher

Over het algemeen gezien kun je deze klasse goed vergelijken met de KafkaPublisher klasse. Het grootste verschil zit men in de publish methode. De annotatie @Component boven de klasse heeft hetzelfde doel als bij de KafkaPublisher. Er is hier één attribuut meer en dat is de ObjectMapper. Deze wordt gebruikt om later in de publish methode onze data om te zetten naar een String. Uiteraard is de KafkaTemplate hier vervangen door een RabbitTemplate. Deze hebben dezelfde functionaliteit maar bij een verschillende technologie. Het is dus ook een template die je gebruikt om de data te verzenden naar de exchange. Hier heb je niet de naam van de topic nodig zoals bij de KafkaPublisher maar de naam van de exchange.

In de constructor wordt de `rabbitTemplate` en de `objectMapper` opnieuw automatisch gecreëerd door Spring Boot. De naam van de exchange wordt weer via de `application.yml` file geïnjecteerd. In de `publish` methode overlopen we weer een lijst van objecten van `Data`. Bij iedere elementje uit de lijst voeren we dezelfde operaties uit. We gebruiken de `objectMapper` om ons object om te zetten naar een `String`. Het resultaat van deze `String` is in een `Json` formaat. Dit kunnen we ook verifiëren en daarom is als laatste lijn in de `for`-lus een `print`-statement toegevoegd. Deze `String` wordt uitgeprint en we kunnen mooi zien dat deze `String` een `Json` formaat heeft. Verder gebruiken we de `rabbitTemplate` om ons bericht te verzenden naar de exchange die we meegeven samen met ons bericht.

```
1  @Component
2  public class RabbitMqPublisher {
3      private RabbitTemplate rabbitTemplate;
4      private ObjectMapper objectMapper;
5      private String exchangeName;
6
7      public RabbitMqPublisher(final RabbitTemplate
8          rabbitTemplate,
9          final ObjectMapper objectMapper,
10         @Value("${rabbitmq.topic.exchange.name}") final String
11             exchangeName) {
12         this.rabbitTemplate = rabbitTemplate;
13         this.objectMapper = objectMapper;
14         this.exchangeName = exchangeName;
15     }
16
17     public void publish(final List<Data> dataList) {
18         try {
19             for (Data data : dataList) {
20                 String stringData = objectMapper.
21                     writeValueAsString(data);
22
23                 rabbitTemplate.convertAndSend(exchangeName, "
24                     ", stringData);
25                 System.out.println("sending messeage RMQ: " +
26                     stringData);
27             }
28         } catch (JsonProcessingException e) {
29             e.printStackTrace();
30         }
31     }
32 }
```

3.1.3 Google Pub/Sub

Op figuur 3.2 kunt u zien hoe de configuratie van de topic is voor de gebruikte subscription in dit onderzoek. We zien dat bij het lezen van een message die op de Pub/Sub staat, dat er een tijdslimiet staat van 60 seconden. Dit wil zeggen dat indien de message binnen de 60 seconden niet gelezen kan worden wanneer de subscriber messages aan het lezen is, dat de actie NACK uitgevoerd wordt op deze message. Dit wil zeggen dat de message nog altijd niet verwijderd is van de topic en dat deze nog altijd gelezen kan worden door de subscriber. Bij Google Pub/Sub kun je deze tijdslimiet instellen tussen 60 en 600 seconden. Onder de "Acknowledgement" Deadline ziet u "Message retention duration" staan. Dit is hoe lang een message op de topic blijft staan. In dit geval staat deze waarde op het maximum: op zeven dagen. Met andere woorden: indien een message binnen de zeven dagen niet gelezen wordt, dan zal deze verwijderd worden en is het niet meer mogelijk om deze message terug te vinden.

projects/happyfleetruleengine/subscriptions/bp-ruben-subscription

Topic

projects/happyfleetruleengine/topics/ruben-bp

Delivery Type

Pull

Subscription expiry ?

The subscription will never expire, no matter the activity.

Acknowledgement Deadline ?

60 Seconds

Message retention duration

7 Days 0 Hours 0 Minutes

Retain acknowledged messages

No

Figuur 3.2: Configuratie van de topic voor de gebruikte subscription

GooglePubSubConfig

Zo ziet de GooglePubSubConfig.class eruit. Er worden twee kanalen aangemaakt van het type MessageChannel. Deze kanalen worden gebruikt om een bericht van een topic te lezen of verzenden.

```

29 @Configuration
30 public class GooglePubSubConfig {
31     @Bean("bachelorproef-channel-input")

```

```

32     public MessageChannel createDeviceEventChannelInput() {
33         return new DirectChannel();
34     }
35
36     @Bean("bachelorproef-channel-output")
37     public MessageChannel createDeviceEventChannelOutput() {
38         return new DirectChannel();
39     }
40
41     @Bean
42     public PubSubInboundChannelAdapter
43         createDeviceChannelAdapter(
44         PubSubOperations pubSubTemplate,
45         @Value("${pubsub.subscription}") String subscriptionName,
46         @Qualifier("bachelorproef-channel-input") MessageChannel
47             inputChannel,
48         @Value("${pubsub.autostartup}") Boolean autoStart) {
49
50         final PubSubInboundChannelAdapter adapter = new
51             PubSubInboundChannelAdapter(pubSubTemplate,
52             subscriptionName);
53         adapter.setOutputChannel(inputChannel);
54         adapter.setAckMode(AckMode.AUTO);
55         adapter.setPayloadType(Event.class);
56         adapter.setAutoStartup(autoStart);
57         return adapter;
58     }
59
60     @Bean
61     @ServiceActivator(inputChannel = "bachelorproef-channel-
62         output")
63     public MessageHandler messageSender(
64         PubSubOperations pubsubTemplate,
65         @Value("${pubsub.topic}") String topicName
66     ) {
67         return new PubSubMessageHandler(pubsubTemplate,
68             topicName);
69     }
70
71     @Bean
72     public JacksonPubSubMessageConverter
73         createJacksonMessageConverter(final ObjectMapper
74             objectMapper) {
75         return new JacksonPubSubMessageConverter(objectMapper)
76             ;
77     }
78 }

```

GooglePubSubPublisher

Deze klasse heeft maar één methode, namelijk `publishData`. Deze methode verstuurd het `Data`-object naar de topic via het kanaal *bachelorproef-channel-output*. Er hoeft geen implementatie te zijn van deze methode, het spring-framework regelt al het andere werk in jouw plaats.

```
71 @MessagingGateway(defaultRequestChannel = "bachelorproef -  
    channel-output")  
72 @Component  
73 public interface GooglePubSubPublisher {  
74     void publishData(Data data);  
75  
76 }
```

3.2 Consumer project

3.3 Onderzoek

4. Conclusie

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

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

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

lacus ut lorem. Sed luctus justo sed enim.

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

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

A. Onderzoeksvoorstel

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

A.1 Introductie

Binnen TVH is er dus heel wat input van data die via microservices naar de juiste componenten verstuurd worden. Het spreekt voor zich dat niet ieder component, alle data nodig heeft. Daarom maakt dit bedrijf voornamelijk gebruik van de technologie *Kafka* om met deze microservices te werken. Dit onderzoek zal nagaan of *Kafka* inderdaad wel de beste technologie is om al deze data te verwerken in dit bedrijf. Aan de hand van deze onderzoeksvraag en deelvragen komt dit onderzoek hopelijk tot een besluit welke technologie het meest geschikt is:

- Welke technologie is het best om met microservices te werken voor het bedrijf TVH?
 - Bestaan er nog alternatieven voor *Kafka* en *RabbitMQ*?
 - Wat zijn de bevindingen van gebruikers?
 - Welke technologie is het snelst?

A.2 Literatuurstudie

In het onderzoek van Shadija, Rezai en Hill (2017) staat te lezen dat microservices de business analysts helpen om grote schaalbare applicaties te maken. Het grote voordeel hiervan is flexibiliteit. Als er nieuwe functionaliteiten moeten worden gemaakt dan is het

door de microservices gemakkelijk te implementeren. Vooral in het Internet of Things (IoT) domein kan dit het werk versoepelen. Dus voor het bedrijf TVH lijkt het de meest geschikte manier om de input van al de verzamelde data te verwerken.

Ook in andere onderzoeken naar microservices wordt *Kafka* gebruikt. Zoals in het onderzoek van Khazaei, Bannazadeh en Leon-Garcia (2017). Als we kijken naar de conclusie uit dit onderzoek, blijkt dat *Kafka* in grote lijnen het best scoort. De andere technologieën die in dit onderzoek gebruikt werden zijn *Spark* en *Cassandra*.

Het verschil van deze bachelorproef-onderzoek met het onderzoek van Shadija e.a. (2017) en het dat van Khazaei e.a. (2017) is dat dit onderzoek nagaat welke technologie het beste is voor het bedrijf TVH. Het besluit van dit onderzoek is dus niet noodzakelijk een algemeen besluit voor alle bedrijven die met microservices werken. Het onderzoek van Khazaei e.a. (2017) sluit hier het dichtst bij aan omdat het ook *Kafka* en andere technologieën vergelijkt, maar het onderzoek legt meer de nadruk hoe flexibel een programmeerbaar, zelf-besturend IoT-platform is, gebruik makend van microservices. Het vergelijken van verschillende technologieën bij Khazaei e.a. (2017) is dus maar een klein onderdeel van het onderzoek en wordt bovendien in een andere architectuur toegepast zoals de titel meedeelt: ‘*SAVI-IoT: A Self-Managing Containerized IoT Platform*’. De meerwaarde van deze vergelijking voor de conclusie is dus niet zo groot voor Khazaei e.a. (2017).

Ook Nycander (2015) en Cherradi, El Bouziri en Boulmakoul (2017) behandelen microservices in hun onderzoek.

A.3 Methodologie

Om te bepalen welke technologie het beste is bij het gebruiken van microservices, zal dit onderzoek de verschillende technologieën vergelijken. Eerst zal er een rondvraag gehouden worden over de bevindingen en de voor- en nadelen van *Kafka* en *RabbitMq*. Er wordt ook gepolst of medewerkers met nog andere technologieën reeds gewerkt hebben en wat daar de bevindingen zijn. Er zal voornamelijk gewerkt worden met open vragen waardoor er veel nieuwe nuttige informatie zal ontstaan voor dit onderzoek.

Dan zal er onderzocht worden of er effectief nog alternatieven bestaan voor *Kafka* en *RabbitMq*.

Als laatste zal op een virtuele machine de realiteit nagebootst worden. Dit wordt gedaan door elke technologie op deze virtuele machine te zetten en daarna te gaan meten wat de snelheden zijn bij het opvragen, verwerken, ... van voorbeelddata.

A.4 Verwachte resultaten

Het resultaat van dit onderzoek zal hopelijk aanwijzen welke technologie het meest geschikt is om deze hoeveelheid van data aan te kunnen. We hopen dit te zien door cijfergegevens

van de snelheden van uitvoering.

A.5 Verwachte conclusies

Aangezien TVH al gebruik maakt van *Kafka*, en ook in andere onderzoeken *Kafka* gebruikt werd of bestempeld werd als beste oplossing, kunnen we in dit onderzoek hopelijk ook concluderen dat *Kafka* de beste oplossing is om met microservices om te gaan binnen TVH met hun specifieke data.

Bibliografie

- Cherradi, G., El Bouziri, A. & Boulmakoul, A. (2017, november 1). A Generic microservice-based architecture for Smart HazMat Transportation Ecosystem.
- Google. (2019). What Is Cloud Pub/Sub?
- Guidi, C., Lanese, I., Mazzara, M. & Montesi, F. (2017). *Microservices: A Language-Based Approach*. Springer International Publishing.
- i-scoop. (2019). What is the Internet of Things?
- Johansson, L. (2015). Part 1: RabbitMQ for beginners - What is RabbitMQ?
- Johansson, L. (2016). Part 1: Apache Kafka for beginners - What is Apache Kafka?
- Khazaei, H., Bannazadeh, H. & Leon-Garcia, A. (2017, augustus 1). *SAVI-IoT: A Self-Managing Containerized IoT Platform* (masterscriptie, University of Toronto, Ontario, Canada).
- lombok. (g.d.).
- McClelland, C. (2019). What Is IoT? – A Simple Explanation of the Internet of Things.
- Nycander, P. (2015, juni 1). *Learning-Based Testing of Microservices* (masterscriptie, KTH Royal Institute of Technology, School of Computer Science en Communication (CSC)).
- Shadija, D., Rezai, M. & Hill, R. (2017, september 7). *Towards an Understanding of Microservices* (masterscriptie, Department of Computing Sheffield Hallam University, UK, School of Computing en Engineering University of Huddersfield, UK).
- Sookocheff, K. (2015). Kafka in a Nutshell.
- Trends, G. (2019). Kafka vs RabbitMq.
- Velthoven, J. K. (2016). 10 dingen die je moet weten over microservices.
- Xu, J. (2019). Service discovery in microservice architecture.