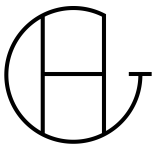


HoGent

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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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, consectetuer 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

Lijst van figuren

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 technologiën 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

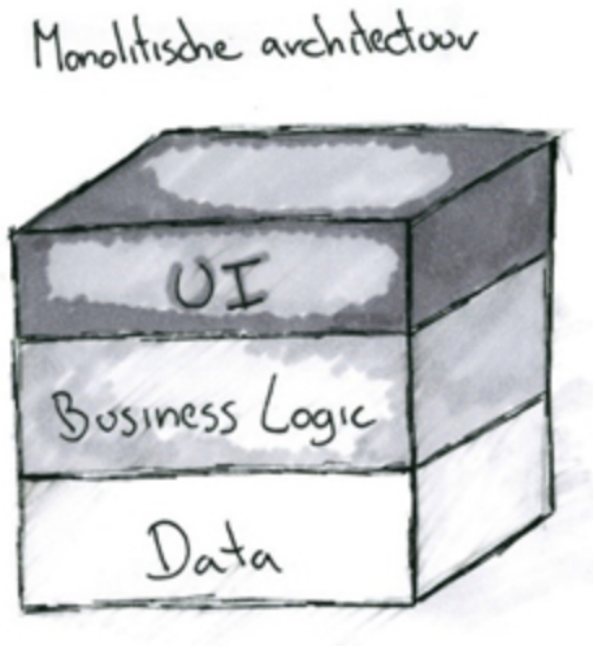
In dit hoofdstuk gaan we Kafka en RabbitMq bespreken. Want natuurlijk zijn de begrippen zoals *Kafka* en *RabbitMq* niet voor iedereen even duidelijk. Ook de term microservices zal bij sommigen de wenkbrauwen wel eens doen fronsen. In dit hoofdstuk is het de bedoeling uit te leggen wat al deze begrippen 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 datalaag. 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 mogelijks 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 verwijs 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.

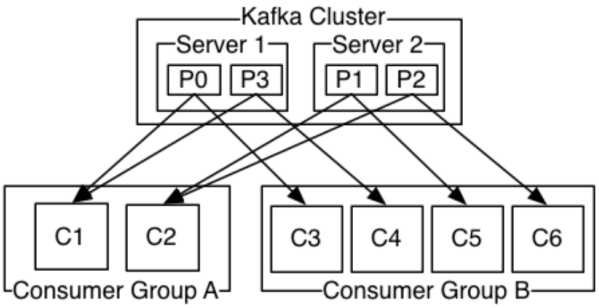
Als producer kun je een bericht verzenden naar uw gewenste topic. Topics zijn, zoals de naam al doet vermoeden, verschillende onderwerpen. Alle berichten zijn gegroepeerd in een van deze topics. Een topic kan verschillende nodes hebben, maar het kan ook zijn dat er maar één node is. Als er verschillende 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.

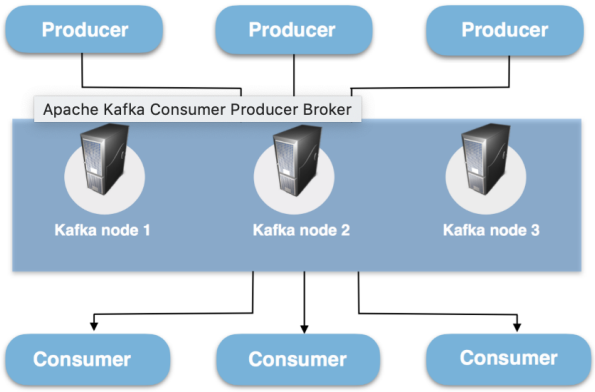


Figuur 2.2: Wisselwerking tussen partities en consumer-groepen, (Sookocheff, 2015)

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.

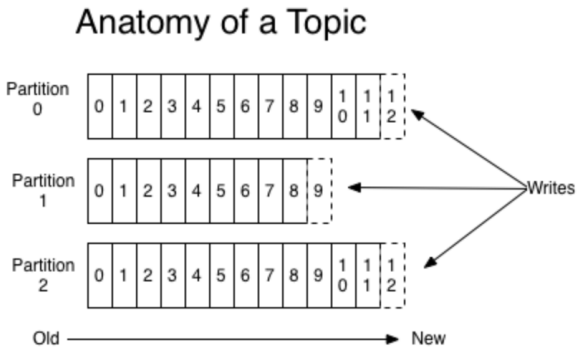
Het laatste woord dat nog moet verduidelijkt worden is een broker. *Kafka* draait op een cluster. Iedere cluster bestaat uit één of meerdere nodes(servers). Per Kafka cluster kunnen er verschillende producers en consumers zijn, zoals te zien is op figuur 2.3.

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.

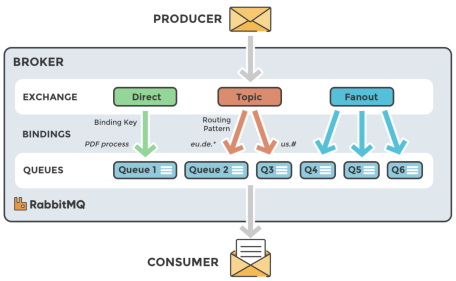
RabbitMq kan een voordeel zijn om te gebruiken in verschillende situaties. Als je bijvoor-

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.



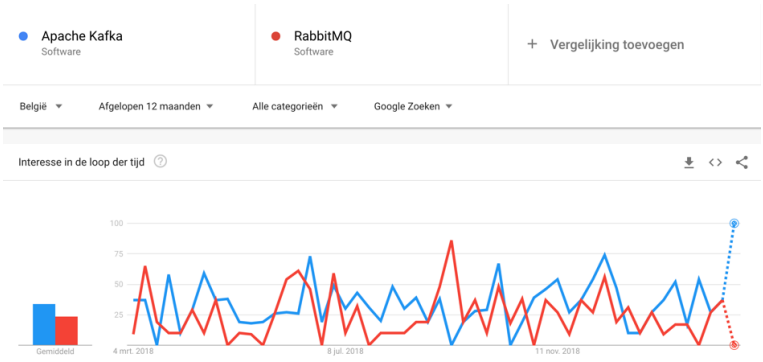
Figuur 2.5: Werking drie verschillende exchanges bij *RabbitMq*, (Johansson, 2015)

(Johansson, 2015)

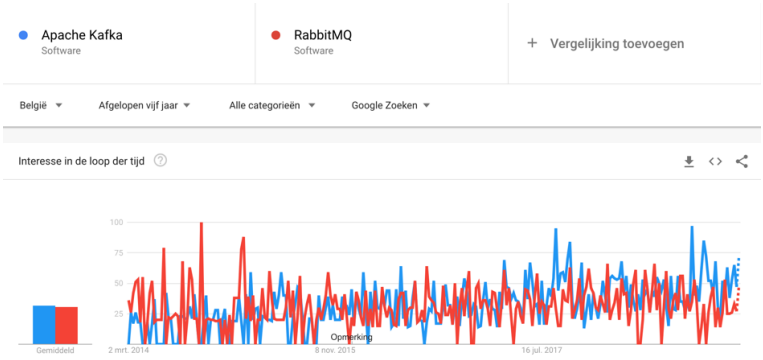
2.4 Kafka vs RabbitMq

Als we kijken naar Google Trends om te bepalen welke van deze twee technologiën 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

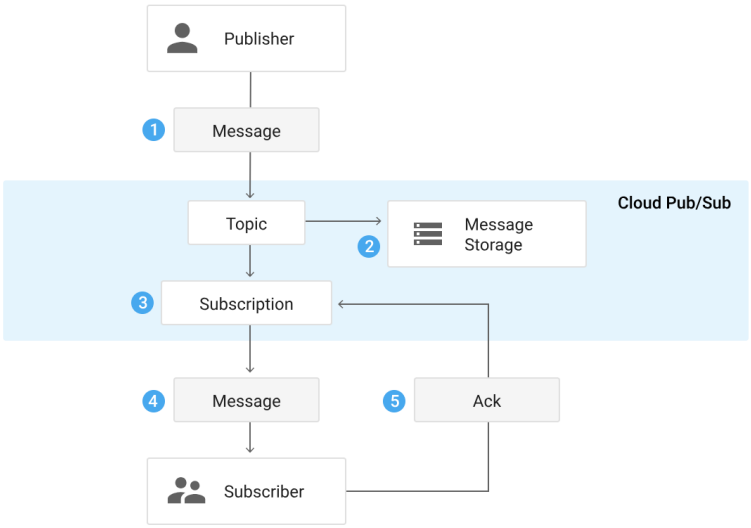
Tijdens het academiejaar dat dit onderzoek is uitgevoerd, zijn de gebruikte technologieën die TVH gebruikt ook wat veranderd. Er is momenteel een nieuwe technologie bij gekomen, namelijk *Google Pub/Sub*. Zoals de naam doet vermoeden is Google eigenaar van deze technologie.

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.

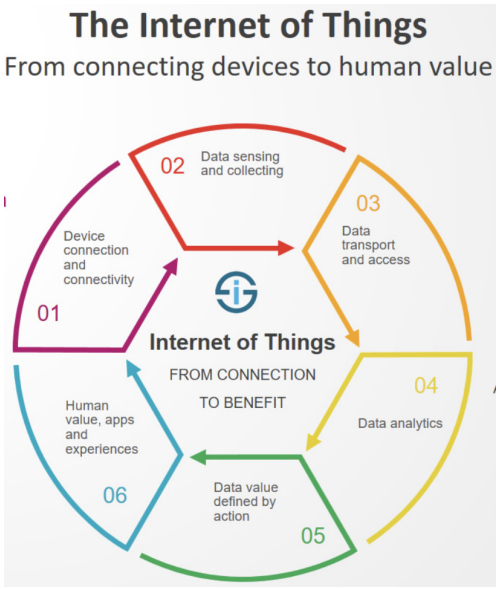
(Google, 2019)

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: bewegins-, 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 dichter bij 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.

```
1 @Getter
```

```
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class Data {
    private int id;
    private String name;
    private String description;
    private Date sendedDate;
}
```

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 gebeurd. 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.

```
<dependency>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<optional>true</optional>
</dependency>
```

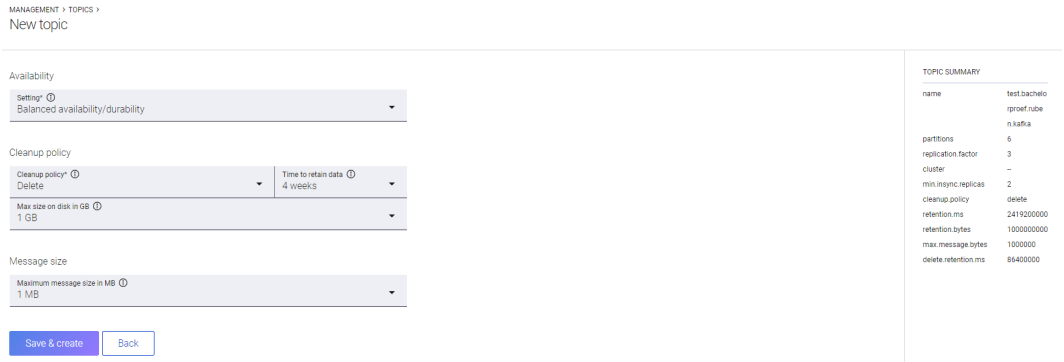
Deze geeft op voorhand al enkele handige implementaties. De gebruikte annotaties zullen we uitleggen, want natuurlijk bestaan er meer dan enkel deze die dit onderzoek gebruikt. Voor de andere mogelijkheden verwijs ik u graag door naar de site van Project Lombok waar u alle mogelijkheden op een rijtje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



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
                KafkaProperties();
12
13          KafkaProperties.Producer producer =
                kafkaProperties.getProducer();
14
```

```
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 gebeurd. 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 klasses 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 enige 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 geserializeerd.

```
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.
                   withPayload(data)
16             .setHeader(KafkaHeaders.MESSAGE_KEY, String.
                   valueOf(data.getId()))
17             .setHeader(KafkaHeaders.TOPIC, topicName)
18             .build();
19
20             kafkaTemplate.send(message);
```

```
            }
        }
    }
```

3.1.2 RabbitMq

RabbitMqConfig

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

```
spring:
    rabbitmq:
        host: "hostadress"
        port: 16379
        username: ${rabbitmq_username}
        password: ${rabbitmq_password}
        virtual-host: "rabbitmq"
        ssl:
            enabled: true
            algorithm: "TLSv1.2"
        listener:
         simple:
            default-requeue-rejected: false
        direct:
            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 ”hostadress” 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

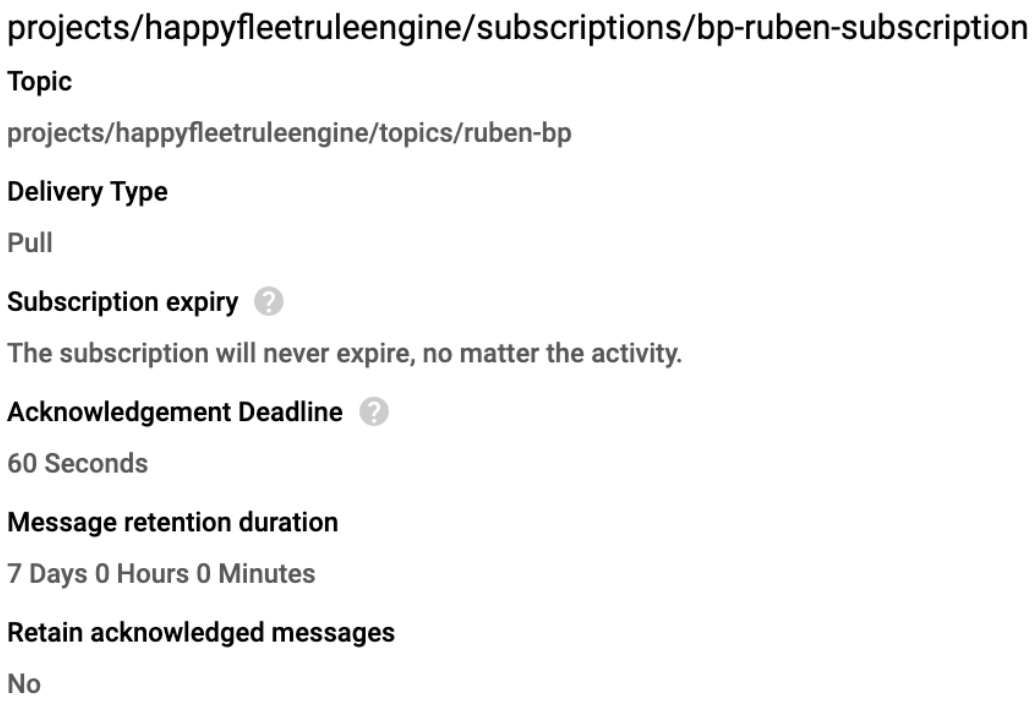
Over het algemeen gezien kun je deze klasse goed vergelijken met de KafkaPublisher klasse. Het grootste verschil zit men in de publish methode. De annotatie @Component boven de klasse heeft hetzelfde doel als bij de KafkaPublisher. Er is hier één attribuut meer en dat is de ObjectMapper. Deze wordt gebruikt 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.

```
@Component
public class RabbitMqPublisher {
    private RabbitTemplate rabbitTemplate;
    private ObjectMapper objectMapper;
    private String exchangeName;

    public RabbitMqPublisher(final RabbitTemplate
       rabbitTemplate,
    final ObjectMapper objectMapper,
    @Value("${rabbitmq.topic.exchange.name}") final String
       exchangeName) {
        this.rabbitTemplate = rabbitTemplate;
        this.objectMapper = objectMapper;
        this.exchangeName = exchangeName;
    }

    public void publish(final List<Data> dataList) {
        try {
            for (Data data : dataList) {
                String stringData = objectMapper.
                   writeValueAsString(data);

                rabbitTemplate.convertAndSend(exchangeName, "
                   ", stringData);
                System.out.println("sending messeage RMQ: " +
                    stringData);
            }

        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
    }
}
```

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 tijdlimiet 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.



Figuur 3.2: Configuratie van de topic voor de gebruikte subscription

GooglePubSubConfig

Voor Google Pub/Sub is de configuratie iets meer verschillend in vergelijking met Kafka en RabbitMq. Hier gebruiken we zowel een configuration klasse als de application.yml file. De configuratie klasse wordt aangeduid met de annotatie van Spring: @Configuration. Ook de application.yml file wordt gebruikt voor drie properties. Laat ons eerst de application.yml file overlopen. Deze wordt gebruikt om de naam van de topic in op te slaan. Want we willen ook niet hier de naam van onze topic in onze code hebben. Dan zijn er nog twee zaken die hier geconfigureert worden: spring.cloud.gcp.project.id

en spring.cloud.gcp.credentials.location. Het eerste is het project id van het cloud platform waar uw Pub/Sub zich bevindt. Het tweede is het path naar de credentials file om toegang te krijgen tot de cloud platform. Al het andere configuratie werk wordt in de GooglePubSubConfig klasse gedaan. De eerste methode createBachelorproefChannelOutput geeft een MessageChannel terug. Deze heeft een annotatie @Bean die de naam 'bachelorproef-channel-output" krijgt. Dit is letterlijk het 'kanaal' die we gaan gebruiken om onze messages te verzenden. De volgende methode geeft een MessageHandler terug en meer bepaald in dit geval een PubSubMessageHandler. Deze heeft een pubsubTemplate die je kunt vergelijken met de kafkaTemplate of rabbitTemplate. Ook wordt er verwacht dat je de naam van uw topic meegeeft. Naar deze topic zal hij de messages sturen. Zoals te zien is gebruiken we weer de @Value annotatie van het Spring framework om de naam van onze topic niet in onze code te hoeven plaatsen. Boven deze methode zie je niet alleen de annotatie @Bean maar ook @ServiceActivator met de naam van de bean uit de eerste methode. Dit zal ik zo meteen uitleggen wanneer we aan het stukje komen van de GooglePubSubPublisher interface. De derde methode returned een JacksonPubSubMessageConverter. Door deze bean aan te maken weet Spring automatisch dat hij deze klasse moet gebruiken om onze objecten om te zetten naar een Json.

```
@Configuration
public class GooglePubSubConfig {

    @Bean("bachelorproef-channel-output")
    public MessageChannel createBachelorproefChannelOutput() {
        return new DirectChannel();
    }

    @Bean
    @ServiceActivator(inputChannel = "bachelorproef-channel-
       output")
    public MessageHandler messageSender(
    PubSubOperations pubsubTemplate,
    @Value("${pubsub.topic}") String topicName
    ) {
        return new PubSubMessageHandler(pubsubTemplate,
           topicName);
    }

    @Bean
    public JacksonPubSubMessageConverter
       createJacksonMessageConverter(final ObjectMapper
       objectMapper) {
        return new JacksonPubSubMessageConverter(objectMapper)
           ;
    }

}
```

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. Maar toch is het handig om even te verduidelijken hoe dit allemaal in zijn werk gaat. Wanneer de methode publish met een Data object opgeroepen wordt, dan verzend hij deze data via het kanaal dat beschreven staat in de annotatie @MessagingGateway. Maar deze interface kan dit werk niet op zijn eentje doen en zoekt daarom hulp in de configuratie. Als we dan nu eventjes terug gaan naar de GooglePubSubConfig klasse, dan zijn we boven de methode die een MessageHandler terug geeft de @ServiceActivator staan en dezelfde naam van het kanaal. Dus om kort samen te vatten: als de methode publish opgeropen wordt, dan kijkt hij welk kanaal hij nodig geeft om te verzenden. Dan zoekt hij een MessageHandler die geactiveerd wordt wanneer zijn kanaal nodig is. En deze handler gebruikt spring om de message dan effectief te verzenden. Omdat de bean die de MessageChannel terug geeft de naam ”bachelorproef-channel-output” heeft, en deze naam van het kanaal ook in de annotatie @ServiceActivator staat, weet Spring dat dit kanaal gebruikt moet worden.

```
1 @MessagingGateway(defaultRequestChannel = "bachelorproef-
    channel-output")
2 @Component
3 public interface GooglePubSubPublisher {
4     void publishData(Data data);
5
6 }
```

3.1.4 Creëren van de data

Nu alle drie de technologieën klaar zijn om data te verzenden, blijft er voor deze applicatie alleen nog het creëren van de data en het verzenden ervan over. Er zijn nog drie cruciale elementen die besproken moeten worden en die zijn: de ProducerApplication- , de RandomDataProvider- en de RandomDataPublisher.

ProducerApplication

Dit is de eerste klasse die gebruikt wordt om de applicatie te starten. In de methode runner geven we via de parameters alle klasses mee die data publishen. Dit is dus een klasse voor Google Pub/Sub, Kafka en RabbitMq. Ook moeten we de klasse RandomDataPublisher meegeven. Deze klasses worden opnieuw gecreëerd door Spring Boot. In deze methode roep je de methode doStuff op van de RandomDataPublisher klasse. Wat deze methode exact doet bespreken we later. Maar je moet meegeven welke publisher je wilt gebruiken. Deze lijn code is het enige dat je moet aanpassen om data te verzenden voor één van de drie technologieën.

```
1 @SpringBootApplication
```

```
@Slf4j
public class ProducerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProducerApplication.class, args);
    }

    @Bean
    public CommandLineRunner runner(final RandomDataPublisher randomDataPublisher,
        final GooglePubSubPublisher googlePubSubPublisher,
        final KafkaPublisher kafkaPublisher,
        final RabbitMqPublisher rabbitMqPublisher) {
            return args -> {
              randomDataPublisher.doStuff(
                 googlePubSubPublisher);
              //   OF:          randomDataPublisher.doStuff(
                 kafkaPublisher);
              //   OF;          randomDataPublisher.doStuff(
                 rabbitMqPublisher);
        };
    }
}
```

RandomDataProvider

Deze klasse is een hulpklasse die de lijst van Data objecten genereert. Het aantal objecten wordt meegegeven als parameter dat ingesteld wordt in de RandomDataPublisher. De naam van het object en beschrijving is telkens de naam van het attribuut, respectievelijk name en description, plus het id eraan geplakt. Hier wordt ook een nieuwe datum gecreëerd dat dient als sendedDate.

```
@Component
public class RandomDataProvider {
    public List<Data> create(int numberOfEntries) {
        final List<Data> dataList = new ArrayList<>();
        for (int i = 0; i < numberOfEntries; i++) {
            dataList.add(new Data(i, "name" + i, "description"
                + i, new Date()));
        }

        return dataList;
    }
}
```

RandomDataPublisher

Wat is de bedoeling van deze klasse? Dit heeft de functionaliteit om de methode create uit de klasse RandomDataProvider op te roepen om de Data objecten te genereren en door te geven aan de juiste publisher. De constructor heeft opnieuw al de publisher klasses als parameter die gecreëerd worden door Spring Boot. Het heeft ook nog een extra parameter namelijk een instantie van de RandomDataProvider om later de create methode te kunnen gebruiken.

We verdiepen ons even in de methode doStuff. Hierin wordt aan de randomDataProvider gevraagd om een aantal Data objecten te maken, in deze codevoorbeeld is dit 10000 objecten. Deze slaat hij op een lijst van het type Data. Herinnert u zich nog dat ja in de ProducerApplication klasse aan de methode doStuff de publisher moest meegeven die je wou? Hier wordt ze gebruikt. In de methode wordt er gecontroleerd van welke instantie de publisher is die je mee krijgt. Indien de publisher van Kafka of RabbitMq is dan wordt deze lijst verzonden naar de Kafka topic of de RabbitMq exchange. Is het een instantie van de publisher van Google Pub/Sub dan wordt deze lijst overlopen en één voor één verstuurt. Want de publish methode van GooglePubSubPublisher wilt maar één object en geen lijst van objecten.

```
1  @Component
2  public class RandomDataPublisher {
3
4      private final RandomDataProvider randomDataProvider;
5      private final KafkaPublisher kafkaPublisher;
6      private final GooglePubSubPublisher googlePubSubPublisher;
7      private final RabbitMqPublisher rabbitMqPublisher;
8
9      public RandomDataPublisher(RandomDataProvider
           randomDataProvider,
10     KafkaPublisher kafkaPublisher,
11     GooglePubSubPublisher googlePubSubPublisher,
12     RabbitMqPublisher rabbitMqPublisher) {
13         this.randomDataProvider = randomDataProvider;
14         this.kafkaPublisher = kafkaPublisher;
15         this.googlePubSubPublisher = googlePubSubPublisher;
16         this.rabbitMqPublisher = rabbitMqPublisher;
17     }
18
19     public void doStuff(Object object) {
20         final List<Data> dataList = randomDataProvider.create
               (10_000);
21         if(object instanceof GooglePubSubPublisher) {
22             for (Data data : dataList) {
23                 googlePubSubPublisher.publishData(data);
24                 System.out.printf("publishing data: %s", data)
                       ;
25             }
26         } else if (object instanceof KafkaPublisher){
```

```
27              kafkaPublisher.publish(dataList);
28          } else if (object instanceof RabbitMqPublisher){
29              rabbitMqPublisher.publish(dataList);
30          }
31
32      }
33  }
```

3.2 Consumer project

De bedoeling van deze applicatie is dat ze wanneer er iets op een topic of queue geplaatst wordt, dat ze deze informatie weer omzet en opslaat in een lokale databank. Bij het opslaan van de data wordt er meer opgeslagen dan dat er informatie binnen komt om deze te kunnen vergelijken, maar dit wordt straks duidelijker. Wanneer deze applicatie opgestart is, luistert deze voortdurend naar de topics, of naar de queue, of er messages erop geplaatst worden. Indien er een message op een topic of de queue toegekomen is, dan leest hij deze meteen uit. Deze applicatie luistert dus tegelijkertijd naar zowel de topic van Kafka en Google Pub/Sub als naar de queue van RabbitMq. Om dit niet met elkaar te laten beïnvloeden wordt tijdens dit onderzoek enkel data verstuurd van een bepaalde technologie, bijvoorbeeld Kafka, indien de consumer applicatie niets meer leest van een andere technologie. Deze aanpak zorgt er dus voor dat data van verschillende technologieën niet door elkaar gelezen worden. We hebben in deze applicatie hetzelfde domein object van Data. Want we willen het object dat verstuurd werd op dezelfde manier terug krijgen. Door later de juiste deserializers mee te geven, moeten we ons geen zorgen maken en wordt het inkomende object automatisch gemapt naar hetzelfde domein object.

```
1  @Getter
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 }
```

Naast deze Data klasse hebben we ook per technologie een entity klasse en repository. Deze zorgen ervoor dat per technologie een klasse bestaat die we kunnen opslaan in onze databank. De repository zorgt ervoor dat een object van deze klasse effectief opgeslagen kan worden. De namen van deze klasses zijn: DataGPS, DataKafka en DataRMQ. Voor de repository zijn de namen van de interfaces: DataGPSRepository, DataKafkaRepository en DataRMQRepository. De implementatie van deze klasses is telkens dezelfde, maar het is nodig om deze apart te houden om in de databank de verschillende technologieën mooi apart te kunnen opslaan. Hier zullen we één entity klasse en repository uitleggen want

voor de andere is het dezelfde uitleg.

Entity klasse

We bekijken het voorbeeld van de DataGPS klasse.

```
@Entity
public class DataGPS {

    public DataGPS( int dataId, String name, String
       description, Date sendedDate, Date receivedDate, long
       timeDiff) {
        this.dataId = dataId;
        this.name = name;
        this.description = description;
        this.sendedDate = sendedDate;
        this.receivedDate = receivedDate;
        this.timeDiff = timeDiff;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    private int dataId;
    private String name;
    private String description;
    private Date sendedDate;
    private Date receivedDate;
    private long timeDiff;
}
```

Boven de klasse staat er een annotatie uit de package javax.persistence. De @Entity zorgt ervoor dat van deze klasse automatisch een tabel aangemaakt wordt in de databank. De attributen van deze klasse zullen automatisch ook de attributen zijn van deze tabel.

In de constructor zien we dat alle attributen van deze klasse aanwezig zijn als parameter, behalve het attribuut id. Dit is niet nodig om in de constructor mee te geven aangezien deze waarde automatisch gecreëerd wordt. Door de annotatie @Id zeg je dat dit veld het id is van de tabel en dus uniek moet zien. Dit is eveneens de primary key van de tabel. Door de tweede annotatie @GeneratedValue zorg je ervoor dat er automatisch een waarde aangemaakt wordt die uniek is. Je moet je dus voor de rest niets meer aantrekken van dit attribuut.

Repository

Ook hier bekijken we het voorbeeld van de DataGPSRepository.

```
public interface DataGPSRepository extends CrudRepository<
    DataGPS, Long> {
}
```

U ziet dat op het eerste zicht lijkt alsof deze klasse geen implementatie heeft. Toch kan je bij deze repository bijvoorbeeld de methode save(DataGPS data) oproepen om een object van de klasse DataGPS op te slaan. Dit komt omdat deze interface extends van de interface CrudRepository<T, ID>. Deze interface zorgt ervoor dat je een object van de klasse, die je meegeeft in de plaats van T, de basisoperaties kunt uitvoeren zoals save.

3.2.1 Kafka

KafkaConfig

Eerst en vooral zijn er opnieuw enkele waardes die we liever niet weergeven. In dit geval zijn er vier waardes: de naam van de topic, het bootstrapadres (het adres waar de server zich bevindt), een key en een secret. Sommige van deze variabelen zullen gebruikt worden in de application.yml file waar ook deze vier waardes zich bevinden. Anderen, zoals de topic, worden dan weer gebruikt in de implementatie zelf. De namen die we aan deze variabelen gegeven hebben in dit project zijn: kafka.topic.name (voor de naam van de topic), kafka.bootstrapAddress (voor het bootstrapadres), kafka_key en kafka_secret (voor de key en de secret).

Verder wordt er voor de configuratie enkel de application.yml gebruikt en dus geen aparte KafkaConfig klasse. Dit is hoe de configuratie er uit ziet voor Kafka:

```
spring:
    kafka:
        consumer:
            bootstrap-servers: ${kafka.bootstrapAddress}
            group-id: foo
            auto-offset-reset: latest
            key-deserializer: org.apache.kafka.common.
                serialization.StringDeserializer
            value-deserializer: org.apache.kafka.common.
                serialization.StringDeserializer

        properties:
            ssl.endpoint.identification.algorithm: https
            sasl.mechanism: PLAIN
            request.timeout.ms: 20000
            retry.backoff.ms: 500
            sasl.jaas.config: org.apache.kafka.common.security
                .plain.PlainLoginModule required username="${
                kafka_key}" password="${kafka_secret}";
            security.protocol: SASL_SSL
```

We zien dat het bootstrapadres meegegeven wordt, op deze manier weet de applicatie op welke server hij moet gaan zoeken. Hier worden er geen serializers gebruikt maar deserializers, wat voor zich spreekt. Om de waarde van de Json te deserializeren wordt de StringDeserializer gebruikt uit de package org.apache.kafka.common.serialization. Voor de key te deserializeren wordt de StringDeserializer gebruikt uit dezelfde package. Verder hoeven we ons geen zorgen te maken over het deserializeren. Het Spring framework regelt al dit werk voor ons. We kunnen er dus van uit gaan als er een message binnen komt dat dit automatisch op een goede manier omgezet wordt voor ons. Verder zien we ook dat de kafka_key en kafka_secret variabelen gebruikt worden om het wachtwoord en gebruikersnaam mee te geven. Dit is duidelijk waarom we deze waardes niet willen bloot stellen.

KafkaReceiver

Deze klasse is verantwoordelijk voor het ontvangen van objecten die van de Kafka topic komen. Deze klasse heeft twee attributen, een ObjectMapper en de DataKafkaRespository om de objecten op te slaan in de databank. Door deze mee te geven als parameter in de constructor zorgt Spring Boot er opnieuw voor dat deze automatisch gecreëerd wordt.

Deze klasse bevat slechts één methode: listen(String message). Hier zien we dat er boven de annotatie @KafkaListener te lezen valt. Deze annotatie krijgt de naam van de topic mee en het groupId. Hierdoor weet de applicatie van waar hij moet berichten lezen. Door deze annotatie zeg je dat als er een bericht uitgelezen moet worden uit deze topic, dat je deze methode moet uitvoeren. De String die je meekrijgt als parameter werd al automatisch gedeserialiseerd door de configuratie in application.yml.

In deze methode lezen we via de objectMapper de message uit en vormen deze om naar ons domein object van het type Data, dat dezelfde klasse is van de producer applicatie. Hierna maken we een nieuwe datum aan om te registreren wat de tijd was van het uitlezen van de message. Hierna vragen we aan ons object wat de tijd was van verzenden. Dit trekken we van elkaar af en op deze manier komen we te weten hoeveel milliseconden het verschilt tussen het uitlezen van de message en het verzenden ervan. Hierna slaan we deze gegevens op als een object van de klasse DataKafka. Veel van deze informatie halen we uit de oorspronkelijke message van het type Data. Maar voor de analyse van dit onderzoek is het ook handig om extra attributen mee te geven zoals het verschil. Dan wordt de repository opgeroepen met zijn methode save die hij kan gebruiken door de interface CrudRepository. En op deze manier wordt dit object opgeslagen in de databank.

```
1  @Component
2  public class KafkaReceiver {
3
4      private ObjectMapper objectMapper;
5      private DataKafkaRepository dataKafkaRepository;
6
7      public KafkaReceiver(final ObjectMapper objectMapper,
8      final DataKafkaRepository dataKafkaRepository) {
9          this.objectMapper = objectMapper;
10         this.dataKafkaRepository = dataKafkaRepository;
```

```
    }

    @KafkaListener(topics = "${kafka.topic.name}", groupId = "foo")
    public void listen(String message) throws Exception {

        try {
            Data data = objectMapper.readValue(message, Data.class);
            System.out.println(data);

            Date received = new Date();
            Date sended = data.getSendedDate();

            long difference = received.getTime() - sended.getTime();

            dataKafkaRepository.save(new DataKafka(data.getId(), data.getName(), data.getDescription(), data.getSendedDate(),received, difference));
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println("Received Messasge in group foo: " + message);
    }
}
```

3.2.2 RabbitMq

RabbitMqConfig

Ook hier wordt er enkel gebruikt gemaakt van de application.yml om RabbitMq te configureren. Eerst en vooral zijn er ook hier enkele variabelen die we liever niet blootstellen. Eerst en vooral de naam van de queue, deze zal gebruikt worden tijdens de implementatie. Hiernaast willen we ook de username, password en het hostadres niet tonen. Deze zullen gebruikt worden in deze application.yml file. De namen van deze variabelen zijn respectievelijk: rabbitmq.queue.name, rabbitmq_username, rabbitmq_password en rabbitmq_host.

De rest van de configuratie ziet er als volgt uit:

```
spring:
    rabbitmq:
        host: ${rabbitmq_host}
        port: 16379
        username: ${rabbitmq_username}
        password: ${rabbitmq_password}
```

```
        virtual-host: "rabbitmq"
        ssl:
            enabled: true
            algorithm: "TLSv1.2"
        listener:
            simple:
                default-requeue-rejected: false
            direct:
                default-requeue-rejected: false
```

Je ziet dat het hostadres meegegeven wordt, alsook de username en password.

RabbitMqReceiver

Deze klasse is zo goed als dezelfde als de KafkaReceiver. Het is evident dat de DataKafka-Repository vervangen is door de DataRMQRespository. Boven de methode receiveMessage staat ook een andere notatie namelijk @RabbitListener. Door deze annotatie boven de methode te zetten, weet de applicatie dat hij deze methode moet uitvoeren las er op de opgegeven queue een bericht komt. In deze methode wordt de message eerst weer omgezet in een Data object en dan opgeslagen als een DataRMQ object met de waardes van de message en het verschil tussen het verzenden en het ontvangen.

```
@Component
public class RabbitMqReceiver {

    private ObjectMapper objectMapper;

    private DataRMQRepository dataRMQRepository;

    public RabbitMqReceiver(final ObjectMapper objectMapper,
    final DataRMQRepository dataRMQRepository) {
        this.objectMapper = objectMapper;
        this.dataRMQRepository = dataRMQRepository;
    }

    @RabbitListener(queues = "${rabbitmq.queue.name}")
    public void receiveMessage(String message) throws
       Exception {
        try {
            Data data = objectMapper.readValue(message, Data.
               class);
            System.out.println(data);

            Date received = new Date();
            Date sended = data.getSendedDate();

            long difference = received.getTime() - sended.
               getTime();
```

```
24
25              dataRMQRepository.save(new DataRMQ(data.getId(),
                    data.getName(), data.getDescription(), data.
                    getSendedDate(),received, difference));
26          } catch (IOException e) {
27              e.printStackTrace();
28          }
29          System.out.println("Receiving message RabbitMq: " +
                message);
30      }
31
32  }
```

3.2.3 Google Pub/Sub

GooglePubSubConfig

Voor Google Pub/Sub werken we met een config klasse en ook met de application.yml. Er zijn vier zaken die meegegeven worden in deze laatste file. Twee ervan worden gebruikt in de implementatie, de andere twee moet je gewoon meegeven in deze file. De twee die gebruikt worden in de implementatie zijn pubsub.subscription en pubsub.autostartup. De eerste is de naam van de subscription, de tweede is een boolean om aan te duiden of de subscriber moet starten met luisteren als de applicatie wordt opgestart of niet. De andere twee configuraties die je moet meegeven zijn: spring.cloud.gcp.project-id en spring.cloud.gcp.credentials.location. De eerste is het id van het project, de tweede is de plaats waar de credentials staan op te kunnen inloggen op het Google Cloud Platform.

De GooglePubSubConfig klasse ziet er als volgt uit:

```
1  @Configuration
2  public class GooglePubSubConfig {
3
4      @Bean("bachelorproef-channel-input")
5      public MessageChannel createDeviceEventChannelInput() {
6          return new DirectChannel();
7      }
8
9      @Bean
10     public PubSubInboundChannelAdapter
           createDeviceChannelAdapter(
11     PubSubOperations pubSubTemplate,
12     @Value("${pubsub.subscription}") String subsciptionName,
13     @Qualifier("bachelorproef-channel-input") MessageChannel
           inputChannel,
14     @Value("${pubsub.autostartup}") Boolean autoStart) {
15
16         final PubSubInboundChannelAdapter adapter = new
               PubSubInboundChannelAdapter(pubSubTemplate,
```

```
            subsciptionName);
17          adapter.setOutputChannel(inputChannel);
18          adapter.setAckMode(AckMode.AUTO);
19          adapter.setPayloadType(Data.class);
20          adapter.setAutoStartup(autoStart);
21          return adapter;
22      }
23
24      @Bean
25      public JacksonPubSubMessageConverter
            createJacksonMessageConverter(final ObjectMapper
            objectMapper) {
26          return new JacksonPubSubMessageConverter(objectMapper)
                ;
27      }
```

Boven de klasse ziet u opnieuw de annotatie @Configuration staan om aan te geven aan het Spring framework dat dit een configuratie is. Verder wordt er een nieuwe MessageChannel aangemaakt, dit is het kanaal dat de messages die ontvangen worden moeten gebruiken. Er wordt ook een naam gegeven aan dit kanaal namelijk ”bachelorproef-channel-input”. Vervolgens moet er een PubSubInboundChannelAdapter aangemaakt worden die enkele configuraties bevat. Als parameter van deze methode moet je een PubSubOperations meegeven, dit is te vergelijken met de KafkaTemplate of RabbitTemplate. Verder moet je de naam van de subscription meegeven en ook welke MessageChannel hij moet gebruiken om message te ontvangen. Via de @Qualifier annotatie zeg je dat hij de MessageChannel moet gebruiken die we daarnet aangemaakt hebben. Als laatste geef je dan nog mee of het starten met luisteren naar de topic automatisch moet gebeuren of niet. In deze methode stel je dan nog in dat de payload van de message, die een Json is, van het type Data is. Door de laatste methode, die een JacksonPubSubMessageConverter aanmaakt, kan deze Json automatisch omgezet worden in een instantie van Data.

GooglePubSubReceiver

Dit doet ongeveer hetzelfde als de vorige receivers. Alleen hoef je hier geen ObjectMapper te gebruiken omdat deze al is meegegeven aan de JacksonPubSubMessageConverter. Verder wordt er gebruik gemaakt van de DataGPSRepository om objecten van het type DataGPS op te slaan.

De methode storeData krijgt geen message mee in de vorm van een String, want het omzetten van de message naar een object van Data is al gebeurd door de JacksonPubSub-MessageConverter. Je hebt dus meteen het Data object dat binnenkomt. Je hoeft dus enkel maar de juiste waardes op te vragen en mee te geven aan ee nieuwe DataGPS. Ook hier werd het verschil berekent tussen het verzenden en het ontvangen. Boven de methode staat de annotatie @ServiceActivator die als parameter het kanaal heeft waar messages op toekomen. Dit wil zeggen als een message langs dit kanaal binnenkomt, dat de applicatie deze methode moet uitvoeren.

```
@Component
public class GooglePubSubReceiver {

    private DataGPSRepository dataGPSRepository;

    public GooglePubSubReceiver(final DataGPSRepository
       dataGPSRepository) {
        this.dataGPSRepository = dataGPSRepository;
    }

    @ServiceActivator(inputChannel = "bachelorproef-channel-
       input")
    public void storeData(final Data data) throws Exception {
        System.out.println("--- Getting data: name: " + data.
           getName() + ", description: " + data.getDescription
           () +
        ", id: " + data.getId() + ", sended: " + data.
           getSendedDate().toString() + ", received: " + new
           Date().toString() + " ---");


        Date received = new Date();
        Date sended = data.getSendedDate();

        long difference = received.getTime() - sended.getTime
           ();

        dataGPSRepository.save(new DataGPS(data.getId(), data.
           getName(), data.getDescription(), data.
           getSendedDate(), received, difference));
    }

}
```

3.3 Onderzoek

Zoals al eerder vermeld worden de resultaten opgeslagen in een databank. De databank draait lokaal in een Docker container. Dit is de weinige configuratie die nodig is om te databank op te stellen:

```
spring:
    datasource:
        url: "jdbc:postgresql://localhost:5432/postgres"
        username: postgres
        password: postgres
    jpa:
        hibernate:
```

```
8               ddl-auto: create
9           jdbc:
10              lob:
11                  non-contextual-creation: true
```

De URL van de databank wordt meegegeven met daarachter de naam van de databank. Daaronder geef je de gebruikersnaam en wachtwoord mee. Dit is niet erg dat het wachtwoord bloot gegeven wordt, want dit is toch een lokale databank. Verder zie je ook dat indien de applicatie opnieuw opstart, de database opnieuw gecreëerd wordt.

Verder werd dit onderzoek drie keer uitgevoerd. Een keer met 10000, 100000 en 1000000 Data objecten die verzonden werden. Zo kunnen we ook vergelijken hoe sterk het tijdsverschil wordt. De bedoeling was om de methodes die de objecten ontvangen zo klein mogelijk te houden. Want hoe groter deze methodes worden, hoe langer het duurt per object om deze objecten te ontvangen en dit zou kunnen een verkeerd beeld geven anders.

De resultaten per onderzoek werden per technologie opgeslagen in een CSV-file om daarna berekeningen op te kunnen doen. Dit werd gedaan met een Python script.

Eerst werd de CSV uitgelezen via de pandas library, hierna wordt er wat aan data exploratie gedaan om te controleren of de waardes die uitgelezen werden wel effectief konden kloppen.

```
1  import pandas as pd
2  result = pd.read_csv('/Users/rubendesmet/Desktop/gps10_000.csv
       ',sep=',')
3  print(result.head())
4  print(result.columns)
5  print(result.index)
6  print(result.count())
7  print(result.shape)
```

Vervolgens worden de kolommen die niet van toepassing zijn verwijderd. Enkel de tabel timeDiff zou moeten overblijven die het verschil van de ontvangen en verzonden tijd voorstelt. Daarna kijken we hoeveel rijen er in de tabel zitten, worden eventuele NaN verwijderd, en kijken we nog eens hoeveel rijen er in de tabel zitten. Als laatste controlemiddel printen we nog eens de eerste vijf rijen af en kunnen we ook zijn of we kolommen vergeten verwijderen zijn.

```
8  print(result.count())
9  result = result.dropna()
10 print(result.count())
11 print(result.head())
```

Dan rest ons nog één ding. Dit is het berekenen van het gemiddelde van deze kolom, samen met het minimum en maximum.

```
12 print(result['timeDiff'].mean())
```

```
13 print(result['timeDiff'].max())
14 print(result['timeDiff'].min())
```

Al deze stappen worden dus negen keer uitgevoerd. Drie keer per technologie en drie keer per hoeveelheid verzonden data.

Tweede deel van het onderzoek

Na dat dit allemaal onderzocht was, werd er door de onderzoeker beslist dat we ook nog het gebruikte memory zouden onderzoeken. Hiervoor moest er niets aangepast worden aan de producer applicatie, maar wel aan de consumer applicatie.

```
15 // Get the Java runtime
16 Runtime runtime = Runtime.getRuntime();
17 // Run the garbage collector
18 runtime.gc();
19 // Calculate the used memory
20 long memory = runtime.totalMemory() - runtime.freeMemory();
21
22 memory = bytesToMegabytes(memory);
```

Per receiver werd er deze code toegevoegd in de methode die de messages ontvangt. Deze code achterhaalde de gebruikte memory. De hulpmethode bytesToMegabytes(memory) zet de memory om naar megabytes. Dit resultaat werd dan ook meegegeven bij het aanmaken van een object van een entity en zo mee opgeslagen. Dit wil zeggen dat er in DataGPS, DataKafka en DataRMQ een attribuut memory bijkwam. Bij de resultaten van dit gedeelte van het onderzoek werden enkel rekening gehouden met de memory. Kijken naar het verschil tussen het verzenden en ontvangen in tijd was niet meer nodig. Het ophalen van het gebruik van memory neemt namelijk zo veel tijd in beslag dat het tijdsverschil niet meer relevant was om te bekijken.

Door dit nieuwe gedeelte van het onderzoek kwamen er 3 CSV bestanden bij. Eén bestand per technologie die de memory berekent. Er werd gekozen om telkens maar 1000 objecten te verzenden. Er kwamen dus ook drie nieuwe Python-scripts bij. Ook één per technologie die het gemiddelde, maximum en minimum van de memory berekende per technologie. Het opbouw van deze script is gelijkaardig met het berekenen van het gemiddelde tijdsverschil. Maar in plaats van het tijdsverschil werd enkel de kolom met de memory bijgehouden.

4. Conclusie

4.1 Snelheid

4.1.1 10 000 objecten

Het eerste wat we gaan analyseren en een conclusie uit trekken is het verzenden van 10 000 objecten. Eerst kijken we naar de waardes apart per technologie, daarna gaan we alle technologieën gaan vergelijken met elkaar.

Google Pub/Sub

De resultaten van de snelheden bij 10 000 objecten van Google Pub/Sub zien er als volgt uit:

	Tijd (ms)
min.	1366
gem.	11665.7694
max.	19365
# ontvangen.	10 000

Tabel 4.1: Verschil tussen ontvangen en verzenden (in ms) - Google Pub/Sub

We zien dus dat het kleinste verschil iets meer dan één seconde is, en het grootste verschil bijna 20 seconden. Het gemiddeld aantal seconden ligt rond de 11,5 seconden.

Kafka

De resultaten van de snelheden bij 10 000 objecten van Kafka zien er als volgt uit:

	Tijd (ms)
min.	1008
gem.	20339.5169
max.	38537
# ontvangen.	10 000

Tabel 4.2: Verschil tussen ontvangen en verzenden (in ms) - Kafka

We zien dus dat het kleinste verschil nog dichter bij de één seconde ligt dan bij Google Pub/Sub. Het grootste verschil is bijna 40 seconden. Het gemiddeld aantal seconden ligt rond de 20 seconden.

RabbitMq

De resultaten van de snelheden bij 10 000 objecten van RabbitMq zien er als volgt uit:

	Tijd (ms)
min.	598
gem.	21780.9111
max.	41805
# ontvangen.	10 000

Tabel 4.3: Verschil tussen ontvangen en verzenden (in ms) - RabbitMq

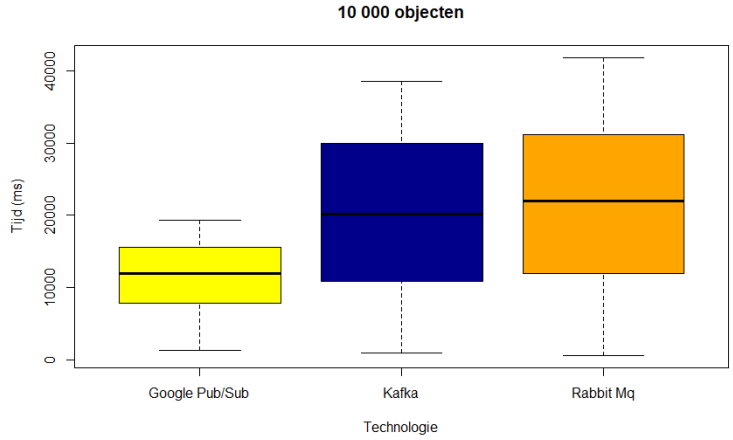
Het kleinste tijdverschil is iets meer dan een halve seconde. Het grootste verschil is meer dan 40 seconden. Het gemiddeld aantal seconden is bijna 42 seconden.

Vergelijking

Om een goede vergelijking te kunnen maken tussen de drie technologieën zetten we de resultaten bij 10 000 objecten van de verschillende technologieën even naast elkaar met behulp van drie boxplots die te zien is op figuur 4.1.

Bij zowel Google Pub/Sub, Kafka en RabbitMq zijn alle 10 000 objecten die verzonden zijn geweest, ook aangekomen. Dit wil zeggen dat bij 10 000 objecten geen enkel object mislukt is om te ontvangen. We kunnen bij deze hoeveelheid dus een duidelijke conclusie trekken als we kijken naar de snelheid tussen ontvangen en verzenden.

Eerst bekijken we de minimum waarden. Daarin is te zien dat RabbitMq de kortste tijd heeft van de drie minimum waarden. Dit is dubbel zo snel als we vergelijken met Kafka, die de tweede kortste tijd heeft. RabbitMq slaagt er in om in ongeveer een halve seconde een object te verzenden en te ontvangen. Kafka doet dit slechts in één seconde. Google



Figuur 4.1: Vergelijking bij 10 000 objecten

Pub/Sub is de slechtste leerling van de klas op het gebied van minimum tijd en doet er ongeveer 1,3 seconden over.

Laat ons nu kijken naar de maximum tijd. Hier doet Google Pub/Sub het veruit beter of de twee andere technologieën. Het doet er nooit langer dan 20 seconden over om een bericht te versturen en weer te lezen. Hier doen Kafka en RabbitMq het veel slechter. Zij doen er maximum 3,8 en 4,1 seconde over, en dit is veel meer of bij Google Pub/Sub.

Het resultaat van de maxima weerspiegelt zich ook in de gemiddeldes. Gemiddeld doet Google Pub/Sub er 11 seconden over. Het gemiddelde van Kafka en RabbitMq ligt dan weer wat hoger met 20 en 21 seconden.

Conclusie bij 10 000 objecten: Alhoewel de minimum tijd bij Google Pub/Sub hoger ligt of bij Kafka en RabbitMq, slaagt Google Pub/Sub er toch in om behoorlijk wat sneller al de data te verwerken.

4.1.2 100 000 objecten

Het lijkt ons ook interessant om eens te bekijken wat te resultaten zijn indien we het aantal objecten wat gaan verhogen. We nemen tien keer zo veel objecten en kijken wat we hiermee kunnen uit concluderen.

Google Pub/Sub

De resultaten van de snelheden bij 100 000 objecten van Google Pub/Sub zien er als volgt uit:

We zien dus dat het kleinste verschil iets minder is dan twee seconden, en het grootste verschil bijna 170 seconden, wat bijna drie minuten is. Het gemiddeld aantal seconden ligt rond de 88 seconden, wat bijna 1,5 minuten is.

	Tijd (ms)
min.	1753
gem.	88603.80396
max.	169580
# ontvangen.	100 000

Tabel 4.4: Verschil tussen ontvangen en verzenden (in ms) - Google Pub/Sub

Kafka

De resultaten van de snelheden bij 100 000 objecten van Kafka zien er als volgt uit:

	Tijd (ms)
min.	1055
gem.	175526.39283
max.	346431
# ontvangen.	100 000

Tabel 4.5: Verschil tussen ontvangen en verzenden (in ms) - Kafka

Het kleinste verschil blijft nog steeds dicht bij de één seconde liggen, alhoewel het verschil toch lichtjes gestegen is. Het grootste verschil ligt rond de 346 seconden, wat bijna zes minuten is. Het gemiddeld aantal seconden ligt rond de 175 seconden, omgerekend is dit bijna drie minuten.

RabbitMq

De resultaten van de snelheden bij 100 000 objecten van RabbitMq zien er als volgt uit:

	Tijd (ms)
min.	458
gem.	114365.12135045651
max.	224467
# ontvangen.	59 802

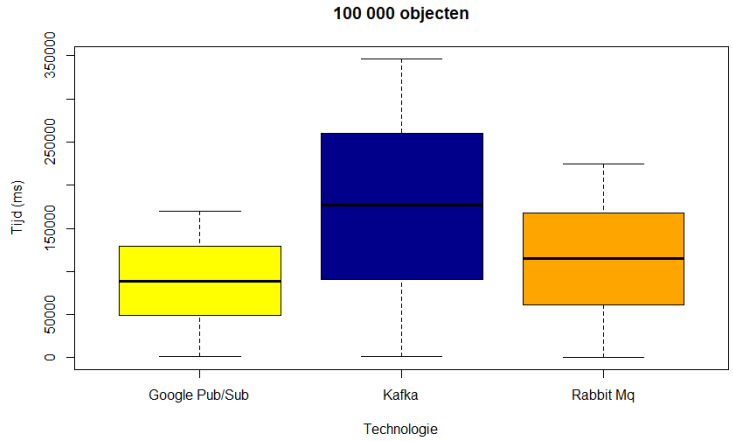
Tabel 4.6: Verschil tussen ontvangen en verzenden (in ms) - RabbitMq

Wat meteen opvalt, is dat bij deze hoeveelheid RabbitMq er niet in geslaagd is om tijdens dezelfde run alle 100 000 objecten binnen te halen. Natuurlijk haalt hij deze wel nog binnen indien de applicatie opnieuw opgestart wordt, maar het is toch opvallend dat maar iets meer of de helft meteen ontvangen wordt.

De minimum tijd is nog gedaald in vergelijking met 10 000 objecten. Hij heeft nu slechts minder dan een halve seconde nodig om te verzenden en te ontvangen. Het gemiddelde ligt rond de 114 seconden en het maximum iets meer dan 224 seconden. Natuurlijk zijn het gemiddelde en maximum minder belangrijk om te vergelijken aangezien niet alle 100 000 objecten zijn ontvangen.

Vergelijking

Ook bij deze hoeveelheid aan objecten zetten we de drie technologieën eens naast elkaar. Deze boxplots zijn te zien op figuur 4.2



Figuur 4.2: Vergelijking bij 100 000 objecten

Hierbij is het minder relevant om RabbitMq helemaal te gaan vergelijken met de twee andere technologieën aangezien maar iets meer dan de helft van de objecten ook ontvangen zijn geweest.

Eerst gaan we de minimum waarden bekijken, hier kunnen we ook nog vergelijken met RabbitMq. Daarin is te zien dat RabbitMq de kortste tijd heeft van de drie minimum waarden. Deze waarde is zelf nog minder dan bij de vorige vergelijking van 10 000 objecten. Kafka blijft redelijk stabiel van de kortste tijd en schommelt nog steeds rond de seconde. Bij Google Pub/Sub zien we toch een stijging en deze waarde is nu 1,7 seconden geworden.

Laat ons nu kijken naar de maximum tijd. Hier doet opnieuw Google Pub/Sub het veruit beter of de twee andere technologieën. We zouden kunnen zeggen dat de waarde bij RabbitMq ook nog goed meevalt, maar dit zou een vertekend beeld geven omdat niet alle objecten ontvangen zijn geweest. Dus het heeft enkel zin om voor de maximum waarde Google Pub/Sub te gaan vergelijken met Kafka. Bij Kafka ligt de maximum waarde veel hoger. Het heeft er maximum ongeveer 346 seconden voor nodig, wat bijna dubbel zo veel is dan bij Google Pub/Sub.

Het resultaat van de maxima weerspiegelt zich ook opnieuw in de gemiddeldes. Ook hier houden we geen rekening met RabbitMq want niet alle objecten zijn ontvangen. We kunnen eigenlijk al zeker concluderen dat bij 100 000 objecten Google Pub/Sub sowieso sneller zou geweest zijn dan RabbitMq aangezien het gemiddelde voor minder ontvangen waarden al hoger ligt dan bij Google Pub/Sub. Gemiddeld doet Google Pub/Sub er 88 seconden over. Het gemiddelde van Kafka ligt dan weer wat hoger met 175 seconden.

Conclusie bij 100 000 objecten: Alhoewel de minimum tijd bij Google Pub/Sub opnieuw

hoger ligt of bij Kafka en RabbitMq, slaagt Google Pub/Sub er toch in om behoorlijk wat sneller al de data te verwerken.

4.1.3 1 000 000 objecten

De laatste hoeveelheid objecten die we onderzoeken is nog eens tien keer zo veel. We kijken ook eens wat het resultaat is bij 1 000 000 objecten.

Google Pub/Sub

De resultaten van de snelheden bij 1 000 000 objecten van Google Pub/Sub zien er als volgt uit:

	Tijd (ms)
min.	3327
gem.	577596.5258098603
max.	1037761
# ontvangen.	556 454

Tabel 4.7: Verschil tussen ontvangen en verzenden (in ms) - Google Pub/Sub

We zien dus dat het kleinste verschil iets meer is dan drie seconden, en het grootste verschil bijna 1037 seconden, wat bijna 17 minuten is. Het gemiddeld aantal seconden ligt rond de 577 seconden, wat iets meer dan 9,5 minuten is. Hier is Google Pub/Sub voor het eerst niet in geslaagd om alle objecten te ontvangen. Het zal dus ook hier wat moeilijker zijn om het gemiddelde en het maximum te vergelijken.

Kafka

De resultaten van de snelheden bij 1 000 000 objecten van Kafka zien er als volgt uit:

	Tijd (ms)
min.	3930
gem.	1719014.245389
max.	3424980
# ontvangen.	1 000 000

Tabel 4.8: Verschil tussen ontvangen en verzenden (in ms) - Kafka

Het kleinste verschil ligt nu niet meer rond de seconde, maar is nu gestegen naar bijna vier seconden. Het grootste verschil ligt rond de 3424 seconden, wat bijna een uur is. Het gemiddeld aantal seconden ligt rond de 1719 seconden, omgerekend is dit bijna een half uur.

RabbitMq

De resultaten van de snelheden bij 1 000 000 objecten van RabbitMq zien er als volgt uit:

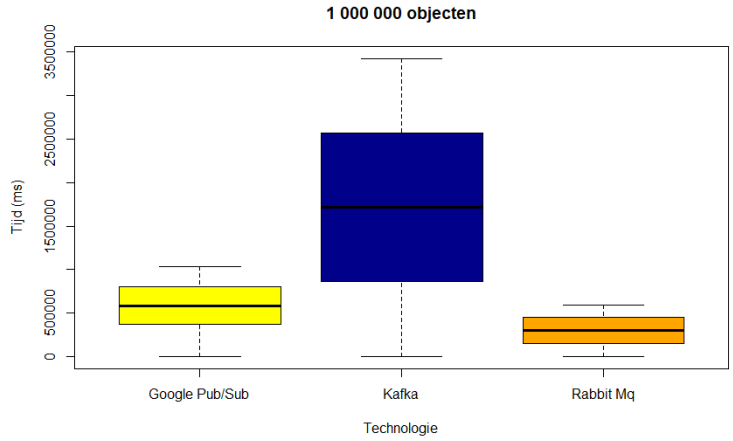
	Tijd (ms)
min.	1345
gem.	300640.46033216227
max.	590104
# ontvangen.	59 802

Tabel 4.9: Verschil tussen ontvangen en verzenden (in ms) - RabbitMq

RabbitMq is ook hier niet in geslaagd om alle objecten te ontvangen. De minimum tijd is voor het eerst gestegen naar iets meer dan een seconde. Het gemiddelde ligt op 300 seconden. En het duurt nooit langer dan 590 seconden of een kleine tien minuten om een object te ontvangen.

Vergelijking

Ook bij deze hoeveelheid aan objecten zetten we de drie technologieën eens naast elkaar. Deze boxplots zijn te zien op figuur 4.3



Figuur 4.3: Vergelijking bij 1 000 000 objecten

Nu is het wat moeilijker om de snelheden met elkaar te gaan vergelijken aangezien de drie verschillende technologiëen verschillend aantal objecten hebben ontvangen. Toch overlopen we de technologieën eens.

Eerst gaan we de minimum waarden bekijken, hier kunnen we nog alle drie met elkaar vergelijken. Daarin is te zien dat RabbitMq nog steeds de kortste tijd heeft van de drie minimum waarden. Deze waarde is wel gestegen naar iets boven de seconde, maar blijft nog steeds de beste. Kafka is voor het eerst ook opvallend gestegen. De waarde is bijna vier keer zo groot geworden. Bij Google Pub/Sub zien we ook een stijging, het minimum

ligt nu ook rond de drie seconden. Kafka heeft dus bij deze hoeveelheid het meeste tijd nodig als minimum waarde.

Laat ons nu kijken naar de maximum tijd. Dit is wat moeilijker om te vergelijken aangezien al de technologieën verschillend aantal objecten ontvangen hebben. Kafka heeft het meeste tijd nodig, maar heeft ook het meeste objecten ontvangen. RabbitMq en Google Pub/Sub hebben ongeveer evenveel ontvangen, en daar zien we RabbitMq toch duidelijk beter zijn voor de maximum waarde.

Het resultaat van de maxima weerspiegelt zich ook opnieuw in de gemiddeldes. De verhoudingen zijn hetzelfde als bij de maximum waardes

Conclusie bij 1 000 000 objecten: RabbitMq doet het beter dan Google Pub/Sub. Kafka is moeilijk te vergelijken met de twee andere.

4.1.4 Conclusie snelheid

Op het gebied van snelheid is Kafka over het algemeen gezien de slechtste. Google Pub/Sub doet het over het algemeen gezien het beste, behalve bij super veel objecten (hier: 1 000 000) doet RabbitMq beter dan Google Pub/Sub.

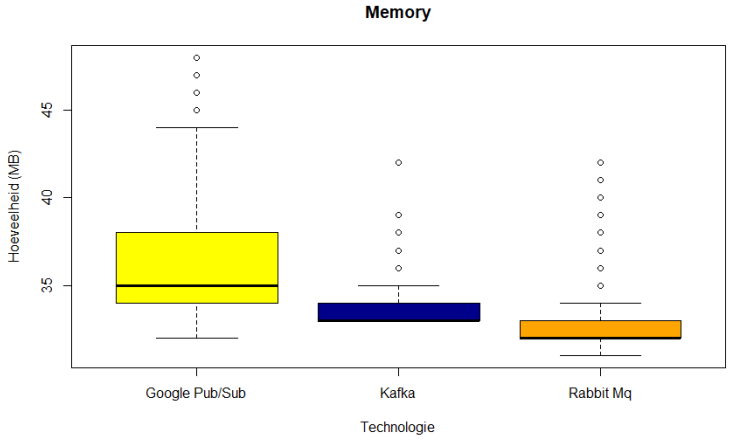
4.2 Memory

4.2.1 1 000 objecten

Omdat de operatie om aan de Java runtime de gebruikte memory te vragen te lang duurt, wordt er enkel gekeken bij 1 000 objecten. Er wordt hier geen rekening gehouden met de snelheden.

4.2.2 Conclusie memory

Op figuur 4.4 ziet u de boxplots van het resultaat van het memory gebruik. Het is duidelijk te zien dat er bij alle technologieën enkele uitschieters zijn. Maar het is ook duidelijk dat over het algemeen voor het memory gebruik RabbitMq beter is. Het gemiddelde, maximum en het minimum liggen lager in vergelijking met de andere. Gemiddeld heeft RabbitMq 33.138 MB nodig. Google Pub/Sub en Kafka hebben gemiddeld 36.31 en 33.601 nodig. RabbitMq is dus beter maar het verschil met Kafka is klein. Het is enkel Google Pub/Sub die veel meer memory nodig heeft dan de rest.



Figuur 4.4: Vergelijking memory

4.3 Algemene conclusie

Wanneer snelheid voor een bedrijf het belangrijkste is, dan lijkt Google Pub/Sub het beste. Tenzij je heel veel objecten hebt (meer dan 1 000 000) dan lijkt RabbitMq sneller te werken. Het memory gebruik is het best bij RabbitMq maar Kafka doet het niet veel slechter. Bij Kafka ben je het meest zeker dat je effectief al je verzonden data meteen ontvangt. Hier slaagt RabbitMq het slechtste op.

Voor TVH specifiek lijkt het belangrijkste dat alle data zeker meteen ontvangen wordt. Omdat Kafka hierop het beste scoort en bijna de beste is op basis van memory gebruik, lijkt Kafka de beste technologie om te gebruiken binnen dit bedrijf.

Een verder onderzoek zou dit onderzoek enkele keren kunnen herhalen om te bepalen hoe frequent verzonden data niet meteen aankomt.

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 technologiën 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.