



VERANTWOORDINGSVERSLAG

Solution Architecture

Retail: ball.com

Michael van Zundert, Kevin Gerretsen, Lynn Vissers, Floris
Botermans, Jorrit Meeuwissen
2124598, 2050253, 2124332, 2128810, 2126991

Inhoudsopgave

1. Inleiding	2
2. Requirements engineering	3
2.1 Functionele requirements	3
2.2 Niet-functionele requirements.....	3
3. Context map	4
4. Event storm	5
5. ArchiMate model	6
6. Microservices.....	7
6.1 Orderservice	7
6.2 CustomerManagementService.....	9
6.2.1 Externe systemen	10
6.3 Inventoryservice	11
6.4 Paymentservice	13
6.4.1 Event Sourcing & CQRS.....	13
6.4.2 Eventual Consistency.....	13
6.5 CustomerSupportservice	14
6.5.1 Event-driven architectuur	14
6.5.2 Eventual Consistency.....	15
6.5.3 Event Sourcing.....	16
6.6 Notificationservice.....	17

1. Inleiding

Dit verslag is een toelichting over het project van het vak Solution Architecture uit periode 3.4 van de opleiding Informatica. Het doel van het verslag is om de keuzes duidelijk te maken en waarom er voor de huidige oplossing is gekozen.

Er is gekozen voor de retail casus van ball.com voor dit verslag. Het doel van dit project was om de principes van Solution Architecture toe te passen en te beheersen. Het domein waarbinnen de opdracht gerealiseerd is, is daarom een denkbeeldige webwinkel met producten, bestellingen en een klantenservice.

In het eerste hoofdstuk worden alle functionele- en niet-functionele requirements beschreven. Hierna wordt de context map getoond en nog de event storm en het ArchiMate model voor dit project. In het laatste hoofdstuk worden alle microservices uitgelegd met de onderliggende concepten.

2. Requirements engineering

De functionele- en niet-functionele requirements zijn uitgewerkt vanuit de casus. De requirements zijn uitgewerkt om de casus af te bakenen en duidelijk te beschrijven wat er uitgewerkt is.

2.1 Functionele requirements

De functionele requirements beschrijven wat de applicatie moet kunnen en wat er uiteindelijk uitgewerkt moet zijn in de code. De functionele requirements zijn gebaseerd op aannames uit de casus. De functionele requirements zijn hieronder zichtbaar.

- Als gebruiker wil ik de voorraad van producten kunnen inzien.
- Als gebruiker wil ik een bestelling kunnen plaatsen met producten.
- Als gebruiker wil ik de status van een bestelling kunnen inzien.
- Als gebruiker wil ik kunnen kiezen om vooraf of achteraf te betalen.
- Als gebruiker wil ik een vraag kunnen stellen aan de klantenservice.
- Als klantenservice wil ik kunnen antwoorden op een vraag.
- Als klantenservice wil ik de vragen kunnen afronden.
- Als beheerder wil ik producten toe kunnen voegen aan de voorraad.
- Als beheerder wil ik verkopers kunnen toevoegen aan het systeem.
- Als beheerder wil ik een vervoerder kunnen toevoegen aan een order.

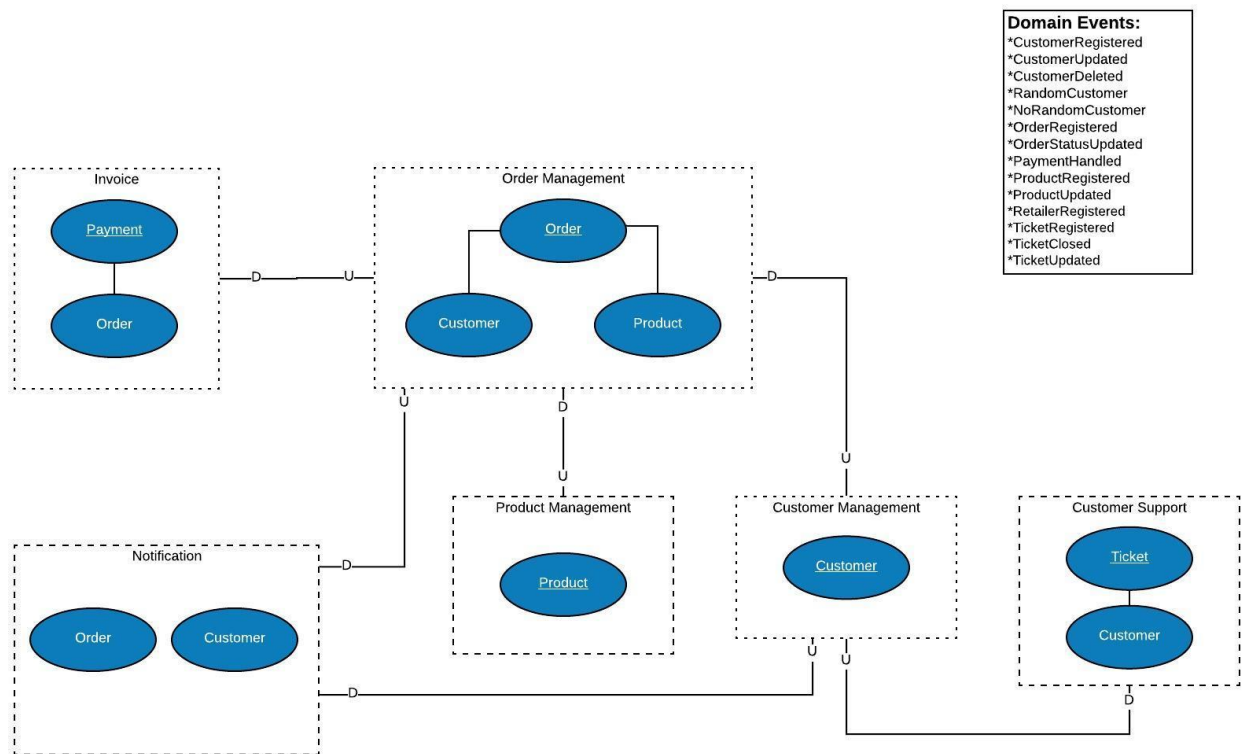
2.2 Niet-functionele requirements

De applicatie moet gebouwd worden op het principe van microservices. Microservices zijn modulair. Modulariteit van de applicatie is dus een essentieel onderdeel. Ten tweede moeten delen van de applicatie gemakkelijk te modificeren zijn, zonder dat andere onderdelen van de applicatie hier hinder aan ondervinden. Ten derde moet de applicatie in zekere mate beschikbaar blijven als bepaalde onderdelen uitvallen. Dit valt onder de non-functionele eis "Availability". De non-functionele requirements staan hieronder beschreven.

- Een order kan bestaan van 1 tot 20 items.
- Logistics company wordt gekozen op basis van laagste prijs.
- De applicatie moet modulair opgebouwd zijn
- Delen van de applicatie moeten gemakkelijk te modificeren zijn
- De applicatie moet in zekere mate beschikbaar blijven als een onderdeel uitvalt

3. Context map

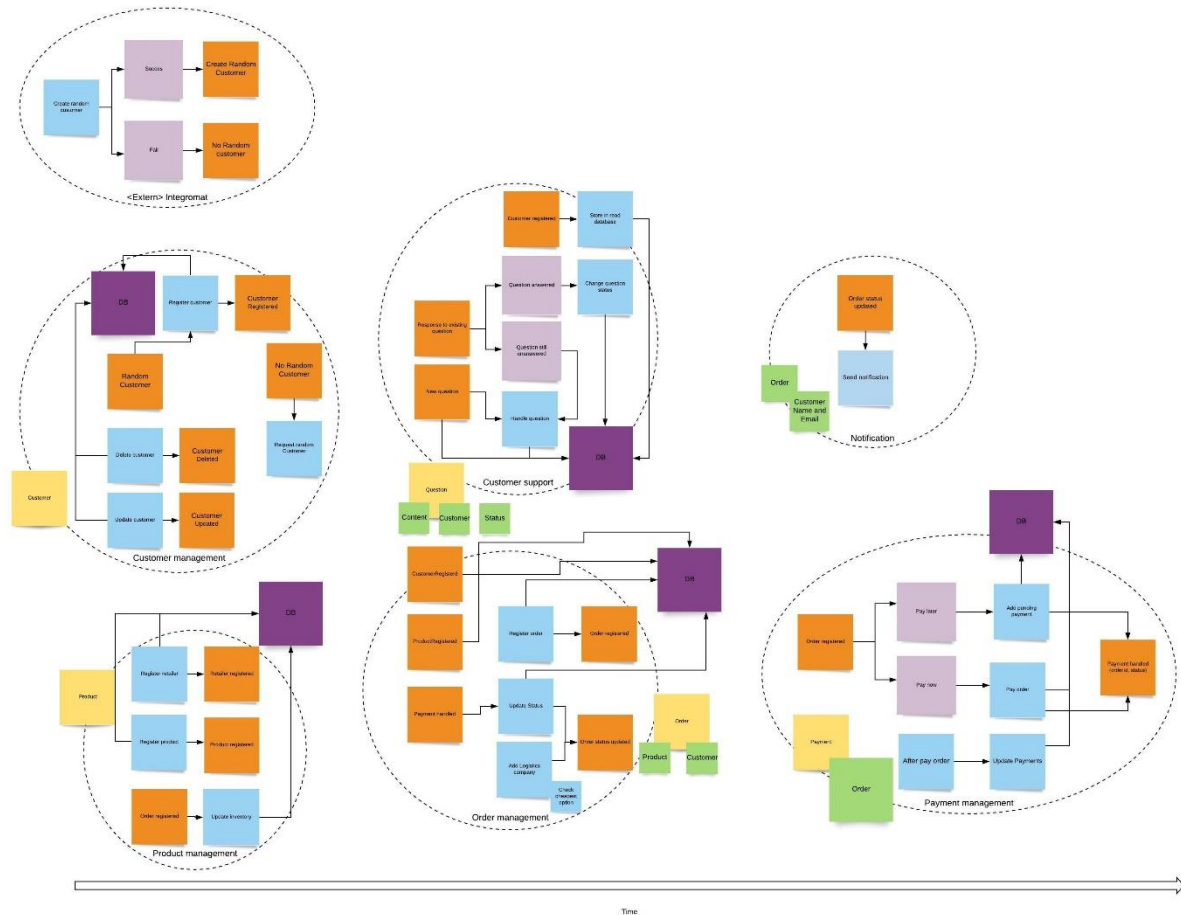
De context map bevat alle modellen van de casus en hoe deze zich tot elkaar verhouden. Bounded contexts zijn aangegeven door de stippellijnen. Een bounded context is de grens binnen een domein waar een bepaald domeinmodel van toepassing is. In een microservice architectuur wordt een bounded context waarschijnlijk een microservice. De context map die gebruikt wordt voor dit project is te zien in Figuur 1.



Figuur 1: Context map

4. Event storm

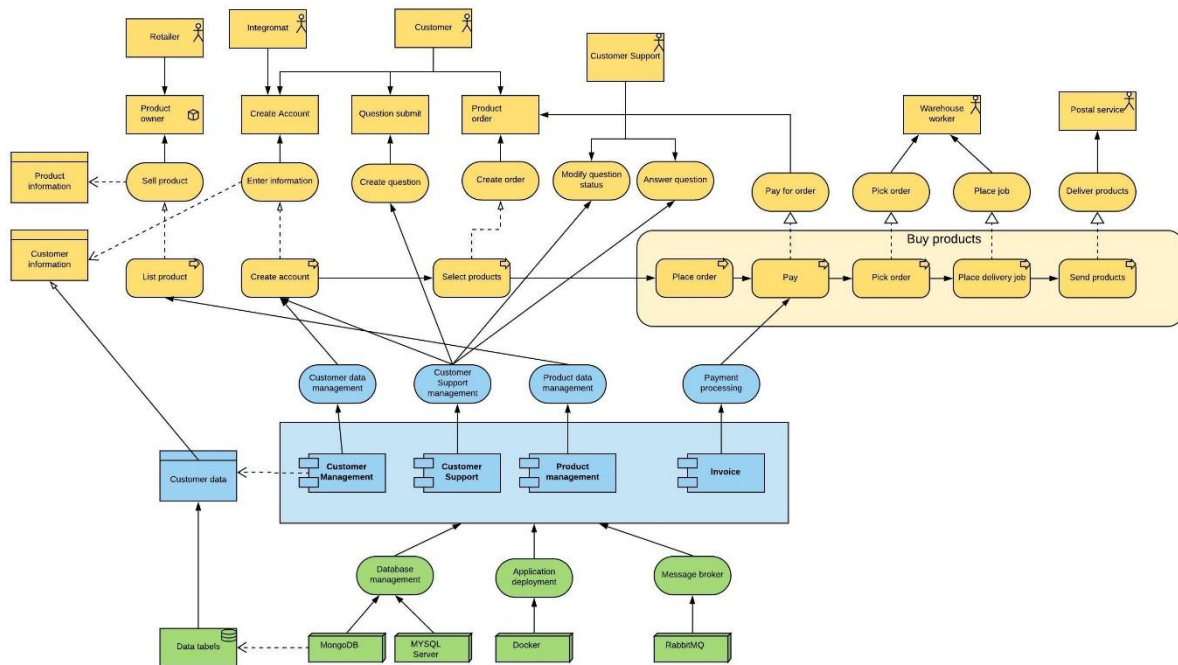
Event storming is een workshop-gebaseerde methode om snel te achterhalen wat er gebeurt in het domein van een applicatie. Met behulp van sticky notes kan je de bedrijfsprocessen goed in beeld zetten. De event storm voor deze casus is te zien in Figuur 2.



Figuur 2: Event storm

5. ArchiMate model

ArchiMate is een enterprise architectuur modelleertaal die de processen binnen een bedrijf beschrijft. Naast dat de processen en artefacten beschreven worden, wordt hierin ook de vertaling gemaakt naar de uitwerking in een geautomatiseerd systeem. Het ArchiMate model voor deze casus is te zien in Figuur 3.



Figuur 3: Archimate model

6. Microservices

De applicatie is opgedeeld in zes microservices. Voor deze architectuur is gekozen omdat dit de flexibiliteit en beschikbaarheid van de applicatie ten goede komen, iets wat gewenst is in de niet-functionele requirements.

De microservices praten richting een messagebroker wij hebben gekozen om hier rabbitMQ voor te gebruiken. De event die de microservices versturen worden richting de exchange verstuurd. Deze exchange plaatst dit event in alle queue's die verbonden zijn met deze exchange elke microservices luisterd naar ze eigen queue en haalt hier de events vandaan.

Bovendien is voor deze architectuur gekozen zodat iedere microservice apart ontwikkeld kan worden. Hierdoor kan iedere student aan een eigen microservice werken, wat de ontwikkeltijd van de applicatie ten goede komt. Ook kan met deze architectuur per onderdeel van de applicatie een geschikte technologie gekozen worden.

Hieronder wordt elke service uitgelegd met de bijhorende technieken en onderliggende concepten.

Als laatst word er Containerization toegepast op alle microservices. Wij gebruiken Docker om containerization te verwezelijken. In onze docker compose file staan al onze service met de bijbehorende data servers die hier voor nodig zijn. Voor de orderService is alleen MongoDB nodig omdat de RabbitMQ online word gehost voor de implementatie van het externe systeem.

6.1 Orderservice

Order service is gekozen als **microservice** omdat het domein Order een groot deel is van de applicatie ball.com. Door de keuze om hiervoor een microservice te gebruiken zijn alle handelingen die gedaan worden met een order losgekoppeld van de rest van de applicatie. In de gehele applicatie wordt gebruik gemaakt van **Event driven architecture based on messaging**. Dit heeft als reden dat hierdoor **loose coupling** ontstaat omdat de microservices geen kennis bevatten over wat er met de events gebeurt die ze versturen. De microservices hebben ook geen kennis over hoe een event tot stand is gekomen als ze er naar luisteren. Als message broker waar de message naar toe gestuurd wordt en in een queue geplaatst wordt gebruiken wij **RabbitMQ**. In de Order service wordt geluisterd naar de volgende events:

- CustomerRegistered:
Dit event komt van CustomerManagementService. In dit event wordt een customer meegegeven die zojuist is aangemaakt door deze microservice. Alleen het customerID wordt opgeslagen in de readdatabase van Order service.
- ProductRegistered:
Dit event komt van InventoryService. In dit event wordt een product meegegeven die zojuist is aangemaakt door deze microservice. In de readdatabase van Order service wordt het id, de naam en de hoeveelheid producten er beschikbaar zijn.
- PaymentUpdated:
Dit event komt van InventoryService. In dit event wordt een product meegegeven die zojuist is bijgewerkt in deze microservice. In de readdatabase van Order service wordt het product met hetzelfde id bijgewerkt.
- PaymentHandled:
Dit event komt van PaymentService. In dit event wordt een orderID meegegeven en de status of het betaald is. In de order service wordt een event toegevoegd in de write

database van order service met de gekoppelde orderID. In dit event staat wat er aan de order is bijgewerkt. Hierna wordt de read database van order service geupdate met het gekoppelde orderID.

De events customerRegistered, ProductRegistered en PaymentUpdated worden gebruikt om **Eventual consistency** te realiseren. Dit wordt gedaan omdat customer en product een **aggregate** zijn met als **aggregate root** order. Omdat Order service zijn eigen customer en product data bijhoudt moet deze consistent blijven met de data vanuit de microservices die hiervoor bedoeld zijn. Dit wordt gerealiseerd door te luisteren naar de events die hierboven genoemd zijn en de data te updaten op basis van deze events.

Naast het luisteren naar events verstuurt de orderService ook verschillende events. Deze events worden verstuurd of nadat er een endpoint is aangeroepen of omdat er geluisterd is naar een ander event en er veranderingen heeft plaatsgevonden. In de Order service worden de volgende events verstuurd:

- **OrderRegistered:**
Dit event wordt verstuurd nadat de endpoint aangeroepen is om een order aan te maken. De order die meegegeven is in de body van de request wordt opgeslagen als event in de event store en daarna als readmodel in de read database. Als dit allemaal succesvol verloopt verstuurt hij het event OrderRegistered naar de exchange ball.com
- **OrderStatusUpdated:**
Dit event wordt verstuurd na 2 verschillende reacties. De eerste keer dat dit event wordt verstuurd is nadat het event PaymentHandled binnen komt. De tweede keer is nadat de endpoint aangeroepen wordt om een logistic company aan de order toe te passen. Bij beide keren wordt de status van de order veranderd waardoor het event OrderStatusUpdated wordt verstuurd.

In de orderservice wordt ook **CQRS** toegepast. Hiermee scheiden we de schrijf en lees data. Dit zorgt ervoor dat ze afzonderlijk van elkaar opgeschaald kunnen worden wanneer je ziet dat een van de twee meer gebruikt wordt dan de andere. In de orderservice wordt dit gedaan door een commandcontroller en een query controller. De commandcontroller spreekt richting de write database en de query controller haalt data uit de read database.

De CQRS in orderservice is gecombineerd met **eventsourcing**. In plaats van de gehele orders op te slaan in de write database wordt er een lijst aangemaakt per order waarin event worden toegevoegd. Deze worden opgeslagen in een **eventstore**. In een event wordt een eventtype meegegeven en data over de order. Daarnaast wordt er een readmodel opgeslagen in de read database. Op het moment dat de status moet worden opgehaald wordt de list met events gelijk aan het gekoppelde order id opgehaald. De order service loopt dan door al deze events heen en reconstrueert het order object. Vanuit dit object wordt dan de status gegeven als response.

6.2 CustomerManagementService

De **microservice** CustomerManagement is gekozen om het beheer van de customers te laten afhandelen. Alle handelingen omtrent de informatie van een customer zullen door deze service afgehandeld worden. Wanneer andere services informatie over customers nodig hebben zullen zij luisteren naar de events van deze service (**Event driven architecture based on messaging**). Hierdoor ontstaat loose coupling omdat elke service op zichzelf kan functioneren en de data die hierbij nodig is in de database van de betreffende service staat. **RabbitMQ** is gebruikt als message broker, hier worden alle events naartoe gestuurd en op een exchange gezet, services kunnen hun queue koppelen aan deze exchange en zo luisteren naar events.

In de CustomerManagement service wordt geluisterd naar de volgende events:

- RandomCustomer
Dit Event wordt op de queue geplaatst door de externe applicatie, Integromat, wanneer er een aanvraag wordt gedaan om een random customer aan te maken. De data hiervan komt vanuit de RandomUser.me API. Na dit event wordt er een user aangemaakt in de service en wordt er een CustomerRegistered event gestuurd.
- NoRandomCustomer
Dit event komt op de queue wanneer er een random user is aangevraagd maar de API van RandomUser.me momenteel geen status 200 terugstuurt. Wanneer dit gebeurt zal de service de webhook van de API nogmaals aanroepen.

De Events die CustomerManagement uitzend zijn:

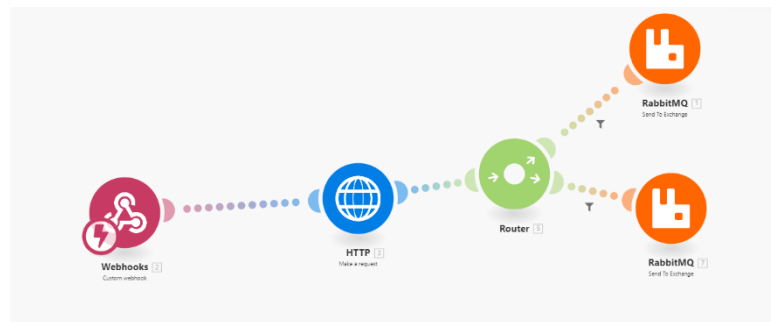
- CustomerRegistered
Wanneer er een customer wordt aangemaakt in de service zullen de andere services op de hoogte gesteld moeten worden. Hiervoor wordt dit event uit gestuurd met daarin de data van de customer.
- CustomerUpdated
Wanneer er een customer wordt aangepast in de service zullen de andere services op de hoogte gesteld moeten worden. Hiervoor wordt dit event uit gestuurd met daarin de vernieuwde data van de customer.
- CustomerDeleted
Wanneer er een customer wordt verwijderd in de service zullen de andere services op de hoogte gesteld moeten worden. Hiervoor wordt dit event uit gestuurd met daarin het id van de verwijderde customer.

De events die de service uitstuurt worden gebruikt voor **eventual consistency**, waarbij alle services die deze data nodig hebben de events kunnen verwerken wanneer hun beschikbaar zijn en zo hun data consistent kunnen hebben.

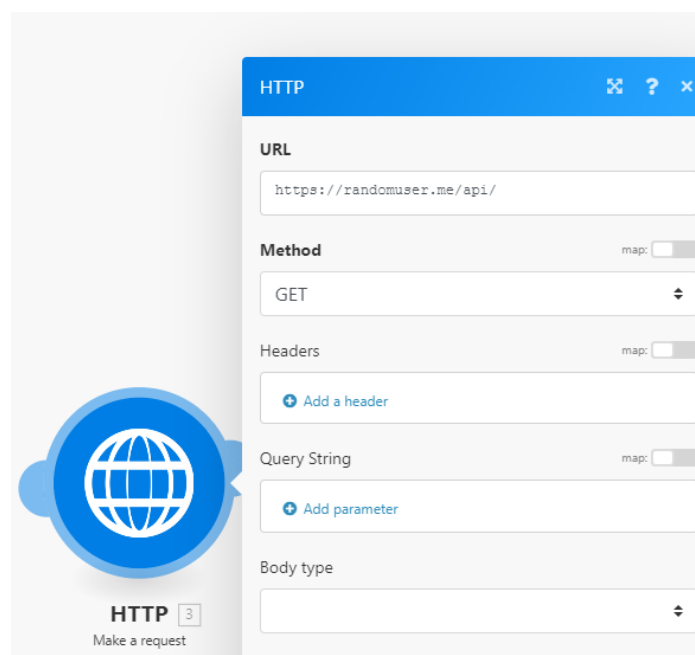
Om acties uit te laten voeren door de service is er een API met drie endpoints voor het aanmaken, updaten en verwijderen van een customer.

6.2.1 Externe systemen

Er is gebruikt gemaakt van een Integromat pipeline, die luistert naar een webhook. Vervolgens wordt een request gestuurd naar de random user API. De Integromat pipeline zet dan vervolgens de data wanneer deze er is op de queue. Dit verbindt de servers met een alleenstaand apart programma. Dit is een vorm van ad hoc applicatie integratie.



Figuur 4 Integromat Webhook

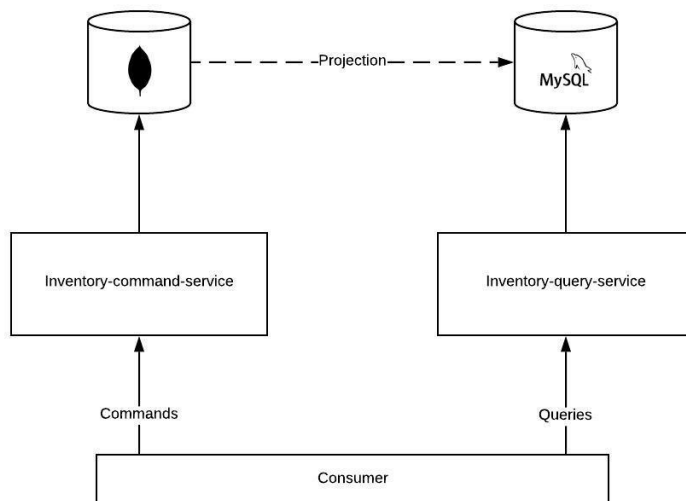


Figuur 5: HTTP Request

6.3 Inventoryservice

De microservice Inventoryservice is gekozen om het beheer van de producten af te handelen. In de applicatie wordt er gebruik gemaakt van Event driven architecture based on messaging. Dit heeft als reden dat hierdoor loose coupling ontstaat omdat de microservices geen kennis bevatten over wat er met de events gebeurt die ze versturen en de microservices hebben ook geen kennis over hoe een event tot stand is gekomen als ze er naar luisteren. Er wordt gebruik gemaakt van RabbitMQ als message broker waar de messages naartoe gestuurd worden en in een queue geplaatst worden. Om de availability te behouden zijn alle microservices van elkaar apart. Hierdoor kan een gedeelte van het systeem online blijven en is het niet afhankelijk van elkaar. Door gebruik te maken van events die de service uitstuurt wordt eventual consistency behouden. Alle services die de data nodig hebben kunnen de events verwerken wanneer ze beschikbaar zijn en zo hun data uiteindelijk consistent te hebben.

De service maakt gebruik van een write database die draait op MongoDB en een read database die draait op MySQL. De read database luistert naar de events die de write database uitstuurt. Hierdoor is er eventual consistency. Op deze manier wordt CQRS toegepast. Hieronder is te zien hoe CQRS is toegepast in een diagram.



Figuur 6: CQRS inventory

De microservice is gescheiden door twee verschillende services. Een command service en een query service. Met de command service worden alle commands uitgevoerd om iets te schrijven en de query service om data op te halen van de inventory.

De command service heeft twee POST endpoints. De endpoints zijn hieronder beschreven.

1. `/inventory/retailer`

Dit endpoint roept de functie `postretailer` aan in de `retailercontroller`. Deze slaat een retailer op in de MongoDB database en stuurt een `RetailerRegistered` event uit. De query service luistert naar het event door een consumer en slaat ook hiermee de retailer toe aan zijn MySQL database. De body bevat een naam voor de retailer en je krijgt een response terug met de naam en het id.

2. `/inventory`

Dit endpoint roept de functie `postProduct` aan in de `productcontroller`. Deze functie slaat een product op in de MongoDB database en stuurt een `ProductRegistered` event uit. De query

service luistert hier ook weer naar en slaat dan het product ook op in zijn MySQL database. De body van het request bevat de productgegevens en het id van de bijhorende retailer. Als response krijg je het aangemaakt product terug met het bijhorende id.

De query service heeft vier GET endpoints. De endpoints zijn hieronder beschreven.

1. `/inventory/retailer`
Met dit endpoint wordt de functie `getretailer` in de `retailerController` aangeroepen. In deze functie worden alle retailers opgehaald uit de MySQL database. De response die je dan terugkrijgt is een array met retailers.
2. `/inventory/retailer/:id`
Met dit endpoint wordt de functie `getOneretailer` in de `retailerController` aangeroepen. In deze functie wordt de retailer met het id dat in de parameter staat opgehaald. De response is een JSON object van de retailer.
3. `/inventory`
Met dit endpoint wordt de functie `getProducts` in de `inventoryController` aangeroepen. In deze functie worden alle producten opgehaald die in de MySQL database staan. De response die je dan terugkrijgt is een array met producten.
4. `/inventory/:id`
Met dit endpoint wordt de functie `getOneretailer` in de `inventoryController` aangeroepen. In deze functie wordt het product met het id opgehaald dat in de parameter staat. De response is een JSON object van het product.

Er wordt ook naar een extern event geluisterd die op RabbitMQ staat. Er wordt geluisterd naar het `OrderRegisteredEvent`. Als er een order aangemaakt wordt zal het inventory amount geüpdatet worden. In de command service wordt er naar dit event geluisterd en wordt er door de order gelopen. Voor elk product in de order wordt het amount geüpdatet. Hierna wordt er nog een event genaamd `ProductUpdated` uitgestuurd per product. Hier luistert de query service naar en update hier ook zijn MySQL database met het nieuwe amount.

6.4 Paymentservice

De payment service krijgt een order registered event binnen van de order management service. Zodra dit event binnenkomt wordt de order opgeslagen in de read database. Dit wordt gedaan zodat de payment service onafhankelijk van de andere services kan functioneren. Hierdoor wordt ook het loose coupling principe ondersteund.

Vervolgens wordt er een nieuwe payment aangemaakt. Op basis van het payment type dat binnenkomt, die “now” of “later” kan bevatten, wordt de status op “true” of “false” gezet. Deze status wordt vervolgens met een payment handled event op de queue gezet, waar de order service weer naar luistert. Wanneer de status op “false” staat kan deze, zodra de order is betaald, op “true” worden gezet door middel van de API endpoint.

6.4.1 Event Sourcing & CQRS

Door event sourcing toe te passen worden alle veranderingen aan de state van de applicatie sequentieel opgeslagen. Om tot een gewenste applicatie state te komen moeten deze events opnieuw opgebouwd worden. In de payment service wordt dit opbouwen van events niet gedaan, omdat hier geen logische implementatie voor is te bedenken. Wel worden de events gepersisteerd in een event store.

Hier komt ook CQRS bij kijken. Er zijn namelijk aparte modellen gebruikt voor het opslaan van de events en voor het uitlezen van de database. Zo wordt voor het uitlezen van de database het query model gebruikt. Daarnaast wordt voor het aanmaken van een nieuwe payment of het updaten van de status van de payment het command model gebruikt om ze op te slaan in de event store. In figuur 1 is te zien dat bij het opslaan van het event onder andere het versie nummer wordt bijgehouden. Hierdoor kan precies gezien worden in welke volgorde de events gebeurd zijn.

Key	Value	Type
▼ (1) ObjectId("60a454f1d9a3f156a4f916ca")	{ 6 fields }	Object
_id	ObjectId("60a454f1d9a3f156a4f916ca")	ObjectId
version	1	Int32
timestamp	2021-05-18 23:59:45.341Z	Date
messageType	PaymentHandled	String
▼ eventData	{ 5 fields }	Object
_id	ObjectId("60a454f1d9a3f156a4f916c9")	ObjectId
date	2021-05-18 23:59:45.267Z	Date
isPaid	false	Boolean
orderNumber	24	Int32
__v	0	Int32
__v	0	Int32

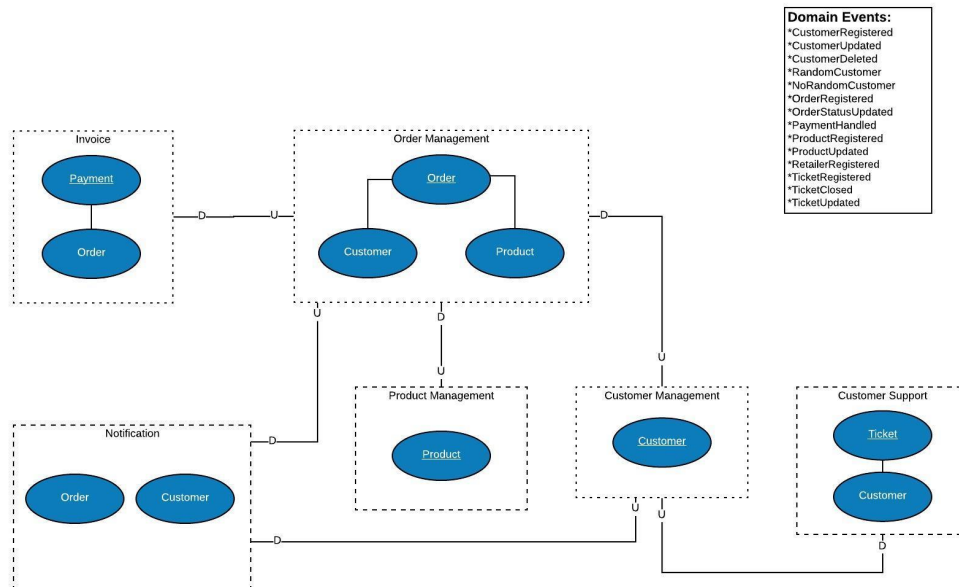
Figuur 7: Event object

6.4.2 Eventual Consistency

Eventual consistency is toegepast in de payment service bij het verzenden van de events. Wanneer de payment service tijdelijk offline zou gaan, maar de order service stuurt wel een event, dan is de payment service tijdelijk inconsistent met de order service. Wanneer de payment service weer online gaat, komt het event dat op de queue staat binnen en wordt het event afgehandeld, waardoor de payment service weer consistent wordt met de order service.

6.5 CustomerSupportservice

De customer support server regelt het ticket systeem. Het is gebaseerd op het domein model van tickets, en werkt samen met andere services die draaien in aparte containers binnen een Docker app. Binnen rabbitmq wordt gebruik gemaakt van het publish-subscribe integratie patroon.



Figuur 8 Context Map

De customer support service ontvangt customers vanuit de customer management service, en implementeert een abstracte implementatie van een ticket event publishing systeem, mocht hier in de toekomst in de context een belang van zijn.

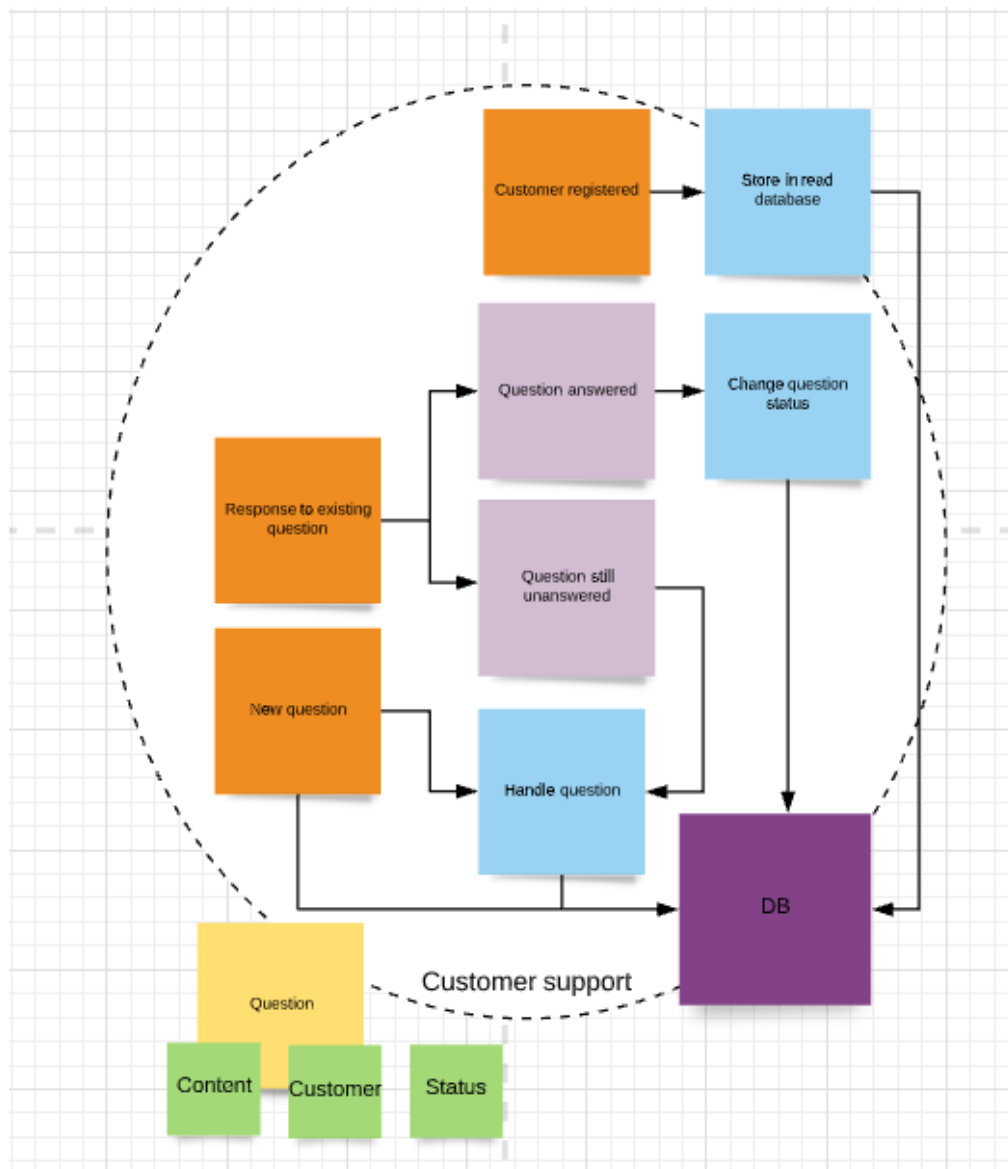
Omdat deze services niet direct van elkaar afhankelijk zijn, en zij gebaseerd zijn op hun eigen unieke domeinen, is het mogelijk om de services los van elkaar te gebruiken. Dit waarborgt de 'loose coupling' en 'containerization' principes.

6.5.1 Event-driven architectuur

Voor communicatie tussen de servers wordt rabbitmq gebruikt. Dit systeem vormt een berichten bus tussen de servers. In plaats van een typische request-response architectuur, sturen de servers geen bericht terug als er een event binnen is gekomen. Andere krachten van event-based architectuur die zijn toegepast, staan beschreven in dit hoofdstuk.

6.5.2 Eventual Consistency

De customer support service luistert naar events om haar eigen read database bij te houden. Deze events leven op de rabbitmq server, en zijn gespecificeerd met gebruik van headers.

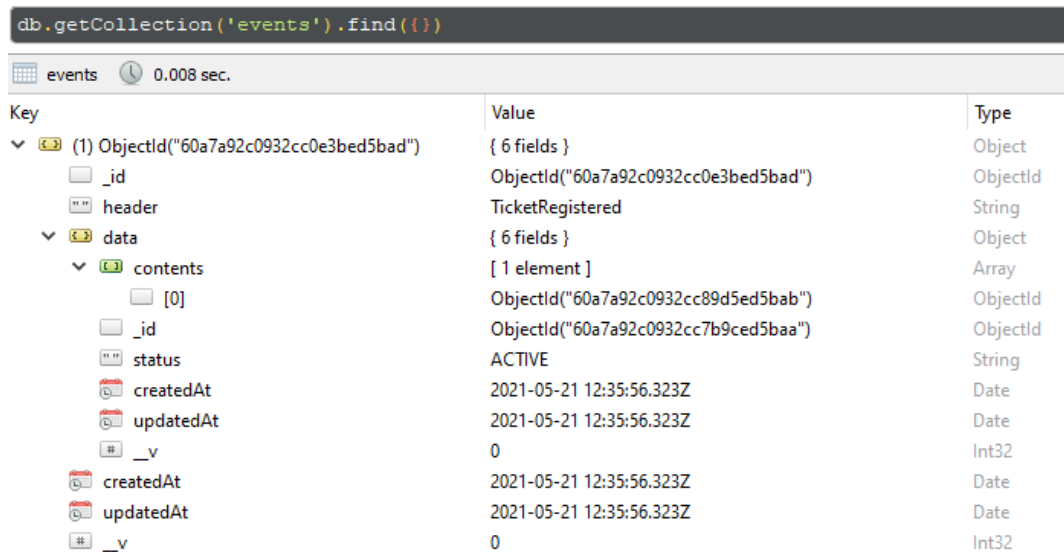


Figuur 9 Event Storm - Customer Support

Als een klant een account aanmaakt, of er iets aan dit account veranderd, wordt de read database bijgehouden. Dit zorgt ervoor dat als alle events gelezen worden, de server dezelfde database opbouwt als de andere servers. Dit proces waarborgt de eventual consistency van het systeem.

6.5.3 Event Sourcing

Om te controleren of de structuur binnen de lokale database consistent is, kan een lijst van events uitgevoerd worden vanuit een event store. In dit geval houdt de customer support service een lijst van events bij, die dan afgevuurd kunnen worden.



```
db.getCollection('events').find({})
```

events 0.008 sec.

Key	Value	Type
(1) ObjectId("60a7a92c0932cc0e3bed5bad")	{ 6 fields }	Object
_id	ObjectId("60a7a92c0932cc0e3bed5bad")	ObjectId
header	TicketRegistered	String
data	{ 6 fields }	Object
contents	[1 element]	Array
[0]	ObjectId("60a7a92c0932cc89d5ed5bab")	ObjectId
_id	ObjectId("60a7a92c0932cc7b9ced5baa")	ObjectId
status	ACTIVE	String
createdAt	2021-05-21 12:35:56.323Z	Date
updatedAt	2021-05-21 12:35:56.323Z	Date
_v	0	Int32
createdAt	2021-05-21 12:35:56.323Z	Date
updatedAt	2021-05-21 12:35:56.323Z	Date
_v	0	Int32

Figuur 10 Event Objecten in Database

Omdat de read database en event store apart worden bijgewerkt en beschikbaar zijn, is het mogelijk om queries uit te voeren op de read database, en write acties uit te voeren op de event store. Deze structuur zorgt ervoor dat query en commands van elkaar gesplitst zijn door ze naar andere databases of collections te sturen. Dit waarborgt het principe van CQRS.

6.6 Notificationservice

De notificationservice zorgt voor het versturen van notificaties door middel van email naar de klant. De service staat los van de andere service en hierdoor veroorzaakt het een loose coupling. De service luistert naar verschillende events op RabbitMQ. Hieronder staan de verschillende events met uitleg wat de service met de events doet.

- CustomerRegistered
Bij het ontvangen van dit event maakt hij een klant aan in zijn eigen MongoDB database. Om de klant zelf op te slaan met het event veroorzaakt het eventual consistency. De data van de klant wordt gebruikt om mails te sturen.
- OrderRegisteredEvent
Bij het ontvangen van dit event wordt de klant opgehaald met het customerId in de order. Er wordt een mail gestuurd naar de klant zijn email adres. In de mail wordt de klant op de hoogte gehouden met de status van zijn order.
- OrderStatusUpdatedEvent
Bij het ontvangen van dit event wordt de klant opgehaald met het customerId in de order. Er wordt een mail gestuurd naar de klant zijn email adres. In de mail wordt de klant op de hoogte gehouden met de status van zijn order.