

Productverantwoordingsverslag Solution Architecture

Door: Tom Schoonbeek (2032257), Tom Smits (2124227),   
Robin la Rondelle (2128503) en Sjoerd Schepers (2122696)  
  
*zondag 17 mei 2020*

Inhoud

[Hoofdstuk 1: Inleiding 2](#_Toc40648370)

[Hoofdstuk 2: Requirements engineering 3](#_Toc40648371)

[2.1 Functionele requirements 3](#_Toc40648372)

[2.2 Niet-functionele requirements 4](#_Toc40648373)

[2.3 Architectural constraints 4](#_Toc40648374)

[Hoofdstuk 3: Context map 5](#_Toc40648375)

[Hoofdstuk 5: Uitwerking concepten 6](#_Toc40648376)

[5.1 Microservices en Domain Driven Design 6](#_Toc40648377)

[5.2 Eventual consistency 7](#_Toc40648378)

[5.3 Event driven architecture 8](#_Toc40648379)

[5.4 Command Query Responsibility Segregation (CQRS) 8](#_Toc40648380)

[5.5 Event sourcing 9](#_Toc40648381)

[5.6 Enterprise integration patterns 10](#_Toc40648382)

[5.7 Containerization 11](#_Toc40648383)

# Hoofdstuk 1: Inleiding

Dit verslag is een toelichting op de gemaakte keuzes ten aanzien van ontwerpprincipes van het vak Solution Architecture uit periode 3.4 van de opleiding informatica te Avans Breda. Het doel van dit verslag is om duidelijk te maken welke keuzes gemaakt zijn, waarom voor een specifieke oplossing is gekozen en waar de consequenties van deze keuzes in de broncode terug te vinden zijn.

De structuur van het verslag is bepaald door de zogenaamde “deliverables”. Dit zijn de zaken die opgeleverd dienen te worden om deze opdracht succesvol af te ronden. Het is belangrijk te realiseren dat de uitgewerkte casus niet als doel had om een zo volledig mogelijke uitwerking te geven, maar om aan te geven dat de principes uit Solution Architecture beheerst worden en toegepast kunnen worden. Door ons is gekozen voor de retail casus. Het domein waarbinnen de opdracht is gerealiseerd is daarom een denkbeeldige webwinkel met producten, bestellingen en een klantenservice.

In het eerste inhoudelijke hoofdstuk worden alle functionele- en niet-functionele requirements beschreven samen met de architectural constraints. Daarna is de context map bijgevoegd met een lijst van alle domain events. Het ArchiMate model van de enterprise architecture is daarna bijgevoegd. In het laatste hoofdstuk komen alle onderliggende concepten aan bod. Dit zijn achtereenvolgend: microservices, eventual consistency, event driven architecture, CQRS, event sourcing, enterprise integration patterns en containerization.

# Hoofdstuk 2: Requirements engineering

De functionele- en niet-functionele requirements zijn, net als de architectural constraints, gebaseerd op een combinatie van voorkennis, literatuuronderzoek en aannames. Deze eigenschappen dienen exclusief om de casus af te bakenen en duidelijk te beschrijven wat wel en niet uitgewerkt is.

## 2.1 Functionele requirements

De functionele requirements beschrijven expliciet wat de applicatie moet kunnen en dus ook wat uitgewerkt moet zijn in de uiteindelijk opgeleverde code. De functionele requirements zijn, zoals eerder benoemd, gebaseerd op aannames en weerspiegelen daarom niet één op één een webshop in de echte wereld. De functionele requirements zijn hieronder zichtbaar (tabel 1).

|  |  |  |
| --- | --- | --- |
| **#** | **Rol** | **Beschrijving** |
| F1 | Gebruiker | Als gebruiker wil ik de catalogus kunnen inzien. |
| F2 | Gebruiker | Als gebruiker wil ik producten uit de catalogus kunnen bestellen. |
| F3 | Gebruiker | Als gebruiker wil ik kunnen kiezen of ik vooraf of achteraf betaal. |
| F4 | Gebruiker | Als gebruiker wil ik mijn bestelling kunnen volgen. |
| F5 | Gebruiker | Als klant wil ik een vraag kunnen stellen aan de klantenservice |
| F6 | Beheerder | Als beheerder wil ik producten toe kunnen voegen aan de catalogus. |
| F7 | Beheerder | Als beheerder wil ik producten kunnen verwijderen uit de catalogus. |
| F8 | Beheerder | Als beheerder wil ik derden toegang kunnen geven om producten toe te voegen aan de catalogus. |
| F9 | Beheerder | Als beheerder wil ik toegang tot derden in kunnen trekken zonder bestaande bestellingen te verwijderen. |
| F10 | Beheerder | Als beheerder wil ik dat de vervoerder aan de hand van de prijs automatisch gekozen wordt. |
| F11 | Beheerder | Als beheerder wil ik vervoerders kunnen toevoegen. |
| F12 | Beheerder | Als beheerder wil ik vervoerders kunnen verwijderen. |
| F13 | Magazijnmedewerker | Als magazijnmedewerker wil ik een overzicht van de te verzamelen pakketten kunnen inzien. |
| F14 | Magazijnmedewerker | Als magazijnmedewerker wil ik een bestelling af kunnen ronden als die verstuurd is. |
| F15 | Klantenservice | Als klantenservicemedewerker wil ik een overzicht kunnen zien van alle support aanvragen |
| F16 | Klantenservice | Als klantenservicemedewerker wil ik een reactie kunnen toevoegen aan een support aanvraag |
| F17 | Klantenservice | Als klantenservicemedewerker wil ik een bestelling kunnen bewerken. |
| F18 | Klantenservice | Als klantenservicemedewerker wil ik alle bestellingen kunnen inzien. |
| F19 | Klantenservice | Als klantenservicemedewerker wil ik een bestelling kunnen verwijderen. |
| F20 | Externe verkoper | Als externe verkoper wil ik producten toe kunnen voegen aan de catalogus. |
| F21 | Externe verkoper | Als externe verkoper wil ik een overzicht van toegevoegde producten kunnen inzien. |
| F22 | Externe verkoper | Als externe verkoper wil ik producten kunnen verwijderen uit de catalogus. |
| F23 | Externe verkoper | Als externe verkoper wil ik toegevoegde producten kunnen wijzigen. |

Tabel ) De functionele requirements zoals toegepast in de uitwerking van de casus

## 2.2 Niet-functionele requirements

Het doel van de opdracht is om een applicatie te bouwen die op het principe van microservices is gebaseerd. Microservices zijn inherent modulair. Modulariteit van de applicatie is dus een essentieel onderdeel om te meten. 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 beschreven in onderstaande tabel (tabel 2).

|  |  |
| --- | --- |
| **#** | **Beschrijving** |
| NF1 | De applicatie moet modulair opgebouwd zijn |
| NF2 | Delen van de applicatie moeten gemakkelijk te modificeren zijn |
| NF3 | De applicatie moet in zekere mate beschikbaar blijven als een onderdeel uitvalt |

Tabel ) De niet-functionele requirements zoals toegepast in de uitwerking van de casus

## 2.3 Architectural constraints

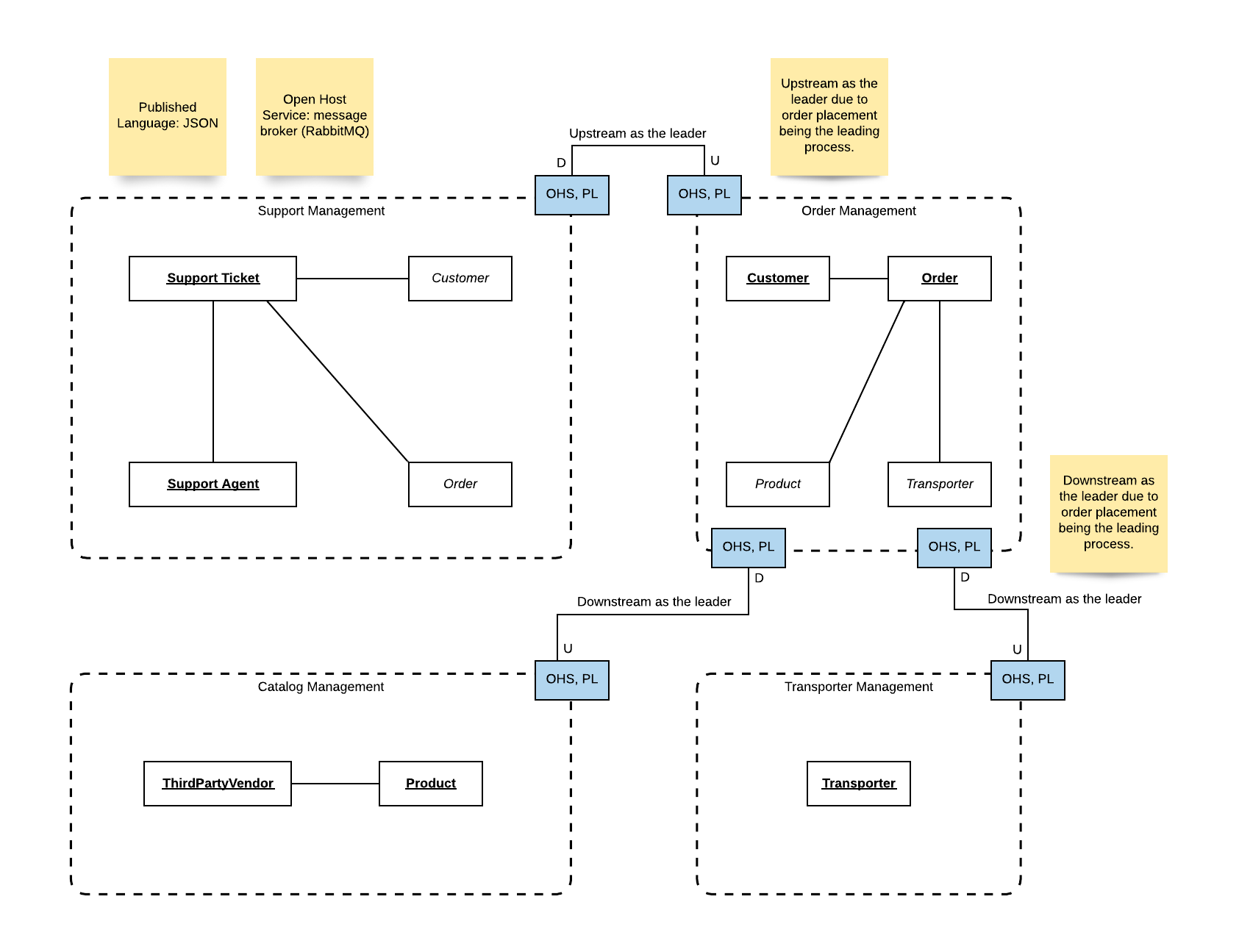
Architectural constraints zijn zaken die buiten de controle van de ontwikkelaars of de opdrachtgever ligt. Een voorbeeld hiervan is bijvoorbeeld wetgeving of vereisten van opdrachtgevers. De architectural constraints voor deze casus zijn weergegeven in de onderstaande tabel (tabel 3).

|  |  |
| --- | --- |
| **#** | **Beschrijving** |
| AC1 | De applicatie moet voldoen aan de Nederlandse wetgeving |
| AC2 | De applicatie moet aan alle eisen voldoen zoals aangeleverd door Avans Hogeschool in opdracht van het vak Solution Architecture |

Tabel ) De architectural constraints zoals toegepast in de uitwerking van de casus

# Hoofdstuk 3: Context map

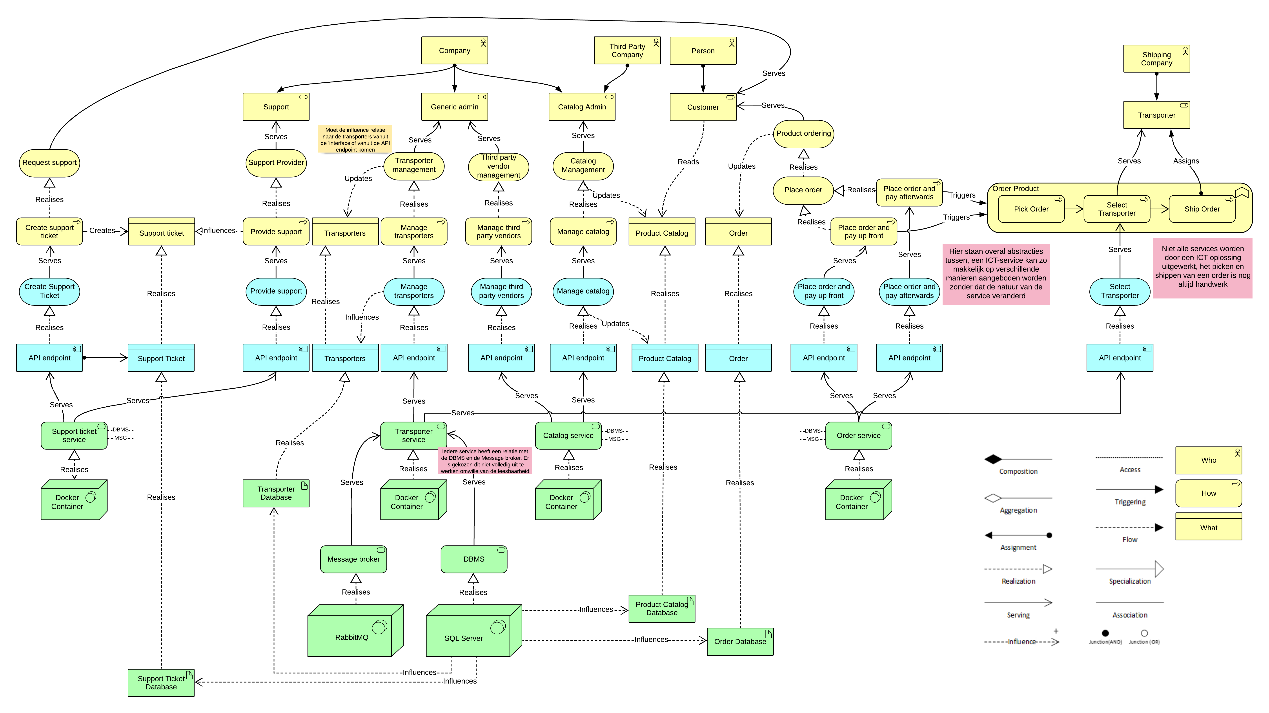
Een context map bevat alle modellen van de casus en hoe die zich tot elkaar verhouden. Het kan voorkomen dat binnen een bedrijf bijvoorbeeld meerdere betekenissen voor het principe bestaan, afhankelijk van met wie je praat. Dit zijn bounded contexts (aangegeven door de stippellijnen). Vrij vertaald naar een microservice architectuur betekent dit dat een bounded context waarschijnlijk (hoeft dus niet) een microservice wordt, omdat de betekenissen van het domein ondubbelzinnig zijn. De context map die gebruikt wordt voor de uitwerking van deze casus is weergegeven in Afbeelding 1



Afbeelding : Context map

# Hoofdstuk 4: ArchiMate model

ArchiMate is een enterprise architectuur modelleertaal die 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 waarop deze casus gebaseerd is, is zichtbaar in Afbeelding 2. In het zip-bestand waar dit document zich in bevind is een hoge kwaliteit bijlage van dit model toegevoegd.



Afbeelding : Archimate diagram

# Hoofdstuk 5: Uitwerking concepten

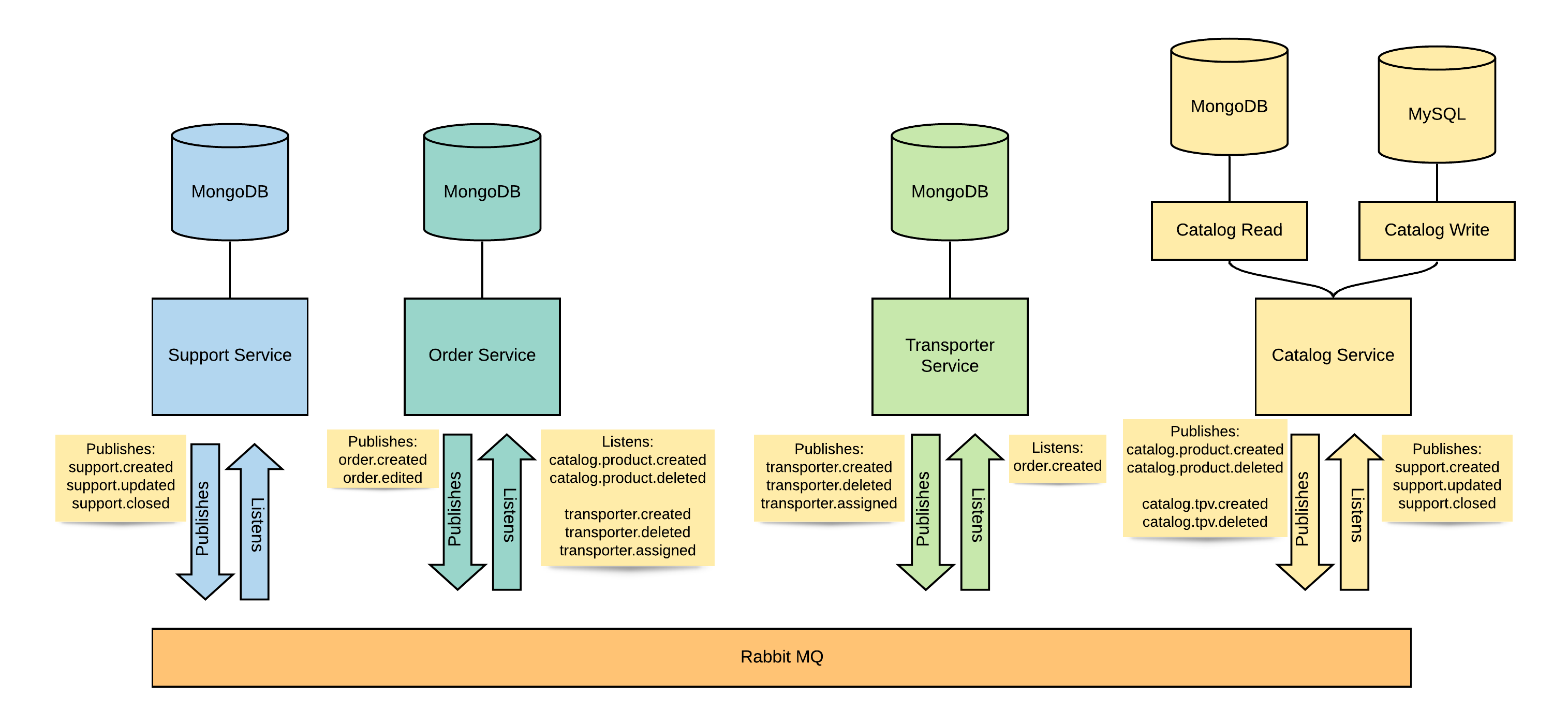
Het aantonen dat de principes uit de lessen van Solution Architecture begrepen en toegepast kunnen worden staat centraal in de uitwerking van deze casus. Per principe is daarom een hoofdstuk om aan te geven waar dit voorkomt en een onderbouwing waarom dat principe juist daar voorkomt.

## 5.1 Microservices en Domain Driven Design

De applicatie is opgedeeld in vier microservices, ondersteund met een drietal servers (MongoDB Server, RabbitMQ Server en MySQL Server. 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.

Elke microservice heft zijn eigen definitie van de modellen van de applicatie, een eigen domein dus. Niet elke service heeft iets te maken met alle modellen van de applicatie, andere zijn juist weer afhankelijk van de modellen van andere microservices. Met dit in gedachte heeft elke microservice zijn eigen domein ontworpen met daarin zijn eigen eisen en wensen qua modellen. De microservices worden onderling met elkaar verbonden middels een Rabbit MQ Exchange, die berichten van de ene microservice naar de andere microservice kan sturen.

Bovendien is voor deze architectuur gekozen zodat iedere microservice apart ontwikkeld kan worden. Aangezien de projectgroep uit vier studenten bestaat, 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.

**Architectuur van de applicatie**

Afbeelding : Architectuur applicatie

De globale architectuur van de applicatie is in de bovenstaande afbeelding (Afbeelding 3) weergeven. Hierin zijn de microservices terug te vinden, met de bijbehorende databases. Alle microservices zijn, bij toeval, in Node.js geschreven. Alle microservices communiceren met de RabbitMQ servers om middels Events hun staat door te geven aan andere microservices. Een scherp oog valt op dat de Events van de Support Service door geen enkele microservice worden ontvangen. Dit is met het oog op de toekomst, mocht er uitbreiding van het systeem plaats vinden waarbij Events van de Support Service nodig zijn.

In bovenstaande afbeelding valt ook op dat het CQRS-principe is toegepast in de Catalog Service, waarbij updates van de catalog worden wegeschreven in een MongoDB, maar dezelfde data wordt uitgelezen in een MySQL tabel. In hoofdstuk 5.4 komt hier meer over terug.

## 5.2 Eventual consistency

Eventual consistency wordt in onze solution gewaarborgd door de RabbitMQ queues. Een aantal services luisteren namelijk naar events van andere services om hun eigen data bij te werken, een aantal voorbeelden zijn: de catalog-query-service luistert naar de catalog-command-service en de order-service luistert naar zowel de catalog-command-service en de transporter-service. Het wordt pas interessant wanneer één van die services tijdelijk niet beschikbaar is. Dan zouden normaal alle messages verloren raken en zou de data out-of-sync gaan. Momenteel is de data ook out of sync, maar omdat alle messages bewaard blijven op de queue gaat de data uiteindelijk (wanneer alle services weer online zijn) weer consistent zijn. Pas als een message verwerkt is door een service wordt deze acknowledged en verwijderd uit de queue.

## 5.3 Event driven architecture

Event driven architecture is gerealiseerd doormiddel van een message broker, in ons geval RabbitMQ. In het geval dat er een verandering plaatsvind in één van de services gaat er automatisch een bericht met de verandering de exchange op. Er is gekozen voor een topic exchange en dat betekend dat alle events een key meekrijgen, en dat queues doormiddel van een soort regex kunnen kiezen wat voor soort events ze ontvangen.

In onderstaande tabel (tabel 4) zijn alle events zichtbaar die bestaan binnen onze solution. Het is belangrijk hierbij op te merken dat er momenteel niet op al deze events gelistened wordt, en dat is ook niet nodig. Doordat deze events sowieso op de exchange geplaatst worden kunnen teams onafhankelijk van elkaar ontwikkelen en is het systeem toekomstbestendiger dan anders.

|  |  |
| --- | --- |
| **Event** | **Beschrijving** |
| catalog.product.created | Er is een product aangemaakt |
| catalog.product.deleted | Er is een product verwijderd |
| catalog.tpv.created | Er is een third party vendor aangemaakt |
| catalog.tpv.deleted | Er is een third party vendor verwijderd |
| support.created | Er is een nieuw support ticket aangemaakt |
| support.replied | Er is gereageerd op een bestaand support ticket |
| support.closed | Er is een support ticket afgerond |
| order.created | Er is een nieuw order gemaakt |
| order.edited | Een bestaand order is bewerkt |
| transporter.created | Een nieuwe transporter is gemaakt |
| transporter.deleted | Een nieuwe transporter is verwijderd |
| transporter.assigned | Een nieuwe transporter is gevonden om een order te versturen |

Tabel ) Alle events zoals gebruikt in onze solution.

## 5.4 Command Query Responsibility Segregation (CQRS)

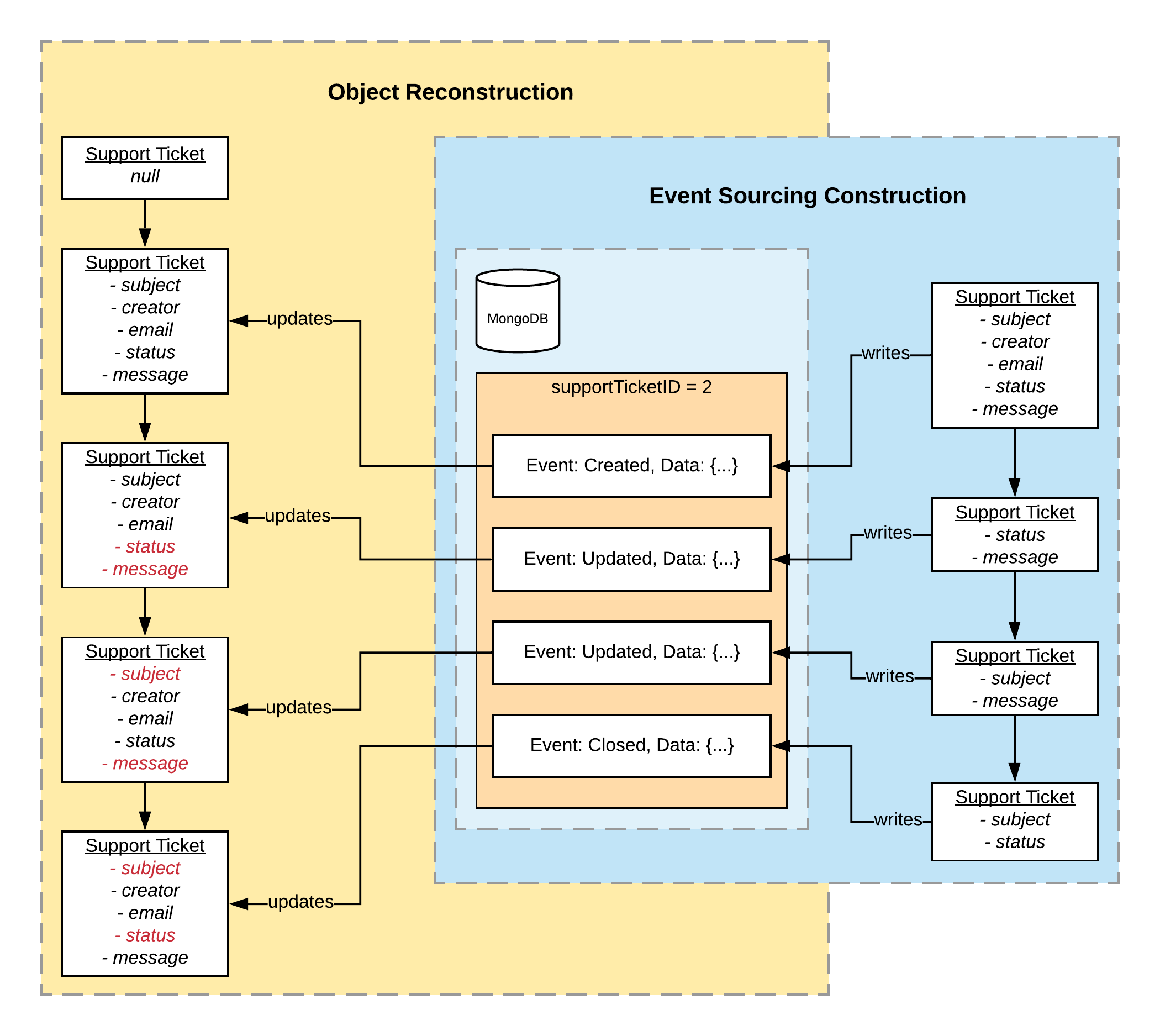
Command Query Responsibility Segregation (CQRS) is een manier om availability en scalability van een applicatie te vergroten, performance te verbeteren en de complexiteit te verminderen. Er is voor gekozen om CQRS toe te passen op de catalog service, deze service is verantwoordelijk voor de producten in de catalogus en te zoeken op de naam.

Er is gekozen om CQRS toe te passen op de catalog-service omdat er in verhouding veel meer reads gedaan worden op een catalogus dan dat er producten toegevoegd, bewerkt of verwijderd worden. Door dit te splitsen en twee databases te gebruiken kan er één geoptimaliseerd worden voor lezen, in dit geval een MySQL database met indexes op de \_id en name column. De andere is dan geoptimaliseerd op het genormaliseerd en netjes opslaan en valideren van de data, in dit geval een MongoDB.

De reden dat er voor twee databases en twee services gekozen is, is om de schaalbaarheid te verbeteren. Door deze architectuur is het namelijk gemakkelijk om de reads op de catalog horizontaal te schalen. Het is namelijk mogelijk om zoveel MySQL databases en catalog-query-services te deployen als je wilt doordat de MySQL databases bijgewerkt worden door events vanuit de command service. Een bijkomend voordeel hiervan is een verbeterde availability, als je tien identieke query services hebt draaien is de kans dat ze allemaal tegelijk unavailable zijn minimaal.

## 5.5 Event sourcing

Event Sourcing in de applicatie komt voor in de Support Service. In deze service vindt de afhandeling van support tickets plaats. Een klant of bezoeker van een support ticket indienen. Naar verloop van tijd kijkt een medewerker naar deze ticket en bepaald aan de hand daarvan de actie die ondernomen moet worden. Hij update hiervoor de ticket zodat de klant weet wat de laatste status van zijn ticket is. Bij het ‘verwijderen’ van een ticket wordt zijn status op ‘closed’ gezet, zodat de klant of indiener van de support ticket later nog kan terugkijken.

Om de geschiedenis van een support ticket te kunnen bewaren, en om de staat van een support ticket te kunnen opbouwen naar een bepaald moment in de tijd, is er gekozen om het principe ‘Event Sourcing’ toe te passen. In Afbeelding 4 is te zien hoe dit in zijn werk gaat. Een nieuwe support ticket kan worden aangemaakt via de POST/support endpoint. Deze nieuwe ticket wordt opgeslagen in de MongoDB, waarbij het supportTicketID gelijk staat aan de door MongoDB gegenereede ‘*\_id’* waarde. In dit document worden de verschillende events opgeslagen, met als eerste event het creëren van de support ticket. Nieuwe events worden gekoppeld in dit document. Bij het ophalen van het meest actuele support ticket object worden alle events uit een support ticket gelezen. De waardes die per event veranderen worden geüpdate, en dit uiteindelijke object wordt verstuurd. Elk event heeft een timestamp, dus de staat van een support ticket kan worden hersteld naar elk gewenst punt in tijd

Afbeelding : Weergave Event Sourcing

## 5.6 Enterprise integration patterns

Het kan voorkomen dat er in de toekomst een applicatie van buitenaf moet kunnen integreren met de applicatie die door ons ontwikkeld is. Denkbare voorbeelden omvatten maar zijn niet beperkt tot: een extern betalingssysteem, voorraadsysteem of een administratieve applicatie. Het is zeer aannemelijk dat dit systeem niet direct mee kan werken met de door ons opgezette applicatie. Er is hier dus sprake van een ad hoc applicatie integratie.

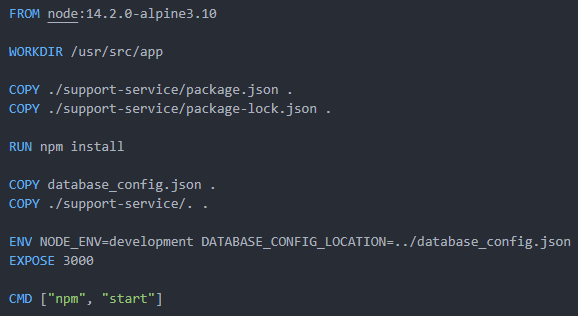
We gaan ervan uit dat we de applicatie kunnen benaderen met RabbitMQ, is dit niet het geval zal er een adapter applicatie moeten komen die wel met de te integreren applicatie kan communiceren. Om deze applicatie te integreren gaan we gebruik maken van integratie patterns.

Een voorbeeld van een integratie pattern is een splitter deze splitst een bericht met een array op in een aparte berichten met één item uit de array. Een ander voorbeeld is een aggregator, die data uit meerdere berichten combineert tot één bericht. Verder zijn er nog content enrichers, message translators en vele andere patterns die allemaal één ding doen: namelijk het veranderen van de data zodat de te integreren applicatie er mee kan werken.

In ons geval zouden wij dit integreren door een service toe te voegen die luistert naar alle berichten die benodigd zijn, intern de berichten aanpast en publiceert in een formaat met een key waar de te integreren applicatie naar luistert. In het geval dat we een schakelapplicatie nodig hebben omdat de te integreren applicatie niet met RabbitMQ kan werken zou dit dezelfde applicatie kunnen zijn.

## 5.7 Containerization

Elke microservice draait in zijn eigen Docker container. Deze Docker container wordt opgebouwd met een eigen Dockerfile. Een fragment van zo’n Dockerfile is te vinden in Afbeelding 5. De context van de Dockerimage ligt in de root van het project (1 map boven de map van de microservice). Dit is vanwege de database\_config.json die elke microservice nodig heeft. Naast deze 4 microservices die hun eigen Dockerfile hebben, draaien er nog 3 andere containers: 1 voor de MongoDB server; 1 voor de MySQL server en 1 voor de RabbitMQ server. Om te voorkomen dat de images van Docker te groot worden, worden de node\_modules niet gekopieerd middels een .dockerignore bestand.

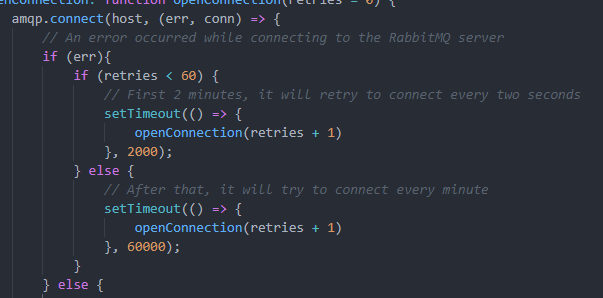


Afbeelding 5: Dockerfile van Support Service

Om deze containers met elkaar te laten communiceren (zoals met de MongoDB server bijvoorbeeld) is gebruik gemaakt van Docker Compose. Docker compose is een tool van Docker om meerdere containers op dezelfde host te laten draaien. Hierbij wordt gebruik gemaakt van een speciaal intern netwerk waarmee containers onderling kunnen communiceren. De structuur van deze containers en het onderlinge netwerk wordt gedefinieerd in de docker-compose.yaml. Een fragment van dit bestand is te zien in Afbeelding 6. De containers kunnen onderling met elkaar communiceren aan de hand van de ‘service name’ (de eerste regel van de afbeelding, mongodb, bijvoorbeeld).

Afbeelding 6: Fragment docker-compose.yaml

Het kan voorkomen dat een aantal microservices eerder zijn opgestart dan zijn afhankelijkheden, zoals de RabbitMQ server. Gebleken is dat de “depends\_on” waarde van de docker-compose wel werkt als het gaat om wachten tot de container draait, maar deze waarde wacht niet tot de server zelf daadwerkelijk is opgestart. Hier is een retry-mechanisme voor geschreven, die een aantal keer probeert verbinding te maken met de RabbitMQ container zonder de applicatie te laten crashen. Een uitwerking hiervan is te vinden in Afbeelding 7



Afbeelding 7: Retry mechanisme