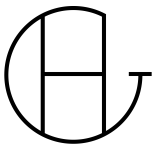


HoGent

Faculteit Bedrijf en Organisatie

Onderzoek naar de meerwaarde van EventSourcing als gegevensbeheer techniek bij Skedify, waar ze gespecialiseerd zijn in online afspraak beheersoftware.

Robin Malfait

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

Promotor:
Lieven Smits
Co-promotor:
Mathias Verraes

Instelling: —

Academiejaar: 2016-2017

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Onderzoek naar de meerwaarde van EventSourcing als gegevensbeheer techniek bij Skedify, waar ze gespecialiseerd zijn in online afspraak beheersoftware.

Robin Malfait

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

Promotor:
Lieven Smits
Co-promotor:
Mathias Verraes

Instelling: —

Academiejaar: 2016-2017

Tweede examenperiode

Samenvatting

De maatschappij draait om data. Data en informatie zijn geld waard. Data zijn de gegevens die gepersisteerd zijn in een databank. Informatie is een afgeleide van deze data. Daarom is het van belang dat er voorzichtig met data wordt omgegaan zodat er geen gegevens verloren gaan bij mutaties in het systeem. In systemen waar relationele modellen gebruikt worden gaat er data verloren wanneer er niet genoeg nagedacht wordt. Bij elke wijziging of verwijdering van een rij, is de voorgaande informatie die ter beschikking was verloren gegaan. Het probleem hierbij is dat deze databank gebruikt wordt als single source of truth. Dit kan echter opgelost worden door logs te gebruiken, maar vermits deze niet de single source of truth zijn, en er geen huidige staat van berekend wordt is de kans groter dat deze fouten bevat of onbruikbaar is om de oude gegevens te achterhalen. Het kan ook zijn dat er gegevens niet gelogd werden door een vergeetachtigheid van een developer. Door gebruik te maken van EventSourcing gaat er geen data verloren. Dit komt omdat EventSourcing gebruik maakt van een EventStore (zie Hoofdstuk 5.7) als single source of truth waar de huidige staat van berekend werd. Elke actie die gebeurd wordt eerst gepersisteerd in de EventStore, en daarna worden de projecties (zie Hoofdstuk 5.9) pas uitgevoerd. Het is van belang om informatie bij te houden, dit kan altijd interessant zijn in de toekomst wanneer er specifieke business vragen komen waar er momenteel nog geen antwoord op is. Eerst werd er een literatuurstudie gedaan omtrent EventSourcing en zijn voor- en nadelen. Daarna wordt er gekeken naar alles wat er nodig is om aan een EventSourcing applicatie te beginnen. Dit gaat van Command Query Separation (CQS), Command Query Responsibility Segregation (CQRS) tot de effectieve onderdelen van EventSourcing. Er wordt ook uitleg gegeven over Skedify zodat de concrete businesscase duidelijk gescoped wordt.

Voorwoord

Ik ben al een 5 jaar lang gepassioneerd door data en informatie. Door gebruik te maken van EventSourcing gaat er geen informatie verloren, en is er een mogelijkheid om te tijdreizen naar het verleden tot op het heden. Het idee van deze bachelorproef kwam uit gesprekken met een aantal mensen via sociale media Twitter. Deze gesprekken waren met een aantal mensen die nu bekend zijn in de DDD (Domain Driven Design) en EventSourcing wereld. Een van deze mensen in Shawn McCool, die het platform https://eventsourcery.com heeft gemaakt, wat een introductie is tot domain modeling, CQRS (Command Query Responsibility Segregation, zie Hoofdstuk 4) en EventSourcing (zie Hoofdstuk 5).

Dankzij Twitter heb ik ook mijn co-promotor, Mathias Verraes leren kennen. Ik heb hem ook ontmoed op een conferentie in Nederland, namelijk Laracon EU 2014. Ik zou hem heel graag willen bedanken voor het realiseren van deze bachelorproef. Ik kon altijd met al mijn vragen raad bij hem, en hij zorgde bezorgde mij ook de nodige boeken, blog posts en andere resources omtrent EventSourcing. Ik kreeg ook de kans om naar meetups (mini conferenties waar 2 à 3-tal mensen een presentatie gegeven omtrent een bepaald onderwerp en waar je aan netwerking kan doen) te gaan, maar dit was niet altijd even gemakkelijk omdat ze niet altijd bij de deur plaatsvonden.

Ik zou ook graag de product owner van Skedify bedanken, Christophe Thelen om mij interessante gegevens over Skedify te bezorgen. Alsook resources te bezorgen in verband meet EventSourcing omdat hij hier ook al in de praktijk mee gewerkt heeft.

Daarnaast zou ik ook graag mijn promotor, Lieven Smits willen bedanken voor het benadrukken van mijn spellingsfouten om deze bachelorproef tot een goed einde te brengen.

Inhoudsopgave

1. Inleiding

1.1 Stand van zaken

1.2 Probleemstelling en Onderzoeksvragen

Data is het belangrijkste gegeven van een bedrijf. Rapportering is van belang om de eisen van de klant nog beter te vervullen. Voorspellen van vragen die de business kan vragen binnen dit en 5 jaar is de dag van vandaag moeilijk. Daarom is het interessant om alle informatie bij te houden om hier interessante rapportering op te kunnen uitvoeren. In een systeem met een standaard relationele databank wordt niet alle informatie bijgehouden. Bij het verwijderen of wijzigen van informatie is de voorgaande informatie verloren gegaan. Alle informatie bijhouden is dus interessant, maar wat zijn hier de voor en nadelen van?

De onderzoeksvraag luidt als volgt: Wanneer is EventSourcing een meerwaarde voor een bedrijf zoals Skedify (Hoofdstuk 3), waar ze gespecialiseerd zijn in online afspraakbeheer-software?

Hierbij worden ook volgende subvragen beantwoord:

- Wat is het verschil tussen een systeem met relationele databank modellen en een systeem met EventSourcing?
- Hoe kan een applicatie die gebruik maakt van EventSourcing getest worden?
- Welke delen moeten er EventSourced worden, en hoe kan je deze delen bepalen?

1.3 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 11, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Methodologie

Voor er een proof of concept gemaakt kan worden in verband met EventSourcing, moet men eerst kijken naar wat EventSourcing is en wat het inhoudt. Naast EventSourcing zijn er ook concepten zoals CQRS die even aan bod moeten komen omdat dit een interessante rol speelt bij EventSourcing. CQRS is een splitsing van de lees- en schrijfkant in een systeem. Binnen EventSourcing heb je ook Commands (schrijfkant) en Queries (leeskant), en daarom is dit een goed hoofdstuk om binnen deze bachelorproef op te nemen. Daarnaast draait het in dit onderzoek rond Skedify (Hoofdstuk 3), het bedrijf die als concrete case van toepassing is voor deze bachelorproef. In het deel rond Skedify, kan er meer te weten gekomen worden over wat ze doen. Daarna zal er gekeken worden naar de huidige manier van werken bij het bedrijf omtrent hunt data. Tot slot gaat alles worden samengevoegd: Skedify en de huidige manier van werken, en Skedify en EventSourcing. Hierna worden de voor- en nadelen van beide mogelijkheden afgegaan om te zien of EventSourcing nu effectief een meerwaarde biedt voor Skedify. Het onderzoek is in grote delen opgesplitst: Skedify, EventSourcing, huidige manier van werken en de conclusies.

In het deel over Skedify zal er onderzocht worden wat ze doen, en hoe ze dit doen. Er zal ook een gesprek komen met de business kant van Skedify om vragen te kunnen beantwoorden en om duidelijke conclusies te kunnen trekken.

In het deel over EventSourcing zal er dieper onderzocht worden hoe EventSourcing juist werkt en hoe het gebruikt kan worden.

2.1 Literatuurstudie

De eerste fase van dit onderzoek start met het vergaren van informatie door middel van een literatuurstudie. Het doel van deze literatuurstudie is om een basisinzicht en kennis te verkrijgen met betrekking tot EventSourcing. Alvorens een antwoord op de onderzoeksvragen kan geformuleerd worden moet er vanzelfsprekend een basiskennis zijn over de belangrijkste zaken die te maken hebben met EventSourcing. Het is van belang om eerst de basis principes van het CQRS en EventSourcing te begrijpen.

2.2 Proof of concept

Voor de proof-of-concept zullen bepaalde functies van Skedify geschreven worden in het huidige systeem en in een EventSourcing systeem. Daarbij zal er gekeken worden naar voordelen en nadelen van beide systemen. De zaken waar rekening mee gehouden zal worden zijn de volgende:

- Hoe lang werd er gewerkt aan een bepaalde feature?
- Hoe leesbaar/onderhoudbaar is de code?
- Hoe kan een bepaalde feature getest worden?
- Is de feature een 1-op-1 mapping met de business?

3. Skedify

Skedify is een Gentse tech start-up dat online afspraakbeheersoftware ontwikkelt voor onder andere de banksector, ziekenfondsen, verzekeringen, immobiliën en HR-bedrijven. Deze bachelorproef gaat op zoek naar een antwoord of EventSourcing een meerwaarde kan bieden voor Skedify. Skedify zorgt er voor dat een persoon in eender welke sector een afspraak kan maken op een korte tijd voor eender welk probleem, en dit met de juiste contact persoon van het bedrijf. Dit moet in een zo kort mogelijke tijd verlopen. Skedify is een business-to-business tool, die onder andere dynamisch formulieren kan gaan opbouwen om te gebruiken in de plugin die op de website van de klant terecht komt. Alles wordt in een relationele databank opgeslagen.

Een van de mogelijkheden die Skedify bied is het wijzigen van een afspraak, het kan interessant zijn om te weten hoeveel keer dit gebeurt en of er specifieke maatregelen kunnen genomen worden.

Rapportering is tot op de dag vandaag nog niet beschikbaar omdat er geen historiek wordt bijgehouden.

4. CQRS

CQRS is een term uitgevonden door Greg Young, de heer Young gaat ook verder in op EventSourcing. Een goede uitleg hieromtrent is te vinden in een presentatie die hij gaf op een conferentie (Young, 2014). CQS kort voor Command Query Seperation, is een principe die uitgevonden is door Bertrand Meyer (Meyer, 1988). Het is een API design principe dat beschrijft hoe er met een object of een systeem gecommuniceerd moet worden. CQRS daarentegen is een architectuurstijl die gaat over hoe het aan de binnenkant geïmplementeerd is. Als er gekeken wordt naar CQS is dit al een eerste vorm van goede, overzichtelijke code schrijven. CQS zorgt er voor dat getters en setters gescheiden zijn. Getters zijn louter bedoeld om een waarde uit de huidige state te halen en deze terug te geven. Setters zijn bedoeld om een wijziging te doen (of een algemene actie uit te voeren), setters geven geen waarde terug maar void, dit principe wordt ook uitgelegd op de blog van Martin Fowler (Fowler, 2005a). Een goede heuristiek voor het toepassen van CQS is: Ëen vraag stellen, mag het antwoord niet wijzigen.¨ met andere woorden, een getter heeft nooit gevolgen op de huidige state of op andere zaken.

Het doel van CQS is vooral de leesbaarheid verhogen, het is een principe die developers gebruiken en begrijpen waardoor communicatie makkelijker gaat. Een tweede probleem is dat getters en setters in een en dezelfde klasse gedefinieerd zijn, voor eenvoudige klassen is dit geen probleem, maar wanneer er gelezen of geschreven wordt naar een databank is dit wel een probleem... Er is geen strikte scheiding tussen de lees kant en de schrijf kant.

Listing 4.1: Voorbeeld van CQS

```
1 class Appointment {
2   private DateTime start;
3   private DateTime end;
4   private String subject;
5
6   public DateTime getStart() {
7     return this.start;
```

```
8     }
9
10    public DateTime getEnd() {
11      return this.end;
12    }
13
14    public String getSubject() {
15      return this.subject;
16    }
17
18    public void setStart(DateTime start) {
19      this.start = start;
20    }
21
22    public void setEnd(DateTime end) {
23      this.end = end;
24    }
25
26    public void setSubject(String subject) {
27      this.subject = subject;
28    }
29  }
```

In dit voorbeeld zijn getters en setters strikt gescheiden. Het opvragen van een datum of subject wijzigt niets aan de huidige state van dat object. Wanneer deze klasse ook verbonden is aan een databank, dan zullen er zich problemen vormen, wat te zien is in het volgende voorbeeld.

Listing 4.2: Voorbeeld van CQS met databank

```
1   @Entity
2   @Table(name="appointments")
3   class Appointment {
4     private DateTime start;
5     private DateTime end;
6     private String subject;
7
8     public DateTime getStart() {
9       return this.start;
10    }
11
12    public DateTime getEnd() {
13      return this.end;
14    }
15
16    public String getSubject() {
17      return this.subject;
18    }
19
20    public void setStart(DateTime start) {
21      this.start = start;
22    }
23
24    public void setEnd(DateTime end) {
25      this.end = end;
26    }
27
28    public void setSubject(String subject) {
29      this.subject = subject;
30    }
31  }
```

Er is nu geen manier om de lees- en schrijfkant los te koppelen van elkaar. De Appointment klasse zal zowel voor het persisteren van data met deze databank verbinden als voor het uitlezen van gegevens. Er is momenteel geen manier om te bepalen dat het lezen van data

via een andere database connectie moet gaan. Het lezen en schrijven, is gekoppeld aan deze klasse waardoor er geen scheiding mogelijk is.

Dit is waar CQRS komt kijken, Command Query Responsibility Segregation. CQRS zorgt er voor dat de lees- en schrijfkant kant strikt gescheiden zijn. Het zijn bijna 2 applicaties die naast elkaar staan. Een Command wordt afgehandeld aan de schrijfkant en een Query wordt afgehandeld aan de leeskant. Zowel een Command als een Query zijn messages, dit wordt verder besproken in hoofdstuk 5.3. De leeskant gaat zijn informatie halen bij de databank (of een andere vorm van opslagmechanisme), dit kan via sql queries, ORM tools, enzovoort. De manier waarop dit gebeurt staat volledig los van hoe de schrijfkant communiceert met het opslagmechanisme.

De meeste applicaties, onder andere ook die van Skedify zijn intensiever aan de leeskant dan aan de schrijfkant (Tabel 4.1). De lees- en schrijfkant zijn nu strikt gescheiden en er kan gebruik gemaakt worden van schalingsmechanismen. Beter nog, de leeskant en schrijfkant kunnen individueel geschaald worden. Er kan zelfs geopteerd worden om lees- en schrijfkant in verschillende programmeertalen te schrijven.

Tabel 4.1: Aantal requests vergeleken voor de lees- en schrijfkant. Er zijn 5.548 (~18.02%) keer zoveel lees requests dan schrijf requests.

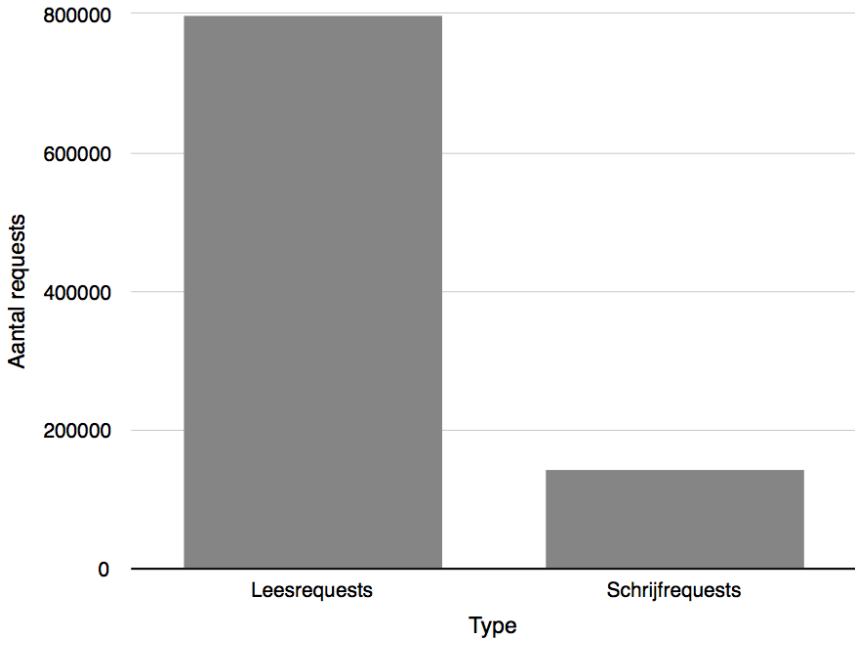
Methode	# Requests	Type
GET	612 000	LEZEN
OPTIONS	148 000	LEZEN
POST	129 000	SCHRIJVEN
HEAD	362 000	LEZEN
PATCH	13 400	SCHRIJVEN
DELETE	1 100	SCHRIJVEN

Door de zwaardere druk langst de leeskant, kan er moeilijk geoptimaliseerd worden voor alle cases waar zowel de lees- als schrijfkant voordeel uit haalt. De splitsing is daarom een voordeel om langs beide kanten te kunnen optimaliseren.

Wanneer de lees- en schrijfkant niet gescheiden zouden worden, dan maken ze gebruik van de dezelfde databank. Wanneer er gelezen wordt en door een andere partij geschreven wordt, kan dit elkaar hinderen omdat beide kanten dezelfde resources gebruiken.

CQRS speelt en grote rol bij EventSourcing, vandaar dit korte hoofdstuk.

Figuur 4.1: Visuele representatie van tabel 4.1



5. EventSourcing

EventSourcing is geen nieuwe uitvinding, maar wordt al jaren gebruikt in andere sectoren. Bekende voorbeelden zijn de bankindustrie, wetgeving, en patiënten fisches die opgeslagen zijn bij dokters. Onze wetgeving is gebasseerd op de grondwet, elke nieuwe wet is een addendum op deze basis wetgeving („Loi - Wet”, g.d.).

EventSourcing, is zoals de naam al verklapt, dat events de bron zijn van de applicatie. Het is ook zo dat de events de s̈ingle source of truthz̈ijn van de applicatie om andere state uit af te leiden. Een log bestaat ook uit een geschiedenis van events, maar deze wordt niet gebruikt als s̈ingle source of truthöp de huidige state af te leiden.

EventSourcing wordt door Microsoft aanzien als een patroon. Microsoft heeft onderzoek gedaan naar wanneer je dit patroon het best gebruikt en wanneer niet (Microsoft, 2017).

Zoals eerder vermeld, is EventSourcing niet nieuw voor sectoren zoals de bankindustrie, daarom zijn veel voorbeelden te vinden over bank transactions en e-commerce, zoals te vinden is bij Microsoft (2017) als voorbeeld. In deze bachelorproef wordt er naar Skedify gekeken, wat niets te maken heeft met de standaard voorbeelden die te vinden zijn.

Als er naar de bankindustrie wordt gekeken dan is het bedrag op iemand zijn rekening niet een getal dat enkel en alleen opgeslagen is in een databank. Er is een lijst van transacties die tot dit getal komen. Het getal op je zichtrekening zit ook opgeslagen in een databank, maar dit is puur als caching mechanisme, zodat men niet elke keer opnieuw alle transacties moet afgaan om dit getal te bepalen.

Het is zo dat nu pas, de laatste jaren, EventSourcing opkomt in de informatica sector. In de volgende hoofdstukken zal er dieper ingegaan worden op onderdelen van EventSourcing.

5.1 Aggregates

Aggregate is een patroon die in de DDD (Domain Driven Development) wereld gebruikt wordt Fowler (2005b). Aggregaten zijn de entiteiten die behoren tot het domein. Een aggregaat is een entiteit die zal zorgen dat invariants (besproken in 5.5) gegarandeerd zullen worden. Eens dit het geval is zal die er voor zorgen dat de juiste domain events opgeslagen worden en de state van de applicatie wordt gewijzigd.

Deze aggregaten moeten correct gekozen worden, zodat er geen gevaar is dat er 'god' aggregaten ontstaan. Een 'god' aggregaat is een aggregaat die alles in een applicatie doet, in de meeste gevallen is dit een 'User' aggregaat.

In het geval van Skedify zijn goede aggregaten: Appointment, Subject, Question, Answer, Office, ... omdat deze core concepts zijn bij Skedify.

5.2 Value Objects

Value objects zijn objecten die zorgen dat een bepaalde waarde altijd in een geldige staat is. Bijvoorbeeld, er kan een value object zijn voor een Email. Dit email adres moet ten alle tijden geldig zijn. Dit kan afgedwongen worden door in de constructor van een klasse een waarde te ontvangen en deze te controleren. Indien deze waarde fout is wordt er een exceptie gegooid. Deze exceptie zal dan opgevangen worden in de lagen daarboven.

Een belangrijke regel bij Value objects is dat deze geen identiteit bevatten, ze hebben geen unieke identifier en zijn daarom geen entiteit. Value objects kunnen wel tot een entiteit behoren zoals Bijvoorbeeld een User entiteit.

Value objects bevatten ook bepaalde logica of methoden die met dat object gepaard gaan.

Value objects geven ook betekenis aan de code. In het volgende code voorbeeld, is het niet meteen duidelijk welke parameter de verzender is, en welke de ontvanger is. Een mogelijke oplossing is goede namen geven aan de parameters.

Listing 5.1: Een minder goed voorbeeld van Value Objects

```
1  // 1. slecht
2  class AddAppointment {
3    private String person1;
4    private String person2;
5    private DateTime date;
6
7    // Wanneer er een slechte naamkeuze wordt gemaakt in de naam
8    // van de variabele, kan dit tot bugs lijden.
9    public AddAppointment(String person1, String person2, DateTime date) {
10     this.person1 = person1;
11     this.person2 = person2;
12     this.date = date;
13   }
14 }
15
16 // 2. beter
17 class AddAppointment {
18   private String agent;
```

```
19    private String client;
20    private DateTime date;
21
22    // Dit is al beter leesbaar, het gevaar bestaat er in agen & client
23    // Van plaats te verwisselen. Bij 2 geldige strings zal dit dus geen
24    // Exceptie gooien
25    public AddAppointment(String agent, String client, DateTime date) {
26      this.agent = agent;
27      this.client = client;
28      this.date = date;
29    }
30  }
```

Wanneer er nu een AddAppointment object gemaakt wordt, is er wel nog steeds het gevaar de agent en de klant om te draaien. Als we geldige strings gebruiken, zal de code geen exceptie gooien en zit er een bug in het systeem.

Dit probleem kan opgelost worden door Value objects te gebruiken. Bij het verwisselen van de parameters zal er een exceptie gegooid worden omdat de types niet overeen komen.

Listing 5.2: Een voorbeeld van Value Objects

```
1   class Id {
2     private GUID id;
3
4     public Id(GUID id) {
5       this.id = id;
6     }
7
8     public GUID getId() {
9       return this.id;
10    }
11
12    public void setId(GUID id) {
13      this.id = id;
14    }
15  }
16
17  class ClientId extends Id {}
18  class AgentId extends Id {}
19
20  class AddAppointment {
21    private AgentId agent;
22    private ClientId client;
23    private DateTime date;
24
25    public AddAppointment(AgentId agent, ClientId client, DateTime date) {
26      this.agent = agent;
27      this.client = client;
28      this.date = date;
29    }
30  }
```

5.3 Messages

Communicatie in een applicatie draait rond messages. Er zijn 3 soorten messages die terug komen in een applicatie: informational, interrogatory en imperative. Verraes (2015) Binnen een applicatie met EventSourcing wordt er ook gebruik gemaakt van messages.

5.3.1 Imperative Messages

De imperative of imperatieve messages, zijn messages die de intentie hebben om de ontvanger van dit bericht een actie te laten uitvoeren. Binnen EventSourcing zijn is een typisch voorbeeld een Command, waarbij het Command de intentie heeft om een actie uit te voeren. Binnen een systeem zijn deze Commands iets typisch dat uit de business kant komt. Wanneer het volgende voorbeeld door iemand van de business gelezen wordt, weet die ook meteen wat er zal gebeuren, als er gewerkt wordt met deze Command.

Listing 5.3: Een voorbeeld van een Command

```
class ChangeAppointmentStartDate {
  private AppointmentId appointmentId;
  private DateTime start;

  public ChangeAppointmentStartDate(
    AppointmentId appointmentId,
    DateTime start
  ) {
    this.appointmentId = appointmentId;
    this.start = start;
  }

  public AppointmentId getAppointmentId() {
    return this.appointmentId;
  }

  public DateTime getStart() {
    return this.start;
  }
}
```

Wanneer er met deze klasse gewerkt wordt (of met objecten van deze klasse), dan wordt er verwacht dat de ontvanger iets zal wijzigen.

5.3.2 Interrogatory Messages

De interrogatory of ondervragende messages, zijn messages die iets vragen over de huidige state van de applicatie. In elke applicatie worden er zaken opgevraagd. Binnen EventSourcing wordt dit ook gedaan aan de hand van Queries, deze Queries gaan informatie opvragen van de huidige state. Deze queries worden ook opgelegd door de business kant, de business kan, wanneer ze het voorbeeld zien, meteen begrijpen wat er zal gebeuren als er met deze Query wordt gewerkt omdat het een 1-op-1 vertaling is van de business.

Listing 5.4: Een voorbeeld van een Query

```
class MyAppointmentsForToday {
  private UserId userId;
  private Date today;

  public MyAppointmentsForToday(UserId userId) {
    this.userId = userId;
    this.today = new Date();
  }
}
```

Bij dit voorbeeld is het duidelijk det er informatie opgevraagd wordt en dat er geen wijzigingen aan de huidige state zullen gebeuren.

5.3.3 Informational Messages

De informational of de informatieve messages, zijn messages die iets over zichzelf willen vertellen. Bij EventSourcing wordt dit uitgedrukt in DomainEvents. DomainEvents zijn de messages die opgeslagen zullen worden in de databank. DomainEvents worden ook meestal in de verleden tijd geschreven omdat ze benadrukken dat er iets gebeurt is.

Listing 5.5: Een voorbeeld van een DomainEvent

```
1  class AppointmentsStartDateHasBeenChanged {
2    private AppointmentId appointmentId;
3    private DateTime start;
4
5    public AppointmentsStartDateHasBeenChanged(
6      AppointmentId appointmentId,
7      DateTime start
8    ) {
9      this.appointmentId = appointmentId;
10     this.start = start;
11   }
12
13   public AppointmentId getAppointmentId() {
14     return this.appointmentId;
15   }
16
17   public DateTime getStart() {
18     return this.start;
19   }
20 }
```

Dit voorbeeld lijkt sterk om het voorbeeld van de Command, maar de intentie van beide klassen is totaal verschillend. De klasnamen zijn altijd expliciet, soms zijn ze wat lang maar het is duidelijk wat de bedoeling is. Doordat de messages zo expliciet geschreven zijn, kan er gemakkelijk mee gecommuniceerd worden wat het hele doel is van messages.

Er mag ook geen gemeenschappelijke naam gezocht worden voor deze klassen om 'dubbele code' te verwijderen. Het zijn totaal aparte concepten, zelfs wanneer verschillende commands tot hetzelfde resultaat lijden, moet dit niet vervangen worden door 1 command (Verraes, 2014). Vanaf dit gebeurt komen er problemen naar boven. Wat als er extra informatie in een Command moet maar niet in een DomainEvent of vice versa. De leesbaarheid van de code gaat ook achteruit, de communicatie wordt onduidelijk.

5.4 Domain Events

Domain events zijn een essentieel onderdeel van EventSourcing. Domain Events zijn, net als commands and queries, messages. Domain Events zijn de effectieve events die zullen opgeslagen worden in een append-only database. Een event is iets dat gebeurt is en nooit meer kan veranderen. Er zijn een paar belangrijke eigenschappen aan deze events.

- Ze bevatten een unieke id, die op voorhand vastgelegd is. Dit kan een Globally Unique Identifier (GUID) zijn.
- Ze bevatten enkel de data die gewijzigd is ten opzichte van de vorige versie.
- Alle data die ze bevatten, is correct en kan niet meer aangepast worden.

Domain events worden opgeslagen in een append-only database, maar wat als er een fout gemaakt is gemaakt? Indien een fout is opgetreden, moet er een nieuw domain event gemaakt worden, die het vorige event corrigeert. Op deze manier blijft al de data correct, en gaat er geen belangrijke informatie verloren. Er is ook niet geprutst met de historiek van deze events, wat van belang is omdat de geschiedenis herschrijven gevolgen kan hebben voor een systeem. Het houdt in dat de audit log (Hoofdstuk 5.8) niet meer geldig is. Het houdt ook in dat elke projectie opnieuw zal moeten opgebouwd, omdat een wijziging in 1 event een totaal andere uitkomst kan creëren in de huidige state.

Domain events hebben ook een naam, deze naam is specifiek voor de use case. Het verteld meer over wat er gebeurt is of wat er zal gebeuren. Het is ook aangeraden dat deze naam in de verleden tijd is opgesteld. Zo kan er gedifferentieerd worden tussen de soorten messages (Hoofdstuk 5.3) en kan de leesbaarheid verhoogd worden naar andere partijen toe (developers, business). Een event is tenslotte gebeurd in het verleden. Als er in context van Skedify gesproken wordt, dan is AppointmentWasRescheduled een goede naam voor een domain event. Het is ook van belang dat er geen CRUD (Create Read Update Delete) events gemaakt worden zoals OfficeWasCreated, want er werd niet effectief een office gemaakt in het echte leven, er is een office toegevoegd die bestaat in het echt en die nu aan de applicatie is toegevoegd. Een betere naam zou zijn OfficeWasAdded.

5.5 Invariants

Invariants zijn regels die opgelegd zijn door de business. Invariants worden gecontroleerd alvorens een domain event opgeslagen wordt. Elke invariant moet goedgekeurd worden, want eens een domain event opgeslagen is, kan dit niet meer ongedaan gemaakt worden. Er kan wel een nieuw domain event opgeslagen worden om deze wijziging te niet te doen. Invariant controle wordt uitgevoerd bij het triggeren van een command, telkens wanneer een input niet aan deze invariant voldoet moet er een exception gegooid worden. Op deze manier kunnen er inconsistenties opgevangen worden.

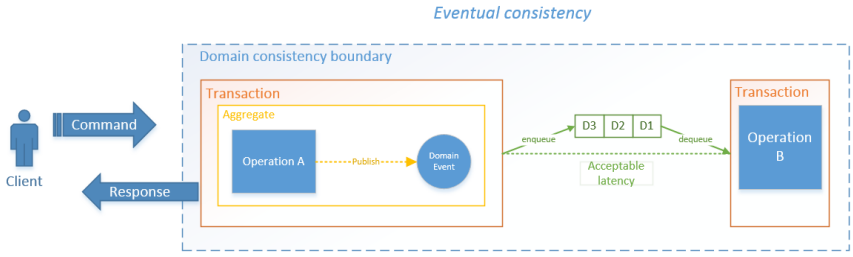
Invariants kunnen ook gecontroleerd worden in Value Objects (hoofdstuk 5.2) en ook deze zullen een exception gooien indien er een probleem optreed. Een exceptie is een zeer eenvoudige manier om de applicatie vroegtijdig te laten stoppen en een bericht aan de gebruiker terug te geven.

5.6 Eventual Consistency

Elk commando dat in het systeem terecht komt, kan op een queue geplaatst worden. Het grote voordeel hier van is dat het systeem niet over belast wordt en dat elke queue job afgehandeld kan worden (King, 2015). Vandaar de term eventual consistency, uiteindelijk zal het systeem consistent zijn. Een ander groot voordeel is dat eventual consistency automatisch samen kan werken met schaalbaarheid van een applicatie, zoals beschreven in Hoofdstuk 6. Wanneer er een commando uitgevoerd wordt op het systeem, weet de client

al de data al. Er is momenteel geen nood om meteen een response terug te krijgen. Daarom is het ook belangrijk bij commands om een GUID op voorhand te definiëren zodat de identiteit al bekend is. Met deze identiteit kan er eventueel al een pagina getoond worden waar de data getoond wordt.

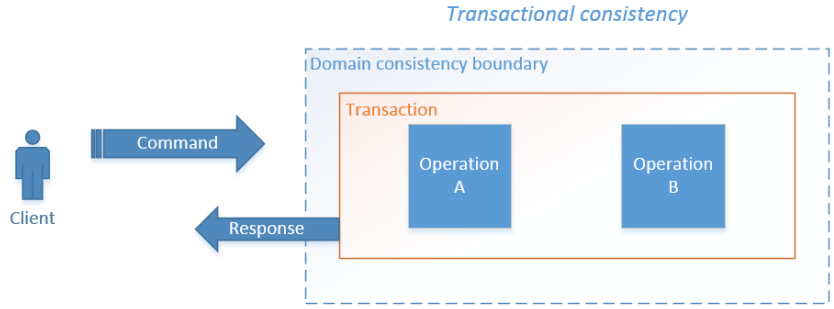
Figuur 5.1: Eventual Consistency



5.6.1 Transactional Consistency

Het voordeel aan transactional consistency is dat de response meteen terug gegeven wordt. Alle delen van de applicatie zijn doorlopen en alle gegevens zijn gepersisteerd. Het grote nadeel is dat de applicatie volledig doorlopen moet worden en dat de response veel trager kan zijn. Indien de applicatie exponentieel hoog aantal requests binnen krijgt en dus alles meteen moet afhandelen. Kan dit resulteren in het crashen van de applicatie. Transactional consistency is ook moeilijker schaalbaar omdat alles in 1 keer uitgevoerd wordt.

Figuur 5.2: Transactional Consistency



5.7 Event Store

De EventStore is een relevant concept bij EventSourcing. De EventStore is de plaats waar alle DomainEvents opgeslagen zullen worden. De EventStore kan elk soort databank zijn, er is ook een speciaal gemaakt en is te vinden op https://geteventstore.com. Voor deze bachelorproef, en de proof-of-concept zal er gebruik gemaakt worden van een simpele EventStore in MySQL. Een EventStore is helemaal niet zo moeilijk qua structuur, het bevat volgende minimale velden:

- id, een simpele id, die autoincrementeel kan zijn

- stream_id, een id die bij een Aggregate hoort, dit is een GUID. Alle events die horen tot een bepaalde aggregate zullen dezelfde stream_id hebben.
- stream_version, een versie die incrementeel is per aggregaat. Dit zorgt er voor dat event in de juiste volgorde kunnen worden afgespeeld indien nodig.
- event_name, de naam van het DomainEvent dat opgeslagen moet worden. Via deze naam kan het juiste object weer opgebouwd worden in de code.
- payload, dit bevat de effectieve data van het DomainEvent. Dit kan een json object zijn. Dit wordt mee gebruikt, naast de naam om het object weer op te bouwen.
- recorded_at, een timestamp waarop het event opgeslagen is geweest. Dit kan handig zijn voor de audit log.

5.8 Audit Log

Wanneer er gebruik gemaakt wordt van EventSourcing, is er automatisch een gratis audit log beschikbaar omdat de EventStore append-only is. Deze audit log is een bewijs van alle events die tot een bepaalde waarde (projection) lijden. Wanneer de EventStore op een Write Once Read Many (WORM) drive gezet wordt, dan kan er niet geknoeid worden met deze lijst van events. Het voordeel hieraan is dat er gemakkelijk bewijs kan geleverd worden naar andere partijen indien dat nodig is. Stel er komt een rechtzaak tussen 2 bedrijven omdat er geknoeid is met de data. Dan kan dit zwart op wit bewezen worden dat dit niet het geval is.

Deze audit-log heeft ook nog andere voordelen, als er een fout zit in de applicatie en de applicatie moet gedebugged worden, dan kan men de audit log nemen en deze opnieuw afspelen tot de huidige staat. Zo kan er per event gekeken worden of de uitkomst van de huidige staat al dan niet correct is.

5.9 Projections

Projections zijn een projectie van de huidige staat die berekend is van alle voorbijgaande events. Bij een systeem waar er met een relationele databank wordt gewerkt, is de databank zelf de projectie. Met als grote verschil dat daar ook de databank de enige echte bron van waarheid is. Dat is niet het geval bij een systeem met EventSourcing, daar is de bron van waarheid de events die opgeslagen zijn.

Projections hebben het voordeel dat er meerdere kunnen zijn. Er kan een databank gegenereerd worden voor de business kant die dan allerlei statistieken kan bekijken. Er kan ook een databank gegenereerd worden voor werknemers die met een bepaalde applicatie werken. Er kan zelf een databank of ander opslagsysteem gevuld worden los van de huidige databanken voor de andere partijen. Op deze manier zijn het systeem en de databank losgekoppeld van elkaar en dit door de flexibele projecties.

Deze projecties zijn een berekening van de events, dit wilt ook zeggen dat er niet geoptimaliseerd moet worden voor de schrijfkant (Hoofdstuk 4). Er kan geoptimaliseerd worden

voor de leeskant, er is geen nood om een aantal tabellen elke keer te gaan joinen, als er rechtstreeks naar 1 tabel geschreven wordt.

Om een wijziging te kunnen doen in een aggregate, moet eerst de huidige state opgebouwd worden vanuit de DomainEvents. Dit kan echter inefficiënt zijn wanneer er duizenden events beschikbaar zijn. Een projectie kan dan perfect dienen als een cache, om dit probleem op te lossen.

5.10 Testen

Zoals besproken in Hoofdstuk 5.3, zijn alle queries en commandoś klassen die leesbaar zijn, en de naamgeving komt rechtstreeks van de business kant. Testen schrijven voor de domain logica zoals commandoś en queries kunnen ook opgesteld worden door de business en gevalideerd worden door hen. De volgende structuur van testen kan aangenomen worden:

- Given: een lijst van events
- When: een commando uitgevoerd wordt
- Then: worden er Domain Events (Hoofdstuk 5.4) geretourneerd of excepties gegooid (Hoofdstuk 5.5)

Deze manier van testen levert automatisch documentatie op die gevalideerd kan worden door de business. Wanneer er nieuwe developers bijkomen, kunnen ze door naar de testen te kijken te weten komen hoe bepaalde zaken werken.

5.10.1 Debugging

Het debuggen van een EventSourced systeem kan op volgende manier gebeuren: alle events worden opgeslagen, wat een developer kan doen is een reeks aan events programmeren of kopiëren uit een log tot waar het fout ging om zo stap voor stap door het process te lopen.

6. Schaalbaarheid

Een systeem dat gebruik maakt van EventSourcing, kan geschaald worden, het schalen is relatief gemakkelijk door volgende redenen. De EventStore is een append-only database, er kan altijd een kopie genomen worden van deze database. Er moet niet gekeken worden of er al dan niet wijzigingen zijn in de data en welke de correcte data is. Synchroniseren van meerdere EventStores is een kwestie van ontbrekende rijen aan te vullen. Hiervoor kan er een mechanisme gebruikt worden zoals ‘geef alle events na id X.’ Schalen van de applicatie zelf kan als volgt gebeuren: er kunnen projecties bijgemaakt worden die zichzelf gaan ’voeden’ met alle events uit de EventStore. Van zodra ze up-to-date zijn met de EventStore kunnen ze als projection listener geregistreerd worden, om live naar de events te luisteren.

7. Voordelen

Er zijn meerdere voordelen verbonden aan EventSourcing. Een voordeel is dat er van technologie gewisseld kan worden voor de projecties. Stel er werd gebruik gemaakt van MySQL om de huidige staat in op te slaan. Wanneer er gemerkt wordt dat er nood is aan een graph database in plaats van een relationele database dan kan er een projectie bijgemaakt worden die simultaan met de MySQL database loopt. Zodra de graph database volledig gevoed is met de events uit de EventStore, dan kan de MySQL databank weggegooid worden. Doordat een projectie opgebouwd is in code, en in een versiebeheersysteem zit, moet er niet nagedacht worden om een eventuele back-up te nemen van deze MySQL databank. Indien er later opnieuw van deze databank gebruik gemaakt moet worden, dan kan deze projectie terug ingeladen worden en volledig opnieuw opgebouwd worden. Er werd net gesproken over de graph database, stel dat de applicatie al 3 jaar draait en nu komt deze graph database er bij. Zodra de events volledig gevoed zijn in de graph database, dan staat deze op het punt alsof ze al van in begin geprogrammeerd was. Dit is het grote voordeel van al de events bij te houden. Een ander voordeel van EventSourcing is dat enkel de EventStore gebackupped moet worden. Indien elke server corrupt of kapot gaat en alle MySQL instanties zijn kapot, dan kunnen deze opnieuw opgebouwd worden. Dit is mogelijk omdat de EventStore, de single source of truth is.

8. Nadelen

...TODO...

9. Kosten

De kosten van EventSourcing kunnen opgedeeld worden in 3 grote delen. Het eerste deel gaat over de fysieke infrastructuur kosten die EventSourcing met zich meebrengt. Het opslagen van alle belangrijke business data doorheen de jaren dat de applicatie werkt brengt een bepaalde kost met zich mee. Het tweede deel gaat over de kosten van de developers, EventSourcing is een andere manier om met gegevens om te gaan en dus ook een andere manier om code te schrijven. Tot slot het derde deel gaat over de kost van performance vermits elk event overlopen moet worden kan dit een performance probleem met zich mee brengen.

9.0.1 Infrastructurele kosten

Er van uitgaande dat events opgeslagen worden als json objecten kan hier vrij snel een berekening op gedaan worden. Hier zijn volgende zaken waar er rekening mee gehouden moet worden: grote van het json object en het aantal requests. De grote van een json object voor een bepaald event is niet zo groot. Om dit te weten te komen wordt de grootte van de payload om een nieuwe afspraak te maken genomen en als json object opgeslagen. Nu we dit event hebben opgeslagen kan de grootte in bytes uitgelezen worden. De keuze om de payload van een nieuwe appointment te gebruiken is omdat deze het meest voorkomt in de applicatie van Skedify. De grootte van dit event bedraagt 168 bytes. Uit informatie van New Relic, een platform dat door Skedify gebruikt wordt om aan monitoring te doen blijkt dat er in 1 maand tijd 143500 requests worden uitgevoerd die een schrijf actie uitvoeren. Met deze gegevens kan er berekend worden hoeveel disk capaciteit er nodig is om deze

events voor lange tijd bij te houden.

$$\frac{143500 \text{ requests}}{\text{maand}} \times 168 \text{ bytes} \times 5 \text{ jaar} = 1.37 \text{ GB} \tag{9.1}$$

Dit wilt zeggen dat er 1.37GB plaats moet voorzien worden als men de events wilt opslaan voor 5 jaar. In de volgende tabel zal er een assumptie gemaakt worden dat het aantal requests doorheen de jaren zal stijgen en dat het aantal bytes voor een event ook zal stijgen.

Requests / maand	Event grootte	Periode	Nodige disk capaciteit
150 000	500 bytes	10 jaar	8.5 GB
200 000	750 bytes	10 jaar	17.01 GB
300 000	1 kB	10 jaar	34.83 GB
350 000	1.5 kB	10 jaar	60.69 GB
400 000	1.5 kB	10 jaar	69.67 GB
500 000	1.5 kB	10 jaar	87.08 GB
800 000	2 kB	50 jaar	928.88 GB

Tabel 9.1: Disk capaciteit nodig om alle events op te slaan[1]

Zoals te zien in tabel 9.1 is het helemaal geen probleem om de events op te slaan omdat deze helemaal niet zo groot zijn. De kost is 1 harde schijf extra, die elk jaar goedkoper en goedkoper wordt. Meer nog, zolang de data hoeveelheid onder Kryderś law blijft, wordt de data alleen maar goedkoper over tijd.

Een tweede kost waar er rekening mee moet worden gehouden zijn backups. Zoals vermeld in Hoofdstuk 5.9 zijn de databanken die gebruikt worden projecties van de afgebeelde events. Vermits deze een berekening zijn volstaat het om enkel van de events een backup te nemen, omdat de projecties opnieuw berekend kunnen worden[2].

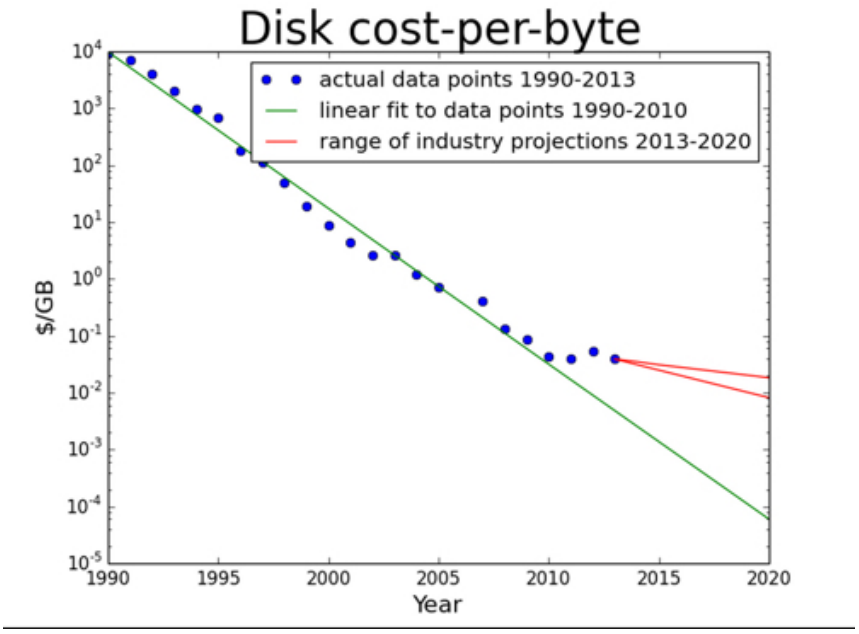
9.0.2 Developer kosten

Het is een andere manier van werken dus moeten de developers ook op de hoogte gebracht worden hoe ze met deze technologie moeten omgaan. Er zijn verschillende resources beschikbaar, waaronder https://eventsourcery.com. Zowel bij Skedify, als bij andere bedrijven wordt er gebruik gemaakt van OOP (Object oriented programming). EventSourcing gaat erg goed samen met functioneel programming omdat de huidige state een left fold is

[1] De gegevens staan voor de herkenbaarheid met het kilobyte en gigabyte symbool, maar eigenlijk wordt er kibibyte en gibibyte bedoelt.

[2] Er mag niet vergeten worden om een backup te nemen van data die niet in de databank wordt opgeslagen zoals afbeeldingen

Figuur 9.1: Kryder slowdown. Chart by Preeti Gupta at UCSC.



van vorige events.

$$\text{current state} = f(\text{events}, \text{action}) \tag{9.2}$$

Functioneel leren programmeren is dus een meerwaarde.

9.0.3 Performance kosten

Telkens wanneer er een nieuwe command uitgevoerd wordt zoals vernoemd in Hoofdstuk 5.3.1, moeten alle events uit de EventStore overlopen worden om de huidige staat van het object op te bouwen. Dit wil zeggen dat bij elk nieuw event de applicatie trager zal worden. In kleine en middelmatige applicaties heeft dit geen gevolgen, naar grotere applicaties heeft dit grotere gevolgen. Hiervoor is een oplossing van snapshots. Een snapshot is de huidige state van een object na het overlopen van x aantal events. Deze kunnen in de EventStore opgeslagen worden, in een tabel naast de events zelf. Telkens wanneer de klasse beschrijving zou veranderen, moeten de snapshots opnieuw gegenereerd worden. Een tweede optie is om de current state ook op te slagen in memory, via deze weg moeten er geen events opnieuw afgespeeld worden om de huidige state van een object te kunnen bepalen omdat deze reeds beschikbaar is.

10. Proof of concept

...TODO...

11. Conclusie

...TODO...

Lijst van acroniemen

CQRS Command Query Responsibility Segregation. 3, 17
CQS Command Query Separation. 3, 17, 18

GUID Globally Unique Identifier. 25, 27, 28

WORM Write Once Read Many. 28

Verklarende Woordenlijst

MySQL MySQL is een managementsysteem voor relationele databases (relational database management system (RDBMS)). 27, 33

Bibliografie

Fowler, M. (2005a, december). Command query separation. https://www.martinfowler.com/bliki/CommandQuerySeparation.html. Blog. Last visited on 2017-04-20.

Fowler, M. (2005b, december). Ddd aggregate. https://www.martinfowler.com/bliki/DDD_Aggregate.html. Blog. Last visited on 2017-04-20.

King, J. (2015, juli). Jef king - eventual consistency. https://www.youtube.com/watch?v=fIfH-kUaX4c. YouTube. Last visited on 2017-05-13.

Meyer, B. (1988). *Object-oriented software construction*. United States: Prentice Hall.

Microsoft. (2017, maart). Event sourcing. https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing. Blog. Last visited on 2017-04-20.

Loi - Wet. (g.d.). http://www.ejustice.just.fgov.be/wet/wet.htm. Last visited on 2017-04-20.

Verraes, M. (2014, januari). Domain-driven design is linguistic - just because two behaviours lead to the same outcome, does not mean they are replaceable. http://verraes.net/2014/01/domain-driven-design-is-linguistic/. Blog. Last visited on 2017-03-22.

Verraes, M. (2015, januari). Messaging flavours - differentiating between informational, interrogatory, and imperative messages, and keeping them nicely separated. http://verraes.net/2015/01/messaging-flavours/. Blog. Last visited on 2017-03-19.

Young, G. (2014, september). Greg young - cqrs and event sourcing. https://www.youtube.com/watch?v=JHGkaShoyNs. YouTube. Last visited on 2017-04-19.

Lijst van figuren

Lijst van tabellen