



HoGent

Faculteit Bedrijf en Organisatie

Opslag en synchronisatie van offline data bij mobiele applicaties

Simon Jang

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

Promotor:
Stefaan De Cock
Co-promotor:
Sam Verschueren

Instelling: Pridiktiv.care - into.care

Academiejaar: 2016-2017

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Opslag en synchronisatie van offline data bij mobiele applicaties

Simon Jang

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

Promotor:
Stefaan De Cock
Co-promotor:
Sam Verschueren

Instelling: Pridiktiv.care - into.care

Academiejaar: 2016-2017

Tweede examenperiode

Samenvatting

Mobiele applicatieontwikkeling beschouwt toegang tot internet vaak als vanzelfsprekend wanneer een mobiele applicatie wordt gebruikt. Dit is echter niet het geval in elke (werk)omgeving. Daarom moet de ontwikkelaar ervoor zorgen dat de applicatie de data ook lokaal kan opslaan indien de wijzigingen niet meteen kunnen worden doorgevoerd naar de achterliggende infrastructuur. Dit probleem vormt voor ontwikkelaars een uitdaging omdat de data integriteit moet worden gewaarborgd wanneer het toestel terug verbonden is met het Internet. Het gebruik van een performante en betrouwbare methode voor de data die offline wordt ingegeven te synchroniseren met de online cloud-based databank is dus essentieel. Hierbij is het belangrijk om een onderscheid te maken tussen use cases waarbij de developer gebruik maakt van fully-managed databases zoals onder andere DynamoDB van Amazon Web Services en use cases waarbij men zelf de database(s) beheert op verschillende virtuele machines. In het onderzoek is synchronisatie specifiek onderzocht voor de fully-managed database DynamoDB, een backend die gebruikt maakt van een serverless microservices architectuur en een Angular applicatie als client applicatie.

Na onderzoek en reserach is gebleken dat er verschillende manieren van synchronisatie zijn. Technologieën als CouchDB en Firebase automatiseren het synchronisatieproces. Maar wanneer men zelf de synchronisatie wenst te realiseren is het belangrijk om de verschillende use cases onder te verdelen volgens synchronisatie methode en zo verder te werken. Er zijn scenario's waar er geen conflicten zijn maar performantie de belangrijkste factor is bv. Read-Only Optimised. Wanneer er conflicten optreden kan men dan kiezen voor First Write Wins of Last Write Wins waarbij er onvermijdelijk data verloren gaat of andere strategieën van conflict resolution. Caching speelt een belangrijke rol in de transitie van online naar offline en synchronisatie is belangrijk bij de overgang van offline naar online. Met behulp van caching en synchronisatie is het mogelijk om een robuuste applicatie te ontwikkelen waarbij data integriteit en functionaliteit worden gegarandeerd.

Voorwoord

Mijn interesse in mobiele- en web applicaties was voor mij de reden om de opleiding Toegepaste Informatica aan de Hogeschool Gent te starten. Deze bachelorproef vormt het sluitstuk in mijn opleiding en de keuze van het onderwerp is tot stand gekomen door de samenwerking met mijn stageplaats Pridiktiv.care - into.care. Er was nood aan een onderzoek naar offline data opslag en synchronisatie bij hun mobiele applicatie. Met de groei van IoT en de digitalisering binnen verschillende sectoren, lijkt offline en online synchronisatie relevanter dan ooit.

Het schrijven van een bachelorpaper is geen eenvoudige opdracht en ik zou daarom enkele personen willen bedanken voor hun ondersteuning en expertise. Eerst en vooral wil ik mijn co-promotor en stage-mentor Sam Verschueren bedanken voor zijn inzet en geduld bij de talloze vragen die ik het gesteld in verband met webontwikkeling. Daarnaast ben ik ook zeer dankbaar voor de intense begeleiding en feedback die ik heb ontvangen van mijn promotor Stefaan De Cock. Tenslotte wil ook mijn partner en vrienden bedanken voor de hulp die ze hebben aangeboden bij het lezen van mijn bachelorproef en de morele ondersteuning.

Inhoudsopgave

1	Glossarium	11
2	Inleiding	13
2.1	Business Case	13
2.1.1	Voorstelling bedrijf	13
2.1.2	Context en probleemstelling van de business case	14
2.1.3	De Pridiktiv applicaties en technologiestack	14
2.2	Terminologie	15
2.2.1	Huidige Web APIs voor lokale opslag	15
2.2.2	Web Storage	15
2.2.3	Scalable Angular Architecture	17
2.2.4	Redux store: een state container	18
2.2.5	Reactive programming paradigm	19
2.2.6	Offline First	19

2.2.7	Amazon Web Services	20
2.3	Stand van zaken	20
2.4	Probleemstelling en Onderzoeksvragen	21
2.5	Opzet van deze bachelorproef	22
3	Methodologie	23
3.1	Research	23
3.1.1	Research naar gebruikte technologieën in de business case	23
3.1.2	Beperkingen in de business case	24
3.1.3	Research naar bestaande synchronisatie methodes	24
3.1.4	Bespreken research met expert	24
3.2	Ontwerpen en prototyping van architectuur voor synchronisatie	24
3.3	Toepassen van synchronisatie oplossing op business case	25
3.4	Conclusie	25
4	Opstelling	27
4.1	Client	27
4.1.1	Client: Backoffice applicatie	27
4.1.2	Client: Mobiele applicatie	28
4.2	Server	29
5	Synchronisatie methodes	31
5.1	Bestaande synchronisatie mogelijkheden	31
5.2	Alternatieve synchronisatie mogelijkheden	32
5.2.1	Read-Only	32

5.2.2	Read-Only Optimized	32
5.2.3	First/Last Write Wins	33
5.2.4	Write with Conflict Detection	33

6 Onderzoek 35

6.1 Online/Offline status registratie 35

6.2 Caching van requests 37

6.3 Data preloading in de client 38

6.3.1 Beperkingen door single-threaded omgeving 38

6.3.2 Preloading methodes 38

6.4 Data synchronisatie: planning bij development 40

6.5 Read-Only Optimised 41

6.5.1 Overzicht: Client 41

6.5.2 Overzicht: Server 41

6.5.3 Opmerkingen 42

6.5.4 Toepassing 42

6.6 Last/First Write Wins 42

6.6.1 Overzicht: Client 43

6.6.2 Overzicht: Server 43

6.6.3 Opmerkingen 43

6.6.4 Toepassing 43

6.7 TODO ergens tussenplaatsen 44

6.8 Conflict Resolution 44

6.8.1 Overzicht: Client 44

6.8.2 Overzicht: Server 44

6.8.3	Toepassing	45
6.9	Open-source libraries	45
6.9.1	aws-sqs-push	46
6.9.2	aws-sqs-poll	47
6.9.3	aws-sqs-deletemessage	47
6.9.4	aws-sqs-geturl	48
7	Conclusie	49
	Bibliografie	52

1. Glossarium

Single-Page Application, SPA : Een single-page application (SPA) is een web applicatie of website waarbij noodzakelijke HTML, CSS en JavaScript dynamisch wordt ingeladen of bij de eerste page load. De volledige pagina wordt bij een SPA nooit volledig herladen. Het is wel mogelijk dat onderdelen van de pagina dynamisch wordt gewijzigd. De data van de SPA wordt dynamisch opgevraagd van de web server. Met SPA is mogelijk een om efficiënt om te springen met de beschikbare bandbreedte.

Angular CLI : Angular CLI is een command line interface tool waarbij een volledige Angular project wordt gebouwd met minimale configuratie. Er wordt een basis structuur, tests, root module en root component aangemaakt. Met behulp van Webpack worden alle files dan gebundeld in enkele static files. Met Angular CLI wordt er heel wat tijd uitgespaard omdat een groot deel van de configuratie wegvalt. Angular CLI wordt gebruikt de business case van Pridiktiv.

Progressive enhancement : Progressive enhancement is een ontwikkelstrategie bij web development waarbij de focus wordt gelegd op de belangrijkste business requirements die de web applicatie of website moet invullen. Afhankelijk van de browser en de connectie van de eindgebruiker, kunnen er 'lagen' van functionaliteit (features) worden toegevoegd aan de applicatie of website. Met deze strategie kan een website of web applicatie zich aanpassen aan de eindgebruiker en altijd een basisfunctionaliteit garanderen.

Single Source Of Truth, SSOT : In de context van het ontwerp van informatica systemen is de single source of truth een techniek om data op een bepaalde manier te structureren zodat die niet gedupliceerd wordt binnen de applicatie en alle referenties verwijzen naar dezelfde 'bron'. Alle informatie wordt opgehaald in een centraal punt en dat is voor de applicatie en haar componenten de enige plaats waar die informatie kan worden opgehaald. De ngrx/redux store vervult die rol in een Angular applicatie.

- Conflict resolution** : Een applicatie kan data van een remote databank lokaal opslaan voor offline functionaliteit aan te bieden aan de gebruiker. Wanneer de applicatie terug online gaat en de applicatie of remote databank een verschil opmerkt, dan is er sprake van een conflict. Conflict resolution duidt op de methode(s) die worden gebruikt voor het synchroniseren van de data tussen de verschillende databanken.
- Serverless computing** : Een andere naam voor Function as a Service (FaaS) is een cloud-computing executie model waarbij de cloud provider het opstarten en stoppen van een functie volledig zelf beheert. Het opstarten van een functie is op basis van een event. Een mogelijk event is bijvoorbeeld een API call naar een HTTP endpoint van een functie. Met serverless computing hoeft een developer zich niet bezig te houden met het configureren en beheren van verschillende servers of virtual machines. Daarnaast spelen schaalbaarheid en kostenefficiëntie ook een belangrijke rol bij serverless computing. Als FaaS-gebruiker betaal je enkel voor de executietijd en wanneer er meer rekencapaciteit nodig is, gebeurt de schaling volledig automatisch.
- ReactiveX** : Reactive Extensions is een verzameling van operatoren die imperatieve¹ programmeertalen toelaten om een datasequentie te verwerken zonder rekening te houden met het (a)synchrone karakter van de data.
- Event-driven Architectuur** : Een software architectuur waarbij de verschillende componenten events genereren, detecteren en op een gepaste manier reageren. De event-driven architectuur vormt de basis van ReactiveX en serverless computing.
- Asynchroon** : Asynchroon of 'Asynchronous' in de context van web development wil zeggen dat een bepaalde handeling niet real-time maar periodiek van aard is. Dit is belangrijk wanneer gebruikers geen stabiele verbinding hebben tot een netwerk, wanneer men optimaal gebruik wenst te maken van de beschikbare bandbreedte of eenvoudigweg bij het uitvoeren van een HTTP request. Een voorbeeld is RxJS dat gebaseerd is op de ReactiveX library en volledig gebouwd rond asynchroon programmeren.
- Polling** : Bij polling (of pulling) controleert een applicatie met een vast interval of er op een invoer -of uitvoerapparaat nieuwe data beschikbaar is. Het interval of frequentie wanneer de applicatie polling uitvoert, noemt de pollfrequentie. Polling wordt steeds geïnitieerd door de ontvanger van de informatie. Te frequente polling kan een negatieve impact hebben op de performantie van de databron.
- Pushing** : Bij push technologie wordt de request voor een datatransactie geïnitieerd door de server of 'publisher' van de informatie via internet. Een variant is het push/subscriber model waarbij clients
- Microservices Architecture** : Microservices is een architectuur voor het opstellen van server-side enterprise applicaties. Daarbij worden alle onderdelen opgebouwd als een set van losgekoppelde en samenwerkende services. Elke service heeft een beperkte functionaliteit en een beperkte verantwoordelijkheid die kan worden aangeroepen door andere microservices. Dankzij microservices is het mogelijk om de functionaliteit van een monolitische backend op te delen in verschillende kleine services.

¹Imperatieve programmeertalen kunnen de state van een applicatie veranderen. JavaScript is een voorbeeld van een imperatieve programmeertaal

2. Inleiding

Het onderzoek zal verschillende methodes voor offline opslag en synchronisatie analyseren en onderzoeken. Het onderzoek zal van elke methode de voor- en nadelen overlopen. In de sectie 2.1 'Business case' wordt de business case van Pridiktiv.care - into.care toegelicht. Daarna worden in sectie 2.2 'Terminologie' de verschillende relevante termen overlopen. In sectie 2.3 'Stand van zaken' komt de context en noodzaak van het onderzoek aan bod. Tenslotte volgt de probleemstelling, de onderzoeksvraag en de opzet van de bachelorproef.

2.1 Business Case

2.1.1 Voorstelling bedrijf

Pridiktiv - into.care is een start-up die een mobiel platform aanbiedt aan zorgverstrekkers om de administratielast te verlichten. Op die manier is er meer tijd voor zorg en verliezen de zorgverleners minder tijd bij het invullen van documenten. UX¹ en mobile vormen de speerpunten van de business doelstellingen en vormen de belangrijkste factoren om zich te onderscheiden van andere concurrenten. Op termijn wil de start-up een modern, mobile-friendly ERP-systeem aanbieden aan woonzorgcentra.

¹UX staat voor User Experience. Een verzamelnaam die alle emoties capteert die gebruikers ervaren bij het gebruiken van een product.

2.1.2 Context en probleemstelling van de business case

De applicatie draait op een smartphone in een woonzorgcentrum. Het woonzorgcentrum beschikt over draadloos internet maar de internetdekking in het gebouw is niet volledig. Dus moet de applicatie ook offline werken. Wanneer verschillende waarden worden geregistreerd met de applicatie (zoals bloeddruk, gewicht, inname medicatie) worden die offline opgeslagen in een IndexedDB. Wanneer het toestel terug online komt, moeten deze waarden worden doorgestuurd naar de backend. Het is essentieel dat er geen data verloren gaat aangezien het gaat om medische data. Bij de start van het onderzoek gebruikte de applicatie een algoritme die een HTTP call uitvoert wanneer een waarde wordt ingegeven. Wanneer dit niet lukt, veronderstelt het algoritme dat de applicatie offline is en worden de data lokaal opgeslagen. Wanneer het toestel terug online komt, dan worden de offline data gesynchroniseerd met de server. De realiteit wijst echter uit dat betrouwbaar internet eerder uitzonderlijk is in een woonzorgcentrum. In deze context is er dus nood aan een 'offline first' - oplossing.

2.1.3 De Pridiktiv applicaties en technologiestack

Pridiktiv maakt gebruik van verschillende applicaties om de business doelstellingen te realiseren. In hoofdstuk 4 kan u een gedetailleerd bespreking terugvinden van de verschillende applicaties.

Client: Mobile applicatie

De huidige applicatie is een Angular² applicatie met Redux als state container. Voor lokale opslag wordt momenteel gebruik gemaakt van Mozilla's localForage³ library. De applicatie is gebundeld als een Cordova applicatie.

Client: Backoffice applicatie

De backoffice applicatie is net als de mobiele applicatie een Angular applicatie maar dan gericht voor desktop gebruik. Synchronisatie en caching is voor deze applicatie niet relevant. Deze applicatie komt niet verder aan bod in dit onderzoek

²Wanneer de term 'Angular' wordt gebruikt zonder versienummer, dan duidt dat op alle versies vanaf Angular 2. AngularJS is de verwijzing naar de 'oude' Angular versie

³localForage maakt gebruik van de verschillende DOM Storage methodes en voorziet een uniforme interface voor het opslaan van data in de browser.

Server side: Serverless architectuur

De backend van de businesscase maakt gebruik van een serverless architectuur in de Amazon Web Services Cloud en bestaat uit een verzameling van verschillende microservices die AWS⁴ Lambda gebruiken. Om de data te persisteren wordt DynamoDB van AWS gebruikt.

2.2 Terminologie

In deze sectie komen de verschillende begrippen met betrekking tot de componenten van het onderzoek, het prototype en tools aan bod. De volledige lijst met begrippen en termen kan u terugvinden onder het hoofdstuk 'Glossarium'. Er is een minimum kennis in verband met software ontwikkeling vereist van de lezer om deze sectie volledig te begrijpen.

2.2.1 Huidige Web APIs voor lokale opslag

Er zijn momenteel 4 APIs (Mozilla, 2017d) voor lokale opslag die ondersteund worden door verschillende browsers. Deze APIs worden ondersteund (WHATWG, 2017) door de Web Hypertext Application Technology Working Group, aangegeven door WHATWG. De specificatie van de API wordt dan gestandaardiseerd door het World Wide Web Consortium, aangegeven door W3C. Dit proces is belangrijk omdat de APIs dan door de populaire browsers zoals Chrome, Firefox en Safari worden geïntegreerd. Op die manier kunnen webapplicaties gebruik maken van de verschillende APIs. Het is belangrijk om deze APIs kort te overlopen omdat ze steeds gebruikt worden bij de verschillende caching technieken om data lokaal op te slaan.

2.2.2 Web Storage

localStorage en sessionStorage APIs vallen onder Web Storage of DOM Storage (Camden, 2016). Wanneer de browser ondersteuning biedt voor Web Storage, zijn beide beschikbaar op het globale window. Web Storage wordt vaak vergeleken met cookies. Terwijl die vergelijkbaar zijn in functie, verschilt Web Storage in volgende aspecten:

- Opslag ruimte: Afhankelijk van browser maar meestal 5 MB beschikbare opslagruimte in vergelijking met maar 4 kb voor cookies.
- Client-side interface: Cookies kunnen zowel door server als client side worden gebruikt. Web storage valt exclusief onder client-side scripting.
- Twee verschillende storage omgevingen: localStorage en sessionStorage.
- Een eenvoudigere programmeerbare interface in vergelijking met cookies.

⁴AWS is de afkorting voor Amazon Web Services. Gemakkelijkheidshalve wordt in dit onderzoek AWS gebruikt om te verwijzen naar AWS

localStorage

Data die in localStorage wordt opgeslagen is persistent tenzij die manueel wordt verwijderd door de gebruiker of applicatie. localStorage is dus een belangrijke kandidaat om data lokaal op te slaan in geval een applicatie offline moet kunnen worden gebruikt. Bij gevoelige data zoals medische data is het belangrijk om de data te verwijderen uit de localStorage wanneer die niet meer moet worden gecached.

sessionStorage

Het grote verschil (Mozilla, 2017d) met localStorage is dat sessionStorage een vervaltijd heeft en de inhoud van de sessionStorage wordt verwijderd wanneer de sessie vervalt. Een sessie vervalt bijvoorbeeld bij het openen van een nieuw tabblad of browser venster. Refreshen van de browser heeft geen impact op de sessionStorage. sessionStorage laat toe om instances van een webapplicatie te runnen in verschillende browser windows, zonder dat er conflicten optreden.

IndexedDB

IndexedDB (Leemans, 2013) is een Web API die gebruikt wordt het opslaan van relatief grote data structuren in browsers. Dankzij indexering is het mogelijk om sneller en performanter (Camden, 2016) te zoeken in de databank in vergelijking met localStorage en sessionStorage. Net zoals SQL-databanken is IndexedDB een transactional database system. Het grote verschil is echter het gebruik van JSON objecten in plaats van fixed columns tables om data op te slaan, vergelijkbaar met andere NoSQL databanken zoals MongoDB of CouchDB. Onder impuls van Mozilla wordt IndexedDB waarschijnlijk de alternatieve storage standaard voor het web in de nabije toekomst.

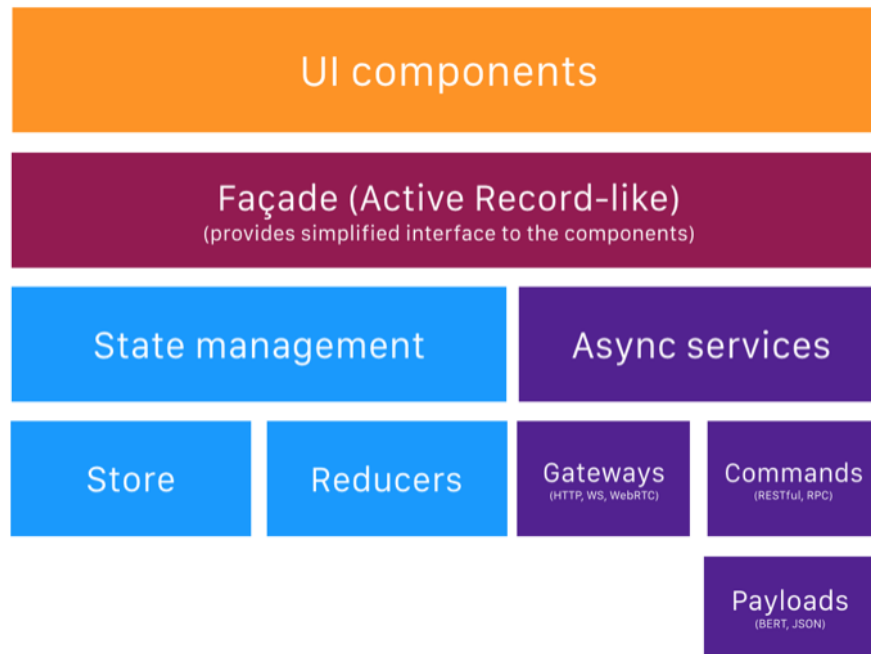
Web SQL - deprecated

Net zoals bij IndexedDB biedt Web SQL toegang tot een databank waar data structuren kunnen worden opgeslagen. Met een SQL variant is het dan mogelijk om queries uit te voeren op de Web SQL database. Momenteel biedt enkel SQLite een database systeem voor Web SQL. W3C werkt momenteel niet meer verder aan de specificatie (W3C, 2017a) van Web SQL omdat er te weinig onafhankelijke implementaties zijn van Web SQL. Het gebruik van Web SQL wordt sterk afgeraden door het deprecated status⁵ en wordt niet gebruikt in het onderzoek en de business case.

⁵'Deprecated' binnen de context van softwareontwikkeling wil aangeven dat de functionaliteit niet meer verder wordt ondersteund of ontwikkeld

2.2.3 Scalable Angular Architecture

Figuur 2.1: Overzicht van een schaalbare web applicatie



Bij de start van de ontwikkeling van een Angular applicatie is het belangrijk om na te denken over de architectuur (Gechev, 2016) van de applicatie. Moderne SPA-technologieën zoals React en Angular maken gebruik van components. Een component is combinatie van HTML, JavaScript en optioneel CSS. Door alles in componenten onder te verdelen is het eenvoudiger om de applicatie te onderhouden omdat alle relevante data wordt gegroepeerd. Componenten zelf kunnen 'dumb' zijn als ze enkel data voorstellen of 'smart' wanneer ze data opvragen of verwerken.

De dataflow tussen componenten is ook belangrijk om een beheersbare (Billiet, 2016) applicatie te bouwen. Zo communiceren child components enkel maar met hun parent en interageert een parent met een model dat op zijn beurt communiceert met een 'store'⁶ aan de hand van acties. Wanneer de state wordt gewijzigd in de store, dan wordt de volledige component tree opnieuw geëvalueerd. Een ander belangrijk aspect van een scalable SPA is het inperken van de communicatiemogelijkheden van smart components. Door het gebruik van een extra abstractie layer, zoals een model of sandbox, is het mogelijk om de communicatie te verwerken in de abstractielaag en op die manier een microservice te genereren die gemakkelijk kan worden refactored indien er aanpassingen moeten gebeuren.

⁶Zie 2.2.5 Redux store: een state container

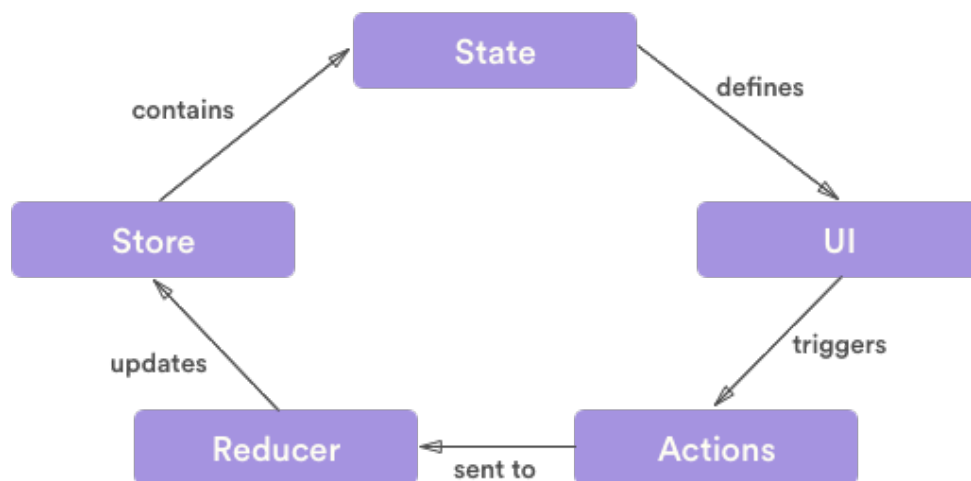
2.2.4 Redux store: een state container

Redux store is een container die de state van de applicatie bijhoudt waarbij performantie en consistentie centraal staan. De redux store is de 'single source of truth', verder benoemd als SSOT, voor de applicatie en moet ervoor zorgen dat de data van alle componenten consistent is met elkaar en er geen duplicate data is. Alle updates aan de data worden eerste weggeschreven naar de store en daarna gecachet. In het geval dat de gebruiker toegang heeft tot het Internet worden de data verstuurd naar de achterliggende systemen. Indien er geen of slechte internetverbinding is, krijgt de user de gecachte data te zien dankzij de SSOT. De store werkt met volgende (ngrx, 2017) principes :

- de state is een single immutable JSON data structure, alle aanpassingen in de state creëren een nieuwe state die de oude state overschrijft
- trigger of dispatch events initialiseren acties in de reducers
- reducers verwerken acties en wijzigen de state door een nieuwe state te creëren. Een reducer is een idempotente⁷ functie die een actie moet verwerken
- de state van de store kan asynchroon worden opgevraagd

In de applicatie van de business case wordt er gebruikt gemaakt van een Angular implementatie van Redux, de ngrx store. Net als Angular maakt ngrx gebruik van RxJS. Zo is het mogelijk om RxJS componenten zoals Observables en Subjects te gebruiken bij het opvragen van de state.

Figuur 2.2: Voorbeeld van de flow bij een dispatch in Redux



⁷Een idempotente functie is een functie die een operatie uitvoert zonder neveneffecten als die meerdere keren met dezelfde input wordt opgeroepen

2.2.5 Reactive programming paradigm

Het reactive programming paradigm (ReactiveX, 2017) is gebouwd rond veranderende data flows, waarbij een applicatie kan reageren op nieuwe of andere input. Dankzij de ReactiveX library is het mogelijk om in verschillende programmeertalen componenten van reactive programming te gebruiken. De library laat toe om asynchroon en event-driven applicaties te ontwikkelen, die in real-time data kunnen manipuleren en tonen. Het breidt het Observer design pattern uit waarbij het mogelijk is om verschillende operators te 'chainen' met elkaar zonder rekening te houden met low level concerns zoals threading, synchronisatie, thread safety, concurrent data structures en blocking I/O. De RxJS library van ReactiveX is volledig geïntegreerd in Angular en ngrx store.

Met Promises (Mozilla, 2017a) was het reeds mogelijk om asynchroon te programmeren in JavaScript dus wat zijn de voordelen van RxJS ten opzicht van promises in JavaScript? Een Promise kan maar een enkel event afhandelen en kan niet worden gestopt. Een Observable kan worden beschouwd als een stream van events waarop verschillende manipulaties zoals map, reduce en filter kunnen worden op toegepast.

```
1 // een event wordt elke 2 seconden gegenereerd
2 const source = Rx.Observable.interval(2000);
3 // mappen van elke event naar 'Hello World' string
4 const example = source.mapTo('HELLO WORLD!');
5 // subscriben op operator om waarden te ontvangen
6 const subscribe = example.subscribe(val => console.log(val));
7 //output: 'HELLO WORLD!'...'HELLO WORLD!'...'HELLO WORLD!'...
```

Listing 2.1: Voorbeeld de mapTo operator in RxJS

2.2.6 Offline First

'Offline First' is een stroming (First, 2017) binnen web development waarbij offline gebruik als de basis wordt beschouwd van de applicatie. Net zoals bij 'Progressive Enhancement' wordt online functionaliteit beschouwd als een extra laag van features die de applicatie kan aanbieden. Het idee is gegroeid vanuit de teleurstelling dat 'always online' omwille van technische, geografische, financiële en praktische redenen nog niet haalbaar is voor de nabije toekomst.

2.2.7 Amazon Web Services

De applicaties en backend van de business case maken uitgebreid gebruik van Amazon Web Services (AWS). Volgende componenten komen aan bod in het onderzoek:

- Amazon SQS: Simpelweg Simple Queue Service is een message queue service die verschillende AWS services toelaat om data te plaatsen in een queue. SQS voorziet 2 varianten. De standard queue biedt de snelste en meer performante oplossing maar hanteert geen FIFO-principe. Een FIFO-queue, zoals de naam reeds aangeeft, hanteert wel het FIFO-principe en elke message in de queue wordt altijd maar 1 keer geleverd. Andere AWS services kunnen de queue pollen voor nieuwe data.
- Lambda: Met AWS Lambda is het mogelijk om code, functies en operaties uit te voeren in de cloud dus zonder servers zelf te beheren en vormt de basis voor de serverless architectuur. De input voor de functie kan worden geleverd door HTTP endpoints maar ook door andere AWS services en is een voorbeeld van een event-driven architectuur. De output kan dan bijvoorbeeld worden gepersisteerd in DynamoDB of teruggestuurd worden naar de client. De verschillende Lambda functies vormen de microservice architectuur in de backend van de businesscase.
- Amazon SNS: Amazon Simple Notification Service is een push notificatie service die toelaat om berichten te sturen naar mobiele toestellen, email adressen of andere AWS services. Met SNS is het onder andere mogelijk om polling van andere services te vermijden.
- DynamoDB: DynamoDB is een fully-managed⁸ NoSQL database van AWS en wordt gebruikt door de business case voor de data opslag. DynamoDB voorziet geen automatische synchronisatie met clients.
- API Gateway: API Gateway is een fully managed service die toelaat om de toegang tot API's te beheren.

2.3 Stand van zaken

Wanneer een ontwikkelaar de eindgebruiker van zijn web -of mobiele applicatie wil voorstellen, dan denkt hij vaak aan een gebruiker met dezelfde eigenschappen als zichzelf (laatste smartphone, up-to-date besturingssysteem, snelle internet verbinding). De ontwikkelomgeving (snelle desktop/laptop met een betrouwbare en snelle internet verbinding) simuleert amper de omgeving van de eindgebruiker (Google, 2017). Bepaalde omgevingsfactoren zoals tunnels, trein en vliegtuig hebben een grote invloed op de betrouwbaarheid van de connectie. Het klassieke client - server model waarbij de client enkel maar als het ware een view is van de data die door de server worden bijgehouden, is achterhaald, want elke onderbreking in de internetconnectie zorgt ervoor dat de applicatie niet meer kan worden gebruikt.

⁸fully-managed houdt in dat dat het management en beheer van een service of databank wordt 'outsourced' naar de service provider. In het geval van DynamoDB moet de developer zich niet meer bezighouden met installatie, upgrades, provisioning, deployment, backups, restores en database availability.

Meeste mobiele- en webapplicaties hebben twee momenten waarbij er problemen kunnen optreden door de status van de connectie:

1. client stuurt request naar de server
2. server pusht request naar client

Afhankelijk van de business context van de applicatie, zijn er verschillende mogelijkheden:

- De gebruiker al dan niet op de hoogte brengen van veranderingen in de status van de connectie. Bijvoorbeeld: bij het versturen van een bericht kan de gebruiker op de hoogte worden gebracht dat het bericht pas wordt verstuurd wanneer de applicatie terug online is.
- Offline client-side creatie en manipulatie van data toelaten aan de hand van caching. Wanneer de applicatie terug online, deze data synchroniseren met de server.
- Uitschakelen of aanpassen van bepaalde features wanneer de applicatie offline is.

Daarnaast is het mogelijk dat de server data wil pushen naar de applicatie van de gebruiker omdat deze door een andere client werd gewijzigd. De applicatie moet de gebruiker dan waarschuwen dat er nieuwere, recentere data beschikbaar zijn en indien nodig, een conflict-resolution aanbieden. Momenteel zijn er al enkele frameworks en databank systemen die toelaten om (offline) automatisch data te synchroniseren van een web- of mobiele applicatie met de achterliggende databank. Deze oplossing is echter niet compatibel met de bestaande backend en komen enkel kort aan bod in hoofdstuk 5 'Synchronisatie methodes'

2.4 Probleemstelling en Onderzoeksvragen

Het doel van dit onderzoek is het ontwerpen van een algoritme of architectuur die compatibel is met de huidige applicaties en technologie stack. Hoe wordt conflict-resolution opgelost en moet de gebruiker daar zelf een keuze maken of kan de applicatie of achterliggende systemen zelf alle conflicten oplossen? Het is de intentie van het onderzoek om tot een betrouwbare oplossing te komen voor het probleem van de business case. Zoals reeds vermeld moet in het geval van de business case, de applicatie zonder onderbreking gemakkelijk kunnen overstappen van een offline status naar een online status en omgekeerd, zonder dat er daarbij belangrijke data verloren gaat. De applicatie werkt met vertrouwelijke medische data waardoor het verlies van data onaanvaardbaar.

2.5 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In het volgende hoofdstuk 3 wordt de methodologie toegelicht die werd gehanteerd voor dit onderzoek. De applicatie van de business case, backend infrastructuur voor dit onderzoek worden toegelicht in hoofdstuk 4. Het onderzoek start in hoofdstuk 5 met een opsomming van verschillende synchronisatie methodes. In hoofdstuk 6 worden de oplossingen op de verschillende problemen bij synchronisatie overlopen. Tenslotte kan u in hoofdstuk 7 de conclusie en een oplossing vinden. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein. Indien bepaalde begrippen onduidelijk zijn, vindt u een beknopte uitleg in het hoofdstuk 1.

3. Methodologie

In dit hoofdstuk wordt de modus operandi overlopen die gehanteerd werd tijdens het onderzoek.

3.1 Research

Het onderzoek begon met research naar de gebruikte technologieën in de business case en bestaande synchronisatie mogelijkheden. Bij het uitvoeren van deze fase, viel het op dat er weinig literatuur over het onderwerp te vinden is. Hierdoor bestaat het overgrote deel van de bronvermeldingen en referenties uit hyperlinks naar documentatie, blogs en artikelen van experts.

3.1.1 Research naar gebruikte technologieën in de business case

Tijdens deze fase lag de focus van het onderzoek om de verschillende technologieën die gebruikt worden in de business case in kaart brengen. Op basis van die resultaten zijn de grenzen bepaald waar binnen dit onderzoek wordt uitgevoerd.

3.1.2 Beperkingen in de business case

De business case van Pridiktiv laat wijzigingen aan de bestaande backend toe. Een belangrijke voorwaarde echter is het behouden van DynamoDB als database maar die biedt geen synchronisatie aan. Een andere belangrijke beperking is de eis dat er een abstractielaag¹ is tussen de database en de client.

3.1.3 Research naar bestaande synchronisatie methodes

Het tweede deel van de researchfase bestond uit onderzoek naar bestaande synchronisatie mogelijkheden. Op basis van het resultaat van het eerste deel en de beperkingen van de technologieën van de business case, werd er een selectie gemaakt met de geschikte technologieën. Er zijn reeds oplossingen voor synchronisatie van offline data maar die hebben bepaalde implementatievoorwaarden die niet voldoen aan de beperkingen van de business case. Deze worden toegelicht in hoofdstuk 4 'Opstelling'.

3.1.4 Bespreken research met expert

Na het uitvoeren van de researchfase werden de voorlopige conclusies van de researchfase getoetst bij de expert en co-promoter Sam Verschueren. Bij deze bespreking werden verschillende onderzoekspistes uitgestippeld voor het ontwikkelen van een oplossing.

3.2 Ontwerpen en prototyping van architectuur voor synchronisatie

Met behulp van een sandbox² AWS account was het mogelijk om een architectuur te ontwerpen voor de oplossing. Er werden ook prototype lambda's ontwikkeld voor het testen van de libraries die gebruikt worden bij de synchronisatie.

¹Bij complexe use cases is het niet wenselijk om een client rechtstreeks te laten communiceren met de database. Client en databank zijn te sterk aan elkaar gekoppeld waardoor database refactoring zeer moeilijk wordt.

²Een sandbox is een omgeving waar zonder neveneffecten bepaalde prototypes of tests kunnen worden uitgevoerd als 'proof of concept'. Deze dienen dan als basis voor een concrete implementatie

3.3 Toepassen van synchronisatie oplossing op business case

Terwijl synchronisatie voornamelijk afspeelt server side heeft de client binnen 'Offline First' ook een rol. De grootste uitdagingen in de client applicatie zijn de caching en de pre-caching van data in het geval de applicatie plots van status³ zou veranderen. Caching is niet de focus van dit onderzoek maar speelt een niet-onbelangrijke rol bij de transitie van online naar offline en omgekeerd. In het hoofdstuk 6 'Onderzoek' wordt verder in detail ingegaan op de caching in de client en de synchronisatie op server. De implementatie op basis van de resultaten van het onderzoek vond plaats tijdens de stage periode bij Pridiktiv. De oplossing is opgenomen in de pilootversie van de Pridiktiv applicatie en is momenteel operationeel in productie.

3.4 Conclusie

Met behulp van de literatuurstudie, research van documentatie van verschillende technologieën en prototyping is er een oplossing aangeboden voor synchronisatie en caching. In het laatste onderdeel van het onderzoek wordt er op basis van de ervaringen en bevindingen van het onderzoek een synthese gevormd.

³Van offline naar online en omgekeerd

4. Opstelling

In dit hoofdstuk wordt er dieper ingegaan op de verschillende onderdelen van het Pridiktiv platform¹. Met behulp van component diagrammen² worden de architectuur en interactie visueel voorgesteld.

4.1 Client

4.1.1 Client: Backoffice applicatie

De backoffice is een Angular applicatie en bestaat functioneel uit twee grote delen. Er is een zorgplanner en takenplanner voor de hoofdverplegers/hoofdverpleegsters waar zij het takenpakket voor een patient kunnen samenstellen. Voor het management is er een dashboard die inzicht biedt in de operationele werking van het woonzorgcentrum. De applicatie is afhankelijk van een internetconnectie om te kunnen functioneren. Omdat offline functionaliteit voor deze applicatie niet relevant is, komt het in het onderzoek niet verder aan bod.

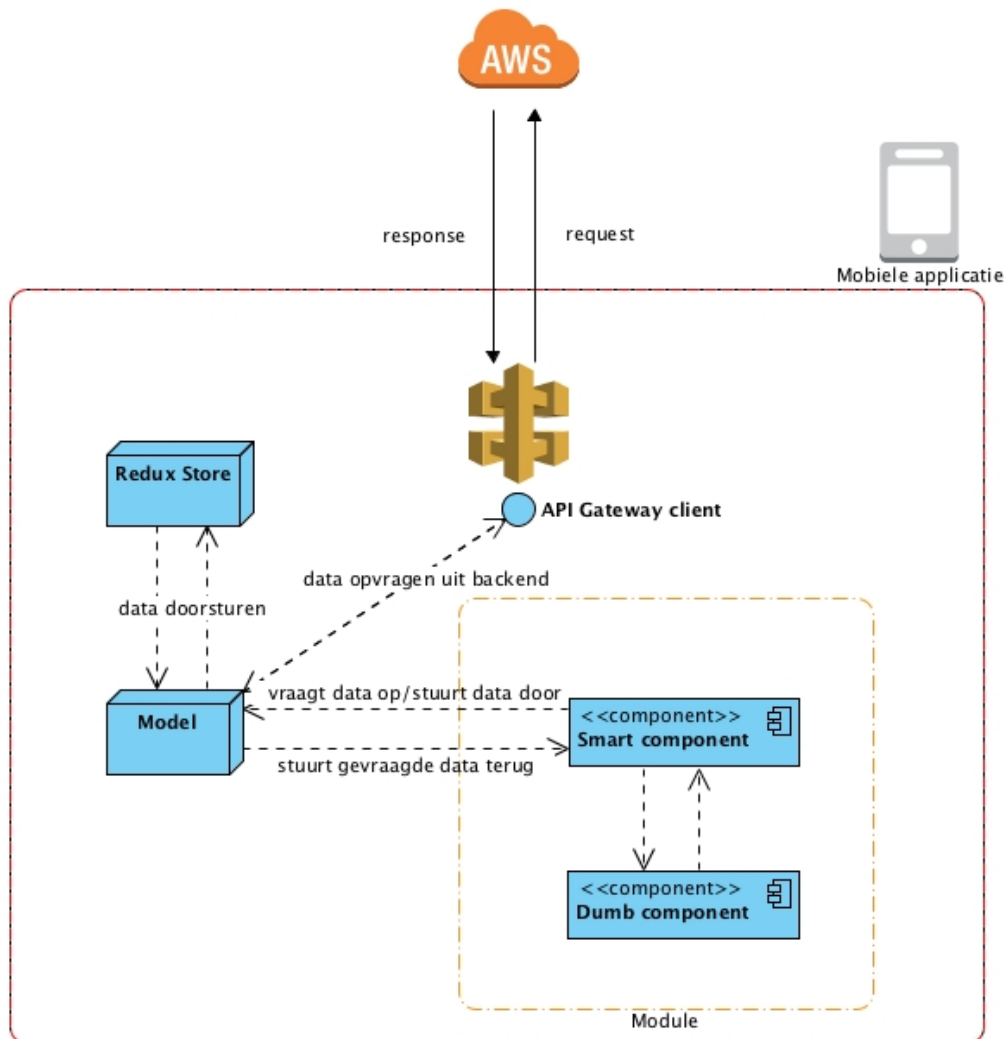
¹platform of 'solution stack': Een set van componenten en software subsystemen die een geheel vormen waarmee de business doelstellingen worden bereikt

²Een component diagram is een UML diagram die visueel voorstelt hoe componenten samenwerken om een software systeem te vormen

4.1.2 Client: Mobiele applicatie

De mobiele applicatie is net als de backoffice een Angular applicatie maar gebundeld als een Cordova applicatie. De applicatie wordt gebruikt door verplegers/verpleegsters en zorgkundigen in woonzorgcentra voor het registreren van verschillende handelingen.

Figuur 4.1: Flow van de mobiele applicatie



De applicatie maakt gebruik van verschillende design principes(Billiet, 2016)(Gechev, 2016) die ervoor zorgen dat het een schaalbare Angular applicatie is. De applicatie bestaat uit verschillende Angular modules³ die elke een feature voorstellen in de applicatie.

³Een Angular module is een verzameling van componenten en services. Modules organiseren een applicatie in coherente blokken van functionaliteit

Modules die data nodig hebben gebruiken een model⁴ voor het ophalen van de data uit de redux store. Indien de data niet in de store aanwezig is wordt die met behulp van de AWS API Gateway client opgevraagd aan de backend. De opgevraagde data wordt meteen in de store gestopt om het SSOT-principe te respecteren. Een smart component kan dan via zijn model data uit de redux store opvragen en stuurt die dan door naar een dumb component of meerdere dumb components. Wanneer er data wordt gewijzigd of gecreeërd in een dumb component, stuurt het component de data terug naar zijn parent smart component. Via het model van de smart component wordt er in de store een nieuwe state gecreeërd op basis van de nieuwe of aangepaste data.

Er wordt momenteel geen rekening gehouden met het cachen en precachen van data indien de applicatie plots offline is. Dit komt wel aan bod in het hoofdstuk 6 'Onderzoek'

4.2 Server

De backend is opgebouwd volgens het 'serverless' (Richardson, 2015) principe. Hierbij vormen AWS Lambda's de ruggengraat van de backend infrastructuur. Er wordt gebruik gemaakt van verschillende microservices die elk een beperkte maar specifieke verantwoordelijkheid hebben over een bepaalde functionaliteit. Voor de persistentie van de data wordt per data 'type'⁵ een DynamoDB tabel gebruikt. Met behulp van SNS kunnen Lambda's of andere AWS services op de hoogte worden gebracht wanneer er een neveneffect moet optreden bij het afhandelen van een request. De API Gateway is de toeganspoort tot de backend en is verantwoordelijk voor het delegeren van de binnenkomende requests. In de business case wordt er gebruik gemaakt van een proxy⁶ lambda die de requests mapped naar de correcte lambda. Die voert dan de noodzakelijke stappen uit voor de request te verwerken.

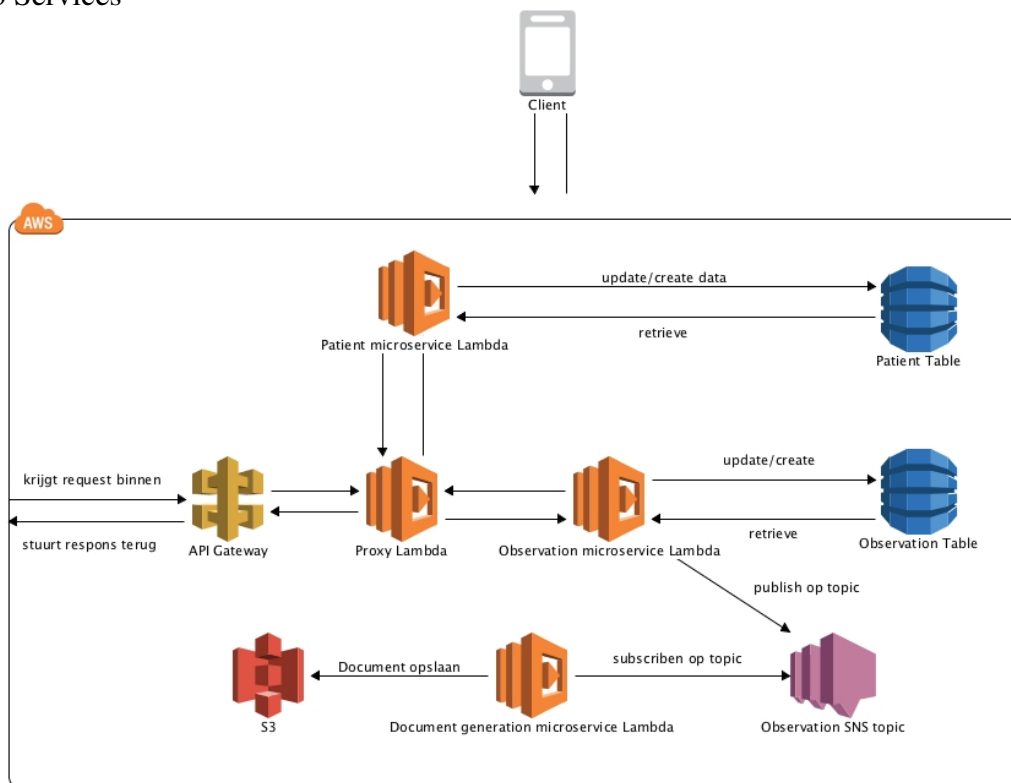
Figuur 4.2 is een voorbeeld van verschillende lambda's waarbij manipulaties worden uitgevoerd op patient -en observatiedata. De API Gateway stuurt een request naar de proxy lambda. Die triggert op zijn beurt de relevante lambda. Wanneer er een aanpassing gebeurt in de observatie data, plaatst de observatie microservice lambda een bericht op een SNS topic. Dit triggert dan de documentatie lambda die subscribed is op het SNS topic. Lambda en SNS werken dus perfect in tandem in het kader van een event-driven architectuur.

⁴'model' verwijst in deze context naar een 'service' in Angular terminologie

⁵Bijvoorbeeld patiënten en observaties zijn 2 verschillende datatypes

⁶Wanneer een lambda met proxy integration wordt gebruikt, hanteert de API Gateway een greedy 'catch-all' principe. Hierbij worden alle requests opgevangen en doorgestuurd naar de proxy lambda

Figuur 4.2: Voorbeeld van een serverless architectuur opgezet met verschillende Amazon Web Services



5. Synchronisatie methodes

In dit hoofdstuk worden enkele synchronisatie methodes en technieken besproken die data kunnen synchroniseren met de databank. In sectie 5.1 'Bestaande synchronisatie mogelijkheden' worden kort de bestaande oplossingen overlopen. Alternatieve synchronisatie patronen worden overlopen in sectie 5.2 'Alternatieve synchronisatie mogelijkheden'.

5.1 Bestaande synchronisatie mogelijkheden

Op basis van die research en de beperkingen van de business case was het immers eenvoudiger om te bepalen welke technologieën al dan niet in aanmerking komen voor het onderzoek. Synchronisatie technologieën zoals Firebase, CouchDB en Azure Mobile Apps laten eenvoudig toe dat een applicatie de data synchroniseert met de backend. De oplossingen maken gebruik van database replicatie. De client kopieert een deel van de database en bij het synchroniseren wordt de database in de backend dan aangepast op basis van de lokale kopie die de client bezit. De database rechtstreeks vanuit de client applicatie manipuleren brengt ook enkele andere nadelen met zich mee. De client applicatie en de databankstructuur zijn hard gekoppeld. Hierdoor is het moeilijk om de database te refactoren dan moet de client ook worden aangepast. De client kan ook bepaalde handelingen op de database uitvoeren die niet wenselijk zijn. Dit maakt bestaande synchronisatie oplossingen niet aantrekkelijk om het probleem van de business case op te lossen.

5.2 Alternatieve synchronisatie mogelijkheden

Door de beperkingen van de business case is het noodzakelijk om andere methodes te onderzoeken. In het volgende deel worden methodes overlopen die zowel kunnen gebruikt worden bij caching als bij synchronisatie. In hoofdstuk 6 'Onderzoek' wordt er dieper ingegaan op de toepassing van de verschillende methodes op de business case.

5.2.1 Read-Only

Deze caching methode laat toe dat de gebruiker de applicatie verder kan gebruiken wanneer die offline is. Door het beperken van de interacties in de client naar Read-Only is in dit scenario geen sprake van synchronisatie. De gebruiker mag immers de data niet manipuleren. Bij Read-Only is de richting van het dataverkeer unidirectioneel, van server naar client applicatie. De methode heeft volgende eigenschappen.

- De client vraagt de data¹ op van de server. De server is de SSOT en houdt alle data bij. Die data kan wijzigen wanneer de client offline is.
- De server retourneert de data. De opgevraagde data wordt gecached in de store. De store is nu de nieuwe SSOT voor de client.
- Alle manipulaties op de data worden geblokkeerd wanneer de applicatie offline is. De client dient enkel als view voor de data die lokaal is bijgehouden.
- De client synchroniseren wanneer die terug online is houdt in dat de oude data lokaal worden overschreven door de nieuwe opgevraagde data.
- Wanneer een view in de client wordt geladen, probeert de client de server te pollen voor nieuwe data. Indien de applicatie offline is, wordt de gecachte data uit de store gebruikt.

5.2.2 Read-Only Optimized

Het Read-Only Optimized caching methode is bijna identiek aan de Read-Only methode maar met enkele verschillen. Bij het synchroniseren worden enkel de data opgevraagd die gewijzigd zijn. Dit kan op verschillende manieren worden geïmplementeerd. Er kan een timestamp worden bijgehouden van de laatste wijziging of een version number die incrementeert bij elke wijziging. Op basis van de vergelijking tussen de lokale data en de data in de databank, kan de applicatie al dan niet beslissen om de lokale data te overschrijven. De richting van het dataverkeer is bidirectioneel omdat de client data moet meesturen met de request. Een nadeel van de Read-Only Optimised caching is de complexiteit die deze methode introduceert.

Read-Only Optimized kan ook server side wordt gebruikt voor het samenstellen van de state². Hier wordt verder op ingegaan in het hoofdstuk 6 'Onderzoek'.

¹ Applicatie moet online zijn om de initiële state op te bouwen.

² Redux state in JSON, voor de redux store in de client

5.2.3 First/Last Write Wins

Bij de First/Last Write Wins techniek gaat de server er van uit dat writes steeds in de juiste volgorde worden uitgevoerd. Er zijn twee varianten die elk, afhankelijk van de use case, de data kunnen synchroniseren. Bij First Write Wins zijn de data impliciet final. De data kunnen dan maar een keer worden aangepast. Alle andere pogingen worden verworpen. Bij de tweede variant Last Write Wins, worden de data steeds overschreven door de meest recente versie en kunnen de data op ieder moment worden aangepast door recentere data. De techniek wordt enkel maar server side gebruikt en verwerkt de gecachte³ transacties van de client.

Met deze methode gaat er onherroepelijk data verloren. Bij sommige use cases kan deze compromis worden gemaakt. Indien dit niet mogelijk is dan biedt 5.2.4 'Write with Conflict Detection' een betere oplossing.

5.2.4 Write with Conflict Detection

Het Read/Write with Conflict Detection methode is het complexere variant van First/Last Write Wins waarbij verschillende clients dezelfde data manipuleren op het moment dat de applicatie offline is. Bij deze techniek ligt de focus om minimale data te verliezen bij de synchronisatie. Wanneer er server side data moet worden overschreven zoals bij Last Write Wins, dan wordt de bestaande data toegevoegd aan de nieuwe data. Bij First-Write Wins kan er ook data worden toegevoegd zonder dat de oorspronkelijke data wordt overschreven. Indien de server een conflict vaststelt waarbij er geen beslissing kan worden genomen⁴, dan moet de eindgebruiker worden verwittigd. De gebruiker maakt dan de keuze welke data er moeten worden bewaard en welke er mogen worden overschreven. Bij voorkeur lost de server het conflict autonoom op, zonder tussenkomst van de gebruiker. Dit is echter niet bij elke use case mogelijk. Write with Conflict Detection zou volgende stappen kunnen doorlopen:

1. de server database houdt al de data bij
2. de applicatie houdt lokaal een subset van de data bij die kan worden gewijzigd
3. bij synchronisatie worden de aangepaste data die lokaal wordt opgeslagen naar de server en omgekeerd
4. in de server worden de data aangepast en alle conflicten worden geregistreerd voor verdere behandeling
5. de server verwittigt de gebruiker voor een conflict of de server kan zelf het conflict oplossen en de data aanpassen

³De client cacht de acties wanneer er geen verbinding is met de server. Zie hoofdstuk 6

⁴Bv. een UPDATE op een record dat verwijderd is door een DELETE operatie

6. Onderzoek

In dit hoofdstuk wordt dieper ingegaan op de verschillende caching -en synchronisatie methodes, hun toepassing op de businesscase en het proces voor het implementeren van synchronisatie. Volgende onderdelen komen aan bod in dit hoofdstuk:

- Online/Offline status registstratie
- Caching, preloading en Read-Only Optimised
- Planning bij het bepalen van synchronisatie
- Last/First Write Wins en Conflict resolution
- De gebruikte AWS services voor de synchronisatie in de backend

6.1 Online/Offline status registratie

Voor dat de verschillende caching- en synchronisatie methodes kunnen worden overlopen, is het belangrijk om te onderzoeken hoe de applicatie op een betrouwbare manier de status van de internetverbinding kan controleren. De status van de verbinding vormt immers de belangrijkste voorwaarde bij de beslissing om de data¹ al dan niet te cachen.

¹de request, in het geval van de business case

1. DOM API's aanspreken vanuit JavaScript. Helaas is er geen consistente (Mozilla, 2017b) cross-browser support op het controleren van de status van de verbinding zonder gebruik te maken van requests naar externe data. Een voorbeeld is luisteren naar events van het document.window object om online en offline status te controleren werkt niet op alle browsers. Een andere optie waarbij de status van de verbinding wordt opgevraagd met window.navigator.onLine werkt ook niet consistent in alle browsers.
2. Een request uitvoeren naar een externe bron: als de applicatie een request uitvoert naar een externe bron en de HTTP code controleert van de respons, dan kan de applicatie gemakkelijk afleiden wat de status van de verbinding is. Deze methode garandeert dat de applicatie correct kan vaststellen wanneer de status van de verbinding verandert. Deze methode ook gebruikt in libraries zoals Offline.js².

```
1 // met jQuery
2 $(window).bind("online", applicationBackOnline);
3 $(window).bind("offline", applicationOffline);
4
5 //IE met vendor specifieke objecten
6 window.onload = function() {
7     document.body.ononline = ConnectionEvent;
8     document.body.onoffline = ConnectionEvent;
9 }
10 // Met gebruik van event listeners
11 document.body.addEventListener("offline", function () {
12     //alert("offline")
13 }, false);
14 document.body.addEventListener("online", function () {
15     //alert("online")
16 }, false);
17
18 // met native window.navigator
19 if (navigator.onLine) {
20     // Online
21 } else {
22     // Offline
23 }
```

Listing 6.1: Verschillende methodes (StackOverflow, 2015) voor controleren offline en online status.

²Zie <https://github.com/hubspot/offline>

6.2 Caching van requests

Voor het onderzoek werd geopteerd op gebruik te maken van de methode waarbij een call wordt uitgevoerd om te controleren of de client online of offline is. Wanneer de applicatie registreert dat er geen verbinding meer is, worden alle POST/PUT/UPDATE³ requests gecached als afzonderlijke objecten in localStorage. Hiervoor wordt er gebruik gemaakt van localForage. Wanneer de applicatie terug online is, wordt er 1 request naar de server alle gecachte data doorgestuurd. Requests die een UPDATE uitvoeren op een object dat aangemaakt werd (CREATE) tijdens de offline status, worden allemaal gebundeld in een enkel object. Op die manier wordt de volgorde gegarandeerd voor het verwerken van de verschillende requests. Wanneer er dan synchronisatieproblemen optreden, moeten die in de lambda's worden opgevangen die de individuele request moeten verwerken.

```

1 private cache(key: string, command: Request) {
2   // Haalt de huidige cache uit localStorage als Promise
3   return storage.getItem<Action[]>(key).then(actions => {
4     // Default lege array indien actions niet bestaat
5     actions = actions || [];
6
7     // Omzetten van request naar data object
8     const data: any = command.serialize();
9     data.method = command.method;
10    data.path = command.path;
11
12    // Toevoegen aan array van requestobjecten
13    actions.push(data);
14
15    // Terug in storage plaatsen wanneer gecachte actie is
16    // localForage bibliotheek kiest opslagmedia (localStorage of
17    // IndexedDB)
18    return storage.setItem<Action[]>(key, actions);
19  });
20 }

```

Listing 6.2: Caching van een request

De connectie status van de de applicatie wordt bewaard als state in de redux store. Die kan worden opgevraagd als een Observable wanneer requests moeten worden uitgevoerd naar de backend of voor het tonen van een waarschuwing in de UI. Wanneer de status van de verbinding zou wijzingen, dan wordt de subscribe methode opnieuw aangeroepen. Op die manier kunnen verschillende configuraties worden ingesteld afhankelijk van de verbinding.

```

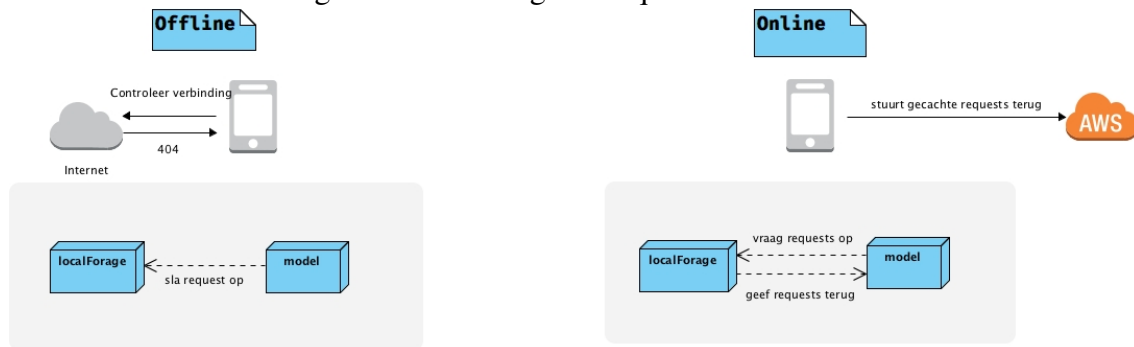
1 // Bevragen van de store
2 this.store.select(state => state.common.network.online)
3 // Retourneert een Observable
4 .subscribe(online =>
5   // reactie op boolean 'online'
6 )

```

Listing 6.3: Opvragen van de status van de verbinding aan de hand van de redux store

³Alle requests die geen POST/PUT/UPDATE request zijn worden genegeerd want die hebben geen invloed op de synchronisatie

Figuur 6.1: Caching van requests in de client



6.3 Data preloading in de client

Bij een applicatie die zowel online als offline moet werken, is het belangrijk dat de data wordt preloaded⁴ in de applicatie. Zo kan de gebruiker blijven verder werken indien de applicatie plots offline is. De redux store is de SSOT en bevat de preloaded data in-memory. De applicatie kan de preloaded data uit de store gebruiken bij het laden van de views. Het belangrijkste aandachtspunt is hierbij de schaalbaarheid van de oplossing. In de business case van Pridiktiv worden verschillende requests naar de server uitgevoerd naar de server om de data op te vragen. Wanneer de functionaliteit en data capaciteit van de applicatie toeneemt kan het aantal requests kan wel dramatisch stijgen. Door het single-threaded karakter van JavaScript is het echter niet mogelijk om parallel te werken. Een mogelijke oplossing voor dat probleem kan een web worker(Mozilla, 2017c) zijn.

6.3.1 Beperkingen door single-threaded omgeving

JavaScript is een single-threaded omgeving waardoor meerdere scripts niet parallel kunnen worden uitgevoerd. Dit vormt problemen wanneer een webapplicatie UI events, queries, een groot volume API data en tegelijk de DOM aanpassingen moet verwerken. Met technieken zoals `setTimeout()` en `setInterval()` is het mogelijk om concurrency te simuleren maar dit maakt de applicatie meteen een stuk complexer. Dankzij de HTML5 specificatie is het mogelijk om Web Workers te gebruiken. Deze laten toe om scripts in de achtergrond uit te voeren. Hierdoor is het mogelijk om complexe berekeningen concurrent uit te voeren zonder dat dit een performantie impact op de applicatie heeft.

6.3.2 Preloading methodes

Er zijn 3 methodes voor preloading die van toepassing zijn voor de business case.

⁴Preloading is het op voorhand inladen van data.

Serieel inladen van de data

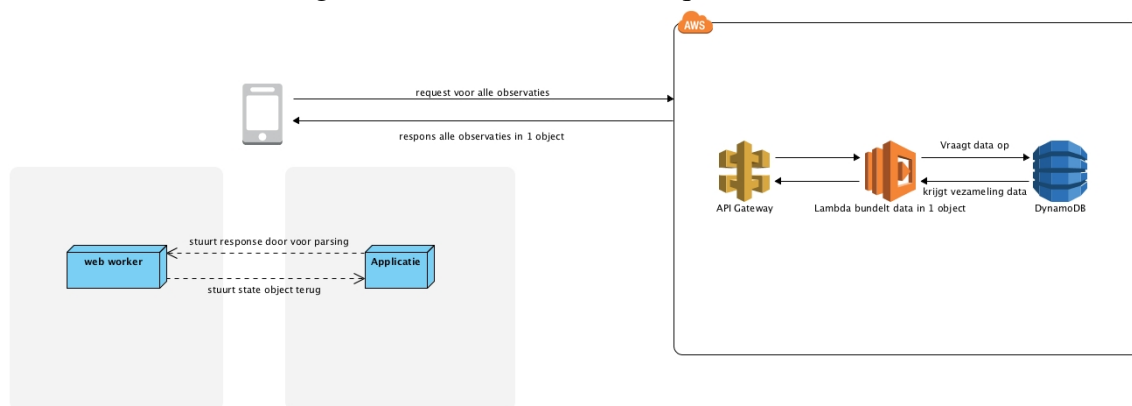
Bij deze methode wordt de data serieel opgevraagd van de server. Dit is de minst complexe oplossing en is ideaal wanneer het volume van de data beperkt is. Echter, hoe groter het data volume, hoe groter de impact op de performantie van de client applicatie. Door de vele requests en de vele aanpassingen van de in-memory Redux state wordt de main thread geblocked waardoor de UI niet kan worden gerenderd. Dit wekt de indruk bij de gebruiker dat de applicatie niet responsief is. Deze methode is dus enkel geschikt wanneer er maar een beperkt aantal requests zijn.

Web worker

Aan de hand van web workers is het mogelijk om parallel een ander Javascript applicatie te gebruiken. De API voor web workers wordt door alle (CanIUse.com, 2017) moderne browsers ondersteund. Met behulp van web workers is het mogelijk om CPU-intensieve taken onafhankelijk van de web pagina uit te voeren. Als men dit toepast op preloading kunnen web workers een van de volgende taken op zich nemen.

1. Data ophalen van de server en verwerken naar een state object.
2. Omzetten van response/data object naar state object voor de redux store

Figuur 6.2: Web worker die request verwerkt



Een nadeel bij web workers is de complexiteit wanneer externe scripts worden gebruikt. Als er meerdere dependencies zijn voor het uitvoeren van de code of als een superset zoals TypeScript wordt gebruikt, dan moet men gebruik maken van SystemJS⁵. SystemJS is echter niet combineerbaar met Angular CLI, de build tool die gebruikt wordt bij de business case. Door deze beperking is het werken met web workers niet de ideale oplossing voor de business case. Het is ook mogelijk om de Webpack⁶-configuratie file die Angular CLI gebruikt handmatig te configureren. Dit wordt echter sterk afgeraden omdat de meerwaarde van de bundling die door Angular CLI wordt uitgevoerd dan volledig verloren gaat.

⁵SystemJS is een dynamische module loader voor JavaScript ES6

⁶Webpack is een module bundler. Webpack laadt net als SystemJS de verschillende modules in maar bundelt die ook netjes in verschillende files. Op die manier moet een applicatie die heel wat dependencies heeft, enkel maar een file inladen.

```
1 TODO
```

Listing 6.4: Creatie van een web worker

```
1 TODO
```

Listing 6.5: Communicatie tussen applicatie en webworker

Redux state server side opbouwen met Read-Only Optimised

Een derde methode is het opzetten van een redux state in de backend die door de client kan worden opgevraagd. Op die manier moet enkel het state object worden toegevoegd aan de store wanneer het is opgevraagd van de server. Een nadeel van deze methode is de state van de client die nu sterk gekoppeld is aan de backend. Grote voordeel is dat de backend nu verantwoordelijk is voor alle intensieve operaties en dat de client applicatie enkel maar zijn state moet injecteren in de store. Bij deze methode kan de Read-Only Optimised techniek gebruikt worden bij het opstellen van de client state.

Toepassing business case

In de business case wordt de request serieel bevestigd van de server. Nadat de patiëntenlijst is opgevraagd, worden voor alle patiënten de relevante data opgevraagd. In de piloot versie waren dat ongeveer 75 personen die elk taken, notities, observaties, medische parameters en wondzorgdossiers kunnen bevatten. Dit leidt tot een 500 requests die in een zeer korte periode worden uitgevoerd. Het gebruik van web workers (W3C, 2017b) gecombineerd met een gebundelde data zou het aantal requests sterk kunnen verminderen. In de business case is er ook sprake van afbeeldingen die moeten worden gesynchroniseerd met de applicatie maar het cachen van BLOBs valt buiten de scope van dit onderzoek.

6.4 Data synchronisatie: planning bij development

Wanneer een synchronisatieoplossing moet worden ontwikkeld voor een applicatie, is het belangrijk om alle use cases in kaart te brengen waarbij er rekening moet worden gehouden met synchronisatie. Dit zorgt ervoor dat er geen functionaliteit over het hoofd wordt gezien bij het ontwikkelen van de synchronisatieoplossing. Ter illustratie wordt in de onderstaande voorbeeld de business case van Pridiktiv ontleed.

Functionaliteit	Synchronisatie
Taken	First-Write Wins of conflict resolution
Observaties	geen conflict mogelijk
Medische data	geen conflict mogelijk
Wondzorg	First-Write Wins of conflict resolution
Notities	Geen conflict mogelijk

De onderverdeling is het uitgangspunt voor de volgende stap namelijk het bepalen van de verschillende synchronisatie mogelijkheden. Zo kan bijvoorbeeld worden bepaald dat de gebruiker bij een conflict zelf een beslissing moet nemen. Of indien er een conflict optreedt bij een synchronisatie van bepaalde data, dat er automatisch voor Last/First Write Wins wordt gekozen. In het geval waarbij er conflict resolution moet gebeuren door de gebruiker, kunnen kan worden bepaald welke keuzes de gebruiker krijgen en welke keuzes de server zelf kan maken. Bovenstaande overzicht vormt dan als het ware de basis voor de ontwikkeling van de synchronisatie. Een ander voorbeeld is het deactiveren van bepaalde functionaliteit bij wanneer de applicatie offline is. Hierdoor kunnen potentieel complexe synchronisatieproblemen worden vermeden. Dit heeft echter als nadeel dat de applicatie niet meer de volledige functionaliteit kan aanbieden wanneer de verbinding is verbroken. Het al dan niet uitschakelen van bepaalde functionaliteit bij offline gebruik is sterk afhankelijk van de business case

6.5 Read-Only Optimised

De belangrijkste insteek voor deze methode is om zo efficiënt mogelijk gebruik te maken van de bandbreedte van de client of processor tijd serverside. Read-Only Optimised is dan ook eerder een optimalisatietechniek dan een synchronisatiemethode. Deze techniek laat toe om enkel nieuwe data binnen te halen waardoor data die de applicatie reeds bezit niet opnieuw worden opgevraagd. De complexiteit van deze methode stijgt evenredig met het aantal parameters dat in de data kan worden aangepast. Deze methode wordt vooral gebruikt bij het cachen van data voor offline gebruik.

6.5.1 Overzicht: Client

Bij Read-Only Optimised staat de synchronisatie van de client centraal. Tijdens de periode dat de client offline was, is het mogelijk dat er nieuwe data zijn aangemaakt. Om deze data te synchroniseren kan de client alle data opnieuw opvragen maar deze manier heeft enkele nadelen. Indien de data klein is zoals enkel tekst, dan is Read-Only Optimised triviaal en kunnen alle data opnieuw worden opgevraagd. Wanneer er echter grotere data moet worden ingeladen zoals afbeeldingen, dan kan Read-Only Optimised heel wat bandbreedte uitsparen. Dankzij de ngrx store is het ook gemakkelijk om manipulaties uit te voeren op de data, waardoor er eenvoudig objecten kunnen worden toegevoegd aan de data in de store. Belangrijke voorwaarde voor het uitvoeren van deze actie is het bijhouden in de client van de timestamp van de laatste update indien de data aanwezig zijn. Indien er geen timestamp aanwezig is, dan weet de client dat alle data moeten worden opgevraagd.

6.5.2 Overzicht: Server

De server houdt rekening met de verschillende timestamps om te bepalen welke data er moet worden geretourneerd.

6.5.3 Opmerkingen

Ondanks de perceptie dat dit een eenvoudige methode is om te implementeren, zijn er enkele opmerkingen zij deze methode:

1. In bovenstaand scenario wordt er enkel maar uitgegaan van nieuwe data, dus CREATE operaties. Indien de methode ook voor UPDATE moet werken is het in bepaalde gevallen noodzakelijk om ook de databank structuur aan te passen.
2. Bij klassieke SQL databanken kan gemakkelijk worden gefilterd op bepaalde velden zoals een 'updated veld' binnen een tabel. Bij DynamoDB is dit moeilijker als de waarde van de query geen deel uitmaakt van de sort -of partition key. In dat geval moet er dan gebruik worden gemaakt van een extra secondary index niet telkens alle partition key moet worden overlopen telkens wanneer er een GET request is.
3. Het is bij kleine data vaak eenvoudiger om steeds alle data op te vragen en geen gebruik te maken van de Read-Only Optimised methode. De complexiteit die Read-Only Optimised toevoegt bij eenvoudige scenario's maakt deze methode soms overbodig.

6.5.4 Toepassing

Met behulp van twee timestamps wordt er bijgehouden wanneer de laatste GET is uitgevoerd en wanneer de laatste aanpassing in een tabel heeft plaatsgevonden. Op die manier kan de backend bepalen welke informatie er moet worden geretourneerd. Wanneer er maar een beperkt volume data wordt geretourneerd dan zorgt de Read-Only Optimised methode nodeloos voor extra complexiteit. Het is daarom enkel aangeraden om enkel Read-Only Optimised te gebruiken indien er met grotere datasets wordt gewerkt.

Toepassing op de business case

In de backoffice en mobiele applicatie van Pridiktiv wordt er momenteel geen gebruikt gemaakt van Read-Only Optimised. Het Read-Only Optimised patroon zou in de business case van Pridiktiv eerder geschikt zijn wanneer de state server side wordt opgebouwd. Als de state voor de caching serverside wordt opgebouwd, dan zou men gebruik kunnen maken van Read-Only Optimised. Door de opgebouwde state te persisteren en de tegelijk de timestamps van de laatste change binnen een bepaalde categorie bij te houden in een 'changes' tabel zou het aantal queries dat nodig is om de state op te bouwen sterk kunnen verminderen.

6.6 Last/First Write Wins

Bij Last/First Write Wins gaat er onherroepelijk informatie verloren. Afhankelijk van de gekozen conflict resolution methode is het wordt de eerste of laatste write bijgehouden. Dit is de eenvoudigste manier van conflict resolution maar men moet bereidt zijn om een compromis te sluiten en informatie op te offeren in ruil voor minder complexiteit bij het

synchroniseren. Wanneer heel snel naar een offline/online synchronisatiemethode moet worden gezocht, biedt die de gemakkelijkste oplossing.

6.6.1 Overzicht: Client

Bij deze methode is de impact van de client miniem want de client weet niet dat er een synchronisatieprobleem zal optreden wanneer die data doorstuurt naar de server.

6.6.2 Overzicht: Server

Deze methode vindt plaats wanneer verschillende clients een verandering aanbrengen bij hetzelfde bestaande object. Hierdoor krijgt de databank verschillende waarden binnen en moet dat verplicht de databank er toe om te reageren. Afhankelijk van de methode die gekozen zijn er twee scenario's mogelijk bij deze synchronisatiemethode.

1. First Write Wins. Hier wordt enkel de eerste write van een object of row bijgehouden. Indien hetzelfde object opnieuw wordt aangepast dan wordt er een exceptie geworpen. Dit is enkel maar mogelijk in use cases waarbij een aanpassing in een object finaal is. Een voorbeeld uit de use case is bijvoorbeeld het volbrengen van een taak. Wanneer een taak is voltooid, is het niet meer mogelijk om die aan te passen. Met behulp van een completed flag kan de state van de taak worden bijgehouden. Wanneer de aanpassingen in een object niet finaal zijn, is een last Write wins een betere methode.
2. Last Write Wins. Bij Last Write Wins wordt enkel maar de laatste write operatie behouden. Er wordt geen vergelijking gemaakt met de waarde die reeds in de databank beschikbaar is en de nieuwe waarde die wordt doorgegeven. Men hanteert deze methode dan best enkel bij bestaande objecten waarbij het mogelijk is om UPDATE operaties op uit te voeren.

6.6.3 Opmerkingen

First/Last Write wins is een aantrekkelijke methode wanneer er op korte termijn synchronisatie moet worden gerealiseerd maar er zijn wel enkele opmerkingen bij deze methode.

- Er gaat onherroepelijk data verloren op deze manier. Indien de use case dit niet toelaat dan moet er worden gekeken naar andere conflict resolution policy.
- Het type van de data (finaal of niet-finaal) sluit de andere methode uit. Niet-finale data met First Write Wins maken de data impliciet finaal en immutable.

6.6.4 Toepassing

Toepassing op de business case

Momenteel hanteert de backend van Pridiktiv enkel een First/Last Write principe bij data. Naar de toekomst zou dit moeten veranderen naar een bredere oplossing die conflict

resolution kan aanbieden.

6.7 TODO ergens tussenplaatsen

De API Gateway geeft het object door aan de proxy lambda die de objecten op een queue plaatst. Een andere lambda leest alle boodschappen uit de queue en stuurt elke request terug naar de gateway lambda. De API Gateway wijst de correcte lambda aan die verantwoordelijk is voor de verwerking van de oorspronkelijke request.

6.8 Conflict Resolution

Bij synchronisatie gaat er idealiter geen informatie verloren of laat de business case gaan dataverlies toe. Daarom is het ook belangrijk om te kijken hoe data kan worden gesynchroniseerd op een manier waarbij geen data verloren gaan. Om het onderzoek binnen de restricties van de business case te laten passen, worden hier enkel AWS services gebruikt bij het synchronisatie. Een andere belangrijke opmerking dat het enkel maar gaat over UPDATE en DELETE operaties want Een CREATE operatie kan niet leiden tot een conflict wanneer het toestel offline is.

6.8.1 Overzicht: Client

Net zoals bij First/Last Write Wins, is de rol van de client beperkt. Wanneer de data zonder timestamp in de database wordt bewaard, is het belangrijk om een timestamp bij te houden wanneer nieuwe data zijn aangemaakt of aangepast. Zo heeft de server een referentiepunt om de data te verwerken.

6.8.2 Overzicht: Server

Timestamps bieden een eenvoudige oplossing om de controleren wanneer een bepaalde actie heeft plaatsgevonden. De timestamp van het nieuwe aangepaste object wordt vergeleken met de timestamp van het object dat reeds aanwezig is in de database. Zo kan bijvoorbeeld een timestamp worden bijgehouden in een 'modified' veld. Dat veld kan worden vergeleken met de timestamp van de UPDATE of DELETE die wordt doorgestuurd. Zo kan de server bepalen of er al dan niet een conflict is en wat er in het geval van een conflict moet gebeuren. Indien er geen timestamp sinds de laatste update wordt bewaard in het object, dan wordt synchroniseren zeer moeilijk en is First/Last Write Wins een betere oplossing. Het bijhouden van een timestamp alleen is echter niet voldoende. Het is belangrijk om op voorhand te bepalen wat de conflict resolution policies zijn en welke use cases er allemaal van toepassing zijn. Niet alle conflicten kunnen worden opgevangen door de server. Zo kan de server geen beslissing nemen wanneer een DELETE actie wordt uitgevoerd op een object en er na nog een update wordt doorgestuurd. De gebruiker moet

dan een waarschuwing krijgen dat zijn object is verwijderd en dan zijn aanpassingen niet zijn opgeslagen. Indien de gebruiker geen boodschap zou krijgen, dan wekt de server de perceptie dat er iets fout is gegaan bij de verwerking van de data.

6.8.3 Toepassing

Toepassing op de business case

UPDATE operatie

Pridiktiv maakt gebruik van DynamoDB voor het bijhouden van de data. In DynamoDB wordt er gewerkt met een Partition Key die kan worden vergeleken met een Primary Key in SQL-terminologie en optioneel een Sort Key die de Partition Key bevat in combinatie met een attribuut van het object. Daarnaast is het ook nog mogelijk om Secondary Indices te creëren die functioneren zoals Sort keys. Het is belangrijk dat de timestamp de Partition Key is of deel uitmaakt van de samengestelde Sort Key. Indien niet, is de DynamoDB verplicht om een full table scan uit te voeren op de tabel en over die data set te filteren. Omdat de timestamp deel uitmaakt van de Primary Key, is het bij updates eenvoudig om bij te houden wanneer de laatste update is uitgevoerd en om te voorkomen dat een UPDATE gedupliceerd wordt. Dit laatste wordt vermeden doordat de Partition Key unique moet zijn en dus met andere woorden kan die niet worden gedupliceerd zonder dat DynamoDB een exceptie gooit.

DELETE operatie

Het is in de business case van Pridiktiv niet mogelijk om data te verwijderen. Alle data moet zichtbaar blijven in een historiek. Hierdoor is het niet toegelaten om DELETE operaties uit voeren, enkel UPDATE operaties.

6.9 Open-source libraries

Voor het synchroniseren van de data bij Pridiktiv in AWS wordt er gebruik gemaakt van twee verschillende Open Source libraries. Deze zijn gemaakt om een oplossing aan te bieden voor Pridiktiv en in kader van dit onderzoek. De componenten zijn geschreven in JavaScript voor NodeJS 4.3 en hoger. Hierdoor is het mogelijk om nieuwere features uit ES6 te gebruiken zoals Promises. Het is specifiek ontwikkeld om te werken met Simple Queue Service van AWS maar biedt de flexibiliteit om gebruikt te worden buiten AWS Lambda, bv in AWS EC2 instances die ook NodeJS gebruiken. Deze libraries zijn ontwikkeld in kader van dit onderzoek en worden gebruikt binnen de productieomgeving van de Pridiktiv applicaties.

6.9.1 aws-sqs-push

Deze library heeft een functie die berichten op de SQS queue plaatst en een Promise retourneert wanneer er een bericht op de queue is geplaatst. De naam van de queue wordt met een hulpfunctie `aws-sqs-geturl` opgevraagd. Op deze manier is de gebruiker van deze library niet verplicht om de volledige ARN naam van de SQS door te geven. Wanneer de message een object bevat is dan wordt het object met `JSON.stringify()` automatisch omgevormd naar een string.

```
1  const sqsPush = require('aws-sqs-push');
2
3
4  sqsPush('QueueName', 'SomeMessage').then(messageId => {
5      console.log(messageId);
6      //=> '8a98f4d0-078b-5176-9af2-bbd871660ecb'
7  });
8
9  sqsPush('QueueName', 'SomeMessage', {awsAccountId: '123456789101',
10      }).then(messageId => {
11      console.log(messageId);
12      //=> '8a98f4d0-078b-5176-9af2-bbd871660ecb'
13  });
```

Listing 6.6: Voorbeeld dat data plaatst op een SQS queue met behulp van de `aws-sqs-push` library

6.9.2 aws-sqs-poll

De `aws-sqs-poll` library vraagt aan de SQS queue berichten. Die worden dan geretourneerd als een string of indien het een stringified object is, als een object. Net zoals bij de `aws-sqs-push` wordt er gebruikt gemaakt van een hulplibrary voor het opvragen van de ARN naam van de queue.

```

1
2  const awsSqsPoll = require('aws-sqs-poll');
3
4  awsSqsPoll('QueueName')
5    // ["MessageId": "28f61fd2-b9ca-4cb9-879a-71ea8bce4636",
6    //   "ReceiptHandle": "
7    //     AQEB9mnsxtAZlwnDERxn3yADAP96QReOKPbqaKXLvvchqmD4jAr",
8    //   "MD5OfBody": "098f6bcd4621d373cade4e832627b4f6",
9    //   "Body": "test"]
10
11  awsSqsPoll('QueueName', {AwsAccountId: '123456789012',
12    numberOfMessages: 1, timeout: 20, json: false})
13    // ["MessageId": "28f61fd2-b9ca-4cb9-879a-71ea8bce4636",
14    //   "ReceiptHandle": "
15    //     AQEB9mnsxtAZlwnDERxn3yADAP96QReOKPbqaKXLvvchqmD4jAr",
16    //   "MD5OfBody": "098f6bcd4621d373cade4e832627b4f6",
17    //   "Body": "test"]

```

Listing 6.7: Voorbeeld hoe berichten worden opgehaald van de SQS queue

6.9.3 aws-sqs-deletemessage

Een kleine functie die SQS messages verwijderd op basis van de `ReceiptHandle` die wordt meegestuurd wanneer er een bericht wordt opgevraagd. De message is verwerkt dan mag het bericht zonder problemen worden verwijderd van de queue. Net zoals bij de andere bovenstaande functies, heb je altijd de ARN naam nodig van de SQS queue die je wenst op te roepen. Dus hier wordt opnieuw de hulpfunctie `aws-sqs-geturl` nodig.

```

1
2  const awsSqsDeletemessage = require('aws-sqs-deletemessage');
3
4  awsSqsDeletemessage('somequeue', 'SasuWXPJB+
5    CwLj1FjgXUv1uSj1gUPAWV66FU/').then(id => {
6    console.log(id);
7    //=> 'b5293cb5-d306-4a17-9048-b263635abe4
8  });
9
10  awsSqsDeletemessage('somequeue', 'SasuWXPJB+
11    CwLj1FjgXUv1uSj1gUPAWV66FU/', {awsAccountId: '123456789012'}).
12    then(id => {
13    console.log(id);
14    //=> 'b5293cb5-d306-4a17-9048-b263635abe4
15  });

```

Listing 6.8: Voorbeeld hoe een bericht wordt verwijderd van SQS nadat het is verwerkt

6.9.4 aws-sqs-geturl

Hulplibrary die de ARN naam opvraagt van een specifieke SQS queue. Aan de hand van de AWS root account id, die automatisch wordt meegestuurd in een request, kan samen met de naam de ARN naam van een bepaalde SQS queue worden opgehaald. De AWS JavaScript SDK laat dit ook toe maar de functie gebruikt callbacks en een complexere configuratie. Door het gebruik van deze hulplibrary is de configuratie verwaarloosbaar en wordt er geen gebruik gemaakt van callbacks maar van promises

```
1  const awsGetSqsUrl = require('aws-sqs-geturl');
2
3
4  awsGetSqsUrl('somequeue').then(url => {
5    console.log(url);
6    //=> https://sqs.eu-west-1.amazonaws.com/123456789111/somequeue
7  });
8
9  awsGetSqsUrl('anotherqueue', {awsAccountId: '123456789012'}).then
10    (url => {
11      console.log(url);
12      //=> https://sqs.us-west-1.amazonaws.com/123456789012/
13      anotherqueue
14    });
```

Listing 6.9: Voorbeeld hoe de ARN van een SQS wordt opgehaald

7. Conclusie

Afhankelijk van de restricties waarbinnen een applicatie moet worden gebouwd, kan men voor online/offline synchronisatie gebruik maken van de een volledig oplossing waarbij synchronisatie wordt aangeboden als een deel van totaal pakket zoals Microsoft Azure Applications, CouchDB en Google's Firebase. Deze opties werden maar kort toegelicht en het belangrijkste gegeven is hierbij dat de synchronisatie bijna volledig automatisch verloopt met enkel wat. Een allesomvattende oplossing is echter niet geschikt voor elke use case. In het voorbeeld van Pridiktiv is de nood aan een managed database en het gebruik van een serverless architectuur met AWS Lambda's de belangrijkste redenen om zelf de synchronisatie methodes in te bouwen in de huidige applicatie.

Wanneer online/offline synchronisatie moet worden geïmplementeerd in een applicatie, bestaat de eerste stap om de use cases onder te verdelen in de verschillende manieren voor synchronisatie. Wanneer absoluut geen data mag verloren gaan moet men resoluut kiezen voor conflict resolution. Wanneer niet alle data belangrijk is, kan er worden geopteerd voor een First/Last Write Wins techniek waarbij er onherroepelijk informatie verloren gaat. De belangrijkste conclusie die men uit het onderzoek kan trekken is dat er geen allesomvattende methode bestaat maar eerder een verzameling aan technieken op om synchronisatie van data uit de client en server te waarborgen. Wanneer er synchronisatie wordt ingebouwd is het belangrijk om ook rekening te houden met performantie en efficiëntie en de verantwoordelijkheid te verdelen tussen client -en server side. Voor de applicatie van Pridiktiv, waarbij er met gevoelige medische data wordt gewerkt, is het absoluut noodzakelijk dat er geen belangrijke data verloren gaat en alles netjes gesynchroniseerd word met de achterliggende backend. Ook naar user experience is synchronisatie een belangrijke factor. Eindgebruikers hebben enkel maar baat bij een robuuste offline/online synchronisatie. De applicatie kan bij een onderbreking in de connectie nog verder worden gebruikt zonder problemen. Afhankelijk van het belang van de data wordt alles

ook zorgvuldig gesynchroniseerd indien de gebruiker terug online gaat. Teruggekoppeld naar de use case van Pridiktiv, houdt dit in dat de applicatie ook zonder problemen kan worden gebruikt op locaties zoals woonzorgcentra waarbij er vaak maar beperkte of geen internetverbinding beschikbaar is.

Samengevat kan er worden gesteld dat het probleem van synchronisatie zeer specifiek is aan de use case en welke data men wenst te persisteren maar wel essentieel is voor een aangename user experience. Een interessante piste voor Amazon is dan ook zonder twijfel een service waarbij data uit mobiele applicaties of andere externe data bronnen gesynchroniseerd wordt met DynamoDB (of Aurora, een andere managed databank die Amazon Web Services aanbiedt). Op die manier zou het dan ook mogelijk zijn om bij complexe use cases de data te synchroniseren met de databank. Dit zou zonder twijfel de aantrekkelijkheid van AWS verhogen. Terwijl het onderzoek zich voornamelijk heeft gefocust op data synchronisatietechnieken, kan het interessant zijn om bijvoorbeeld de kost in kaart te brengen om over stappen van een fully-managed database naar een andere data source provider die synchronisatie toelaat zonder dat men zelf veel rekening moet houden met synchronisatie.

Bibliografie

- Billiet, B. (2016). Scalable Angular 2 Architecture. Verkregen van <http://blog.brecht.io/A-scalable-angular2-architecture/>
- Camden, R. (2016). *Client-Side Data Storage*. O'Reilly.
- CanIUse.com. (2017). Web Workers browser support. Web. Geraadpleegd op 20 mei 2017.
- First, O. (2017). Offline first mission statement. Verkregen van <http://offlinefirst.org/>
- Gechev, M. (2016). Scalable Angular applications. Verkregen van <http://blog.mgechev.com/2016/04/10/scalable-javascript-single-page-app-angular2-application-architecture/>
- Google. (2017). Why develop for offline? Verkregen van https://developer.chrome.com/apps/offline_apps
- Leemans, B. (2013). Overview of the IndexedDB API. Verkregen van <https://developer.mozilla.org/nl/docs/IndexedDB>
- Mozilla. (2017a). Explanation on Promises. Verkregen van https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- Mozilla. (2017b). Online and Offline Events. Web. Geraadpleegd op 3 maart 2017. Verkregen van https://developer.mozilla.org/en-US/docs/Online_and_offline_events
- Mozilla. (2017c). Using Web Workers. Web. Verkregen van https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
- Mozilla. (2017d). Web API Overview. Verkregen van <https://developer.mozilla.org/nl/docs/WebAPI>
- ngrx. (2017). Documentation for ngrx. Verkregen van <https://github.com/ngrx/store>
- ReactiveX. (2017). Documentation on how to use RX. Verkregen van <http://reactivex.io/intro.html>
- Richardson, C. (2015). Overview of microservices. Verkregen van <http://microservices.io/patterns/microservices.html>

- StackOverflow. (2015). How to detect online and offline events cross-browser. Web. Geraadpleegd op 20 april. Verkregen van <https://stackoverflow.com/questions/3181080/how-to-detect-online-offline-event-cross-browser>
- W3C. (2017a). Overview of W3C Specification. Verkregen van <https://www.w3.org/TR/webdatabase/>
- W3C. (2017b). Specification for Webworkers. Verkregen van <https://w3c.github.io/workers/>
- WHATWG. (2017). Standard for storage by WHATWG. Verkregen van <https://html.spec.whatwg.org/multipage/webstorage.html>

Lijst van figuren

2.1	Overzicht van een schaalbare web applicatie	17
2.2	Voorbeeld van de flow bij een dispatch in Redux	18
4.1	Flow van de mobiele applicatie	28
4.2	Voorbeeld van een serverless architectuur opgezet met verschillende Amazon Web Services	30
6.1	Caching van requests in de client	38
6.2	Web worker die request verwerkt	39

Lijst van tabellen