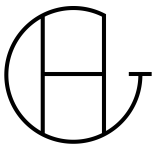


HoGent

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 stagementor 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

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 imperatieve[1] 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 geïnitialiseerd 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 geïnitialiseerd door de server of 'publisher' van de informatie via internet.

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 om de functionaliteit van een monolitische backend op te delen in verschillende kleine services.

[1] Imperatieve 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, onderzoeken en toepassen op de business case. In de sectie 2.1 'Business case' wordt de business case van Pridiktiv.care - into.care toegelicht. Daarna in sectie 2.2 'Terminologie' worden de belangrijkste begrippen van dit onderzoek 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. UX[1] 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.

[1] UX 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 lokaal opgeslagen. Wanneer het toestel terug online komt, moeten deze waarden worden doorgestuurd naar de backend. Omdat het medische data zijn, is essentieel dat er geen data verloren gaat. 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 'Opstelling' kan u een gedetailleerd bespreking terugvinden van de verschillende applicaties.

Client: Mobile applicatie

De huidige applicatie is een Angular[2] applicatie met Redux als state container. Voor lokale opslag wordt momenteel gebruik gemaakt van Mozilla's localForage[3] 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.

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

[3]localForage 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 AWS[4] 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 verschillende componenten van het onderzoek aan bod. De volledige lijst met begrippen en termen kan u terugvinden onder het hoofdstuk 'Glossarium'.

2.2.1 Huidige Web APIs voor lokale opslag

Er zijn momenteel 4 APIs (Mozilla, 2017e) 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 gestandardiseerd 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 sessionStorageAPIs 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.

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

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, 2017e) met localStorage is dat sessionStorage een vervalttijd heeft en de inhoud van de sessionStorage wordt verwijderd wanneer de sessie vervalt. Een sessie vervalt bijvoorbeeld hij 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.

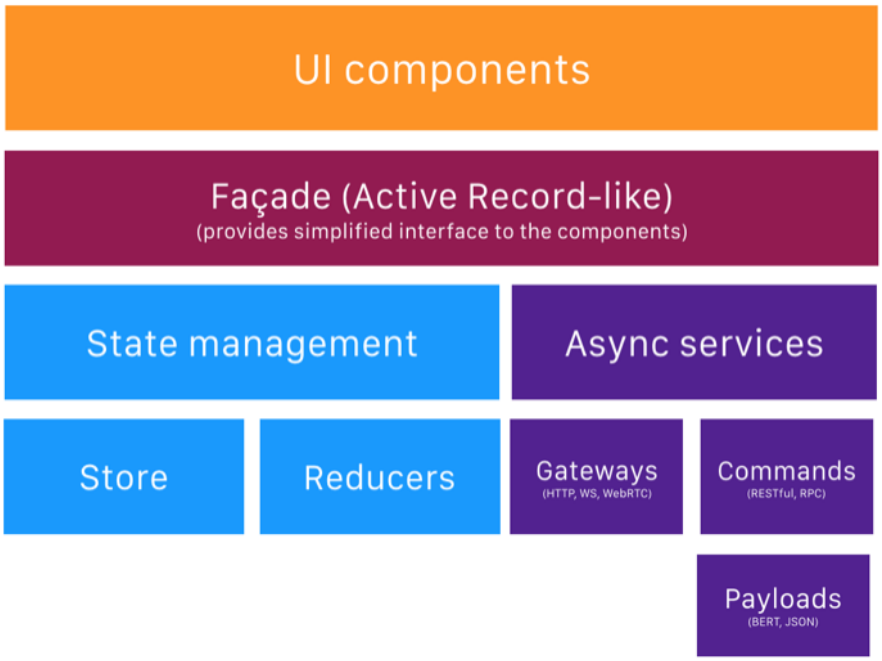
Web SQL - deprecated

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, 2017) van Web SQL omdat er te weinig onafhankelijke implementaties zijn van Web SQL. Het gebruik van Web SQL wordt sterk afgeraden door het deprecated status[5] 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(Gechev, 2016) 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 word 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.

[6]Zie 2.2.5 Redux store: een state container

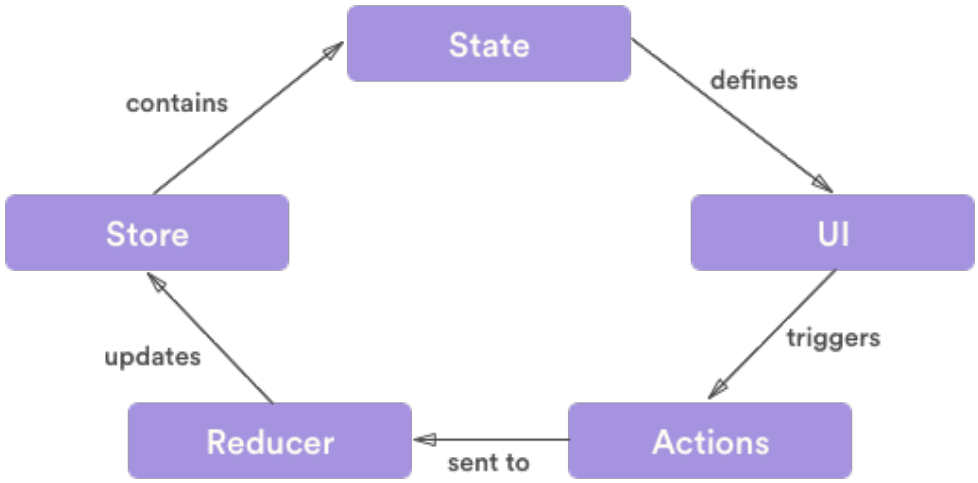
2.2.4 Redux store: een state container

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 creeëren 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 creeën. Een reducer[7] is een 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 gebruikt 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(Meents, 2016) van de flow bij een dispatch in Redux



[7]Een reducer 'verandert' de state door een nieuwe state te genereren. Op die manier wordt de immutability van redux gerespecteerd. De action bepaalt of er al dan niet een nieuwe state moet worden gegenereerd.

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 (Offline First, 2017) binnen web development waarbij offline gebruik als de basis wordt beschouwd van de applicatie. Net zoals bij 'Progressive Enhancement' wordt online functionaliteit beschouwt 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. 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-managed[8] 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[9] als zichzelf. De ontwikkelomgeving[10] simuleert amper de omgeving(Europese Commissie, 2015)(Europese Commissie, 2016) van de eindgebruiker (Google, 2017b). 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.

[8] fully-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.

[9] De laatste smartphone met een up-to-date besturingssysteem en snelle internet verbinding

[10] Een snelle desktop of laptop met een betrouwbare en snelle internet verbinding

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(Go, 2015):

- 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.

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 oplossingen zijn 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 een 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 kunnen de applicatie en 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 overschakelen van een offline status naar een online status en omgekeerd, zonder dat er daarbij belangrijke medische data verloren gaat.

2.5 Opzet van deze bachelorproef

Na de inleiding maakt het onderzoek gebruik van de volgende structuur.

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 de synthese van het onderzoek terugvinden. 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 startte met research naar de gebruikte technologieën in de business case en bestaande synchronisatie mogelijkheden. Bij het uitvoeren van deze fase, gebeurde de vaststelling 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 abstractielaag[1] 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 sandbox[2] 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. Voor het in beeld brengen van de architectuur, wordt er gebruik gemaakt van component diagrammen ontworpen met Visual Paradigm. Die maken gebruik van AWS Service Icons waarvoor AWS toestemming[3] geeft om het te gebruiken in dit onderzoek.

[1] Bij 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.

[2] Een 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] zie https://aws.amazon.com/architecture/icons/

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 preloaden van data in het geval de applicatie plots van status[4] 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 plaatst 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 beperkte 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.

[4] Van offline naar online en omgekeerd

4. Opstelling

In dit hoofdstuk wordt er dieper ingegaan op de verschillende onderdelen van het Pridiktiv platform[1]. Met behulp van component diagrammen[2] worden de architectuur en interactie visueel voorgesteld.

4.1 Client

4.1.1 Client: Backoffice applicatie

De backoffice is een Angular CLI 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.

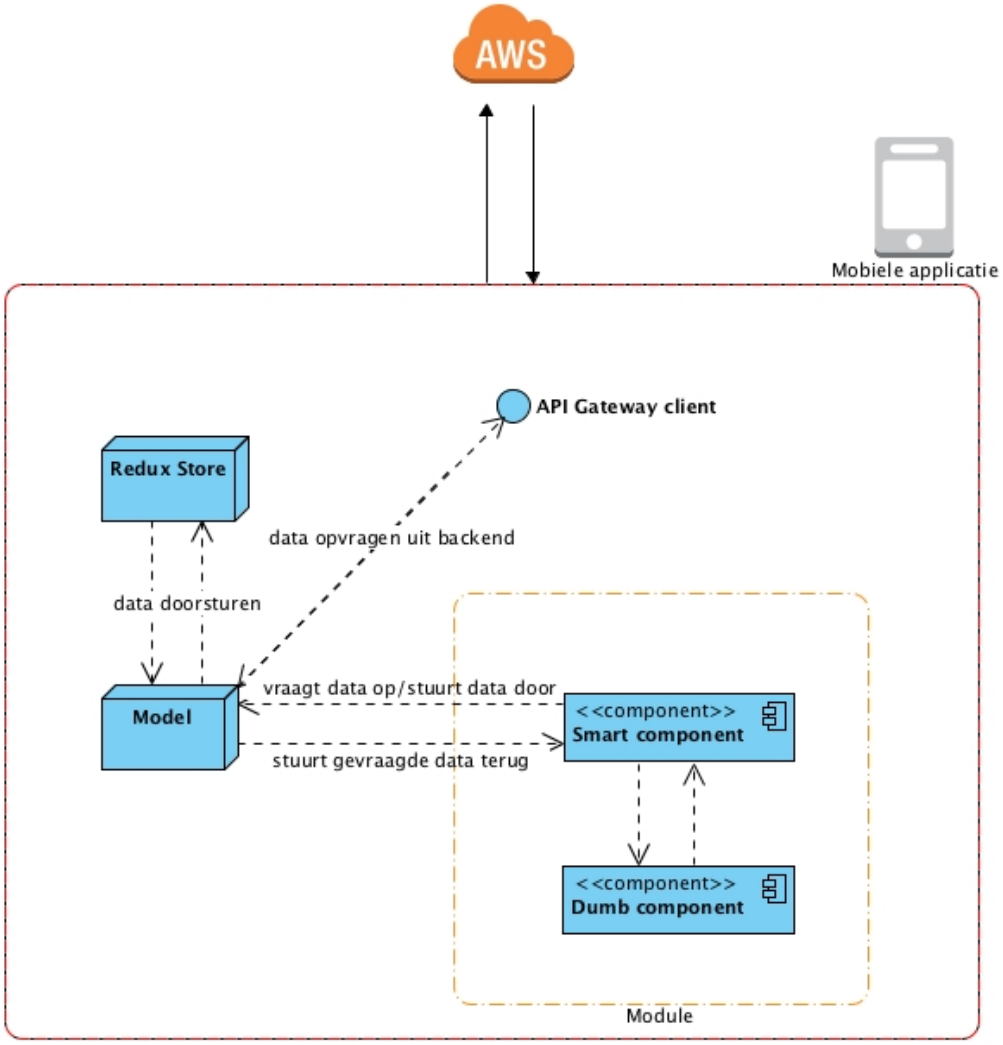
[1] platform of 'solution stack': Een set van componenten en software subsystemen die een geheel vormen waarmee de business doelstellingen worden bereikt

[2] Een 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 CLI 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 modules[3] die elke een feature voorstellen in de applicatie.

[3]Een 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 model[4] 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 proxy[6] lambda die de requests mapped naar de correcte lambda. Die voert dan de noodzakelijke stappen uit voor de request te verwerken.

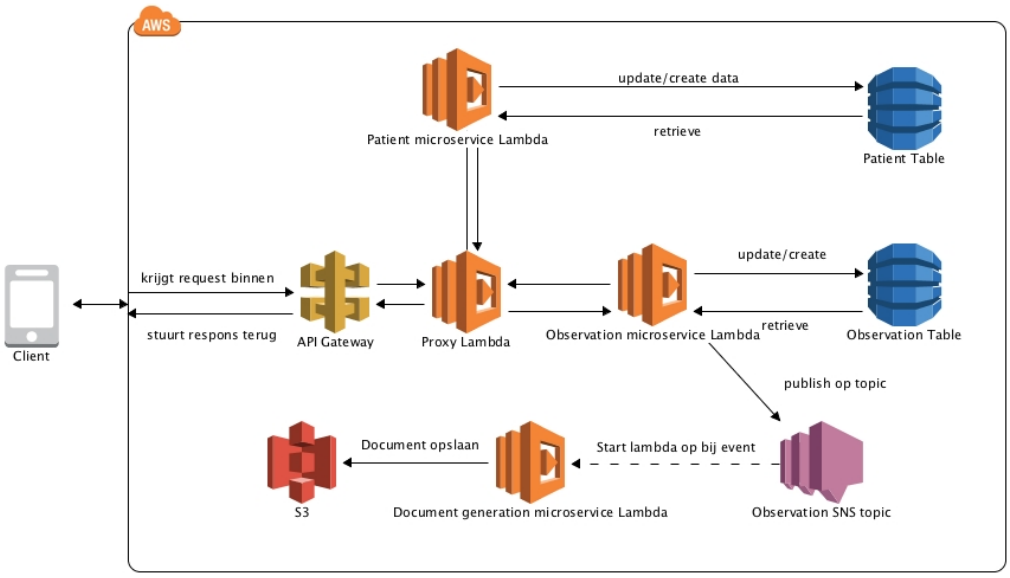
Figuur 4.2 is een voorbeeld van verschillende lambda's waarbij manipulaties worden uitgevoerd op patient -en observatiedata. De API Gateway stuurt een request naar de proxy lambda. Die triggert op zijn beurt de relevante lambda. Wanneer er een aanpassing gebeurt in de observatie data, plaatst de observatie 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

[5] Bijvoorbeeld patiënten en observaties zijn 2 verschillende datatypes

[6] Wanneer 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(Google, 2017a), CouchDB(Apache CouchDB, 2016) en Azure Mobile Apps(Microsoft Azure Documentation, 2017) laten eenvoudig toe dat een applicatie de data synchroniseert met de backend. De oplossingen maken gebruik van database replicatie. De client kopieërt 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(OutSystems, 2016) 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 data[1] 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

De 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 state[2]. 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 gecachte[3] 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. 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 genomen[4], 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 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
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
6. de client vraagt de meest recente data terug

[3] De client bewaart de acties wanneer er geen verbinding is met de server. Zie hoofdstuk 6
[4] Bv. 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 en offline status registratie
- 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
- libraries ontwikkeld in kader van het onderzoek

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 data[1] al dan niet te cachen.

[1] de 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 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 wordt ook gebruikt in libraries zoals Offline.js[2].

```
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(Stack Overflow, 2015) voor controleren offline en online status.

[2]Zie 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/UPDATE[3] 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
           toegevoegd
16     // localForage blibliotheek kiest opslagmedia (localStorage of
           IndexedDB)
17     return storage.setItem<Action[]>(key, actions);
18   });
19 }
```

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

[3] Alle 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 preloaded[4] 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, 2017d) 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.

[4]Preloading is het op voorhand inladen van data.

Serieel inladen van de data

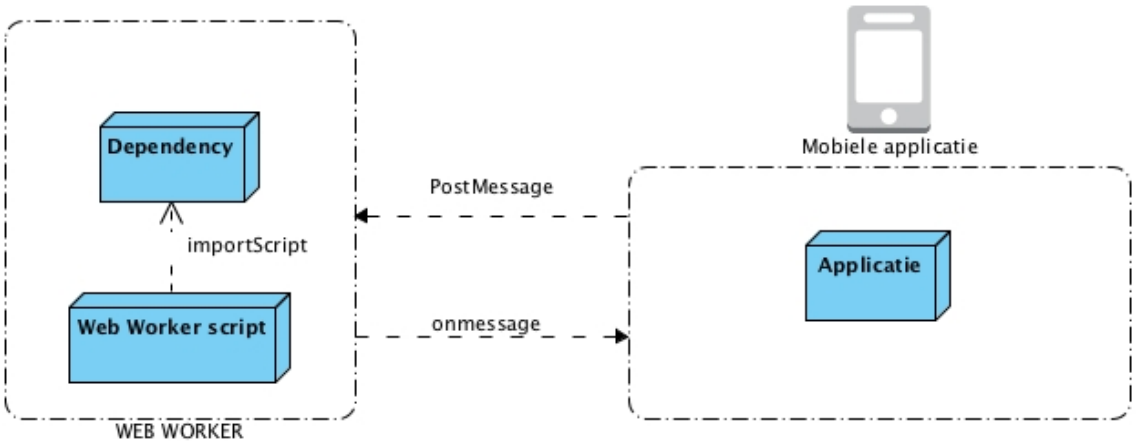
Bij deze methode wordt de data serieel opgevraagd van de server. Dit is de minst complexe oplossing en is ideaal wanneer het volume van de data beperkt is. Echter, hoe groter het data volume, hoe groter de impact op de performantie van de client applicatie. Door de vele requests en de vele aanpassingen 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: Interactie tussen web worker en applicatie



Een nadeel bij het gebruik van 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 van JavaScript zoals TypeScript wordt gebruikt, dan moet men gebruik maken van SystemJS[5]. 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 Webpack[6]-configuratie file die Angular CLI gebruikt handmatig te configureren. Dit wordt

[5]SystemJS is een dynamische module loader voor JavaScript ES6

[6]Webpack 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 maar een beperkt (4) aantal files inladen.

echter sterk afgeraden omdat de meerwaarde van de bundling die door Angular CLI wordt uitgevoerd dan volledig verloren gaat.

```
1 var myWorker = new Worker('my_worker_jsfile.js');
```

Listing 6.4: Creatie van een web worker

Bij de creatie van een web worker moet het pad naar de JavaScript file die de code van de web worker bevat worden opgegeven. Wanneer Angular CLI wordt gebruikt, dan moet het pad ook worden toegevoegd aan de scripts in de configuratie file.

```
1 // data1 en data2 worden doorgegeven aan de web worker
2 myWorker.postMessage([data1, data2]);
```

Listing 6.5: Communicatie tussen applicatie en web worker

Met behulp van berichten kan er worden gecommuniceerd tussen de web worker en de applicatie die de web worker aanspreekt. De data wordt gekopieerd aan de hand van een algoritme(Mozilla, 2017c)[7] en doorgestuurd naar de web worker. Omdat de data niet wordt geshared tussen de applicatie en de web worker

```
1 // Importeren van externe scripts of libraries
2 importScripts('foo.js');
3
4 // functie die het bericht verwerkt dat de client heeft verstuurd
     naar de web worker
5 onmessage = function(e) {
6   var workerResult = 'Result: ' + (e.data[0] * e.data[1]);
7   postMessage(workerResult);
8 }
```

Listing 6.6: Voorbeeld hoe een web worker een boodschap verwerkt

Met behulp van importScript kunnen externe libraries worden gebruikt in de web worker. In de business case zou dat bijvoorbeeld kunnen de API Gateway client kunnen zijn die de communicatie voorziet met de backend. Daarnaast maakt de web worker gebruik van een eenvoudige API om de boodschappen van de applicatie te verwerken.

```
1 // Methode voor het verwerken van de boodschap
2 myWorker.onmessage = function(e) {
3   result = e.data; // Dit result kan dan worden gebruikt in de
       applicatie
4   console.log('Message received from worker');
5 }
```

Listing 6.7: Voorbeeld van hoe de applicatie de respons van een web worker verwerkt

Tenslotte moet de applicatie die de web worker aanroept de boodschap die wordt teruggestuurd van de web worker nog verwerken. Bij de business case zou dit bijvoorbeeld de opgebouwde state kunnen zijn of deel van de state.

[7] Structured Cloning Algoritme laat doe om complexe JavaScript objecten te kopiëren en biedt meer flexibiliteit dan JSON. Net zoals bij JSON worden functies niet toegestaan.

Redux state server side opbouwen met Read-Only Optimised

Een derde methode is het opzetten van een redux state in de backend de client kan opvragen. 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. De belangrijkste insteek voor deze methode is om zo efficiënt mogelijk gebruik te maken van de bandbreedte van de client of processortijd server side. Read-Only Optimised is dan ook eerder een optimalisatietechniek dan een synchronisatiemethode. De caching techniek zorgt ervoor dat de state op een efficiënte manier kan worden gegeneerd. De state kan worden onderverdeeld in de verschillende features de die client applicatie bevat. Daarnaast kan er ook per 'feature state' een timestamp worden bijgehouden van de laatste creatie. Dankzij AWS SNS is het echter gemakkelijker om de state opnieuw te genereren wanneer een event plaatsvind[8] en zo event-driven te werken in plaats van timestamps te vergelijken. Op die manier bevat elke 'feature state' de laatste versie. Wanneer de volledige state dan wordt opgevraagd tijdens het preloaden in de client, wordt via deze methode gegarandeerd dat de client de laatste informatie ontvangt.

Toepassing business case

In de business case wordt momenteel de data serieel opgevraagd aan de server. Bijvoorbeeld nadat de patiëntenlijst is opgevraagd, worden voor alle patiënten de relevante data opgevraagd. In de piloot versie zijn dat ongeveer 75 personen die elk taken, notities, observaties, medische parameters en wondzorgdossiers kunnen bevatten. Dit leidt gemakkelijk tot een 500 requests die in een zeer korte periode worden uitgevoerd. De state in de backend opbouwen is de volgende stap die wordt gepland voor het optimaliseren van de preloading.

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 van het proces

Data synchronisatie moet worden beschouwd als een continue proces(Data Integration, 2015) waarbij synchronisatie niet mag worden beschouwd als een eenmalige gebeurtenis. Het is daarom 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 een synchronisatie oplossing.

[8] SNS bevat een topic waarop een AWS service een boodschap kan 'publishen'. Andere lambda's kunnen 'subscriben' op de SNS en kunnen dan die boodschap verwerken.

Bij de business case was dit dan ook de eerste stap voor het in kaart brengen van de verschillende use cases. Ter illustratie wordt in de onderstaande voorbeeld de business case van Pridiktiv ontleed.

Tabel 6.1: Overzicht planning synchronisatie

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 synchronisatieoplossing. 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 use case.

6.5 Synchronisatie

6.5.1 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 deze methode de gemakkelijkste oplossing.

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. De client is enkel verantwoordelijk voor het correct cachen van de requests wanneer de applicatie offline is.

Overzicht: Server

Deze methode vind 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 volbracht, 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.

Opmerkingen

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

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 een bredere oplossing die conflict resolution kan aanbieden.

6.5.2 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.

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. De belangrijkste bijdrage van de client is het correct cachen van de request voor verdere afhandeling wanneer de applicatie terug online is.

Overzicht: Server

Timestamps bieden een eenvoudige oplossing om om te 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. 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. Naar user experience toe is het beter om het conflict in de server op te lossen. Soms is dat niet mogelijk, zoals een DELETE operatie, en moet de eindgebruiker zelf een keuze maken.

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. 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 date te verwijderen. Alle data moet zichtbaar blijven in een historiek. Hierdoor is het niet toegelaten om DELETE operaties uit voeren, enkel UPDATE operaties. Bij DELETE operaties is het altijd noodzakelijk om de gebruiker een keuze aan te bieden indien er een synchronisatieconflict is. Aangezien dit niet van toepassing is binnen de business case wordt hier niet verder op in gegaan.

6.6 Architectuur server side voor de synchronisatie

Server side wordt er gebruik gemaakt van een queue voor het bewaren van de verschillende acties. Deze queue bevat alle acties die moeten verwerkt door de server voor het synchroniseren van de data. De reden dat er server side een queue wordt gebruikt is omdat die op een eenvoudige manier[9] gemakkelijk overweg kan met een groot aantal requests. Een poll-functie kan dan gemakkelijk van de data uitlezen uit de queue en doorsturen naar de correcte functie.

6.6.1 Toepassing in de business case

Server side zijn er enkele aanpassingen uitgevoerd zodat de synchronisatie automatisch kan verlopen. De API Gateway, die de mapping[10] van de requests tot zijn rekening nam, gebruikt nu een proxy lambda integratie. Deze lambda is nu verantwoordelijk voor de mapping van de request. Wanneer de client gecachte data heeft, wordt die doorgestuurd naar de aparte[11] route in het URL path. Hierdoor weet de proxy lambda dat de payload[12] acties bevat die moeten worden gesynchroniseerd. Deze lambda plaatst vervolgens alle acties op een SQS queue.

Een polling lambda vraagt op vaste intervallen berichten van de queue op aan de hand van een CRON[13] job. De opgehaalde objecten uit de queue bevatten de methode, body en optionele querystrings en die worden dan terug gestuurd als een enkele request naar de proxy lambda. De proxy lambda vervult dan zijn taak als proxy lambda en stuurt de request naar de correcte lambda om te verwerken. Wanneer de Lambda succesvol is uitgevoerd, dan wordt message van de queue verwijderd.

Er is voor SQS gekozen omdat SNS een push strategie voor zijn berichten hanteert. Indien er bij de synchronisatie een fout voorvalt, dan is het bericht en de data onherroepelijk verloren. Met SQS moeten de boodschappen expliciet worden verwijderd.

[9] met behulp van buffers of throttling

[10] Mapping van de request houdt in dat de parameters van de querystring worden gemapped naar een object en de body van een request bij een HTTP POST wordt gemapped naar een object.

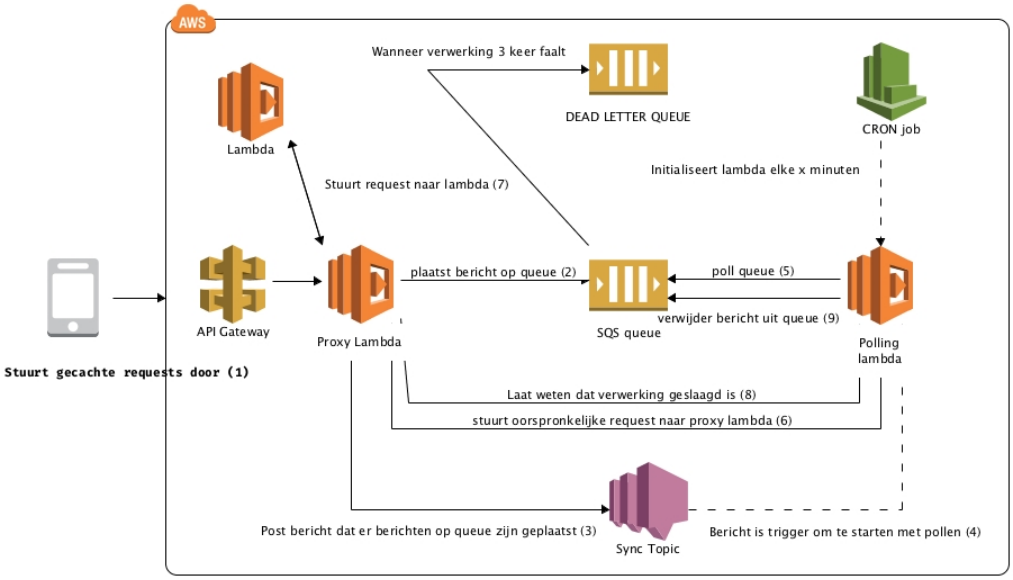
[11] organisatie/sync

[12] De payload is de data die de body bevat, de optionele querystring parameters, het pad (bv /observations) en de methode(POST, UPDATE, PUT)

[13] Bij een CRON job wordt op vaste intervallen een AWS service aangeroepen.

Wanneer de afhandeling van een bericht meerdere keren zou falen, dan komt de terecht in de 'dead letter queue'. Afhankelijk van de configuratie van die 'dead letter queue' blijft het bericht nog 30 dagen in de dead letter queue alvorens het wordt verwijderd. De berichten in de 'dead letter queue' kunnen dan nog altijd manueel worden verwerkt. AWS SNS wordt wel gebruikt wanneer de proxy lambda de lambda die de queue pollt wil laten weten als er nieuwe requests op de queue zijn geplaatst. Voor de kostenefficiëntie kan de CRON-job dan op een interval van 10 minuten worden geplaatst want een SNS boodschap kan de lambda ook triggeren.

Figuur 6.3: De server side architectuur voor de synchronisatie



6.6.2 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 ontwikkeld in kader van dit onderzoek en bieden een oplossing aan voor het synchroniseren van data bij Pridiktiv. De componenten zijn geschreven in JavaScript voor NodeJS 4.3 en hoger. Hierdoor is het mogelijk om nieuwe 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, bijvoorbeeld 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.

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
2   const sqsPush = require('aws-sqs-push');
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'
        }).then(messageId => {
10        console.log(messageId);
11      //=> '8a98f4d0-078b-5176-9af2-bbd871660ecb'
12  });
```

Listing 6.8: Voorbeeld dat data plaatst op een SQS queue met behulp van de aws-sqs-push library

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": "
            AQEB9mnsxtAZlwnDERxn3yADAP96QRe0KPbqaKXLvvchqmD4jAr",
7       //   "MD5OfBody": "098f6bcd4621d373cade4e832627b4f6",
8       //   "Body": "test"]
9
10
11  awsSqsPoll('QueueName', {AwsAccountId: '123456789012',
        numberOfMessages: 1, timeout: 20, json: false})
12      //  ["MessageId": "28f61fd2-b9ca-4cb9-879a-71ea8bce4636",
13      //   "ReceiptHandle": "
            AQEB9mnsxtAZlwnDERxn3yADAP96QRe0KPbqaKXLvvchqmD4jAr",
14      //   "MD5OfBody": "098f6bcd4621d373cade4e832627b4f6",
15      //   "Body": "test"]
```

Listing 6.9: Voorbeeld hoe berichten worden opgehaald van de SQS queue

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+
        CwLj1FjgXUv1uSj1gUPAWV66FU/').then(id => {
5     console.log(id);
6     //=> 'b5293cb5-d306-4a17-9048-b263635abe4
7   });
8
9   awsSqsDeletemessage('somequeue', 'SasuWXPJB+
        CwLj1FjgXUv1uSj1gUPAWV66FU/', {awsAccountId: '123456789012'}).
        then(id => {
10    console.log(id);
11  //=> 'b5293cb5-d306-4a17-9048-b263635abe4
12 });
```

Listing 6.10: Voorbeeld hoe een bericht wordt verwijderd van SQS nadat het is verwerkt

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[14] 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
2   const awsGetSqsUrl = require('aws-sqs-geturl');
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
        (url => {
10    console.log(url);
11  //=> https://sqs.us-west-1.amazonaws.com/123456789012/
        anotherqueue
12  });
```

Listing 6.11: Voorbeeld hoe de ARN van een SQS wordt opgehaald

[14]Een callback is een functie die als argument wordt meegegeven in een functie. De callback wordt uitgevoerd op een bepaald moment tijdens de executie van de functie.

Met behulp van deze libraries is het mogelijk om event-driven en serverless de synchronisatie in de backend uit te voeren. De grootste verdienste van de libraries is het introduceren van een gemakkelijke interface om te werken met SQS. Standaard maakt de AWS NodeJS SDK gebruik van een callback bij het uitvoeren van de verschillende functies. Met behulp van pify[15] retourneren de verschillende functies nu promises.

[15] Pify is een JavaScript library die functies die een callback-argument hebben omzet naar een functie die een Promise retourneert.

7. Conclusie

Afhankelijk van de restricties waarbinnen een applicatie moet worden gebouwd, kan men voor online/offline synchronisatie gebruik maken van een volledig oplossing zoals Microsoft Azure Applications, CouchDB en Google's Firebase. Deze opties werden maar kort toegelicht en de synthese is hierbij dat de synchronisatie automatisch verloopt door middel van database replicatie. Een allesomvattende oplossing is echter niet geschikt voor elke use case. In het voorbeeld van Pridiktiv is de wens er om een van de fully-managed database te gebruiken die Amazon Web Services aanbiedt. Het hanteren van een serverless architectuur met AWS Lambda's is een tweede belangrijke reden om zelf de synchronisatie oplossing te ontwikkelen voor de business applicaties.

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 caching en synchronisatie wordt ingebouwd is het belangrijk om ook rekening te houden met performantie en efficiëntie bij het verdelen van de verantwoordelijkheid 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 medische data verloren gaat en er zoveel mogelijk gesynchroniseerd word met de achterliggende backend. Ook naar user experience is synchronisatie een belangrijke factor. Eindgebruikers hebben enkel maar baat bij een robuste offline en online synchronisatie.

De applicatie kan bij een onderbreking in de verbinding nog verder worden gebruikt en bij het online gaan wordt alle gecachte data automatisch gesynchroniseerd met de server. 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 de data die men wenst te persisteren. 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 synchronisatie en caching, 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

Apache CouchDB. (2016). Database replication for offline sync. Web. Geraadpleegd op 13 april 2017. Verkregen van http://docs.couchdb.org/en/2.0.0/replication/

Billiet, B. (2016). Scalable Angular 2 Architecture. Web. Geraadpleegd op 3 maart 2017. 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. Verkregen van http://caniuse.com/#search=worker

Data Integration. (2015). Data Synchronisation steps. Web. Geraadpleegd op 6 maart 2017. Verkregen van http://www.dataintegration.info/data-synchronization

Europese Commissie. (2015). Study for EC: Broadband coverage in Europe 2015. Web. Geraadpleegd op 5 december 2016. Verkregen van https://ec.europa.eu/digital-single-market/en/connectivity

Europese Commissie. (2016). Integration of Digital Technology in the EU. https://ec.europa.eu/digital-single-market/en/integration-digital-technology. Geraadpleegd op 6 december 2016. Verkregen van https://ec.europa.eu/digital-single-market/en/integration-digital-technology

Gechev, M. (2016). Scalable Angular applications. Web. Geraadpleegd op 3 maart 2017. Verkregen van http://blog.mgechev.com/2016/04/10/scalable-javascript-single-page-app-angular2-application-architecture/

Go, Y. (Red.). (2015). Reliable, Consistent, and Efficient Data Sync for Mobile Apps. USENIX.

Google. (2017a). Enabling Firebase offline sync. Web. Geraadpleegd op 13 april 2017. Verkregen van https://firebase.google.com/docs/database/web/offline-capabilities

Google. (2017b). Why develop for offline? Web. Geraadpleegd op 3 maart 2017. Google. Verkregen van https://developer.chrome.com/apps/offline_apps

Leemans, B. (2013). Overview of the IndexedDB API. Web. Geraadpleegd op 14 maart 2017. Mozilla. Verkregen van https://developer.mozilla.org/nl/docs/IndexedDB

Meents, D. (2016). Managing state with Redux in React. Web. Geraadpleegd op 2 mei 2017. Verkregen van https://www.davidmeents.com/blog/manage-state-connect-to-api-redux-axios/

Microsoft Azure Documentation. (2017). Enable Offline sync for your Cordova app. Web. Geraadpleegd op 14 april 2017. Verkregen van https://docs.microsoft.com/en-us/azure/app-service-mobile/app-service-mobile-cordova-get-started-offline-data

Mozilla. (2017a). Explanation on Promises. Web. Geraadpleegd op 23 april 2017. Mozilla. 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). Structured Cloning Algoritme. Web. Geraadpleegd op 25 mei 2017. Verkregen van https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm

Mozilla. (2017d). Using Web Workers. Web. Geraadpleegd op 23 april 2017. Verkregen van https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

Mozilla. (2017e). Web API Overview. Web. Geraadpleegd op 14 maart 2017. Mozilla. Verkregen van https://developer.mozilla.org/nl/docs/WebAPI

ngrx. (2017). Documentation for ngrx. Web. Geraadpleegd op 3 maart 2017. open source collective. Verkregen van https://github.com/ngrx/store

Offline First. (2017). Offline first mission statement. Web. Geraadpleegd op 24 november 2016. Offline First Open Source. Verkregen van http://offlinefirst.org/

OutSystems. (2016). Offline Synchronisation Patterns. Web. Geraadpleegd op 3 maart 2017. Verkregen van https://success.outsystems.com/Documentation/10/Developing_an_Application/Use_Data/Offline/Offline_Data_Synchronization_Patterns

ReactiveX. (2017). Documentation on how to use RX. Web. Geraadpleegd op 17 maart 2017. ReactiveX. Verkregen van http://reactivex.io/intro.html

Richardson, C. (2015). Overview of microservices. Web. Geraadpleegd op 17 maart 2017. Verkregen van http://microservices.io/patterns/microservices.html

Stack Overflow. (2015). How to detect online and offline events cross-browser. Web. Geraadpleegd op 20 april 2017. Verkregen van https://stackoverflow.com/questions/3181080/how-to-detect-online-offline-event-cross-browser

W3C. (2017). Overview of W3C Specification. Web. Geraadpleegd op 3 maart 2017. W3C. Verkregen van https://www.w3.org/TR/webdatabase/

WHATWG. (2017). Standard for storage by WHATWG. Web. Geraadpleegd op 14 maart 2017. WHATWG. Verkregen van https://html.spec.whatwg.org/multipage/webstorage.html

Lijst van figuren

Lijst van tabellen