



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 beheert op verschillende virtuele machines. In het onderzoek is synchronisatie specifiek onderzocht voor de fully-managed database DynamoDB die gebruikt maakt van een serverless microservices architectuur en een Angular applicatie als client applicatie.

Na onderzoek 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. In het prototype van het onderzoek komt er een voorbeeld aan bod om aan te tonen hoe conflict resolution kan worden opgelost met behulp van timestamps. 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
2.2	Terminologie	14
2.2.1	Huidige Web APIs voor lokale opslag	14
2.2.2	Scalable Angular Architecture	16
2.2.3	Ngrx store: een state container	17
2.2.4	Reactive programming paradigm	17
2.2.5	Offline First	18
2.2.6	Amazon Web Services	18
2.3	Stand van zaken	18
2.4	Probleemstelling en Onderzoeksvragen	19

2.5	Opzet van deze bachelorproef	20
3	Methodologie	21
3.1	Research	21
3.1.1	Beperkingen door business case	22
3.2	Protoyping	22
3.3	Onderzoeken synchronisatiemethodes	22
3.4	Conclusie	22
4	Opstelling	23
4.1	De Pridiktiv applicatie	23
4.2	Applicaties voor het onderzoek	23
4.3	Client side	24
4.4	Server side	24
5	Synchronisatie patterns	27
5.1	Read-Only Data	27
5.2	Read-Only Data Optimized	28
5.3	Read/Write Data Last Write Wins	28
5.4	Read/Write with Conflict Detection	28
6	Onderzoek	31
6.1	Online/Offline status registratie	31
6.1.1	Gecachte data doorsturen naar de server	32

6.2	Data caching uit backend	33
6.2.1	Beperkingen door single-threaded omgeving	33
6.2.2	Toepassing	33
6.3	Data synchronisatie: planning bij development	34
6.4	Read-Only Optimised	35
6.4.1	Overzicht: Client	35
6.4.2	Overzicht: Server	36
6.4.3	Opmerkingen	36
6.4.4	Toepassing	36
6.5	Last/First Write Wins	37
6.5.1	Overzicht: Client	37
6.5.2	Overzicht: Server	37
6.5.3	Opmerkingen	37
6.5.4	Toepassing	38
6.6	Conflict Resolution	38
6.6.1	Overzicht: Client	38
6.6.2	Overzicht: Server	38
6.6.3	Toepassing	39
6.7	Open-source libraries	39
6.7.1	aws-sqs-push	39
6.7.2	aws-sqs-poll	41
6.7.3	aws-sqs-deletemessage	41
6.7.4	aws-sqs-geturl	42
7	Conclusie	43

Bibliografie	46
---------------------------	-----------

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, afhankelijk van de acties van de gebruiker. 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 normaal dynamisch opgevraagd van de web server.

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. NgRx store vervult die rol in een Angular applicatie.

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. Dankzij 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 en in het prototype.

localForage : localForage is een library van Mozilla waarbij het mogelijk is om data lokaal te cachen. De library maakt gebruik van asynchrone storage mogelijkheden als IndexedDB en WebSQL met een localStorage-achtige syntax. De localStorage DOM Storage wordt wel gebruikt indien de webbrowser geen ondersteuning biedt voor IndexedDB of WebSQL. In dit geval is de opslagruimte om te cachen wel beperkter.

Progressive enhancement : Progressive enhancement is een ontwikkel strategie bij web development waarbij de focus wordt gelegd op de belangrijkste business require-

ments die de web applicatie of website moet invullen. Afhankelijk van de end-user's browser en connectie, kunnen er 'lagen' van functionaliteit (features) worden toegevoegd aan de applicatie of website. Op die manier kan er altijd een minimum worden aangeboden aan de end-user.

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.

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) wordt op regelmatige intervallen de status van applicatie, databank of datastructuur opgevraagd. Bij polling vraagt de ontvanger voor nieuwe data of updates.

Pushing : Bij push technologie wordt de request voor een transactie geïnitieerd door de server.

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 die kan worden aangeroepen door andere microservices. Wanneer een microservice dan data nodig heeft die het zelf niet beschikt, dan kan wordt er een andere microservice aangeroepen met een andere verantwoordelijkheid.

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 virtual machines.

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 en de toepassing tonen met behulp van een prototype. In de sectie 'Business Case' wordt de business case van Pridiktiv.care - into.care toegelicht. Daarna worden in 'Terminologie' de verschillende relevante termen overlopen. In '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

De business case van Pridiktiv vormt het vertrekpunt van deze studie en wordt nu nog wat verder toegelicht. De applicatie draait op een smartphone in een woonzorgcentrum. Het woonzorgcentrum beschikt over draadloos internet maar het gebouw is niet volledig gedekt. 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. Het huidige algoritme doet een HTTP call 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. De huidige applicatie is een Angular (het vroegere Angular 2) met NgRx als state container als een Cordova applicatie.

Voor lokale opslag wordt momenteel gebruik gemaakt van Mozilla's localForage library. De backend maakt gebruik van Amazon Web Services met DynamoDB voor de persistentie van de data. Het is de intentie van dit onderzoek om te werken binnen het kader en binnen de restricties van de business case.

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, 2017b) 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 en 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 (Chrome, Firefox, Safari en Opera, Internet Explorer en Edge) 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.

localStorage en sessionStorage

Deze 2 APIs vallen onder Web Storage of DOM Storage (Camden, 2016). Wanneer de browser ondersteuning biedt voor Web Storage, zijn beide beschikbaar als een globaal object. 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.

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, 2017b) 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

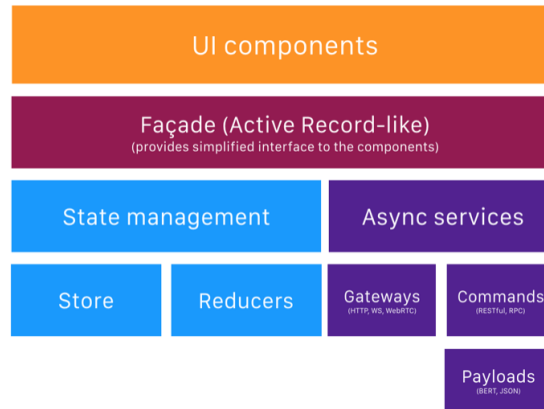
IndexedDB (Leemans, 2013) is een Web API die gebruikt wordt het opslaan van relatief grote data structuren in browsers. Dankzij indexering is het mogelijk om sneller en performanter (Camden, 2016) te zoeken in de databank in vergelijking met localStorage en sessionStorage. Net zoals SQL-databanken is 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

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.

2.2.2 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 word gegroepeerd. Componenten zelf kunnen 'dumb' zijn als ze enkel data voorstellen of 'smart' wanneer ze data opvragen of verwerken.

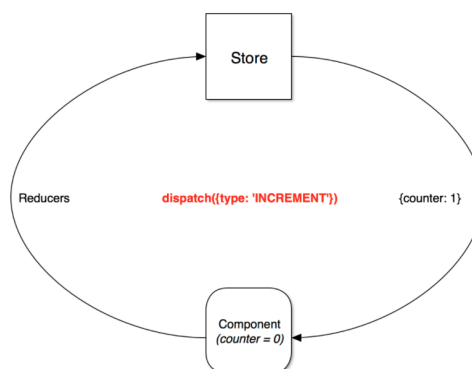
De dataflow tussen componenten is ook belangrijk om een beheersbare (Billiet, 2016) applicatie te bouwen. Zo communiceren child components enkel maar met hun parent en interageert een parent met een model dat op zijn beurt communiceert met een 'store' (.cfr 'ngrx store: een state container') aan de hand 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.

2.2.3 NgRx store: een state container

NgRx store is een container die de state van de applicatie bijhoudt waarbij performantie en consistentie centraal staan. De NgRx 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 NgRx store werkt met volgende (ngrx, 2017) principes :

- de state is een single immutable JSON data structure
- dispatch initialiseren acties en dus veranderingen in de state
- reducers verwerken actions om een 'nieuwe' state te creëren. Een reducer is een functie die een action moet verwerken
- de state van de store wordt geraadpleegd als een Observable van de state die op zijn beurt een observer is van de actions

Figuur 2.2: Voorbeeld van de flow bij een event/action in ngrx



2.2.4 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-based applicaties te ontwikkelen, die in real-time data kunnen manipuleren en tonen. Het breidt het Observer design pattern 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 standaard JavaScript? Een Promise kan maar een single event afhandelen en kan niet worden gecancelled. Een Observable kan worden

beschouwd als een stream van objecten waarop verschillende operatoren zoals map, reduce en filter kunnen worden op toegepast.

2.2.5 Offline First

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.

2.2.6 Amazon Web Services

Om het onderzoek zo dicht mogelijk te laten aansluiten bij de use case van Pridiktiv, wordt gebruikt gemaakt van bepaalde onderdelen (Services, 2017) binnen AWS. Volgende componenten komen aan bod in de bachelor paper:

- Amazon SQS: Simple Queue Service: SQS is een fully managed message queue service die toelaat om te communiceren tussen verschillende AWS services. SQS voorziet 2 varianten. De standard queue biedt de snelste en performantste 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 pollen de queue voor nieuwe data.
- Lambda: Met AWS Lambda is het mogelijk om 'functies' uit te voeren in de cloud dus zonder een specifieke server structuur op te bouwen en vormt dus 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 architecture. De output kan dan bijvoorbeeld worden gepersisteerd door DynamoDB of teruggestuurd worden naar de client. De verschillende Lambda functies vormen de microservices architecture in de backend van de businesscase.
- Amazon SNS, Amazon Simple Notification Service: Een push notificatie service die toelaat om berichten te sturen naar mobile clients, email adressen of andere services. Met SNS is het onder andere mogelijk om polling van andere services te vermijden.
- DynamoDB: DynamoDB is een 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 API's te beheren.

2.3 Stand van zaken

Wanneer een ontwikkelaar de end user van zijn web -of mobiele applicatie wil voorstellen, dan denkt hij vaak aan een user 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. Meeste mobiele- en webapplicaties hebben twee momenten waarbij er problemen kunnen optreden door de status van de connectie:

1. client push request naar de server
2. client pull van server / server push naar client

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

- De gebruiker al dan niet op de hoogte brengen van de veranderingen in de status van de connectie. Bijvoorbeeld: Bij het versturen van bericht, de gebruiker op de hoogte brengen 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 en wanneer de applicatie terug online, deze data synchroniseren met de remote database.
- 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 en dat deze verschillen met de data van de applicatie. 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. CloudBoost, CouchBase en Firebase bieden een all-in-one oplossing. Het nadeel van die oplossing is dat het niet compatibel is met de bestaande backend. Deze opties komen dan ook niet verder aan bod in dit onderzoek.

2.4 Probleemstelling en Onderzoeksvragen

Het doel van dit onderzoek is om bij een bestaande Angular applicatie data synchronisatie zo vlot mogelijk te laten verlopen, zonder onderbreking bij de gebruiker terwijl de data consistent blijft. 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? Op basis van verschillende tests op het prototype, 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. Het is cruciaal dat deze functionaliteit wordt verwerkt aangezien de dekking van draadloos internet in verschillende woonzorgcentra niet volledig is.

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 gebruikte testapplicatie, backend infrastructuur en applicatie van de business case voor dit onderzoek worden toegelicht in hoofdstuk 4. Het onderzoek start in hoofdstuk 5 met een opsomming van verschillende synchronisatieoplossingen. 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

3.1 Research

De eerste stap in het onderzoek was het in kaart brengen van bestaande synchronisatie technologieën en technieken voor data synchronisatie. Synchronisatie technologieën zoals Firebase, CouchBase en Azure Mobile Apps laten toe om na wat configuratie een applicatie 'offline first' te maken waarbij conflict resolution nagenoeg automatisch verloopt. CouchDB en PouchDB zijn twee andere technologieën waarbij er geen 'all-in-one' oplossing wordt aangeboden. Ze werken enkel in tandem voor een oplossing te bieden naar synchronisatie. Wanneer een synchronisatie framework wordt gebruikt, dan gebruikt de client applicatie een specifieke library die de data lokaal beheert indien de gebruiker offline gaat of de applicatie de data wil cachen. Wanneer de gebruiker terug online gaat of de applicatie de gecachte data wil synchroniseren, dan gebeurt de synchronisatie en conflict resolution volledig automatisch. Voorwaarden om dit te kunnen realiseren is de implementatie van een specifieke library om de lokale data te beheren en het gebruik van de correcte backend frameworks of databases. Door de beperkingen van de business case (.cfr "Beperkingen door business case") was er ook nood aan onderzoek naar synchronisatie patterns. Deze patterns zijn vaak geïntegreerd in de vermelde synchronisatie technologieën.

3.1.1 Beperkingen door business case

Er zijn heel wat oplossingen voor synchronisatie van offline data out-of-the-box maar die hebben bepaalde voorwaarden die niet voldoen aan de constraints van de business case. De use case van Pridiktiv laat immers geen volledige refactor van de backend toe. Men heeft gekozen voor DynamoDB voor de persistentie van de data en helaas biedt DynamoDB geen automatische synchronisatie toe. Door deze beperking ligt de focus van dit onderzoek eerder om synchronisatie methodes te bespreken.

3.2 Prototyping

Voor het onderzoek is er een eenvoudige Angular applicatie ontwikkeld die bepaalde use cases van de business case nabootst. Het grootste deel van de synchronisatie vindt echter plaats in de backend. Daarom dient de client applicatie voornamelijk voor het tonen van gesynchroniseerde data en om caching en pre-caching methodes te onderzoeken. 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 'Onderzoek' wordt voornamelijk in detail ingegaan op de interactie van de client en server.

3.3 Onderzoeken synchronisatiemethodes

Het grootste deel van het onderzoek van de synchronisatiemethodes vond plaats tijdens de stage periode bij Pridiktiv. Met behulp van de literatuurstudie en research in de documentatie van verschillende technologieën en APIs, is het mogelijk geweest om een oplossing aan te bieden bij de synchronisatie van de data die offline werd gegenereerd.

3.4 Conclusie

In het laatste onderdeel van het onderzoek wordt er op basis van de resultaten van het onderzoek een synthese gevormd.

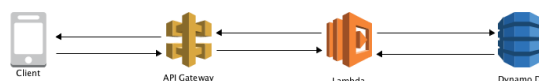
4. Opstelling

In dit hoofdstuk stellen we de technologie stack voor die gebruikt wordt voor dit onderzoek in zowel de testapplicatie als de echte business applicatie. Zo is het voor de lezer ook duidelijk welke architectuur dat er gehanteerd wordt in beide applicaties.

4.1 De Pridiktiv applicatie

Omdat de applicatie het belangrijkste deel vormt van de business waarde van Pridiktiv, kan er niet gedetailleerd worden ingegaan over alle functionaliteit die de applicatie aanbiedt en de architectuur hoe de applicatie is opgebouwd. Om de lezer toch vertrouwd te maken met de architectuur, is er onderstaande afbeelding die een abstract overzicht aanbiedt van de flow binnen de applicaties. In het onderdeel 'Onderzoek' zal wel dieper worden ingegaan bij de architectuur van de synchronisatie oplossing.

Figuur 4.1: Flow van de Pridiktiv applicatie



4.2 Applicaties voor het onderzoek

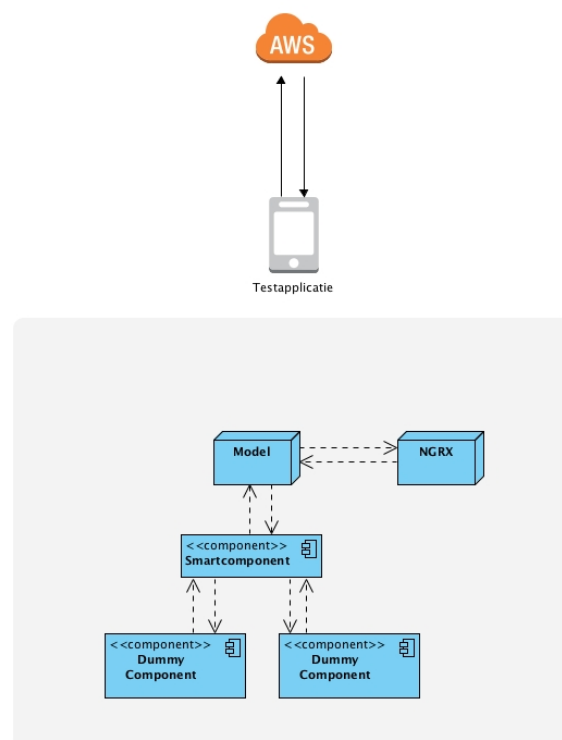
In het onderzoek worden verschillende kleine prototypes gebruikt voor het testen van de verschillende oplossingen. Voor de client wordt steeds dezelfde applicatie gebruikt

mits enkele kleine aanpassingen. Serverside wordt er gebruik gemaakt van verschillende Amazon Web Services.

4.3 Client side

De client applicatie bestaat uit een Angular applicatie, gegenereerd door Angular CLI, die verschillende synchronisatieproblemen moet voorstellen. De client vervult de rol als view voor de data die serverside wordt verwerkt. Het is niet de intentie van dit onderzoek om een volledig werkende applicatie te bouwen. Het doel van de client applicatie is om enerzijds de caching voor te stellen en anderzijds om het belang van het SSOT-principe en de store aan te tonen. Alhoewel de applicatie enkel dient als voorbeeld voor de synchronisatie, wordt er gebruik gemaakt van de design principes (Billiet, 2016)(Gechev, 2016). Hierbij wordt er gebruik gemaakt van een model dat verantwoordelijk is voor het uitvoeren van de requests naar de server, het opslaan en ophalen van de data naar en uit de ngrx store. Ook caching komt kort aan bod met localForage en webworkers.

Figuur 4.2: Flow van een client applicatie die gebruik maakt van scalable angular design patterns en AWS

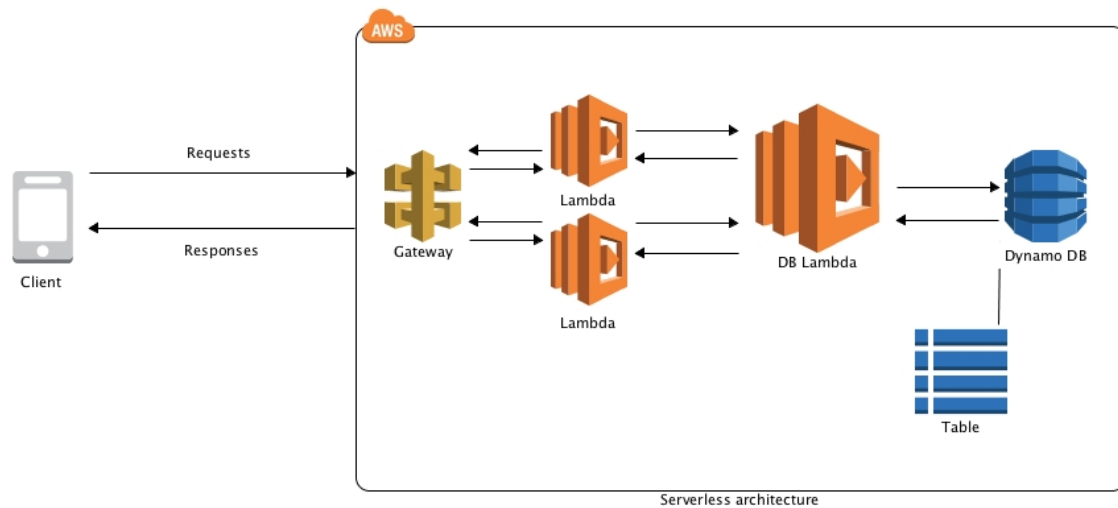


4.4 Server side

Server side wordt er gebruikt gemaakt van AWS Lambda voor het verwerken van de data, SQS voor de synchronisatie en DynamoDB voor de data persistentie. Het grootste deel

van de synchronisatie vindt plaats server side dus vormt dit het omvangrijkste gedeelte van het onderzoek. Volgende afbeelding is een voorbeeld van een serverless (Richardson, 2015) architectuur met de services van Amazon Web Services. Een belangrijke opmerking is dat de testomgeving geen gebruik maakt van de API-Gateway terwijl de API-Gateway service vaak (Richardson, 2015) wel wordt gebruikt bij serverless architectuur.

Figuur 4.3: Voorbeeld van een serverless architectuur opgezet met verschillende Amazon Web Services



5. Synchronisatie patterns

In dit hoofdstuk worden enkele synchronisatie methodes en technieken besproken die data kan synchroniseren met de databank. Indien van toepassing, wordt telkens een concrete situatie van de use case van Pridiktiv gebruikt ter illustratie. Op het einde van dit hoofdstuk worden ook nog andere technieken besproken die niet meteen kunnen worden onderverdeeld onder een specifiek pattern.

5.1 Read-Only Data

Dit data synchronisatie patroon wordt gebruikt wanneer de end user enkel maar data moet kunnen opvragen terwijl de applicatie offline is en die data niet hoeft te manipuleren of de manipulaties op die data niet belangrijk genoeg zijn om te persisteren. Bij Read-Only Data is de richting van het dataverkeer unidirectioneel, van server naar end-user applicatie. Het Read-Only pattern hanteert volgende logica:

1. de end-user (client) vraagt de data op van de server. De server is de SSOT en houdt alle data bij. Die data kan wijzigen wanneer de end-user bijvoorbeeld offline is
2. server retourneert de data. De opgevraagde data wordt lokaal opgeslagen
3. alle manipulaties op de data worden geblokkeerd en worden nooit doorgegeven aan de server. De client kan dus enkel maar GET HTTP requests sturen voor de data op te vragen
4. bij synchronisatie wordt de oude data lokaal verwijderd en vervangen door de nieuwe data
5. de applicatie controleert op regelmatige tijdstippen de data

Een voorbeeld van het Read-Only pattern van in de use case van Pridiktiv is het opvragen

van de patiëntenlijst. Die lijst kan worden gewijzigd door de hoofdverpleger in het dashboard maar niet in de applicatie zelf. Wanneer de applicatie offline is, kan eenvoudig worden verder gewerkt met de applicatie. Indien er een nieuwe patiënt wordt toegevoegd, dan is die vanaf de volgende synchronisatie zichtbaar.

5.2 Read-Only Data Optimized

Het Read-Only Data Optimized pattern is identiek aan het Read-Only Data pattern maar met 1 verschil. Bij de synchronisatie worden enkel de data opgevraagd die gewijzigd zijn en niet alle data. 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 bestaande data en de data in de databank, kan de applicatie al dan niet beslissen om de lokale data te updaten. De richting van het dataverkeer is bidirectioneel, wat ook verschilt met de standaard implementatie van het Read-Only pattern. Omdat de server eerst moet worden gevraagd of er al dan niet data zijn gewijzigd, moet de applicatie ook kunnen communiceren met de server.

5.3 Read/Write Data Last Write Wins

In dit pattern gaat de server er van uit dat writes altijd in de juiste volgorde worden uitgevoerd en de laatste write die naar de databank wordt gestuurd ook de werkelijke laatste wijziging is van de client applicatie. Er wordt geen conflict resolution uitgevoerd. Het pattern blinkt uit wanneer er enkel wordt toegevoegd en de data nooit wordt gemanipuleerd. In de use case van Pridiktiv kan dit worden toegepast bij bijvoorbeeld het toevoegen van notities bij een patient. De notities van een patient worden niet gewijzigd waardoor er geen data kunnen worden overschreven.

5.4 Read/Write with Conflict Detection

Het Read/Write with Conflict Detection pattern is het meest complexe waarbij verschillende end-users dezelfde data wijzigen op het moment dat de applicatie offline is. Bij dit pattern wordt er gesproken van multi-way synchronisatie waarbij een applicatie zowel de data van de server kan updaten en dat de server op zijn beurt alle andere apparaten moet updaten. Een mogelijke flow van het proces zou er als volgt kunnen uitzien:

1. de server database houdt alle 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 conflicting changes worden geregistreerd voor verdere behandeling

5. Vraagt de gebruiker voor conflict resolution of de applicatie kan zelf het conflict oplossen en de data aanpassen

Een voorbeeld uit de use case van Pridiktiv die gebruik zou kunnen maken van het Read/Write with Conflict Resolution pattern is de opvolging bij wondzorg. Wanneer een end-user een bepaalde wijziging aanbrengt in het dossier van de patient, is het belangrijk dat die data niet verloren gaan indien een andere end-user ook het wondzorg dossier van een patient aanpast. Het is belangrijk om conflict resolution 'achter de schermen' op te lossen om op die manier ervoor te zorgen dat er geen aanpassingen verloren gaan.

6. Onderzoek

In dit hoofdstuk wordt dieper ingegaan op de verschillende methodes en de real-world toepassing van die methodes met de testopstelling en de businesscase. De invalshoek voor dit onderzoek is steeds een statusverandering van offline naar online. Volgende methodes komen aan bod:

- Read-only Optimised
- Last/First Write Wins
- Conflict resolution

6.1 Online/Offline status registratie

Voor bovenstaande synchronisatie methodes worden besproken is het belangrijk om aan te geven welke methode er wordt gebruikt zodat de applicatie kan waarnemen dat die al dan niet online is. Er zijn 2 mogelijkheden:

1. DOM API's aanspreken vanuit JavaScript: helaas is er geen consistente crossbrowser support op het controleren van de status van de verbinding zonder gebruik te maken van requests naar externe data. Bijvoorbeeld 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.js.

```

1 // FIREFOX 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
11 // met native window.navigator
12 if (navigator.onLine) {
13     // Online
14 } else {
15     // Offline
16 }

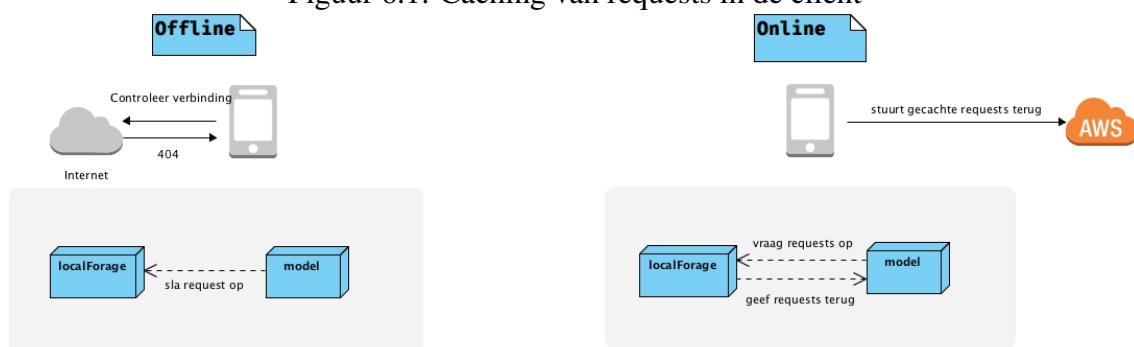
```

Listing 6.1: DOM API's voor controleren offline en online status

6.1.1 Gecachte data doorsturen naar de server

Voor het onderzoek werd geopteerd op gebruik te maken van de tweede methode. Wanneer de applicatie registreert dat er geen verbinding meer is, worden alle requests gecached als afzonderlijke objecten. Wanneer de applicatie dan terug online is, wordt er in een request naar de server alle gecachte data doorgestuurd. De API Gateway geeft het object door aan een gateway 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. Bij requests die een UPDATE uitvoeren op een object dat aangemaakt werd (CREATE) tijdens de offline status, worden allemaal gebundeld in een 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.

Figuur 6.1: Caching van requests in de client



6.2 Data caching uit backend

Bij een applicatie die zowel online als offline moet werken, is het belangrijk dat er data wordt gepreload in de applicatie. Zo kan de gebruiker blijven verder werken indien de applicatie plots offline zou zijn. 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. Een mogelijke oplossing voor dat probleem kan een webworker zijn. Door het single-threaded karakter van JavaScript kan er een webworker worden gebruikt voor het parallel bevragen van de server.

De connectie status van de de applicatie wordt bewaard als state in de ngrx store. Die kan worden bevraagd bij het uitvoeren wanneer er data uit de store moet worden geladen. Zo weet de applicatie of het al dan niet mogelijk is om een request naar de server uit te voeren. De ngrx store is de SSOT en bevat die de gecachte data in-memory. Wanneer de store wordt bevraagd kan de gecachte data uit de store worden geladen. Dit laat de gebruikers toe om zonder onderbreking de applicatie te gebruiken wanneer er verandering in de verbindingstatus is.

6.2.1 Beperkingen door single-threaded omgeving

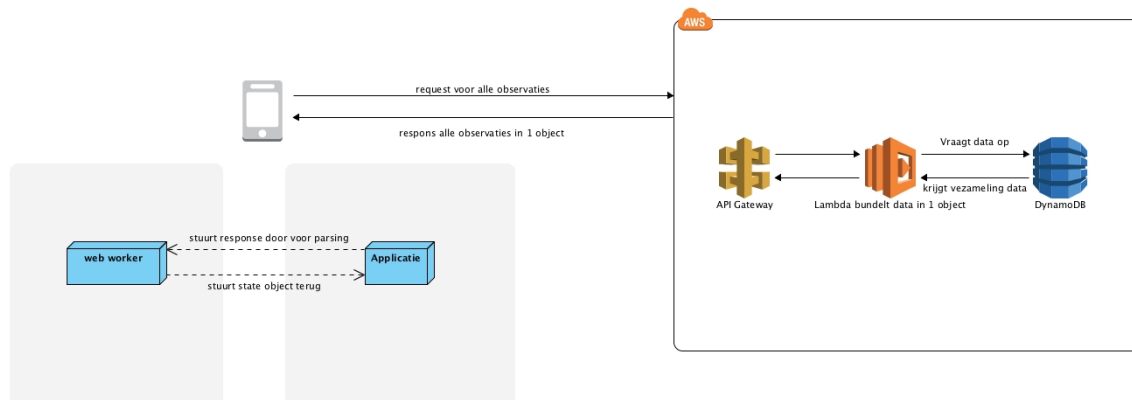
JavaScript is een single-threaded omgeving waardoor meerdere scripts niet simultaan kunnen worden uitgevoerd. Dit vormt problemen wanneer een webapplicatie UI events, queries, een groot volume API data en tegelijk de DOM aanpassingen moet verwerken. Met technieken zoals `setTimeout()` en `setInterval()` 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.

6.2.2 Toepassing

In het onderzoek zijn drie verschillende mogelijkheden voor caching onderzocht. De eerste methode vraagt serieel alle data op van de server. Dit vormt geen probleem indien de data beperkt is, maar eenmaal het volume van de data stijgt, dan moeten er steeds meer requests wordt uitgevoerd naar de server. Dit heeft als nadeel dat de main thread geblocked wordt waardoor er geen UI kan worden gerenderd op het toestel. Dit is dus geen ideaal scenario. Bij de andere 2 methodes wordt er gebruik gemaakt van een webworker. Deze webworker laat doe om parallel met de applicatie bepaalde acties uit te voeren. Met behulp van de webworker is het dan mogelijk om de caching in de achtergrond uit te voeren. Met deze web workers kunnen dan volgende operaties worden uitgevoerd:

1. data ophalen van de server
2. omzetten van opgehaalde data naar state object voor ngrx

Figuur 6.2: Web worker die request verwerkt



Om het aantal requests drastisch te verminderen is het ook mogelijk om bepaalde data te bundelen. Zo kunnen alle observaties voor patiënten worden gebundeld worden in 1 object en de webworker vormt die response dan om naar een geldig state object. Zo worden intensieve berekeningen in de applicatie vermeden. Een derde mogelijkheid is het opzetten van een applicatie state in de backend en dit dan door te sturen. 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.

Toepassing op de business case

In de business case wordt de request serieel bevraagd 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 cachten van BLOBs valt buiten de scope van dit onderzoek.

6.3 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

6.4 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.

6.4.1 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.

6.4.2 Overzicht: Server

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

6.4.3 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.

6.4.4 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.

6.5 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.

6.5.1 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.

6.5.2 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.

6.5.3 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 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.

6.5.4 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.

6.6 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.

6.6.1 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.

6.6.2 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 uses 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.

6.6.3 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 te voeren, enkel UPDATE operaties.

6.7 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.

6.7.1 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
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',
10      }).then(messageId => {
11      console.log(messageId);
12      //=> '8a98f4d0-078b-5176-9af2-bbd871660ecb'
13  });
```

Listing 6.2: Voorbeeld dat data plaatst op een SQS queue met behulp van de `aws-sqs-push` library

6.7.2 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 6.3: Voorbeeld hoe berichten worden opgehaald van de SQS queue

6.7.3 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 6.4: Voorbeeld hoe een bericht wordt verwijderd van SQS nadat het is verwerkt

6.7.4 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 6.5: 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.
- 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). 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>
- Services, A. W. (2017). Overview of the different AWS services. Verkregen van <https://aws.amazon.com/documentation/>
- 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	16
2.2	Voorbeeld van de flow bij een event/action in ngrx	17
4.1	Flow van de Pridiktiv applicatie	23
4.2	Flow van een client applicatie die gebruik maakt van scalable angular design patterns en AWS	24
4.3	Voorbeeld van een serverless architectuur opgezet met verschillende Amazon Web Services	25
6.1	Caching van requests in de client	32
6.2	Web worker die request verwerkt	34

Lijst van tabellen