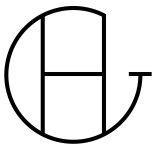

HoGent

Faculteit Bedrijf en Organisatie

Redux uitbreiding gebaseerd op Redux-store

Thibault Gobert

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

Promotor:
Harm De Weirdt

Instelling: —

Academiejaar: 2017-2018

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Redux uitbreiding gebaseerd op Redux-store

Thibault Gobert

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

Promotor:
Harm De Weirdt

Instelling: —

Academiejaar: 2017-2018

Tweede examenperiode

Woord vooraf

Samenvatting

In dit onderzoek worden twee React-Redux projecten opgesteld. React maakt gebruik van JavaScript voor het bouwen van user interfaces, waar Redux een concept is voor data storage en communicatie binnen de applicatie. Redux functioneert als een state container voor JavaScript apps. Een store is een concept van Redux die de hele state tree van de applicatie vasthoudt.

Het probleem situeert zich het samenvoegen van twee React-Redux projecten. Deze hebben beide een store die de overeenkomstige state tree bevat. Er kan dus niet rechtstreeks een component geïmporteerd worden van het ene project naar het andere, omdat het daar geen bekende state zal hebben en er zo geen acties op kunnen uitgevoerd worden.

De bekende oplossing voor een soortgelijk probleem is het kopiëren van de reducers van het tweede project en deze plaatsen in het eerste project. [TODO-REFERENTIE HIER] Deze oplossing heeft echter een groot nadeel: als de reducers van het open source project gewijzigd worden of als er reducers worden toegevoegd/verwijderd, dan zullen deze ook opnieuw gekopieerd moeten worden in het eerste project.

Om het probleem te reproduceren worden dus twee React-Redux projecten opgesteld. Het eerste project is het hoofdproject met een basis inlog functionaliteit. Het tweede project is een fork van het bestaande open source project van Scratch. Om beide projecten met elkaar te laten communiceren zal het tweede project gepubliceerd worden op npm onder een eigen registry. Dit zorgt ervoor dat het project geïnstalleerd kan worden als dependency in het eerste project. Op deze manier kan een component snel geïmporteerd worden.

De bedoeling van dit onderzoek is om een manier te vinden waarbij geen extra aanpassingen moeten gebeuren aan het eerste project, maar enkel een geupdatete versie van het tweede project moet gepublished worden op npm. Deze manier zal verkregen worden door een

extensie te schrijven op Redux die de mogelijheid biedt om de functionaliteit van het tweede project te gebruiken in het eerste project.

Inhoudsopgave

0.1 Context 13

0.2 Probleemstelling 14

0.3 Onderzoeksvraag 15

0.4 Onderzoeksdoelstelling 15

0.5 Opzet van deze bachelorproef 15

1 Stand van zaken 17

1.1 React 17

1.1.1 JSX 17

1.1.2 Components 18

1.1.3 React lifecycle 18

1.2 Redux 20

1.2.1 Drie principes 20

1.2.2 Redux flow 21

1.2.3 Npm 23

2 Methodologie 25

2.1 Analyse 25

2.2 Methodes 25

2.2.1 combineReducers 25

2.2.2 object export 26

2.2.3 Eigen methode 27

3 Conclusie 29

A Onderzoeksvoorstel 31

A.1 Introductie 31

A.2 State-of-the-art 32

A.3 Methodologie 32

A.4 Verwachte resultaten 32

A.5 Verwachte conclusies 33

Bibliografie 35

Lijst van figuren

Lijst van tabellen

Inleiding

0.1 Context

Zoals eerder vermeld, wordt gebruik gemaakt van twee React-Redux projecten. React is een JavaScript library voor het bouwen van user interfaces. React werkt met components. In de component schrijf je de gewenste code die moet gerenderd worden. Een component ontvangt parameters, genaamd *props* en retourneert hiërarchische views die getoond worden door de render methode. Elke component kan onafhankelijk functioneren, wat het dus mogelijk maakt om verschillende components in een andere component in te laden. (React, 2018)

Redux is een state container voor JavaScript apps. Redux kan dus gebruikt worden samen met React of andere view libraries. In Redux zijn er een aantal basis begrippen die moeten uitgelegd worden voor de verdere voortgang van dit onderzoek, zoals actions, reducers, store en containers. Deze begrippen vormen eigenlijk de omkadering van Redux. (Redux, 2018)

De Redux architectuur heeft een unidirectionele data , dit wil zeggen de data steeds dezelfde richting/flow aanneemt. Dit zorgt er voor dat de logica van de applicatie voorspelbaar wordt.

Een container component is een component die verantwoordelijk is voor het verkrijgen van data. Een container component gaat subscriben op de store, om zo een deel van de Redux state tree te kunnen lezen. Het gaat eigenlijk de nodige delen van de data nemen en doorgeven als *props*. Een container component is ook verantwoordelijk voor het dispatchen van actions die veranderingen maken aan de state van de applicatie.

Deze actions zijn payloads van informatie die data verzenden van de applicatie naar de store. Actions zijn tevens de enige soort van informatie voor de store. Het zijn JavaScript objecten die een *type* property moeten hebben om aan de duiden welke actie uitgevoerd wordt. Deze types zijn string constanten die in een aparte module worden opgeslaan.

Als er dan gekeken wordt naar de data flow dan worden er eerst actions gedispatched naar de store. De store zal dan de corresponderende reducer aanroepen met twee argumenten, namelijk de huidige state en de action. Hierna zal de root reducer de output van meerdere reducers combineren in een single state tree. Daarna gaat de Redux store de volledige state tree die geretourneerd werd door de root reducer gaan opslaan. Hier kan de container dan data uit lezen en doorgeven aan een presentationele component.

Reducers specificeren hoe de applicatie zijn state verandert ten gevolge van de actions die verzonden zijn naar de store. Actions beschrijven alleen maar het feit dat er iets gebeurd is, ze beschrijven niet hoe de applicatie zijn state verandert. Vooraleer er kan gezegd worden hoe een reducer dit doet, moet worden uitgelegd wat een pure functie is. Een pure functie is een functie die met dezelfde input altijd dezelfde output produceert. (Elliott, 2016)

Een reducer is een pure functie die de vorige state en een actie neemt en daaruit de nieuwe state retourneert. Het is belangrijk dat de reducer puur blijft, een aantal zaken zijn een no-go zoals API-calls en niet-pure functies aanroepen. De uitkomst moet voorspelbaar blijven. Redux roept de reducer aan met een *undefined* state voor de eerste keer. Daar moet de initial state van de applicatie worden ingesteld. (Redux, 2018)

Actions representeren het feit dat er iets gebeurd is en reducers updaten de state aan de hand van deze actions. De store is een object die deze zaken samenbrengt. Deze store heeft een aantal verantwoordelijkheden:

- vasthouden van de state van de applicatie
- toegang geven tot de state van de applicatie
- toelaten om de state te updaten
- registreren van listeners

Het laatste punt laat toe om een callback te registreren die de redux store zal aanroepen elke keer een actie wordt gedispatched. Op deze manier kan de UI van de applicatie upgedate worden naargelang de state van de applicatie.

0.2 Probleemstelling

Het probleem ligt in het gebruiken van components uit een tweede React-Redux project. Door het importeren van een container component in het eerste project worden de actions die gedispatched worden door die component niet herkent in de store. Deze zitten namelijk in de store van het tweede project. Door de reducers te combineren van beide projecten in een grote root reducer wordt dit probleem opgelost. Dit is zo omdat de store dan de corresponderende reducer kan aanroepen naargelang de action die gedispatched wordt. Het probleem hierbij is dat de reducers van het tweede project gekopieerd moeten worden in

de root reducer van het eerste project. Dit resulteert in onstabiele code wanneer er reducers toegevoegd en/of verwijderd worden in het tweede project. Dit onderzoek heeft een meerwaarde voor bedrijven/personen die een React-Redux project hebben en functionaliteit (componenten) willen gebruiken uit een bestaande React-Redux repository.

0.3 Onderzoeksvraag

Hoe kunnen twee react-redux projecten met elkaar gecombineerd worden? Hoe kunnen twee root reducers met elkaar gecombineerd worden?

0.4 Onderzoeksdoelstelling

Het beoogde resultaat van de bachelorproef is om een minimaal aantal lines of code te hebben. Idealiter is dit een aangepaste versie van de functie *combineReducers* waarbij er twee root reducers worden gecombineerd met elkaar.

0.5 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 1 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.

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

In Hoofdstuk 3, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvraag. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

1. Stand van zaken

inleiding

Zoals eerder vermeld wordt er gebruik gemaakt van twee React-Redux projecten. In dit hoofdstuk wordt meer informatie gegeven omtrent de achtergrond van deze projecten en wat er nodig is om deze met elkaar te laten communiceren.

1.1 React

Volgens (React, 2018) is React een JavaScript library om user interfaces te bouwen. React is component-based. Een component implementeert een *render* methode die data als input neemt en een view retourneert. Om dit te doen wordt JSX gebruikt. Er kunnen simpele views gemaakt worden voor elke state van de applicatie en React zal deze updaten en de juiste componenten renderen wanneer de data verandert.

1.1.1 JSX

JSX is een XML/HTML-achtige syntax gebruikt door React die ECMAScript uitbreidt zodat XML/HTML-achtige text kan bestaan met JavaScript/React code. Deze syntax is bedoeld om gebruikt te worden door preprocessoren (zoals Babel) om HTML text gevonden in JavaScript files om te zetten in standaard JavaScript objecten. Dit wil dus zeggen dat je door JSX te gebruiken HTML structuren en JavaScript code kan schrijven in dezelfde file, Babel zal dan alle uitdrukkingen vertalen naar JavaScript code. Waar vroeger dus JavaScript code in HTML werd geplaatst, laat JSX het toe om HTML in JavaScript te plaatsen. (TODO REF JSX)

1.1.2 Components

Een enkele view van een user interface is opgedeeld in een aantal stukken, in een aantal components. De start component bevat een tree, die tree kan opgedeeld worden in een aantal sub-componenten. Deze kunnen dan weer opgedeeld worden in nog meer sub-componenten. Dit kan dus resulteren in een complexe tree met verschillende React componenten. Er zijn echter ook nog verschillende soorten components, namelijk *simple* en *stateful*. (TODO REF JSX ZELFDE)

Simple component

React components implementeren de *render* methode, ze retourneren wat er moet afgebeeld worden. Daarvoor wordt het eerder aangekaarte JSX gebruikt. Input data die doorgegeven is aan de component kan opgeroepen worden door de *props* aan te spreken van deze component. Sidenote: JSX is optioneel, het gewenste resultaat kan ook bereikt worden door JavaScript code alleen. JSX maakt het wel overzichtelijker om de props aan te spreken. (React, 2018)

Stateful component

In toevoeging met data als input nemen (via *props*), kan een component zijn interne state data aanspreken. Wanneer de state data van een component verandert, wordt de render methode opnieuw aangeropen zodat de juiste data getoond wordt. (React, 2018)

1.1.3 React lifecycle

Met React is het mogelijk om components te creëren door de voorziene methode van React te gebruiken. Deze methode verwacht een *render* methode en zal een lifecycle triggeren, de lifecycle van een component. Deze lifecycle kan onderverdeeld worden in 4 grote fasen:

- initialisatie
- mounten
- updaten
- unmounten

FIGUUR 2

Initialisatie

In deze fase worden de initiële state en de default props ingesteld. De initiële state wordt in de constructor ingesteld, deze kan later altijd veranderd worden door de *setState* methode. *defaultProps* is een property van *Component* die overschreven kan worden met nieuwe waarden voor de props.

mounten

In de tweede fase zijn er een aantal *hook* methodes die aangeropen kunnen worden voor het mounten en na het mounten. Een hook methode is een methode waar je kan inpikken in de lifecycle en bestaande code kan veranderen of verbeteren naar eigen noden.

Al de dingen die moeten gebeuren voor een component gaat mounten moet gedefinieerd zijn in de *componentWillMount*. Deze methode wordt een keer per lifecycle uitgevoerd voor de eerste *render*.

Render zal de component mounten in de browser. Het is een pure methode, dus het geeft altijd dezelfde output gegeven dezelfde input.

Na deze eerste render, wordt de methode *componentDidMount* aangeropen. Deze methode wordt opnieuw een keer per lifecycle uitgevoerd.

updaten

Deze derde fase start wanneer de React component succesvol gerenderd is op de browser. De component kan nu upgedate worden op twee manieren: het verzenden van nieuwe props of het updaten van de state.

Wanneer de component nieuwe props ontvangt of de state is upgedate, wordt in de methode *shouldComponentUpdate* gevraagd of er een re-render moet gebeuren of niet. Deze methode retourneert dus een boolean en zal standaard re-renderen tenzij anders beschreven. Het is dus mogelijk om enkel een re-render uit te voeren als de props veranderen. Deze hook methode wordt vooral gebruikt als renderen een zware methode is. Dan is het niet voordelig om altijd alles opnieuw te renderen.

Als de uitkomst van deze methode true is, dan zal de component updaten en wordt de hook methode *componentWillUpdate* opgeropen. In deze methode worden de nodige voorbereidingen gedaan voor de volgende render, gelijkaardig als *componentWillMount*. Hierna wordt de component dus opnieuw gerenderd.

Wanneer dit succesvol is dan zal *componentDidUpdate* de third party libraries updaten en reloaden.

Wanneer echter de props veranderen en het is niet de eerste render dan zal in *componentWillReceiveProps* de state en de props opnieuw gesynchroniseerd worden met elkaar.

unmounten

In de laatste fase is de component niet meer nodig en zal deze via *componentWillUnmount* verwijderd worden. Hier kan er dan een cleanup gebeuren wat betreft user details en authorization tokens.

1.2 Redux

Door stateful components die elkaar aanroepen verhoogt de complexiteit om de state te beheren. Om tegen te gaan dat de controle verloren gaat over wanneer, hoe en waarom de state upgedate wordt, kan redux geïntroduceerd worden als oplossing om de state te managen. Redux zal eigenlijk proberen om de veranderingen op de state voorspelbaar te maken door bepaalde restricties op te leggen omtrent het updaten van de state. Deze restricties staan beschreven in de *drie principes* van Redux.

1.2.1 Drie principes

Een enkele source

De state van de hele applicatie zit in een object tree in een enkele *store*. Een enkele state maakt het makkelijker om te debuggen of om de applicatie te inspecteren. Sommige functionaliteit kan snel geïmplementeerd worden hierdoor, zoals een undo/redo waar dan geswitched kan worden tussen een vorige en een volgende state van de applicatie. Dit kan dus enkel wanneer de hele state van de applicatie is opgeslaan in de object tree.

State is read-only

De enige manier om een state te veranderen is door het verzenden van een *action*, dit is een object dat beschrijft wat er gebeurt. Dit zorgt er voor dat callbacks nooit direct naar de state gaan schrijven. Ze tonen de intentie om de state te transformeren. Alle veranderingen zijn gecentraliseerd en gebeuren een voor een in een vaste volgorde.

Pure functies voor veranderingen

Om te specificeren hoe de state tree veranderd is ten gevolge van actions, worden pure *reducers* geschreven. Reducers zijn eigenlijk pure functies die dan geëxporteerd worden. Ze nemen de vorige state en de action om zo de volgende state te retourneren. Het is belangrijk om een nieuw state object te retourneren in plaats van het muteren van de vorige state.

Pure functie Een pure functie is een functie die met dezelfde input altijd dezelfde output produceert.

Redux draait nu net om de voorspelbaarheid en betrouwbaarheid. Door een mutatie kan deze state niet meer voorspelbaar of betrouwbaar zijn.

1.2.2 Redux flow

In de subsectie *Drie principes* kwamen een aantal core concepts voor van Redux. Om deze te verduidelijken zal hier een beschrijven van de Redux flow gegeven worden met nodige uitleg van de core concepts.

De Redux architectuur draait rond een strikte, unidirectionele data flow. Dit wil zeggen dat al de data in de applicatie hetzelfde lifecycle patroon volgt. Alle veranderingen zijn gecentraliseerd en gebeuren een voor een in een vaste volgorde. Dit zorgt er voor dat de logica van de applicatie voorspelbaar en betrouwbaar is. Het begunstigt ook data normalisatie, zodat er geen meerdere, onafhankelijke duplicaties zijn van dezelfde data die niet bewust zijn van elkaars bestaan.

De data lifecycle voor om het eender welke Redux app volgt 4 stappen:

- een action wordt gedispatched op de store
- de Redux store roept de reducer functie aan
- de root reducer combineert output van meerdere reducers
- de Redux store saved de volledige state tree

Actions Een action is een object die beschrijft wat er gebeurt. Deze actions zijn payloads van informatie die data verzenden van de applicatie naar de store. Acties zijn tevens de enige soort van informatie voor de store. Het zijn JavaScript objecten die een *type* property moeten hebben om aan de duiden welke action uitgevoerd wordt. Deze types zijn string constanten die in grote applicaties in een aparte module worden opgeslaan om overzichtelijk te houden.

Reducers Reducers specificeren hoe de state van de applicatie zal veranderen ten gevolge van een action die gedispatched wordt naar de store. Waar actions beschrijven dat er iets gebeurd is, beschrijven ze niet hoe de state van de applicatie veranderd is. Een reducer is een pure functie -zie vorige subparagraaf- die de vorige state van de applicatie en een action neemt en daaruit een nieuwe state zal retourneren. Niet-pure functies aanroepen en mutaties zijn dus een no-go om de betrouwbaarheid en voorspelbaarheid van de applicatie te behouden. Redux zal de reducer aanroepen met een undefined state voor de eerste keer. Hier moet er dus een initiële state van de applicatie worden ingesteld.

Elk van de reducers is zijn deel van de globale state aan het beheren. De *state* parameter is verschillend voor elke reducer en hangt samen met het deel van de state die de reducer beheert.

Op deze manier kunnen reducers gesplitst worden in aparte files om zo compleet onafhankelijk verschillende data domeinen te laten beheren door verschillende reducers.

Om deze reducers dan te combineren voor de volledige functionaliteit van het databeheer te hebben, wordt er gebruik gemaakt van de functie *combineReducers*. Deze functie voert

boilerplate logica uit om de reducers op te roepen aan de hand van hun keys en om de resultaten hiervan te combineren in een enkel object.

boilerplate code korte tekst die zonder veel aanpassingen kan worden hergebruikt

Store In de vorige paragrafen werden actions en reducers gedefinieerd. Actions werden gedefinieerd als objecten die representeren wat er gebeurd is terwijl reducers gedefinieerd werden als pure functies die de state gaan updaten volgens de verkregen actions. De store is een object die deze zaken samenbrengt. Dit zorgt ervoor dat de store een aantal verantwoordelijkheden heeft:

- het vasthouden van de state van de applicatie
- toegang geven tot de state
- toelaten dat de state upgedate kan worden door een action te dispatchen
- listeners registreren via subscribe
- listeners unsubscriben via de functie geretourneerd door subscribe

Het laatste punt laat toe om een callback te registreren (subscribe) die de redux store zal aanroepen elke keer een actie wordt gedispatched. Op deze manier kan de UI van de applicatie upgedate worden naargelang de state van de applicatie.

Het is belangrijk om te noteren dat er maar een store in de Redux applicatie is. Wanneer er logica moet gesplitst worden, is het beter om reducer compositie te gebruiken om later te combineren dan om meerdere stores te implementeren. Om deze reden wordt in dit onderzoek ook geopteerd om een extensie te schrijven op combineReducers van Redux in plaats van meerdere stores te implementeren.

Om een store te creëren hebben we een reducer nodig. In een vorige paragraaf werd de methode combineReducers uitgelegd om meerdere reducers te combineren in een reducer. Deze gecombineerde reducer kan geïmporteerd worden en meegegeven worden als parameter aan de methode createStore om een store te creëren.

Als er dan gekeken wordt naar de data flow dan worden er eerst actions gedispatched naar de store. De Redux store zal dan de corresponderende reducer aanroepen met twee argumenten, namelijk de huidige state en de action die gedispatched werd. Deze zal altijd dezelfde output hebben en dus voorspelbaar zijn aangezien reducers pure functies zijn. Hierna zal de root reducer de output van meerdere reducers combineren in een enkele state tree. Dit gebeurt met de functie combineReducers die de verschillende functies voor een bepaald deel van de state zal combineren. Daarna gaat de Redux store de volledige state tree die geretourneerd werd door de root reducer gaan opslaan. Hier kan de container dan data uit lezen en doorgeven aan een presentationele component.

Connectie met React React bindings zitten echter niet standaard in Redux, eerst moet de npm-package react-redux worden geïnstalleerd. Om React en Redux samen te laten

functioneren wordt gebruik gemaakt van een opsplitsing in presentationele en container components. ZIE FIGUUR 1

Presentationele components zijn -zoals de naam aangeeft- prioritair voor de visualisatie. Ze verkrijgen enkel data via de props en sturen geen actions aan.

Daar waar presentationele components gericht zijn op de visualisatie, zijn de container components gericht op het functionele aspect. Ze voorzien andere container (of presentationele) components van data en gedrag. Deze components gaan ook actions aanroepen om aan te duiden dat de data moet veranderen.

Containers Het idee achter containers is dat ze de data gaan ophalen en dan de corresponderende sub-component gaan renderen. Deze data wordt gehaald uit de state en omgezet naar props voor die container. Daarna krijgen de sub-componenten de juiste props mee om hun data te renderen. r

1.2.3 Npm

Om de twee projecten met elkaar te laten communiceren, wordt gebruik gemaakt van npm. De extensie op Redux zal geschreven worden en onder een eigen package gepubliceerd worden op npm, op deze manier kan de geschreven extensie aangesproken worden. (eigen)

npm is 's werelds grootste software register met ongeveer 3 biljoen downloads per week. Het register bevat meer dan 600.000 packages. Open source developers van elk continent gebruiken npm voor het delen en lenen van packages. Veel organisaties gebruiken npm ook voor privaat development, zoals in dit onderzoek ook het geval is.

npm bestaat uit 3 verschillende componenten.

- de website
- de Command Line Interface (CLI)
- het register

De website wordt gebruikt om verschillende packages te zoeken en zelf packages op te publiceren, privaat of publiek. Deze publicatie kan via de CLI gedaan worden (NPM REF)

De grootste reden om npm te gebruiken is om bestaande packages aan te passen en te specialiseren naar eigen apps. Natuurlijk kan ook gebruik gemaakt worden van bestaande custom packages van andere gebruikers.

2. Methodologie

2.1 Analyse

Eerst werd een denk-analyse uitgevoerd om te achterhalen welke manieren er waren om de twee projecten met elkaar te laten communiceren.

Aangezien er aanpassingen moesten gebeuren aan het Scratch project omdat een rechtstreekse import niet werkte door het gebrek aan functionaliteit, werd geopteerd voor een nieuwe versie van het Scratch project te maken en deze te publiceren op npm.

Om deze analyse verder uit te voeren werd gekeken naar wat de store van het hoofdproject nodig had als parameters. Daarna werden een reeks methodes uitgeprobeerd om te zien wat wel werkte en wat niet.

2.2 Methodes

2.2.1 combineReducers

In de eerste methode werd geprobeerd om de root reducer van het scratch project rechtstreeks te integreren in de store van het hoofdproject. In de configuratie van de store wordt meegegeven wat de root reducer is en wat de initiële state van de applicatie is. De bedoeling was om een grote gecombineerde root reducer te maken om daarmee de store te initialiseren. Het enige wat hier aangepast werd aan de scratch-integration package was het exporteren van de root reducer in de index file.

Twee root reducers met elkaar combineren via de methode *combineReducers* gaat echter niet zomaar. Deze methode is een functie met als parameter de reducers die moeten gecombineerd worden. Zoals eerder vermeld bevat een reducer een key en een value. De parameter bevat dus een lijst van reducers met elk hun key en value. De functie *combineReducers* overloopt deze lijst en vraagt eerst alle keys op en schrijft deze weg in een nieuwe variabele. Daarna wordt voor elke reducer key gekeken als de corresponderende value een functie is. Als dit zo is, dan wordt deze reducer weggeschreven in een nieuwe lijst variabele.

Deze functie retourneert dan opnieuw een functie 'combination'. In de scope van deze functie zitten onderandere twee lijsten. Een lijst voor de finale reducer keys en een lijst voor de finale reducers.

Omdat deze na combinatie in de scope van de functie zitten, is het onmogelijk om na een eerste combinatie een tweede combinatie uit te voeren. Dit zorgt er dus voor dat eenmaal er een root reducer gecombineerd is, deze niet opnieuw kan gebruikt worden om nog eens te combineren met een andere root reducer. Om hier verder op in te spelen werd naar een tweede methode gezocht.

2.2.2 object export

Als tweede methode werd geprobeerd om geen rechtstreeks gebruik te maken van de root reducer van Scratch, maar om alle reducers te combineren in een groot object en dan dat object exporteren. Om deze methode mogelijk te maken moet de root reducer van het hoofdproject ook een object zijn en geen gecombineerde functie. Anders keert het probleem van de function scope terug. Met beide root reducers als object is het wel mogelijk om een gecombineerde root reducer af te leveren. De lijsten van de reducer keys en reducers bevatten dan de keys en reducers van beide projecten. Hierdoor wordt dus de volledige functionaliteit omvat van beide projecten.

Nadelen

Een eerste nadeel van deze methode is dat er dan ook met de root reducer van het hoofdproject rekening moet gehouden worden. Deze mag dan namelijk ook niet gecombineerd zijn. Om deze methode uit te voeren was er een gedeeltelijke herstructurering nodig, waardoor deze methode eigenlijk al niet meer optimaal is.

Een tweede nadeel zijn de veranderingen in het Scratch project. Stel dat er een reducer bijkomt of verwijderd wordt, dan moet dit altijd manueel aangepast worden vooraleer er terug een werkende versie geproduceerd wordt. Indien er enkel nood is aan de root reducer die geëxporteerd wordt, dan zou dit geen probleem zijn aangezien deze veranderingen mee opgenomen worden in de combinatie van de root reducer. Door elke reducer afzonderlijk in een nieuw object te steken en zo te exporteren wordt er heel wat extra onderhoudswerk gecreeërd voor deze package.

Een derde nadeel zijn de configuraties en dependencies die nodig zijn om het Scratch

project te laten draaien. Elke dependency die gebruikt wordt, moet ook mee overgenomen worden in de package.json van het hoofdproject. Dit zorgt er ook voor dat de configuraties van webpack ook moeten meegenomen worden. Om dit te realiseren werd een nieuwe package gemaakt van de bestaande react-scripts package met de uitbreidingen die nodig zijn om een succesvolle integratie te verkrijgen.

2.2.3 Eigen methode

Als laatste methode werd gezocht naar een alternatief voor het vele onderhoudswerk die zich voordoet bij een object export. De bedoeling van deze extensie zou als het ware een uncombineReducers methode zijn. Op deze manier kan snel alles los gemaakt worden van elkaar om dan later opnieuw te kunnen combineren.

3. Conclusie

Curabitur nunc magna, posuere eget, venenatis eu, vehicula ac, velit. Aenean ornare, massa a accumsan pulvinar, quam lorem laoreet purus, eu sodales magna risus molestie lorem. Nunc erat velit, hendrerit quis, malesuada ut, aliquam vitae, wisi. Sed posuere. Suspendisse ipsum arcu, scelerisque nec, aliquam eu, molestie tincidunt, justo. Phasellus iaculis. Sed posuere lorem non ipsum. Pellentesque dapibus. Suspendisse quam libero, laoreet a, tincidunt eget, consequat at, est. Nullam ut lectus non enim consequat facilisis. Mauris leo. Quisque pede ligula, auctor vel, pellentesque vel, posuere id, turpis. Cras ipsum sem, cursus et, facilisis ut, tempus euismod, quam. Suspendisse tristique dolor eu orci. Mauris mattis. Aenean semper. Vivamus tortor magna, facilisis id, varius mattis, hendrerit in, justo. Integer purus.

Vivamus adipiscing. Curabitur imperdiet tempus turpis. Vivamus sapien dolor, congue venenatis, euismod eget, porta rhoncus, magna. Proin condimentum pretium enim. Fusce fringilla, libero et venenatis facilisis, eros enim cursus arcu, vitae facilisis odio augue vitae orci. Aliquam varius nibh ut odio. Sed condimentum condimentum nunc. Pellentesque eget massa. Pellentesque quis mauris. Donec ut ligula ac pede pulvinar lobortis. Pellentesque euismod. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent elit. Ut laoreet ornare est. Phasellus gravida vulputate nulla. Donec sit amet arcu ut sem tempor malesuada. Praesent hendrerit augue in urna. Proin enim ante, ornare vel, consequat ut, blandit in, justo. Donec felis elit, dignissim sed, sagittis ut, ullamcorper a, nulla. Aenean pharetra vulputate odio.

Quisque enim. Proin velit neque, tristique eu, eleifend eget, vestibulum nec, lacus. Vivamus odio. Duis odio urna, vehicula in, elementum aliquam, aliquet laoreet, tellus. Sed velit. Sed vel mi ac elit aliquet interdum. Etiam sapien neque, convallis et, aliquet vel, auctor non, arcu. Aliquam suscipit aliquam lectus. Proin tincidunt magna sed wisi. Integer blandit

lacus ut lorem. Sed luctus justo sed enim.

Morbi malesuada hendrerit dui. Nunc mauris leo, dapibus sit amet, vestibulum et, commodo id, est. Pellentesque purus. Pellentesque tristique, nunc ac pulvinar adipiscing, justo eros consequat lectus, sit amet posuere lectus neque vel augue. Cras consectetuer libero ac eros. Ut eget massa. Fusce sit amet enim eleifend sem dictum auctor. In eget risus luctus wisi convallis pulvinar. Vivamus sapien risus, tempor in, viverra in, aliquet pellentesque, eros. Aliquam euismod libero a sem.

Nunc velit augue, scelerisque dignissim, lobortis et, aliquam in, risus. In eu eros. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Curabitur vulputate elit viverra augue. Mauris fringilla, tortor sit amet malesuada mollis, sapien mi dapibus odio, ac imperdiet ligula enim eget nisl. Quisque vitae pede a pede aliquet suscipit. Phasellus tellus pede, viverra vestibulum, gravida id, laoreet in, justo. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Integer commodo luctus lectus. Mauris justo. Duis varius eros. Sed quam. Cras lacus eros, rutrum eget, varius quis, convallis iaculis, velit. Mauris imperdiet, metus at tristique venenatis, purus neque pellentesque mauris, a ultrices elit lacus nec tortor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent malesuada. Nam lacus lectus, auctor sit amet, malesuada vel, elementum eget, metus. Duis neque pede, facilisis eget, egestas elementum, nonummy id, neque.

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introductie

Vooraleer er dieper op het probleem kan worden ingegaan is er meer context nodig rond een aantal basisbegrippen van React en Redux. Volgens de website React (2018) is React een JavaScript library. Met behulp van React worden views gecreëerd voor elke state in onze applicatie. Volgens de website Redux (2018) is Redux een state container voor JavaScript apps. Redux helpt om de afgebeelde data te beheren en beschrijft hoe er gereageerd wordt op user actions. Een store is een object die de volledige state-tree van de applicatie bevat. De enige manier om een state te veranderen in de store is door een action te dispatchen. Een action zal er eigenlijk voor zorgen dat er data van de applicatie verzonden wordt naar de store. Dit is tevens de enige vorm van input voor de store. Actions beschrijven dus het feit dat er iets gebeurd is, maar ze beschrijven niet hoe de state van de applicatie verandert. Dit is de taak van een reducer, deze toont aan hoe de state veranderd is ten gevolge van een action die verzonden werd naar de store. Een container is dan weer verantwoordelijk voor het verkrijgen van data. Om die data te verkrijgen wordt er gebruik gemaakt van de connect functie (voor de store) en een functie die de data uit de state neemt en deze mapt naar zijn eigen props. Een container is ook verantwoordelijk voor het dispatchen van actions. (Redux, 2018) Het probleem zit bij het samenvoegen van 2 react-redux apps tot 1 app. Er is geen manier bekend om beide stores te behouden in 1 project. Dit is relevant omdat er toch een aantal hits op Overflow (2018) te vinden zijn voor dit probleem. 2 React projecten die elk afzonderlijk een Redux store gebruiken kunnen

als ze samengevoegd worden momenteel maar de volledige functionaliteit van 1 store gebruiken. Onderzoeksvraag:

- Op welke manier kunnen de stores van 2 react-redux apps onafhankelijk functioneren in 1 app?

A.2 State-of-the-art

Zoals eerder vermeld is er zeer weinig documentatie te vinden over dit probleem. Er zijn wel een aantal vragen in die richting op Stack Overflow gesteld. De antwoorden die op deze vragen gegeven zijn, bieden geen oplossing op lange termijn of bieden geen schaalbaarheid/uitbreidbaarheid. Er zal veel tijd gestoken worden in het verdiepen van de GitHub repository van Redux, geschreven door Abramov, 2016. Er is wel documentatie te vinden over de Redux-store sharen in meerdere components binnen hetzelfde project. Hier staat een uitgewerkt voorbeeld van op Stack Overflow door b.g (2017). Het wordt echter onduidelijk wanneer we de Redux-store willen sharen tussen meerdere projecten.

A.3 Methodologie

Er zal geprobeerd worden een uitbreiding te schrijven op de bestaande Redux library. Zoals eerder vermeld houdt een redux store eigenlijk de hele state-tree van de applicatie vast. De bedoeling van deze uitbreiding zal zijn om een draaiende app te hebben met 2 onafhankelijke stores van 2 projecten. Deze realisatie zal gebeuren door het ene project om te zetten naar een npm-package. Op deze manier kunnen geëxporteerde componenten makkelijk geïmporteerd worden in het andere project. Met de uitbreiding zal geprobeerd worden om een store toe te voegen aan het project, zonder dat de andere store daar enige invloed van ondervindt. Om dit te kunnen realiseren zal het react-redux patroon goed moeten bestudeerd worden. Daarbij kan ook naar Flux (2015) en MobX (2017) gekeken worden om eventuele analogieën door te trekken. In dit onderzoek zullen een aantal grafieken worden gemaakt met de gemiddelde rendertijd van verschillende oplossingen (met bv: nesting en reducers importeren). Anderszijds wordt er ook naar de lines of code gekeken. Een groot criteria hierbij is dan het dupliceren van code of het vervuilen van de applicatie door code toe te voegen die ze eigenlijk niet nodig zou moeten hebben.

A.4 Verwachte resultaten

Zoals eerder aangegeven worden er een aantal oplossingen verwacht. Enerzijds kan een nesting van de stores een mogelijke oplossing bieden, maar de vraag is dan of alle functionaliteit wordt behouden. Door het schrijven van de uitbreiding kan de rendertijd hoger zijn dan normaal. Een andere oplossing is door de reducers van het ene project samen te voegen met de reducers van het andere project, waarbij een hoger aantal lines of code verwacht wordt.

A.5 Verwachte conclusies

Een verwachte conclusie bij nesting is dat mogelijks niet alle functionaliteit behouden wordt. Een kopie nemen van de reducers van het ene project is geen efficiënte oplossing en wel om deze reden: stel dat er een update gepushed wordt naar de npm-package dan zou men alle reducers opnieuw moeten kopiëren naar het andere project. Tevens vervuilt dit ook de hoofdapplicatie.

Bibliografie

Abramov, D. (2016). Redux [Github]. Verkregen van https://github.com/reactjs/redux

b.g, S. (2017). How to share redux store in multiple components.

Elliott, E. (2016, maart 26). Master the JavaScript interview: What is a pure function? Verkregen van https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-pure-function-d1c076bec976

Flux. (2015). Flux. Verkregen van https://facebook.github.io/flux/

MobX. (2017). MobX.js. Verkregen van https://mobx.js.org/

Overflow, S. (2018). Stack Overflow. Verkregen van https://stackoverflow.com

React. (2018). React.js. Verkregen van https://reactjs.org/

Redux. (2018). Redux.js. Verkregen van https://redux.js.org/