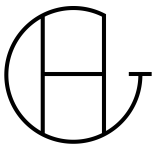


HoGent

Faculteit Bedrijf en Organisatie

Redux uitbreiding voor combinatie van twee React-Redux projecten

Thibault Gobert

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

Promotor:
Harm De Weirdt
Co-promotor:
Simon De Gheselle

Instelling: —

Academiejaar: 2017-2018

Derde examenperiode

Woord vooraf

Op mijn stage werd ik het eerst geconfronteerd met dit probleem. Ik moest daar namelijk de integratie voorzien van een React-Redux project (Scratch). De reden waarom ik dit project ook heb gekozen is omdat ik dit project het beste ken en daardoor zo weinig mogelijk tijd zou verliezen door mij te verdiepen in een ander project. Nu Scratch overgeschakeld is op JavaScript voor hun graphical user interface, wordt het zeer interessant om een makkelijke manier te voorzien om dit project te integreren. Zeker voor een bedrijf zoals CodeFever, waar ik stage heb gedaan. Zij voorzien namelijk programmeerlessen voor kinderen, dus het zou zeer handig zijn moesten zij een ingebouwde versie van Scratch hebben.

Het werd voor mij al snel duidelijk dat er quasi niets van documentatie hierover te vinden was online. Dit vond ik zeer raar, omdat het net zo belangrijk is om dit te kunnen doen. Dit zorgde er ook voor dat ik een oplossing voor dit probleem wou vinden.

Verder wil ik nog Harm De Weirdt bedanken voor de begeleiding alsook Simon De Gheselle om mijn co-promotor te zijn. Finaal bedank ik ook de Hogeschool Gent om het mogelijk te maken deze scriptie te schrijven.

Samenvatting

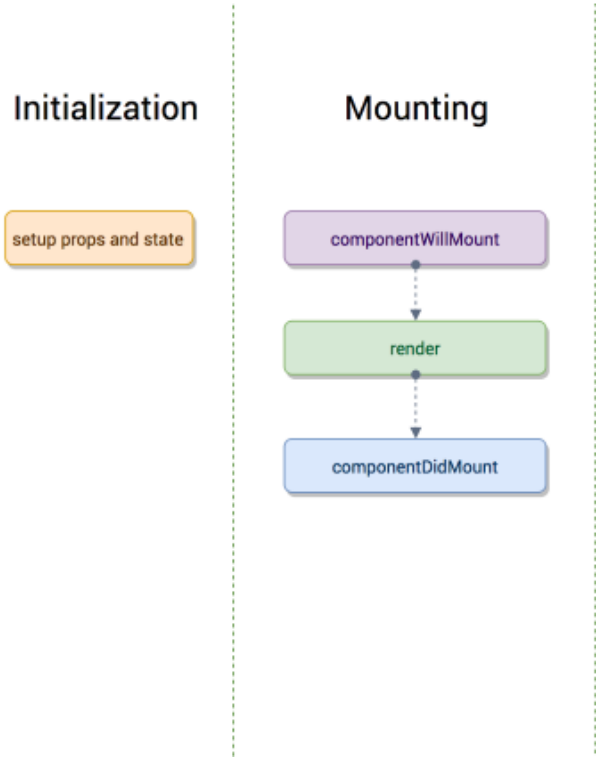
In dit onderzoek worden twee React-Redux projecten opgesteld. React maakt gebruik van JavaScript voor het bouwen van user interfaces, waar Redux een concept is voor data storage en communicatie binnen de applicatie. Een store is een concept van Redux die de hele state tree van de applicatie vasthoudt. Het probleem situeert zich in 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. Om dit 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 (de functie die reducers combineert) die de mogelijkheid biedt om de functionaliteit van het tweede project te gebruiken in het eerste project. Tot dusver is er nog geen bekende manier om de combinatie van een root reducer ongedaan te maken en opnieuw te combineren met de reducers van alle projecten samen. Dit wil zeggen dat de enige bekende manier een export van de bestaande reducers is. Er is dus zeker nog ruimte voor verder onderzoek over hoe reeds gecombineerde reducers opnieuw gecombineerd kunnen worden met elkaar.

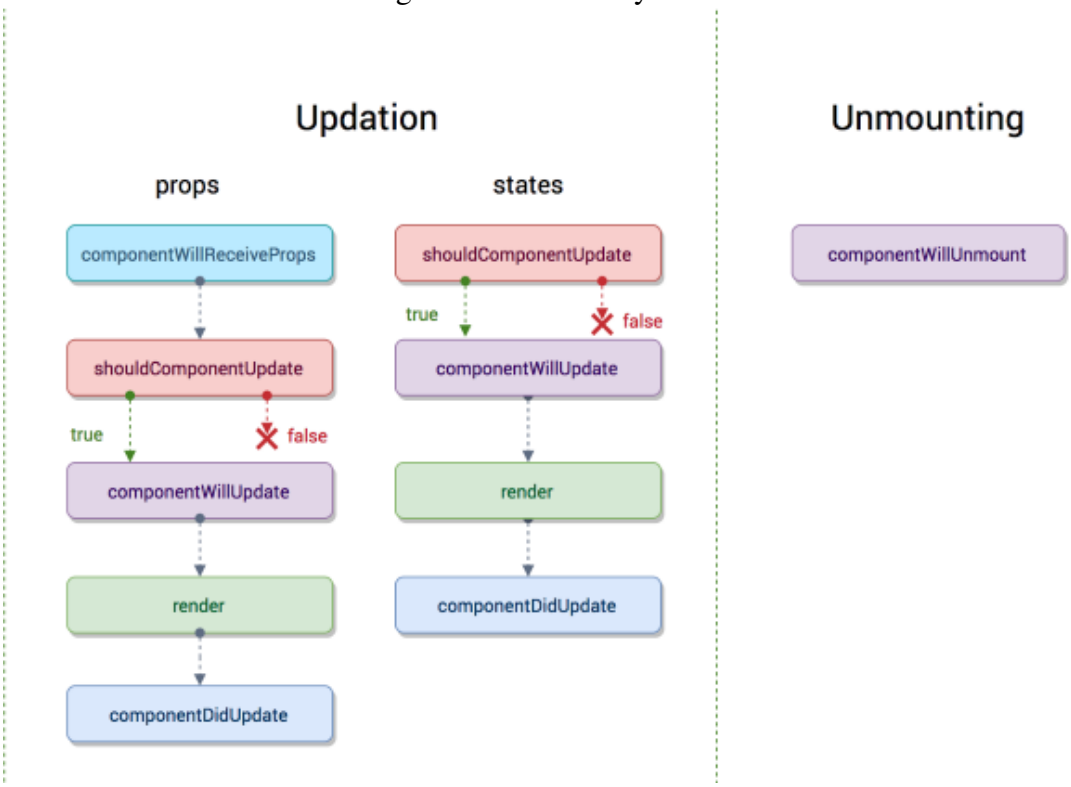
Inhoudsopgave

Lijst van figuren

Figuur 1: React lifecycle deel 1



Figuur 2: React lifecycle deel 2



Figuur 3: Redux devtools

```
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

function configureStoreDev(initialState) {
  const store = createStore(
    rootReducer,
    initialState,
    composeEnhancers(applyMiddleware(...middleware)),
  );
  return store;
}
```

Figuur 4: Presentationeel vs container

	Presentational Components	Container Components
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

Figuur 5: Methode combineReducers

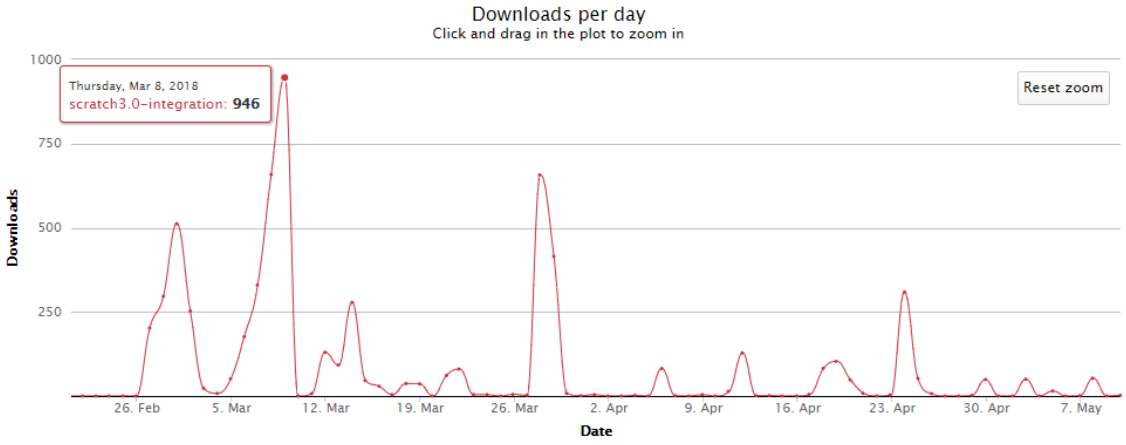
```
export default function combineReducers(reducers) {
  var reducerKeys = Object.keys(reducers)
  var finalReducers = {}
  for (var i = 0; i < reducerKeys.length; i++) {
    var key = reducerKeys[i]
    if (process.env.NODE_ENV !== 'production') {
      if (typeof reducers[key] === 'undefined') {
        warning(`No reducer provided for key "${key}"`)
      }
    }
    if (typeof reducers[key] === 'function') {
      finalReducers[key] = reducers[key]
    }
  }
  var finalReducerKeys = Object.keys(finalReducers)
  function combination(state = {}, action) {
    var hasChanged = false
    var nextState = {}
    for (var i = 0; i < finalReducerKeys.length; i++) {
      var key = finalReducerKeys[i]
      var reducer = finalReducers[key]
      var previousStateForKey = state[key]
      var nextStateForKey = reducer(previousStateForKey, action)
      if (typeof nextStateForKey === 'undefined') {
        var errorMessage = getUndefinedStateErrorMessage(key, action)
        throw new Error(errorMessage)
      }
      nextState[key] = nextStateForKey
      hasChanged = hasChanged || nextStateForKey !== previousStateForKey
    }
    return hasChanged ? nextState : state
  }
}
```

Figuur 6: Object export

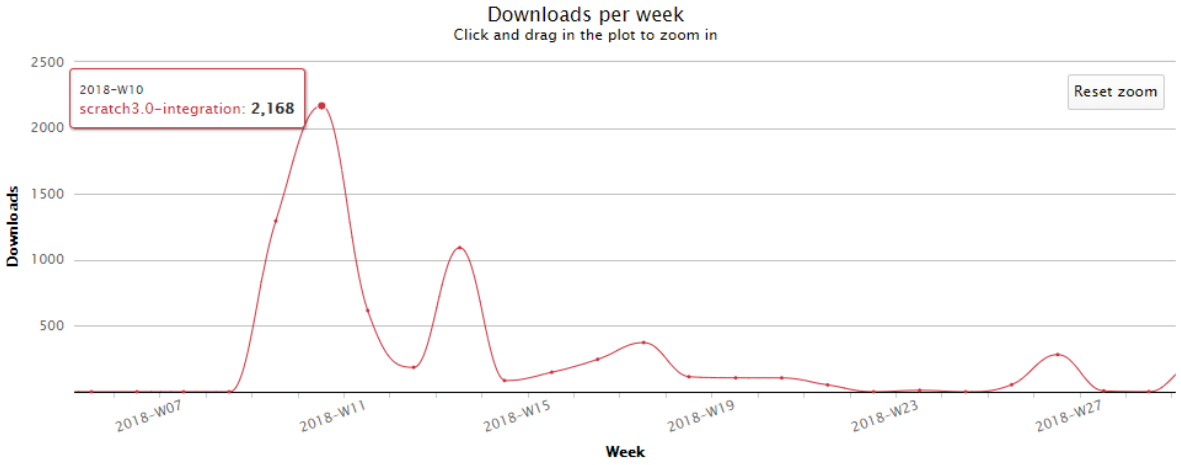
```
const appReducerList = {
  loggedIn,
  profile,
  courses,
  lessons,
  organisation,
  students,
  teachers,
  locations,
  application,
  slides,
  attendances,
  notes,
  evaluations,
  contracts,
  locale,
  payments,
  filters,
  logs,
  events,
  exercises,
};

const appReducer = combineReducers({
  ...appReducerList, ...guiReducers,
});
```

Figuur 7: Downloads per dag



Figuur 8: Downloads per week



Figuur 9: Totale downloads

package	downloads
scratch3.0-integration	7,161

Inleiding

0.1 Context

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 voor dit onderzoek, zoals: actions, reducers, store en containers. Deze begrippen vormen de omkadering van Redux. (Redux, 2018)

De Redux architectuur heeft een unidirectionele data, dit wil zeggen dat 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 op deze manier 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 te 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.

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 action wordt gedispatched. Op deze manier kan de UI van de applicatie upgedate worden naargelang de state van de applicatie.

0.2 Probleemstelling

Een van de problemen is 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 herkend 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 zouden 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

Onderzoeksvraag:

- Hoe kan de functionaliteit van een tweede React-Redux project geïntegreerd worden in een ander React-Redux project?

Er wordt getracht om een manier te vinden waarbij er specifieke functionaliteit uit het ene project kan genomen worden door een container te exporteren. Om deze container te laten werken zullen ook de reducers nodig zijn van dit project.

Dit zorgt voor de deelonderzoeksvraag:

- 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 en onderhoud te hebben. Idealiter is dit een aangepaste versie van de functie *combineReducers* waarbij er twee root reducers worden gecombineerd met elkaar zonder nesting te veroorzaken. Verder is het vooral belangrijk dat de functionaliteit behouden wordt van beide projecten.

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. Literatuurstudie

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. (Lerner, g.d.-b)

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*. (Lerner, g.d.-b)

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 worden opgeroepen 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 aanspreken om daar data uit te halen. Wanneer de data uit de state 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 zoals beschreven in figuur 1 en 2:

- initialisatie
- mounten
- updaten
- unmounten

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, moeten 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 geüpdatet 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 geüpdatet, 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. (Haldar, 2017) (Sharif, 2015)

1.2 Redux

De requirements voor JavaScript single-page applicaties worden meer en meer gecompliceerd. De code moet veel meer state beheren dan vroeger. Deze state kan server responses en gecachete data, maar ook lokaal gecreëerde data die nog niet naar de server is doorgestuurd bevatten. Deze steeds verandere state beheren is moeilijk. Als een model een ander model kan updaten, waar een view dan een ander model kan updaten, die een nieuw model gaat updaten, die op zijn beurt ook nog eens een view kan updaten. Op een gegeven moment wordt het niet meer duidelijk wat er gebeurt in de applicatie. Wanneer een systeem ondoorzichtig en niet-deterministisch is, wordt het moeilijk om bugs te reproduceren of om nieuwe features toe te voegen. Deze complexiteit is moeilijk om mee om te gaan omdat er twee concepten in verwikkeld worden die moeilijk vatbaar zijn: mutatie en asynchroniteit. Beide kunnen apart heel handig zijn, maar samen kan dit voor heel wat problemen zorgen.

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 geüpdatet 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. (Elliott, 2016)

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. (Lerner, g.d.-a)

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. (Holtkamp, 2016)

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 geüpdatet 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 geüpdatet 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. (Redux, 2018)

Een manier om deze flow te controleren of te debuggen is via de redux-devtools extension voor chrome.

Redux devtools extension De Redux devtools extension wordt gebruikt voor het debuggen van veranderingen in de state van de applicatie. Met deze extension kan dus gekeken worden naar de volledige tree alsook naar de actions die verzonden worden. Er kan dan gekozen worden om bepaalde actions uit te voeren of te pauzeren tijdens een aantal actions. Dit zorgt dus voor een preciezere debugging. Vooraleer van deze extension gebruik kan gemaakt worden moet deze extension eerst gedefinieerd worden in de store. Deze wordt toegevoegd als *enhancer*.

Enhancer Een verbetering die ergens aan toegevoegd kan worden, in figuur 3 is het de redux devtools extension die toegevoegd wordt aan de store om een extra debugging tool aan te bieden.

Enkel het hoofdproject wordt geconnecteerd met de redux-devtools extension. Wanneer dus gekeken wordt naar de state tree, dan is dit deze van het hoofdproject. Wanneer er dus meerdere projecten met elkaar gecombineerd worden, zal er nog steeds enkel de state tree van de hoofdapplicatie staan. Het zou dus een absolute meerwaarde zijn om ook deze debug tool te kunnen gebruiken met de gecombineerde state tree van beide projecten. (Zalmoxisus, 2015)

Er werd ook onderzocht als er meerdere stores mogelijk zijn voor gebruik, dan zou er per geïntegreerd project een nieuwe store bijkomen in het hoofdproject. Het originele Flux patroon beschrijft meerdere stores in één applicatie, elke store houdt een verschillend segment van domein data vast. Dit kan problemen veroorzaken zoals een store die geforceerd wordt om te wachten op een andere store die nog geupdate moet worden. Dit is niet noodzakelijk zo in Redux, omdat de splitsing tussen data domeinen al bekomen wordt door het splitsen van één enkele reducer in een reeks kleinere reducers.

Het is mogelijk om meerdere verschillende Redux stores in een applicatie te hebben, maar het voorgenomen patroon bevat slechts één store. Met één store wordt het mogelijk om Redux devtools te gebruiken, het maakt persisteren van data simpeler alsook de subscription logica. Er zijn echter een aantal geldige redenen om meerdere stores te gebruiken in Redux:

- Het oplossen van een performantieprobleem dat veroorzaakt wordt door het te veel updaten van een deel van de state.
- Het isoleren van een Redux applicatie als een component in een grotere applicatie, in dit geval is het aan te raden een store te creëren per root instantie.

Nieuwe stores aanmaken is niet het eerste waar naar gekeken moet worden, het is beter om reducer compositie toe te passen en enkel meerdere stores te gebruiken als het probleem niet opgelost kan worden op deze manier.

Gelijkaardig, het is mogelijk om een de store instantie aan te spreken door deze rechtstreeks te importeren, dit is eveneens geen aan te raden patroon in Redux. Wanneer een instantie van de store wordt gemaakt en geëxporteerd uit de module, dan wordt deze een singleton. Dit wil zeggen dat het moeilijker wordt om een Redux applicatie als een component van een grotere applicatie te isoleren. Er kan dan ook geen server rendering geactiveerd worden, want op de server worden er verschillende store instanties gecreëerd voor elke request die gemaakt wordt.

Het is best practice om de root component te wrappen in een <Provider> en dan gaat React Redux zorgen dat de store doorgegeven wordt naar beneden. Op deze manier moeten components geen store module importeren en worden de isolatie van een Redux app of het activeren van server rendering veel makkelijker om te implementeren. (Redux, 2018)

Dit staat ook beschreven in een provider patroon volgens bloodyowl (2015). Veel React libraries moeten hun data doorgeven doorheen de component tree. Bijvoorbeeld, Redux moet zijn store doorgeven en React Router moet de huidige locatie doorgeven. Dit zou kunnen opgelost worden door een gedeelde, muteerbare state. Dit werkt enkel als er één state is. Wanneer er geprerenderd wordt op de server is het onmogelijk om te vertrouwen op deze implementatie. Gelukkig zorgt React voor een manier om de data van boven naar beneden te krijgen: *context*. Dit kan aanzien worden als het globale object van de component tree.

Helemaal bovenaan in de app moet er dus een provider zijn. De enige rol die deze provider zal hebben is het toevoegen van gewenste data aan de context van de tree zodat alle afstammelingen ook toegang hebben tot deze data.

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 4)

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. (Benedetto, 2016) (Dan Abramov, 2015)

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. („Container Components”, 2015)

1.3 Middleware

In het Redux framework is middleware de code die zich bevindt tussen het krijgen van een request en het verzenden van een response. Express bijvoorbeeld kan CORS headers (mechanisme dat extra HTTP headers gebruikt om te laten weten aan de browser dat een web applicatie met een bepaalde oorsprong gemachtigd is om resources te gebruiken van een server met een verschillende oorsprong), logging, compressie... toevoegen. De beste voorziening van middleware is dat het chainable is, er kan gebruik gemaakt worden van meerdere onafhankelijke third-party middleware in een enkel project.

Redux middleware gaat andere problemen oplossen dan de Express middleware, maar het is conceptueel dezelfde manier. Redux middleware gaat een third-party extension point voorzien tussen het dispatchen van een action en het moment dat deze action een reducer bereikt. Redux middleware wordt voornamelijk gebruikt om te loggen, voor crash reports, routing...

Één van de voordelen van Redux is dat de veranderingen van de state voorspelbaar en transparant zijn. Elke keer een action gedispatched wordt, zal de nieuwe state berekend en opgeslaan worden. De state kan niet veranderd worden door zichzelf, maar enkel ten gevolge van een specifieke action. Via middleware is het mogelijk om elke action die verzonden wordt in de app te loggen samen met de state die erna verkregen wordt. Wanneer iets mis gaat kan er dan makkelijk teruggekeken worden in de log en kan er snel gezien worden welke action de state corrupt heeft gemaakt. De meest naïve oplossing is om zelf de action en de volgende state te loggen elke keer de dispatch functie van de store wordt opgeroepen. Een betere oplossing is om de dispatch functie te vervangen. Met behulp van middleware kan zelf een logger geschreven worden die dan later enkel toegevoegd moet worden aan de store van de applicatie. Middleware wordt op dezelfde manier toegevoegd aan de store als de Redux devtools extension. Waar voor de Redux devtools extension een extra parameter wordt toegevoegd zal voor de middleware ook een parameter worden toegevoegd die meerdere middleware kan bevatten. (Redux, 2018)

Om een betere verstandhouding van Redux te krijgen werd ook naar de voorgangers van Redux gekeken, waaronder Flux. Hieronder worden de verschillen en de gelijkenissen tussen Flux en Redux overlopen. Het eerste verschil is dat Flux een patroon is en Redux een library. Flux is een mooiere naam voor het observer patroon die wat aangepast is voor React. Facebook heeft echter ook een aantal tools ontwikkeld die helpen in de implementatie van het Flux patroon.

Flux en Redux hebben beide actions. Deze Actions kunnen vergeleken worden met events. In Flux is een action een simpel JavaScript object en dat is in het standaard geval ook zo in Redux. Wanneer er gebruik gemaakt wordt van Redux middleware kan het echter zo zijn dat actions ook functies en promises kunnen zijn.

Met Flux is het de conventie dat er meerdere stores per applicatie zijn waarbij elke store een singleton object is. In Redux is het de conventie dat er maar één store per applicatie is.

Flux heeft een enkele dispatcher en alle actions moeten door deze dispatcher gaan, dit is eveneens een singleton object. Een Flux applicatie kan geen meerdere dispatchers hebben. Dit is nodig omdat een Flux applicatie meerdere stores kan hebben en de afhankelijkheden tussen deze stores hebben een manager nodig, de dispatcher. In Redux is er geen dispatcher entiteit. In plaats daarvan heeft de store een dispatch proces van zichzelf. Een Redux store heeft een aantal simpele API functies, één daarvan is het dispatchen van actions.

De logica van wat er moet gebeuren wanneer er een action ontvangen wordt zit in de store zelf bij Flux. De store heeft ook de flexibiliteit om te kiezen welke stukken data deze publiekelijk toegankelijk wil maken. Het slimste object in Flux is de store. In Redux zit de logica dan in de reducer. Een store kan niet gedefinieerd worden zonder reducer functie. Hier is het slimste object de reducer. Redux gaat ook gewoon alle data beschikbaar maken die geretourneerd wordt van de reducer.

Nog een groot verschil is dat de state immutable is in Redux. Dit wordt makkelijk bekomen door van de reducers pure functies te maken. In Flux is hier geen restrictie, de state kan gemuteerd worden zoals gewenst. (Buna, 2017)

1.4 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.

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

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. (npm, g.d.)

1.5 Webpack

De meeste React-Redux projecten maken gebruik van Webpack. Webpack heeft veel populariteit gewonnen door het gebruik in React. Webpack is een statische bundelaar van modules voor JavaScript applicaties. Het kleinste project die gebundeld kan worden met webpack bestaat uit een input en een output. Wanneer webpack bezig is met het verwerken van de applicatie, maakt het intern een *dependency graph* aan. Deze wordt geconstrueerd door alle imports af te lopen. Het bundelen van modules begint bij de door gebruiker gedefinieerde entries.

Een entry point gaat aanduiden welke module webpack het eerst moet beginnen builden vanuit zijn interne dependency graph. Webpack gaat dan kijken van welke andere modules en libraries dit entry point afhankelijk is (direct en indirect).

Standaard is dit entry point de index file, maar deze kan uiteraard veranderd worden of er kunnen zelfs meerdere entry points toegevoegd worden. Samen met de entry point(s) wordt meestal ook de output property gedefinieerd. Via deze property weet webpack naar waar de bundels verzonden moeten worden en hoe deze moeten noemen. De standaardsetting hiervoor is de dist folder.

In de meeste applicaties zitten er echter niet enkel JavaScript files. Soms worden image imports gedaan of wordt er via een url gewerkt. Webpack verstaat echter enkel JavaScript files. Om het mogelijk te maken om andere bestandtypes te converteren in geldige modules moeten loaders toegevoegd worden. Deze modules kunnen dan gebruikt worden door de applicatie en toegevoegd worden aan de dependency graph.

Een loader heeft een *test* en een *use* property. De test property geeft weer welke file of files die er getransformeerd moeten worden. Een use property toont aan welke loader gebruikt moet worden voor deze transformatie.

Loaders worden gedefinieerd in de *rules* property. Waar loaders worden gebruikt voor de transformatie van bepaalde modules, zorgen plugins ervoor dat de bundle geoptimaliseerd wordt. Plugins gaan altijd vooraf aan een require statement en kunnen later aangepast worden door extra opties toe te voegen.

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 Functie 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* (figuur 5) 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 Eigen methode

Als tweede methode werd gezocht naar een alternatief. 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. Op deze manier kan ook een lineaire structuur verkregen worden. Momenteel is het niet mogelijk om van reeds gecombineerde root reducers een nieuwe reducer te combineren met behoud van functionaliteit van beide root reducers en een lineaire structuur in plaats van een geneste structuur.

Analyse

Om een extensie te schrijven is eerst een duidelijk inzicht nodig in de methode combineReducers. Hiervoor werd dus eerst een analyse gedaan van deze methode.

De methode neemt reducers als parameter. Er worden eerst twee lokale variabelen gemaakt, de reducerKeys met daarin de keys van de reducers die als parameter werden meegegeven en de finalReducers. Daarna wordt -zolang er reducerKeys zijn- de juiste reducer opgehaald aan de hand van zijn key en wordt er gekeken als het type een functie is of niet. Als het inderdaad een functie is dan zal deze worden opgeslaan in de finalReducers.

Nu kan er een nieuwe variabele finalReducerKeys aangemaakt worden met daarin de keys van de finalReducers. Daarna wordt er een functie geretourneerd met dezelfde syntax als de reducer (state en action). Er wordt opnieuw per key overlopen, deze keer om een object te verkrijgen via *reduce* aan de hand van de state voor deze key.

Werkwijze

Bij een reducer die voor de eerste keer gecombineerd wordt, zal door de functie *object.keys* elke key aangesproken worden. Hierdoor zal elke reducer opgenomen worden in de combinatie. Wanneer een combinatie tracht gemaakt te worden van een reeds gecombineerde reducer, dan zal door de functie *object.keys* enkel de gecombineerde key aanspreken en dus niet alle onderliggende reducers. Er werd getracht om van een root reducer, die reeds gecombineerd is, de reducer keys aan te spreken van zijn onderliggende reducers. Omdat deze reducers al eens gecombineerd zijn, zitten deze onderliggende reducers in de scope van de functie combineReducers. Daardoor is het niet mogelijk om van een gecombineerde reducers opnieuw zijn reducers aan te spreken, deze kunnen niet worden aangesproken. Indien dit wel mogelijk zou zijn, kon op deze manier een lineaire combinatie gemaakt worden van meerdere root reducers.

Omdat dit niet mogelijk is, wordt de functionaliteit van onderliggende reducers niet meegenomen. Dit zorgt er dan weer voor dat deze reducers niet opgenomen worden in de state. Vanaf het moment dat een reducer nodig zou zijn voor functionaliteit af te handelen, zal de app crashen omdat het een undefined reducer zal oproepen.

Om hier een oplossing voor te bieden werd gekeken naar een alternatief voor de methode combineReducers. Er 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 te exporteren. Om deze methode mogelijk te maken moet de root reducer van het hoofdproject ook een object zijn en geen gecombineerde functie. Een voorbeeld kan gevonden worden in figuur 6. 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.

Dit derde nadeel kan echter wel opgelost worden. Via webpack is het mogelijk om de configuraties van Scratch te bundelen en weg te schrijven in een dist map om zo te exporteren als een library. In dit geval gaat het pas gebeuren wanneer de environment variabele op 'production' staat. Een tweede package op npm met de nodige configuraties voor beide projecten wordt overbodig wanneer deze oplossing gebruikt wordt in de package waar de reducers geëxporteerd worden. Dit bespaart onderhoudswerk en vermindert de complexiteit.

3. Conclusie

3.1 Inleiding

Redux is een framework dat nog maar 3 jaar bestaat. Er is wel al veel documentatie beschikbaar en veel problemen zijn reeds ontdekt en opgelost, maar zeker niet alles staat beschreven.

Bijvoorbeeld hoe 2 React-Redux projecten met elkaar kunnen interageren staat nergens beschreven. Omdat hier geen best practices vermeld werden, is de keuze gemaakt om via npm packages te werken. Op deze manier kan het ene project snel geïntegreerd worden in het andere.

Het was echter niet voldoende om deze package gewoon te installeren. Wanneer dan een component werd aangeropen zou de state niet volledig zijn. De state bevat dan enkel objecten van het hoofdproject en niet van het geïntegreerde project. Om deze reden moeten de reducers geëxporteerd worden uit het geïntegreerde project.

3.2 Performantie

Zoals eerder vermeld was de oplossing om deze reducers in een object te steken en dit te exporteren. Het grootste probleem hierbij is dat veranderingen in het project dat geïntegreerd moet worden grote gevolgen kan hebben. Dit werd ondervonden door het open source project van Scratch. Na een update van Scratch uit en de bijhorende update om de package up to date te houden werkte deze niet meer. Dit kwam door een verandering in de structuur van de reducers, waardoor het niet meer de juiste reducers waren die

geëxporteerd werden. Dit zorgt er dus voor dat er zeer aandachtig moet gebleven worden bij updates. Om deze reden zou er nood zijn aan een methode uncombineReducers of een alternatief waardoor er enkel met de root reducer rekening zou gehouden moeten worden. Dan zouden externe updates geen grote impact meer hebben en kan deze snel terug geïntegreerd worden.

Deze methode is niet optimaal, aangezien er heel wat onderhoudswerk bij komt kijken om de package up to date te houden. Een bijkomend probleem is dat met deze methode ook alle dependencies moeten overgenomen worden alsook de webpack configuraties.

Deze methode wordt al een heel stuk performanter door het te integreren project te combineren in webpack, op deze manier is het niet meer nodig om alle dependencies toe te voegen aan het hoofdproject en wordt de package voor de webpack configuraties overbodig.

Zoals eerder gezegd is deze methode afhankelijk van de veranderingen in het te integreren project. Als hier quasi geen veranderingen in moeten gebeuren zal dit geen grote impact hebben, als er constant veranderingen in gebeuren dan zal deze methode niet meer optimaal zijn.

3.3 Complexiteit

Er zijn een aantal objectieve maatstaven die ontwikkeld zijn voor de complexiteit van software te meten. In grote lijnen kan men deze maatstaven verdelen in maten voor complexiteit die zijn gerelateerd aan de omvang van een software-programma, en in maten voor complexiteit die zijn gerelateerd aan de structuur van zo'n programma. De simpelste maat voor de omvang is het aantal regels programmacode: hoe meer regels, hoe groter de complexiteit. Een ernstig nadeel van deze simpele metriek is dat men het verschil in uitdrukkingskracht van de verschillende programmeertalen volledig verwaarloost.

Men maakt ook geen onderscheid naar soorten regels code. Er zijn echter eenvoudige regels code en er zijn complex samengestelde regels programmacode. Om hieraan tegemoet te komen, heeft Halstead een meer verfijnde metriek ontwikkeld. In zijn theorie is de complexiteit en de daarmee samenhangende programmeerinspanning afhankelijk van de omvang van een programma en van het niveau van structurering. Hoe groter de omvang, hoe hoger de programmeerinspanning. hoe hoger (compacter) het niveau van structurering, hoe lager de inspanning. Zowel omvang als niveau van structurering zijn functies van het aantal verschillende operatoren en operanden in een programma. (Bemelmans & Heemstra, g.d.)

Men kan nu deze metriek toepassen op het voorbeeld met de gemaakte package in deze studie. n1 staat voor het aantal verschillende operatoren, n2 staat voor het verschillende aantal operanden, N1 is het aantal operatoren en N2 is het aantal operanden. Met deze nummers kunnen een aantal metingen berekend worden:

- Vocabulair van het programma: n = $n_1 + n_2 = 3 + 23 = 27$.

- Lengte van het programma: N = N_1 + N_2 = 27 + 43 = 70.
- Berekende lengte van het programma: $N_3 = n_1\ log_2(n_1) + n_2\ log_2(n_2)$ = 4,7549 + 104,0419 = 108,7968.
- Volume: V = N * $log_2(n)$ = 70 * 4,7549 = 332,8422.
- Moeilijkheid: D = (n_1/2) * (N_2/n_2) = 1,5 * 1,8696 = 2,8043.
- Inspanning: E = D * V = 2,8043 * 332,8422 = 933,3894.

De meeteenheid inspanning kan vertaald worden naar codeertijd aan de hand van de volgende relatie: de tijd T = E/18 = 51,8550 seconden. Halstead kan ook een schatting geven van het aantal errors die in de implementatie zitten door naar het aantal geleverde bugs te kijken: B = $E^{2/3}$/3000 = 0,1245.

Deze metriek werd toegepast op het exporteren van de reducers in de gepubliceerde package en de code die nodig is om alle reducers opnieuw te importeren in een hoofdproject. Er kan geconcludeerd worden dat het een snelle codeertijd heeft dat gepaard gaat met een laag aantal geleverde bugs.

Een andere manier om de complexiteit van software te meten is via de cyclomatische complexiteit. Deze metriek is een quantitatieve meting van het aantal lineair onafhankelijke paden in de code en werd ontwikkeld door McCabe. De cyclomatische complexiteit wordt gedefinieerd als M = E - N + 2P.
Waarbij:

- E = het aantal randen, verbindingen in een diagram
- N = het aantal knooppunten in een diagram
- P = het aantal geconnecteerde componenten

(McCabe, 2018)

In dit geval gaat het om een lineaire structuur zoals gezien kan worden in de figuur. De geëxporteerde package bevat 23 verbindingen, 24 knooppunten en 1 geconnecteerde component. Hiermee kan de cyclomatische complexiteit bepaald worden door M = 23 - 24 + 2 = 1. Volgende figuur verkregen uit „Cyclomatic complexity" (g.d.) geeft een overzicht weer van de complexiteit waarde en zijn overeenkomstige betekenis.

Hieruit kan afgelezen worden dat een nummer hoger dan 40 niet getest kan worden en een hoge kost en inspanning heeft. Een waarde tussen de 20 en 40 wordt aanzien als heel complexe code met lage test mogelijkheden en eveneens een hoge kost en inspanning. Een complexiteitswaarde tussen de 10 en 20 duidt op complexe code met gemiddelde testbaarheid, kost en inspanning. Een waarde tussen de één en 10 betekent dat de code goed gestructureerd en geschreven is, met een hoge testbaarheid en een lage kost en inspanning. Aangezien de berekende cyclomatische complexiteit 1 bedraagt, kan de conclusie getrokken worden dat de code gestructureerd is, een lage kost en inspanning heeft en makkelijk getest kan worden.

Een derde manier om de complexiteit van een programma te meten is aan de hand van een maintainability index. De maintainability index is een metriek die meet hoe onderhoudbaar de source code is. Deze index wordt berekend door een aantal factoren, de formule maakt

gebruik van lines of code, cyclomatic complexity en Halstead volume. Deze metriek wordt onderandere gebruikt in verschillende geautomatiseerd software tools in Visual Studio, daar gebruikt het een schaal van 0 tot 100. Om deze index te meten wordt gebruik gemaakt van de formule:
MI = 171 - 5,2 * ln(V) - 0,23 * (G) - 16,2 * ln(LOC)
Hierbij horend:

- V = Halstead volume
- G = Cyclomatische complexiteit
- LOC = het aantal lines of code

(„Maintainability index”, g.d.)

Wanneer deze formule toegepast wordt op het voorbeeld van de npm package met de geëxporteerde reducers en de code die nodig is in het hoofdproject om deze reducers opnieuw te importeren, dan krijgen we:
MI = 171 - 5,2 * ln(332,8422) - 0,23*1 - 16,2*ln(43) = 171 - 30,1999 - 0,23 - 60.9314 = 79,6387.
Volgens zainnab (2011) heeft een programma met hoge onderhoudbaarheid een waarde tussen de 20 en 100. Een waarde tussen 10 en 19 wil zeggen dat de code een gemiddelde onderhoudbaarheid heeft en een waarde tussen de 0 en 9 duidt op een slechte, lage onderhoudbaarheid van het programma. In dit voorbeeld is de maintainability index 79,6387, dit wil zeggen dat er op basis van deze index kan afgeleid worden dat het programma een hoge, goeie onderhoudbaarheid heeft. Men kan echter geen conclusie maken op basis van de maintainability index alleen. Dit komt omdat deze index een bewerking uitvoert met 3 metrieken. Onder deze metrieken bevinden zich de lines of code en de cyclomatische complexiteit. Om deze goeie onderhoudbaarheid te testen kan er gekeken worden naar de andere metingen die zijn uitgevoerd. Halstead had een goeie waarde met een lage codeertijd en een laag aantal geleverde bugs. De cyclomatic complexity had uiteraard ook een goeie waarde aangezien het gaat om een volledig lineaire structuur (M = 1). Deze combinatie van metrieken zorgt ervoor dat er finaal geconcludeerd kan worden dat het programma een hoge onderhoudbaarheid heeft met een lage complexiteit.

3.4 Eindconclusie

Dit onderzoek is nog niet klaar, de basis is wel gelegd om verder onderzoek te doen naar een methode om de root reducers van beide projecten met elkaar te combineren zonder dat daar veel extra onderhoud aan te pas komt. Als uitkomst was wel verwacht dat er een soort uncombineReducers zou gemaakt worden, helaas was dit niet mogelijk door de implementatie van deze functie. Er is wel een alternatief gevonden om toch het probleem op te lossen.

Achteraf gezien bleek het een goede keuze om voor het Scratch open source project te kiezen. Dit is een project dat nog steeds in development is en dat nu overschakelt op JavaScript, waardoor het zeer interessant wordt om dit te kunnen integreren in een eigen

applicatie.

Dit is een belangrijk onderzoek omdat er nog geen soortgelijke onderzoeken zijn gedaan of gedocumenteerd werden. Er zijn een aantal statistieken gevonden die duidelijk maken dat het een probleemstelling is die uitnodigt tot verder onderzoek. De npm package die de reducer exports verzorgde is opgericht eind februari. Dan werd gezocht naar hoe het mogelijk wordt om dit project snel integreerbaar te maken, waar tot een tijdelijke oplossing is gekomen begin maart. Deze package werd sinds dan ruim 7000 keer gedownload (figuur 9) met piekmomenten van 2168 downloads per week (figuur 8) en 946 downloads per dag (figuur 7) in maart. Door tijdsgebrek was het echter onmogelijk om deze package constant up to date te houden. Eind maart kan er nog een update gezien worden door het stijgende aantal downloads, wat duidt op het feit dat er wel vraag naar is.

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 Dan" 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. [Dan]. (2015). Verkregen van https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

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

Bemelmans, T. & Heemstra, F. (g.d.). *Complexiteit en beheersbaarheid van software* (proefschrift, Technische Universiteit Eindhoven).

Benedetto, S. D. (2016). React + Redux container Pattern. Verkregen van https://www.thegreatcodeadventure.com/the-react-plus-redux-container-pattern/

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

bloodyowl. (2015). The Provider and Higher-Order Component patterns with React. Verkregen van https://medium.com/@bloodyowl/the-provider-and-higher-order-component-patterns-with-react-d16ab2d1636

Buna, S. (2017). The difference between Flux and Redux. Verkregen van https://edgecoders.com/the-difference-between-flux-and-redux-71d31b118c1?gi=5b20b1937b84

Container Components. (2015). Verkregen van https://medium.com/@learnreact/container-components-c0e67432e005

Cyclomatic complexity. (g.d.). Verkregen van https://www.guru99.com/cyclomatic-complexity.html

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/

Haldar, M. (2017). reactjs component lifecycle methods. Verkregen van https://hackernoon.com/reactjs-component-lifecycle-methods-a-deep-dive-38275d9d13c0

Holtkamp, J. (2016, oktober 16). Verkregen van https://medium.com/@holtkam2/react-redux-understanding-components-containers-actions-and-reducers-a2f9287bfb92

Lerner, A. (g.d.-a). Redux actions. Verkregen van https://www.fullstackreact.com/30-days-of-react/day-20/

Lerner, A. (g.d.-b). What is JSX? Verkregen van https://www.fullstackreact.com/30-days-of-react/day-2/

Maintainability index. (g.d.). Verkregen van http://www.projectcodemeter.com/cost_estimation/help/GL_maintainability.htm

McCabe, T. J. (2018). Cyclomatic complexity. Verkregen van https://en.wikipedia.org/wiki/Cyclomatic_complexity

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

npm. (g.d.). npm. Verkregen van https://docs.npmjs.com/getting-started/what-is-npm

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/

Sharif, A. (2015). Understanding the React Component Lifecycle. Verkregen van http://busypeoples.github.io/post/react-component-lifecycle/

zainnab. (2011). Code Metrics - Maintainability Index. Verkregen van https://blogs.msdn.microsoft.com/zainnab/2011/05/26/code-metrics-maintainability-index/

Zalmoxisus. (2015). Verkregen van https://github.com/reduxjs/redux-devtools