jamk.fi

Keskitetty tilanhallinta NgRx-kirjastolla

Wikisivuston luominen kirjaston käytöstä Enterprise-tason Angular-sovelluskehityk-sessä

Antti Maaheimo

Opinnäytetyö
Marraskuu 2020
Tekniikan ala
Insinööri (AMK), tieto- ja viestintätekniikan tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä	Julkaisun laji	Päivämäärä
Maaheimo, Antti	Opinnäytetyö, AMK	Marraskuu 2020
	Sivumäärä 53	Julkaisun kieli Suomi
	-	Verkkojulkaisulupa myönnetty: x

Työn nimi

Keskitetty tilanhallinta NgRx-kirjastolla

Wikisivuston luominen kirjaston käytöstä Enterprise-tason Angular-sovelluskehityksessä

Tutkinto-ohjelma

Insinööri (AMK), tieto- ja viestintätekniikan tutkinto-ohjelma

Työn ohjaaja

Ari Rantala

Toimeksiantaja

Landis+Gyr Oy

Tiivistelmä

Landis+Gyr Oy on energia-alan ratkaisuihin keskittynyt yritys. Yritys pohjaa uudet web-pohjaiset käyttöliittymänsä Angular-web-sovelluskehykseen, jonka tilanhallintaratkaisuksi on valittu NgRx-kirjasto. Kirjasto on hyvä esimerkki yksisuuntaisen datavirran tilanhallintaratkaisusta, joka pohjaa tilanhallintansa koko sovelluksen tilan säilömisestä yhteen paikkaan.

NgRx on erinomainen tilanhallintaratkaisu Landis+Gyrin tarpeisiin, mutta sen käytön opettelu ja käsitteiden ymmärtäminen on sovelluskehittäjille haastavaa ja aikaa vievää. NgRx-kirjaston käytön oppimisen haastavuus loi tarpeen yhtiön sisäiselle kattavalle dokumentaatiolle kirjaston käytöstä yhtiön sovelluskehityksessä.

Landis+Gyrin tarpeisiin luotiin kattava, oppimisprosessia tukeva wikisivusto yhtiön sisäiseen wikiin. Wikisivusto sisältää kaiken tarvittavan informaation kirjaston käytöstä ja toiminnasta. Se tukee sovellusten kanssa aloittavia sovelluskehittäjiä ja tarjoaa käytännön esimerkkejä Landis+Gyrin omista sovelluksista. Wikisivustossa konkretisoidut käytänteet luovat perustan tehokkaiden tilanhallintaratkaisuiden kehittämiseen niin uusille kuin myös kirjaston kanssa jo työskenteleville kehittäjille.

Wikisivustoa tukemaan on luotu esimerkkisovellus antaman yksinkertaisen ja helposti lähestyttävän esimerkin NgRx-kirjaston toiminnasta. Se tukee wikisivuston rinnalla kirjaston käytön opettelua ja konkretisoi usein piilotettua logiikkaa käyttöliittymässään.

Luodut oppimistyökalut mahdollistavat NgRx-kirjaston oppimisen juuri Landis+Gyrin tarpeiden ja käytäntöjen kautta. Ne antavat konkreettisen pohjan sovelluskehitykselle keräten kaiken tarvittavan informaation yhteen keskitettyyn paikkaan.

Avainsanat

NgRx, tilanhallinta, Angular, actions, reducers, selectors, store, effects

Muut tiedot

jamk.fi

Description

Author	Type of publication	Date
Maaheimo Antti	Bachelor's thesis	November 2020
		Language of publication: Finnish
	Number of pages 53	Permission for web publication: x

Title of publication

Centralized state management with NgRx library

Creating a wiki page on the use of the library in Enterprise level software developing

Degree programme

Engineer (university of applied sciences), Information and Communication Technology

Supervisor

Ari Rantala

Assigned by

Landis+Gyr Ltd

Abstract

Landis+Gyr Ltd is a company specialized in providing energy management solutions to electricity companies. The companies new web-based user interfaces are made with Angular framework. They use NgRx, a state management library to handle the state of the new applications. NgRx is a good example of state management solution that uses unidirectional data flow. NgRx stores the applications state in one place, creating one and only source of truth for the whole application.

NgRx is an excellent state management solution but it has its setbacks. NgRx is very time consuming and hard to learn as it introduces a new way of thinking the state of an application. This raises a demand for a throughout documentation to specify its use and development principals inside the company.

Wiki pages were created to help new developers in the learning process of the library. This wiki page includes all the required information needed for getting a good understanding on the library and how it has been implemented in Landis+Gyr's development processes. The wiki pages run the reader trough the basics of the library with practical examples from Landis+Gyr's application. It creates a uniform place for new and old developers to get a good understanding on the unique ways that the library is used in Landis+Gyr's applications.

An example application that has been created to help with the understanding of the library gives the developers simple hands-on examples on the workings of the store. It gives a good peek under the hood in its user interface on how the store works and stores the data that is handled throughout the applications lifecycles.

Keywords/tags

NgRx, state management, Angular, actions, reducers, selectors, store, effects

Miscellaneous

Sisältö

1 Johdanto 3

1.1 Toimeksiantaja 3

1.2 Toimeksianto 3

1.3 Vaatimusmäärittely 4

1.4 Tilanhallinta 4

2 Suunnittelu ja teknologiavalinnat 5

2.1 Yleistä 5

2.2 Sovelluskehys 5

2.3 Model to view 6

2.4 Flux ja Redux 8

2.5 Tilanhallinta Angularilla 9

3 NgRx 10

3.1 Yleistä 10

3.2 NgRx Store-rakenne 12

3.3 Käyttöönotto ja kansiorakenne 26

3.4 Työkalut 31

3.5 NgRx-käytänteet 33

4 Työn toteutus 35

4.1 Tavoite 35

4.2 Wikisivuston kirjoitusprosessi ja sisältö 35

4.3 Esimerkkisovelluksen rakentaminen 38

5 Työn tulokset 42

5.1 Työn tavoite 42

5.2 Wikisivusto 42

5.3 Esimerkkisovellus 47
5.4 NgRx datavirta animaatio 48

6 Johtopäätökset ja pohdinta 49
6.1 Tavoitteet ja johtopäätökset 49
6.2 Lähteiden käyttö 49
6.3 Sisällön kriittinen pohdinta 50
6.4 Jatkokehitys 51

Lähteet 52

1 Johdanto

1.1 Toimeksiantaja

Opinnäytetyön toimeksiantaja on Landis+Gyr Oy. Landis+Gyr on kansainvälinen energia-alalla käytettävien mittareiden, järjestelmien sekä palveluiden monipuoliseen tuottamiseen sekä ylläpitämiseen keskittynyt yritys. Suomessa sen toiminta on keskittynyt Jyväskylään.

AIM-järjestelmä on Landis+Gyrin tuote, jonka kehitystiimille opinnäytetyö tehtiin. AIM-järjestelmä koostuu monesta tuotteesta, jotka vastaavat moniin energiayhtiöiden tarpeisiin. AIM-järjestelmä auttaa energiayhtiöitä mittausdatan ja mittaustapahtumien tehokkaassa keräämisessä ja käsittelyssä. Sen ratkaisut mahdollistavat joustavia prosessointi- ja välitysratkaisuja datan siirtoon eri tahojen välillä. Järjestelmä pyörittää maailmanlaajuisesti nykyään yli kahta miljoonaa mittapistettä.

1.2 Toimeksianto

Tuki Adobe Flash -teknologiaan on loppumassa vuoden 2020 joulukuussa (Update on Adobe Flash Player End of Support 2017). Samaa teknologiaa käyttää myös Landis+Gyrin AIM-järjestelmän Dashboard-käyttöliittymä, joka näin tulee myös tiensä päähän. Uuden käyttöliittymän tulisi myös vastata alan kasvavaan paineeseen luoda uusi käyttöliittymä web-sovelluksena. Käyttöliittymä päätettiin uusia kokonaan käyttäen tekniikkana Angular web -sovelluskehystä. Angular soveltuu erinomaisesti Enterprise-tason sovelluskehitykseen ja sovelluskehyksistä parhaiten Landis+Gyrin tarkoituksiin. Angularista puuttuu kuitenkin keskitetty tilanhallinta, joka tämän kokoluokan sovelluksessa todettiin tarpeelliseksi. Tilanhallinta päätettiin toteuttaa käyttäen

NgRx-kirjastoa, joka tuo Angular-sovelluksiin keskitetyn tilanhallinnan. Keskitetty tilanhallinta selkeyttää suurten ohjelmistojen datanhallinnointia huomattavasti. Vaikka keskitetyt tilanhallintaratkaisut tuovat mukanaan valtavia hyötyjä sovelluskehitykseen, niiden ymmärryksen oppimiskynnykset ovat korkeita. Tämän opinnäytetyön tavoitteena on luoda kattava ja käytännöllinen ohje tilanhallintakirjaston kanssa aloittaville kehittäjille Landis+Gyrin sisäiseen wikiin.

1.3 Vaatimusmäärittely

Wikisivuston tarkoitus on lyhentää kehittäjän kirjaston käytön oppimiseen käyttämää aikaa ja helpottaa kirjaston käyttöä sovelluskehityksessä käymällä läpi pääasialliset osat kirjaston toiminnasta ja rakenteesta. Wikin luettuaan kehittäjällä pitäisi olla perustavan laatuinen käsitys kirjaston käyttämisestä ja ymmärrys siitä, miksi sitä lähtökohtaisesti käytetään. Wikisivuston tulisi tukea Landis+Gyrin sovelluskehittäjiä NgRx-kirjaston oppimisprosessissa ja tuoda yhteen kaikki yhtiön sovelluskehityksessä käyttämät tilanhallintakäytänteet.

1.4 Tilanhallinta

Sovelluksen tilalla voidaan viitata kaikkeen siihen informaatioon, mitä applikaatio tietää käyttäjästä, informaatioon siitä, mitä käyttäjä on tehnyt sivulle siirryttyään, ja muuhun globaaliin applikaation toiminnan kannalta tärkeään muuttuvaan informaatioon (Duffy. 2004). Tilaa voisi ajatella siis kaikkien interaktioiden tuloksena, jotka käyttäjä on tehnyt.

Koska tila käsitteenä voi käsittää lähes kaiken applikaation sisältämän muuttuvan informaation, sen sisäistäminen voi yhtenä isona kokonaisuutena olla haastavaa. Sovelluksen tila on kuitenkin aina olemassa, hallitaan sitä tai ei.

Viime aikoina yksisivuisten applikaatioiden noustua lähes web-kehityksen normiksi web-applikaation tila siirtyi kokonaisuudessaan palvelimen sijasta asiakkaan puolelle. Sovelluksen tila selaimessa on helposti muutettavissa mistä vain applikaation osasta käsin. Koodipohjan ja kehittäjätiimin kasvaessa ympäri applikaatiota hajautettu, alati muuttuva tila tuo mukanaan eksponentiaalisesti kasvavia ongelmia. Rajoittamaton ja nopea kyky manipuloida sovelluksen tilaa mistä vain sovelluksen osasta voi hyvinkin nopeasti lähteä kehittäjien käsistä. Yksi tärkeimmistä ylläpidettävän koodin tukipilareista onkin applikaation tilan tehokas hallinta. (Angular Application Architecture - Building Flux Apps with Redux and Immutable.js. 2020.)

2 Suunnittelu ja teknologiavalinnat

2.1 Yleistä

Sovelluskehyksiä ja tilanhallintaratkaisuja on monia. Tämän luvun tarkoituksena on esitellä erilaisia vaihtoehtoja, miksi niitä käytetään ja miksi Landis+Gyr on päätynyt Angularin ja NgRx:n valintaan.

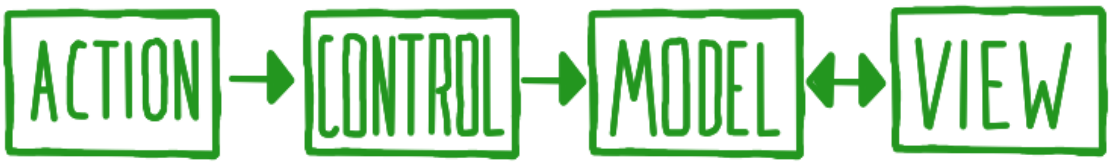
2.2 Sovelluskehys

Kolme suurinta varteenotettavaa sovelluskehysratkaisua ovat Facebookin kehittämä React, Googlen ylläpitämä Angular ja Vue.js. Jokainen näistä sovelluskehyksistä tuo mukanaan omia hyötyjänsä, mutta suurimmaksi osaksi ne perustuvat samoihin periaatteisiin. Kaikki niistä ovat erinomaisia kehikkoja luomaan asiakaspuolen komponenttipohjaisia sovelluksia (Freeman 2020, Comparing Angular to React and Vue.js). Landis+Gyrin sovelluksien sekä myös AIMUIn sovelluskehykseksi on kuitenkin valikoitunut Vue.js:n ja Reactin sijasta Angular. Valinta on tehty yrityksen arkkitehtien ja kehittäjien päätöksestä ja Angularin on todettu vastaavan Landis+Gyrin sovelluskehitystarpeita parhaiten.

Suurimpana erona muihin mainittuihin sovelluskehyksiin Angular käyttää ohjelmointikielenään TypeSriptiä. Poiketen myös Vue.js:stä ja Reactista Angular pyrkii erottamaan komponenttien HTML-, CSS- ja TypeScript-osiot erillisiin tiedostoihin. Vue.js ja React pyrkivät pitämään osat yhden tiedoston alla. Sovelluskehyksissä suurin ero onkin juuri kehityskokemus (Freeman 2020, Comparing Angular to React and Vue.js). Valinta on tehty yrityksen arkkitehtien ja kehittäjien päätöksestä ja Angularin rakenteen ja toimintaperiaatteiden on todettu vastaavan Landis+Gyrin sovelluskehitystarpeisiin parhaiten.

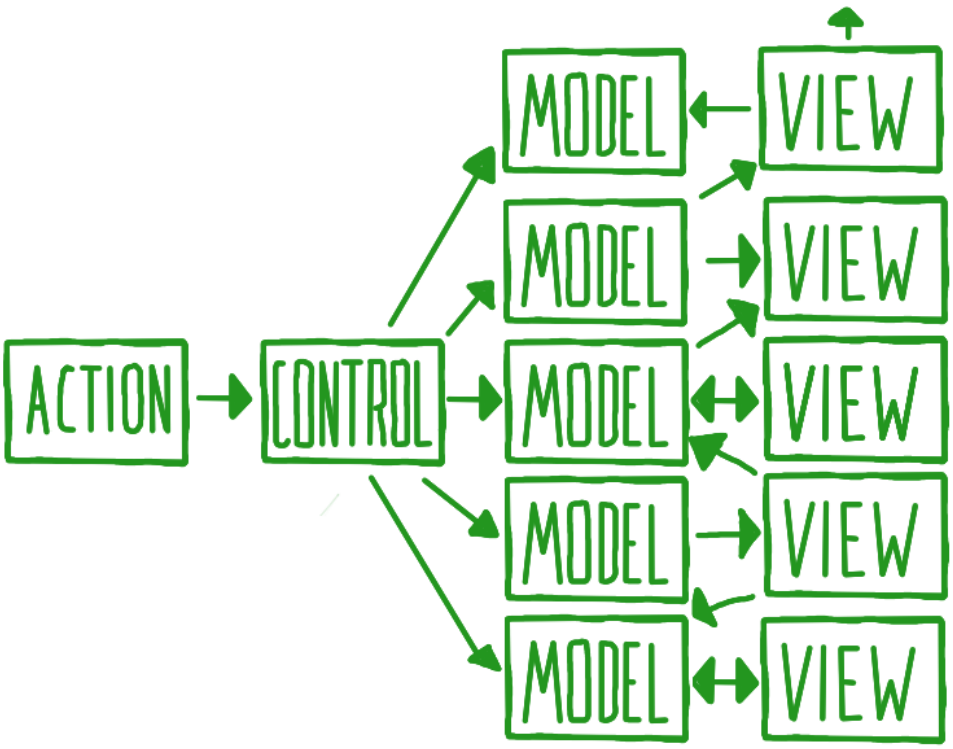
2.3 Model to view

Tyypillinen tilanhallintatoteutus ennen nykyisen mallisia ratkaisuja oli model to view-sovelluskehysmalli. Model to view on tyypillinen esimerkki monisuuntaisesta datavirrasta. Kontrolleri hallitsee datamalleja (models), jotka puolestaan heijastavat tilaansa käyttöliittymän näkymiin (view). (Gackenheimer 2015, Chapter 5 - Introducing Flux: An Application Architecture for React, Introduction to React.). Oheisessa kuviossa on esitetty yksinkertainen model to view-malli.



Kuvio 1. Tekijän laatima Model to view -mallia kuvaava kaavio.

Mallit pitävät hallussaan tilaobjekteja. Näiden tilaobjektien arvot on monissa sovelluskehysratkaisuissa sidottu näkymien arvoihin. Käyttöliittymän näkymissä käyttäjän tekemät muutokset heijastuvat näin myös takaisin malleihin. Malleihin tehdyt muutokset luonnollisesti näkyvät niiden hallitsemissa käyttöliittymän näkymissä. Muutokset näkymissä voivat vaatia myös muutoksia moniin malleihin, jotka saattavat sisältää toisistaan riippuvaisia arvoja. Yksinkertaisessa sovelluksessa model to view -toteutus voi toimia ja kaikkien sovelluksen osien seuraaminen voi olla tehokasta ja selkeää. Käyttöliittymän kasvaessa mallien ja niiden hallitsemien näkymien määrä saattaa kuitenkin kasvaa siihen pisteeseen, että niiden tehokas hallinnointi lähtee kehittäjien käsistä. Kun kerroksia lisätään tarpeeksi ja yhä useammat mallit jakavat arvoja toistensa kesken, on sovelluksen tilan seuraaminen ja ymmärtäminen vaikeaa (ks. kuvio 2).



Kuvio 2. Tekijän laatima Model to view -malli kaavio. Esimerkki kompleksisemmasta toteutuksesta monella mallilla ja näkymällä.

Mitä suurempi osa malleista on riippuvaisia toisistaan, sitä suuremmalla todennäköisyydellä model to view -toteutus voi aiheuttaa ongelmia. Pienetkin muutokset voivat tuoda mukanaan liudan arvaamattomia sivuefektejä. Pitkällä tähtäimellä tämä ei ole ylläpidon kannalta kannattavaa.

2.4 Flux ja Redux

Model to view -tilanhallintaratkaisun tuomat ongelmat inspiroivat Facebookin kehittäjiä luomaan omalle sovelluskehykselleen Reactille uudenlaisen tilanhallintamallin, Fluxin.

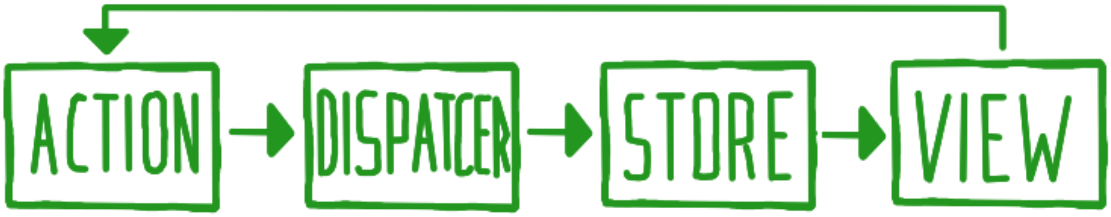
Flux tuo mukanaan uudenlaisen tavan ajatella sovelluksen tilanhallintaa. Monet nykyaikaisista tilanhallintaratkaisuista pohjautuvatkin Fluxin luomaan yksisuuntaiseen datavirtamalliin. Yksisuuntainen datavirta onkin sovellusarkkitehtuurisesti merkittävä muutos aiempiin yleisesti käytettyihin malleihin.

Flux-toteutuksessa koko sovelluksen tila on keskitetty yhdelle entiteetille, storelle. Store päivittää käyttöliittymän näkymille jakamansa tilan aina sen muuttuessa. Käyttöliittymässä tapahtuvien tapahtumien virtaa hallinnoi dispatcher. Jokainen tapahtuma käsitellään dispatcerissä. Dispatcher pitää huolencsitä, että jokainen tapahtuma tekee muutoksensa storeen vasta aiemman tapahtuman muutosten jälkeen. Käytännössä store käsittelee tapahtumia yksi kerrallaan. (Chapter 5 - Introducing Flux: An Application Architecture for React, Introduction to React.)

Flux-datavirtamalli rakentuu seuraavista osista (ks. kuvio 3):

- Actionit aloittavat tilanmuutoksen siirtämällä muutettavan datan eteenpäin
- Dispatcherit hallitsevat actionien virtaa. Dispatcher pitää huolen siitä, että vain yksi action kerrallaan voi tehdä muutoksia storen tilaan.
- Store ottaa vastaan ja säilöö actionien tuomaa dataa.

- Controller View:t kuuntelevat storen muutoksia ja lähettävät uusimmat datan instanssit komponenteille storen tilan muuttuessa.



Kuvio 3. Flux-datavirtamallia kuvaava kaavio.

Facebookin Flux-toteutuksen arkkitehtuurin pohjalta Facebook kehitti Fluxiin pohjautuvan Reduxin. Redux on vapaan lähdekoodin javascript-kirjaston, johon Facebookin nykyinen tilanhallinta pohjaa. Reduxin käyttö ei rajoitu pelkästään Reactin sovelluskehitykseen, vaan sen käyttö on mahdollista käytännössä kaikilla moderneilla web-sovelluskehyksillä. (Why-use-react-redux, React Redux 2020.)

2.5 Tilanhallinta Angularilla

Lähes kaikki varteenotettavat tilanhallintakirjastot Angularille pohjautuvat Reduxin ja Fluxin luomaan yksisuuntaiseen datavirtamalliin. Neljä suosituinta kirjastoa ovat NgRx, NGXS, Akita ja jo mainittu Redux-kirjasto.

Kaikista vaihtoehdoista NgRx valikoitui AIMUIn tilanhallintakirjastoksi. Syitä siihen, miksi juuri NgRx valikoitui käytetyksi tilanhallintaratkaisuksi, on monia:

- NgRx on yksinkertaisesti vaihtoehdoista edistynein ja kehitetyin tilanhallinta ratkaisu Angular-sovelluksille. Sen dokumentaatio on kattava ja sillä on valtavasti käyttäjiä. Ongelmatilanteessa joku toinen onkin jo saattanut kohdata samantapaisen ongelman ja ratkaissutkin sen.
- Kirjastolla on aktiivinen kehittäjäkunta. Kirjastoa päivitetään ja kehitetään jatkuvasti. Pelkoa siitä, että NgRx-kirjasto jäisi kehityksen jalkoihin, ei toistaiseksi ole. Sen voi olettaa pysyvän ajankohtaisena aktiivisen kehityksen ansiosta vielä pitkään.
- NgRx mahdollistaa johdonmukaisen ja selkeän sovelluksen testauksen. Sen mukanaan tuomat testausmenetelmät ja työkalut sopivat parhaiten Landis+Gyrin tarpeisiin.

3 NgRx

3.1 Yleistä

NgRx on oma Reduxiin pohjautuva tilanhallinta viitekehyskirjasto Angular-sovelluksille. Se pohjautuu Reduxin tapaan RxJs-kirjastoon. Fluxin ja Reduxin tapaan se pohjautuu yksisuuntaiseen datavirtaan. Oheisessa kuviossa on esitettynä NgRx Store datavirtaa kuvaava kaavio (ks. kuvio 4).

Kuvio 4. Tekijän laatima NgRx Storen datavirtaa kuvaava kaavio.

1. Käyttäjän vuorovaikutuksen seurauksena sovelluksessa lähetetään tapahtumia (actions).

2. Tapahtumavirtaa kuuntelevat reducerit. Reducerit käsittelevät yhden tapahtuman kerrallaan ja muuttavat storen tilaa tapahtuman tyypin ja tapahtuman mukanaan tuoman metadatan pohjalta. Reducerit ovatkin ainoa taho, joka voi tehdä muutoksia storen tilaan.

3. Ainoa tapa storen tilan noutoon, jotta sitä voisi käyttää sovelluksen muissa osissa, oon käyttää selektoreita. Selektorit muokkaavat storen tilan komponenteille käytettävämpään muotoon. Kaikki tämä tapahtuu reaktiivisesti storen tilan muuttuessa. Storen tilamuutokset tulevat välittömästi asynkronisesti selektoreille, jotka pitävät komponentit aina ajan tasalla. Näin komponentit heijastavat aina storen ja näin myös sovelluksen ajankohtaista tilaa.

4. Jos tapahtumien halutaan laukaisevan uusia tapahtumia, se voidaan tehdä storen sisällä käyttäen efektejä. Efektit kuuntelevat reducerien tapaan tiettyjä niille määrättyjä tapahtumia. Store-toteutuksessa niitä käytetään ulkoisten palveluiden kanssa keskusteluun ja muihin pitkäkestoisiin tapahtumaketjuihin. Jokainen efekti palauttaa lopulta uuden tapahtuman storen tapahtumaketjuun.

3.2 NgRx Store-rakenne

3.2.1 Store

Store koostuu yhdestä objektista, joka sisältää koko sovelluksen tilan tiedot. NgRx-arkkitehtuurimallin mukaan storen tulisi jokaisella hetkellä olla ainoa paikka, joka säilyttää sovelluksen korrektia ja ajankohtaista tilaa.

Store injektoidaan jokaiseen sitä käyttävään komponenttiin, palveluun tai muuhun sovelluksen osaan luokkien konstruktoreissa seuraavalla tavalla:

constructor(private store: Store<JUURITILA>){}

Store on käytännössä kuunneltava muuttuja, joka kuuntelee omassa tilassa tapahtuvia muutoksia. Edellä annetun konstruktorin tyypityksessä annetulla juuritilalla tarkoitetaan moduulin storen tilaa kokonaisuudessaan.

3.2.2 Tapahtumat (actions)

Tapahtumat ilmaisevat uniikkeja tapahtumia, joita suoritetaan läpi sovelluksen elinkaaren. NgRx-toteutuksessa jokainen sovelluksen tilaa muuttava tilanne alkaa tapahtuman lähettämisellä (Actions 2020).

Yksittäinen tapahtuma rakentuu kahdesta osasta:

1. Tyyppi (type): Tapahtuman tyypillä spesifioidaan, mikä tapahtuma on kyseessä. Uniikin tyypin perusteella storen osat kykenevät käsittelemään jokaisen tapahtuman niille erikseen määritetyllä tavalla. Tarkka tyypin määritys on myös tärkeää sovelluskehittäjien kannalta. Hyvä ja selkeä tyypin määritys helpottaa myöhempää storen tapahtumavirran tarkempaa tarkastelua. Tyyppi koostuu tyypillisesti kahdesta osasta. Lähteestä ja tapahtumasta.

   ” [Lähde] Tapahtuma”

   Sovelluksessa saattaakin olla monia lähes identtisiä tapahtumia eri ominaisuuksien välillä. Jotta kehittäjät voisivat erottaa identtiset tapaukset toisistaan, annetaan jokaiselle tapahtuman tyypille myös lähde ilmaisemaan tarkempaa kontekstia. Lähde annetaan hakasulkeiden sisään. Hakasulkeita seuraa itse tapahtuman nimi

2. Tietosisältö (payload): tapahtuman tietosisältö kuljettaa mukanaan tapahtumalle relevanttia metadataa. Jos tapahtuman tarkoituksena on muuttaa storen tilaa, tilamuutokselle relevantit muuttujat lähetetäänkin usein tapahtuman tietosisältönä. Kaikilla tapahtumilla ei välttämättä ole tietosisältöä lainkaan. Tapahtumat voivat toimia esimerkiksi tapahtumaketjussa toisten tapahtumien tai tapahtumaketjujen laukaisijoina. Tietosisältö annetaan tapahtumalle props-metodilla. Metodissa tietosisältö annetaan objektin sisällä.

Esimerkkinä: ”loadMeterings” -tapahtuma (ks. kuva 5). Tapahtuma kuuluu sovelluksen mittariosioon, joten sen tyyppi alkaa hakasulkeisiin suljetulla: ”[Metering Actions]” -osalla. Osalla spesifioidaan lähde tarkemmin. Tyyppi jatkuu itse tapahtumalla. Koska tapahtuma aloittaa mittarien latauksen sovellukseen, tapahtuma on nimeltään: ”Load Meterings”. Koska mittarit ladataan tietyn mittapisteen alle, vaaditaan myös mittapisteen id tapahtuman tietosisältönä. Props-metodille annetaan objekti, jonka ainoana muuttujana on meteringPointId. Id:n tyypiksi on annettu numero. Props-metodi ottaa sisäänsä aina objektin. Objektin sisällä voidaan käytännössä kuljettaa mitä vain tapahtuman kannalta tarpeellista metadataa.

```
export const loadMeterings = createAction('[Metering Actions] Load Meterings',
props<{meteringPointId: number}>());
```

Kuvio 5. Tyypillinen tapahtuma.

Tapahtumia voidaan laukaista käytännössä mistä vain sovelluksen osasta - niin komponenteista, palveluista kuin myös NgRx-efektien kautta jatkotoimenpiteinä aiempiin lähetettyihin tapahtumiin. Tapahtumia voidaan lähettää store.dispatch -metodilla.

Kuvassa (ks. kuvio 6) lähetetään loadMeterings-tapahtuma komponentin initialisoinnin yhteydessä. Sen tietosisällöksi annetaan aiemmin tapahtumalle props-metodilla määritetty mittapisteen id. Tässä tapauksessa id:ksi annetaan luku 1.

```
  constructor(private store: Store<MeteringPointsFeatureState>){}

  ngOnInit(): void {
    this.store.dispatch(MeteringsActions.loadMeterings({meteringPointId: 1}))
  }
}
```

Kuvio 6. loadMeterings-tapahtuman lähetys komponentin ngOnInit-initialisoissinssa.

MeteringActions on importoitu storen actions.index-kansiosta, jossa kaikki sovelluksen tapahtumat kerätään yhden tiedoston alle. Kuviossa 7 mittaritapahtumat tuodaan komponenttin.

```
import { MeteringsActions } from "../store/actions"
```

Kuvio 7, mittaritapahtumien importointi komponenttiin.

NgRx: n virallisessa dokumentaatiossa annetaan seuraavia ohjeita tapahtumien tehokkaaseen ja johdonmukaiseen käyttöön (Actions 2020):

1. Kirjoita tapahtumat ennen varsinaisten ominaisuuksien toteuttamista saadaksesi selkeämmän kuvan toteutettavasta ominaisuudesta.
2. Kategorisoi tapahtumat niiden lähteen mukaan.

3. Kirjoita mahdollisimman monta tapahtumaa, kuvaamaan sovelluksen toimintaa. Tapahtumisen kirjoittaminen on suhteellisen vaivatonta.
4. Kirjoita tapahtumia, älä käskyjä, koska tapahtumien kuvailun ja toteutuksen tulisi olla erotettua toisistaan.

3.2.3 Reducers

Kun tapahtuma on lähetetty, se päätyy siitä vastaavan reducerin käsiteltäväksi. NgRx:ssä reducerit huolehtivat sovelluksen tilan muutoksesta yhdestä tilasta toiseen. Yksinkertaistettuna ne kuuntelevat laukaistuja tapahtumia ja muuttavat tilaa tapahtumien tyyppien ja niiden tietosisällön perusteella. Reducerit ovat puhtaita funktioita, jotka ottavat parametreinaan sisään lähetetyn spesifin kuunnellun tapahtuman sekä viimeisimmän sovelluksen tilan. Näiden kahden parametrin perusteella ne päättelevät, palauttavatko ne uuden tilan vai pitävätkö ne tilan ennallaan. (Reducers 2020.)

NgRx-store koostuu yhdestä tai useammasta reducerista. Reducerit jäsennetään usein sovelluksen eri kokonaisuuksien mukaan. Jokainen reducer on luokka, joka pitää huolen siitä, että sen hallinnoima storen haara saa aina uusimman tiedon sovelluksen tilasta. Reducerit eivät mutatoi yksittäisiä arvoja niiden hallinnoimasta tilan osasta, vaan ne palauttavat aina uuden tilaobjektin kokonaisuudessaan.

Reducereissa alustetaan käytännössä koko storen rakenne. Jokainen reducer-tiedosto hallitsee yhtä storen haaraa, ja se huolehtii myös haaran alustamisesta. Ensimmäinen osa reducer-tiedostoa on reducer-luokan hallitseman store-haaran rajapinnan määritys ja alustus.

Jokainen sovelluksen tilan osaa hallitseva reducer-tiedosto koostuu kolmesta osasta:

1. Määritetään tiedoston hallitseman storen haaran rajapinta (ks. kuvio 8).

2. Alustetaan initiaaliset arvot hallitulle haaralle (ks. kuvio 8).
3. Viimeisenä ovat puhtaat reducer-funktiot, jotka käsittelevät relevanttien tapahtumien laukaisemat tilanmuutokset (ks. kuvio 9).

```
export interface MeteringPointState {
    meteringpoints: Meteringpoint[];
    selected: number;
    loading: boolean;
}

export const initialState: MeteringPointState = {
    meteringpoints: [],
    selected: undefined,
    loading: false
};
```

Kuvio 8. Tilan rajapinnan luonti ja alustus reducer-tiedostossa.

Itse reducer on puhdas funktio *createReducer().* Sen parametreiksi annetaan aiemmin alustettu tila ja lista kuunneltavista tapahtumista. Kuunneltavat tapahtumat eritellään tapahtumavirrasta niille määritettyjen uniikkien tyyppien perusteella. Jokaiselle tapahtumalle on määritetty erikseen se, kuinka reducerin hallinnoimaa tilaa tulee muuttaa sen kohdalla.

```
const meteringpointReducer = createReducer(
  initialState,
  on(meteringPointAction.loadMeteringpoints, (state) => ({...state, loading: true})),
  on(meteringPointAction.loadMeteringpointsSuccess, ((state, {meteringpoints}) =>
  ({...state, loading: false, meteringpoints: meteringpoints}))),
  on(meteringPointAction.loadMeteringpointsFail, (state) => ({...state, loading: false})),
  on(meteringPointAction.selectMeteringpoint, (state, {meteringpointId}) => ({...state, selected: meteringpointId}))
);
```

Kuvio 9. Reducer-funktiot.

Tapahtuman saapuessa reduceriin annetaan reducerille koko store-haaran aiempi tila ja mahdollisesti tapahtuman mukanaan kuljettama tietosisältö. Kuviossa 10 on esimerkki typillisestä reducer-funktiosta.

```
on(meteringAction.loadMeteringsSuccess, (state, { meterings }) => ({
  ...state,
  loading: false,
  meterings: meterings
})),
```

Kuvio 10. reducer-funktio tapahtumalle loadMeteringsSuccess.

Tilaa ja tietosisältöä voidaan manipuloida tässä välissä käytännössä tarpeen mukaan lähes miten vain. Lopuksi reducerille on kuitenkin palautettava kokonaan uusi tila. Jos tapahtuma ei vaadi tilamuutoksia, palautetaan hallitulle tilahaaralle sen vanha tila.

3.2.4 Efektit

Välillä lähetettyjen tapahtumien on johdettava uusien tapahtumien lähettämiseen. Tällöin NgRx-toteutuksessa käytetään sivuefektejä. Efektit ovat injektoitavia palveluita. Efektit kuuntelevat jokaista lähetettyä tapahtumaa. Ne kuitenkin toimivat vain niille määrättyjen yksittäisten tapahtumien kohdalla tapahtumien tyyppien perusteella. Käytännössä ne toimivat siis samalla periaatteella kuin reducerit, mutta efektit suorittavat toimintoja niin asynkronisesti kuin synkronisestikkin ja palauttavat niiden perusteella uuden tapahtuman.

Palvelupohjaisessa Angular-sovelluksessa komponentit keskustelevat ulkoisten lähteiden kanssa palveluiden kautta. Efektit siirtävät vastuun komponenteilta efekteille mahdollistaen näin puhtaampien ja yksikertaisempien komponenttien luonnin. Efektien tehtäväksi jäävät ulkoisten datalähteiden kanssa keskustelu, jotka ne suorittavat käyttäen samoja palveluita, joita komponentit käyttäisivät ilman NgRx-toteutusta. Efekteille on ulkoistettu tapahtumat, joissa vuoropuhelu komponenttien kanssa ei ole tarpeellista. Tällaisia tehtäviä ovat esimerkiksi ulkoisista lähteistä

tulevan datan noutaminen, pitkäkestoiset moniin tapahtumiin johtavat prosessit ja muiden ulkoisten interaktioiden suoritukset. (@ngrx/effects 2020.)

Näiden palveluiden ulkoistaminen efekteille on sovelluksen kasvaessa suositeltavaa. Ilman NgRx-toteutusta monista lähteistä noudettava data ja monesti muista palveluista riippuvaiset palvelut lähtevät hyvin helposti kehittäjien käsistä. Kun palveluiden kanssa kommunikointi tuodaan osaksi NgRx-arkkitehtuuria, voimme pilkkoa prosessit yhä pienemmiksi palasiksi. Palveluiden käsittely osana NgRx-toteutusta mahdollistaa myöhemmin tehokkaamman debuggausprosessin ja ongelmien entistä tarkemman paikantamisen.

Tyypillinen efektin käyttötapaus olisikin datan lataaminen käyttöliittymään API-kutsulla seuraavasti:

1. Käyttäjän interaktioiden tuloksena sovelluksessa laukaistaan tapahtuma.
2. Tapahtumaa kuunteleva efekti vastaanottaa tapahtuman sen uniikin tyypin perusteella. Tapahtuman kutsuu palvelua tietosisällön perusteella. Palvelulle annetaan tietosisältö parametrinä.
3. Palvelu ottaa yhteyttä ulkoiseen APIin, joka palauttaa vastauksen saatuaan datan takaisin sitä odottavalle efektille.
4. Efekti laukaisee uuden tapahtuman, jonka tietosisällöksi annetaan saatu data
5. Uusi tapahtuma jatkaa matkaansa reducereihin lopulta muuttaen storen tilaa datan perusteella vaaditulla tavalla.

Efektit käyttävät hyväkseen RxJs-piippuja. Piippuja käytetään, koska efektien vaatimat tilanteet saattavat monessa tapauksessa olla suhteellisen pitkään kestäviä monivaiheisia prosesseja. Käytännössä piiput ovat puhtaita funktioita, jotka ottavat sisään RxJs-observableja. Tässä tapauksessa observablet ovat kuunneltuja tapahtumia. Piippu suorittaa yksi kerrallaan vaaditut tapahtumat odottaen aina aiemman valmistumista ja lopulta palauttaa ulos uuden observablen. Efektien tapauksessa uusi observable on uusi tapahtuma. (Operators 2020)

Kuviossa 11 (ks. kuvio 11) esimerkki tyypillisestä efektistä, joka hakee tarvittavaa dataa ulkoiselta APIlta. Efekti luodaan createEffect()-funktion sisälle. CreateEffect-funktio mahdollistaa tapahtumavirtojen kuuntelun, ja jokainen efektivirran päätteeksi laukaistu tapahtuma palautetaan storelle. (@ngrx/effects 2020)

```
loadMeterings$ = createEffect(() =>
  this.actions$.pipe(
    ofType(MeteringsActions.loadMeterings),
    map(action => action.meteringPointId),
    switchMap(meteringPointId => {
      return this.meteringService.getMeterings(meteringPointId).pipe(
        map(meterings =>
          MeteringsActions.loadMeteringsSuccess({ meterings: meterings })
        ),
        catchError(error => of(MeteringsActions.loadMeteringsFail()))
      );
    })
  )
);
```

Kuvio 11. Tyypillinen efekti, joka noutaa dataa ulkoiselta APIlta.

Injektoitava Actions-palvelu tarjoaa käyttöön observablen virran kaikista laukaistuista tapahtumista viimeisimmän storen tilamuutoksen jälkeen. Actions$-muuttujan dollarimerkki indikoi muuttujan olevan observable. Action-palvelu injektoidaan luokan konstruktorissa (ks. kuvio 12).

```
constructor(
    private actions$: Actions,
    private meteringService: MeteringService
) {}
```

Kuvio 12. Konstruktorissa injektoitu Actions-palvelu ja mittaripalvelu.

Actions$ observableen tehdään piippu, joka kuuntelee tapahtumavirrasta loadMeterings-tapahtumaa. Tapahtumaa kuunnellaan ofType-operaattorilla (ks. kuvio 13). Se ottaa yhden tai useamman tapahtumatyypin parametreikseen, joilla se päättelee juuri ne tapahtumatyypit, joilla sen kuuluu toimia. (@ngrx/effects 2020.)

```
ofType(MeteringsActions.loadMeterings),
```

Kuvio 13. Efektin kuuntelema tapahtuma määritetynä ofType-metodin sisällä.

LoadMeterings-tapahtuma tuo mukanaan tietosisältönään mittapisteen id:n, joka erotellaan map-operaattorilla tapahtumasta. Seuraavaksi switchMap-operaattorin sisällä palautetaan mittaripalvelun getMeterings()-funktio, joka palauttaa ulkoiselta APIlta parametrinä annettua mittapisteen id:tä vastaavat mittapisteen mittarit.

Palvelun kutsun perään on lisätty piippu. Palvelun palauttaessa mittarit onnistuneesti piipun sisällä palautetaan uusi tapahtuma loadMeteringsSuccess (ks. kuvio 14). Palvelun palauttamat mittarit annetaan tapahtuman parametriksi. Tapahtuma käsitellään seuraavaksi reducereissa ja mittarit lisätään storen tilaan. Jos ulkoisen APIn kutsussa tapahtuu virhe, catchError()-operaattori reagoi siihen, ja laukaistaan tapahtuma loadMeteringsFail laukaistaan.

```
switchMap((meteringPointId) => {
  return this.meteringService.getMeterings(meteringPointId).pipe(
    map((meterings) => MeteringsActions.loadMeteringsSuccess({meterings:
    meterings})),
    catchError(error => of(MeteringsActions.loadMeteringsFail())))
})))
```

Kuvio 14. Efekti palauttaa mittaripalvelun vastauksen tapahtumana.

3.2.5 Selektorit

Selektorien tehtävänä on toimia yksisuuntaisina kommunikoijina storelta komponenteille. NgRx-arkkitehtuurimallissa ne ovat ainoa tapa tilan noutamiseen komponenteille storelta. Selektorit ovat puhtaita funktioita, jotka suoritetaan aina niiden seuraaman storen arvon muuttuessa. Koska storen muutokset laukaisevat selektorit välittömästi, ne pitävät komponentit aina ajan tasalla sovelluksen viimeisimmästä tilasta.

Storen tilaobjektin puhtaat arvot eivät välttämättä ole vielä siinä muodossa, jossa ne olisivat käyttökelpoisia komponenteille. Tämän lisäksi arvoilla saattaa myös olla monia eri käyttötarkoituksia sovelluksen näkymissä. Selektorien toisena tehtävänä ennen tilan tuomista komponenteille onkin datan muuttaminen sellaiseen muotoon, että sitä tarvitsevat komponentit pystyisivät käyttämään sitä mahdollisimman puhtaasti. Mitä suurempi määrä sovelluksen logiikasta suoritetaan NgRx-storessa, sitä puhtaampia komponentteja voidaan kirjoittaa.

Selektorien käyttöön tarvitaan ensiksi sovelluksen juuritilan. Sovelluksen tila kerätään reducereilta yhdeksi objektiksi reducer-kansion index-tiedostossa. Samaan tiedostoon luodaan myös funktio getRootState (ks. kuvio 15).

```
export const getRootState = (state: MeteringPointsFeatureState) => {
  return state
}
```

Kuva 15. GetRootState metodi.

GetRootState palauttaa koko sovelluksen tilan. Kaikki selektorit pohjautuvat getRootState-metodiin, ja niiden tehtävä on pilkkoa juuritilaa yhä pienemmiksi käytettäviksi osiksi komponenteille. Varsinaiset selektori-funktiot luodaan jokaista ominaisuutta vastaaviin tiedostoihin selectors-kansion alle. GetRootState-funktio importoidaan näihin tiedostoihin selektorien käytettäväksi.

Selektorit pohjautuvat NgRx:n createSelector-funktioon. Ensimmäinen selektori käyttää parametrinään getRootState funktion palauttamaa juuritilaa. Selektorit voivat käyttää parametrinään muita selektoreja. Selektoreja voidaankin näin ketjuttaa ja pilkkoa ne näin yhä pienemmiksi helpommin ymmärrettäviksi paloiksi. Kuvassa (ks. kuva 16) getMeteringStaten createSelector-funktio palauttaa importoidun juuritilan pohjalta meterings-tilan. GetMeterings funktio noutaa getMeteringsStaten palauttaman meterings-tilan pohjalta sen alla olevat mittarit. Samaa selektoria käytetään myös meterings-tilan ”loading” boolean-arvon noutamiseen getMeteringsLoading selektorilla.

```
export const getMeteringState = createSelector(
    getRootState,
    (state: MeteringPointsFeatureState) =>  state.meterings
);

export const getMeterings = createSelector(
  getMeteringState,
  (state) => state.meterings
)

export const getMeteringsLoading = createSelector(
  getMeteringState,
  (state) => state.loading
)
```

Kuva 16. Esimerkkejä selektoreista.

Selektoreilla voidaan manipuloida tilaa käytännössä rajattomasti. Kuvassa (ks. kuva 17), getMeteringsWithCapitalLetters-selektori ottaa aiemman getMeterings-selektorin ja muuttaa kaikkien mittareiden nimien kirjoitusasun isoiksi kirjaimiksi. Muutoksen jälkeen taulukko palautetaan.

```
export const getMeteringsWithCapitalLetters = createSelector(
  getMeterings,
  (meterings) => {
    const meteringsNamesWithCapitalLetters = meterings.map(metering => metering.name.toUpperCase)
    return meteringsNamesWithCapitalLetters;
  }
)
```

Kuva 17. Selektori voi suorittaa muutakin logiikkaa ennen arvon palautusta.

Komponentit voivat vastaanottaa selektorien arvoja kahdella eri tavalla: Komponentti voi joko ottaa observable-muuttujan välittäen sen asynkroonisesti lapsi komponentilleen tai muuttaa sen komponentin sisällä käytettäväksi staattiseksi muuttujaksi.

Jos komponentti ottaa selektorin palauttaman observable-muuttujan vastaan antaakseen sen lapsikomponentilleen, voidaan muuttuja ottaa vastaan sellaisenaan.

Komponettiluokan konstruktorissa injektoitu storen (ks. kuvio 18) select-metodi ottaa parametrikseen komponenttiin importoidut selektorit. Select-metodi palauttaa selektorin observable-muuttujan.

```
meterings$: Observable<Metering[]> = this.store.select(getMeterings);
loading$: Observable<boolean> = this.store.select(getMeteringsLoading);

constructor(private store: Store<MeteringPointsFeatureState>){}
```

Kuvio 18. Esimerkki select-metodien käyttämisestä suoraan puhtaiden observable-muuttujien noutoon.

Kun komponentti välittää muuttujan lapsikomponentilleen, voidaan observable-muuttuja välittää async-piipun avulla (ks. kuvio 19). Async-piippu on Angularin sisäänrakennettu piippu. Se pitää huolen siitä, että arvon muuttuessa komponentissa muuttunut arvo päivitetään myös lapsikomponentissa. (AsyncPipe 2020.) Piippu muuttaa myös observablen staattiseksi muuttujaksi. Lapsikomponentin vastaanottamaa @Input-muuttujaa voidaan tämän jälkeen käsitellä normaalina muuttujana.

```
<metering-table [meterings]="meterings$ | async" [loading]="loading$ | async"></metering-table>
```

Kuvio 19. Observable-muuttujat välitys lapsikomponetille @Input-metodilla. Async-piippu muuttaa observable-muuttujat staattisiksi muuttujiksi lapsikompoenntille.

Jos selektorien välittämää arvoa tarvitaan sen vastaanottamassa komponentissa, se vaatii observable-muuttujan muuttamista staattiseksi muuttujaksi. Muutos vaatii Subscribe-metodin käyttöä (ks. Kuvio 20).

Subscribe-metodi on RxJS-metodi, joka voidaan asettaa kuuntelemaan muutoksia määritetyssä observable-muutujassa. (Observable 2020.) Komponentissa määritetään Subscription-muuttuja, joka asetetaan kuuntelemaan haluttua selektoria. Kuuntelija asetetaan komponentin ngOnInit-metodin sisään. OnInit-metodin sisällä kuuntelija alustetaan heti komponentin initialisoituessa. Subscribe-metodin sisällä selektorin palauttaman observablen muuttujan viimeisin arvo annetaan määritetylle muuttujalle.

```
meteringsSub: Subscription;
meterings: Metering[];

ngOnInit(): void {
  this.meteringsSub = this.store.select(getMeterings).subscribe(meterings => {
    this.meterings = meterings
  })
}
```

Kuvio 20. Selektorin käyttö komponetissa käytettävälle muuttujalle.

3.3 Käyttöönotto ja kansiorakenne

3.3.1 Käyttöönotto

NodeJs-pohjaisessa sovelluskehityksessä NgRx Storen voi installoida projektiin komennolla *”npm install @ngrx/store –save”.* Installoinnin jälkeen StoreModule on importoitava siihen moduuliin, jonka kehittäjä haluaa storea käyttävän. StoreModule importoidaan seuraavasti:

import { StoreModule } from '@ngrx/store';

Importoinnin jälkeen StoreModule lisätään moduulin imports-taulukkoon**.**

Seuraavaksi on hyvä lisätä alustava storen rakenne: kirjoittaa ensimmäiset tapahtumat ja niitä vastaavat reducerit. Store tarvitsee jonkinlaisen alustavan tilarakenteen, joka voidaan tuoda moduulille käyttöön. Kun storelle on alustettu haara tai haaroja sen reducer-tiedostoissa, tuodaan kyseiset haarat yhteen reducer-kansion index-tiedostossa.

Reducer-kansion index-tiedostossa alustetaan koko sovelluksen tilan rajapinta (ks. kuvio 21). Näin yksittäisten ominaisuuksien reducer-tiedostoissa alustetuista tiloista rakennetaan koko sovelluksen tai moduulin suurempi rajapinta. Index-tiedostoon määritetään reducers-muuttuja, joka kerää reducer-tilat tapahtumakuuntelijoineen yhteen. Reducer-funktiot on määritetty jokaisen reducer-tiedoston lopussa. Ne ottavat parametreikseen haaran tilan ja tapahtumat.

```
import { ActionReducerMap, createFeatureSelector } from '@ngrx/store';
import * as fromMetering from './metering.reducer'
import * as fromMeteringpoint from './meteringpoint.reducer'
import * as fromStore from './store.reducer'


export interface MeteringPointsFeatureState {
  meteringpoints: fromMeteringpoint.MeteringPointState,
  meterings: fromMetering.MeteringState,
  store: fromStore.StoreState
}

export const reducers: ActionReducerMap<MeteringPointsFeatureState> = {
    meteringpoints: fromMeteringpoint.reducer,
    meterings: fromMetering.reducer,
    store: fromStore.reducer
};
```

Kuvio 21. Index-tiedosto

Reducers-muuttuja on ActionReducerMap, joka käytännössä kuuntelee kaikkia storen tilamuutoksia ja palauttaa aina uusimman tilan storen tilamuutosten jälkeen. Haluttu store-rakenne on näin määritetty ja valmis injektoitavaksi siitä vastaavaan moduuliin.

StoreModule.forRoot(reducers) injektoidaan moduulin imports-taulukkoon. ForRoot määrittää näin sovelluksen juuritilan, jonka parametrinä annetaan aiemmin reducerien index-tiedostossa määritetty reducers-muuttuja.

```
export function reducer(state: MeteringState, action: any) {
    return meteringReducer(state, action);
}
```

Kuvio 22. Reducer-tiedoston lopussa määritetty reducer-funktio.

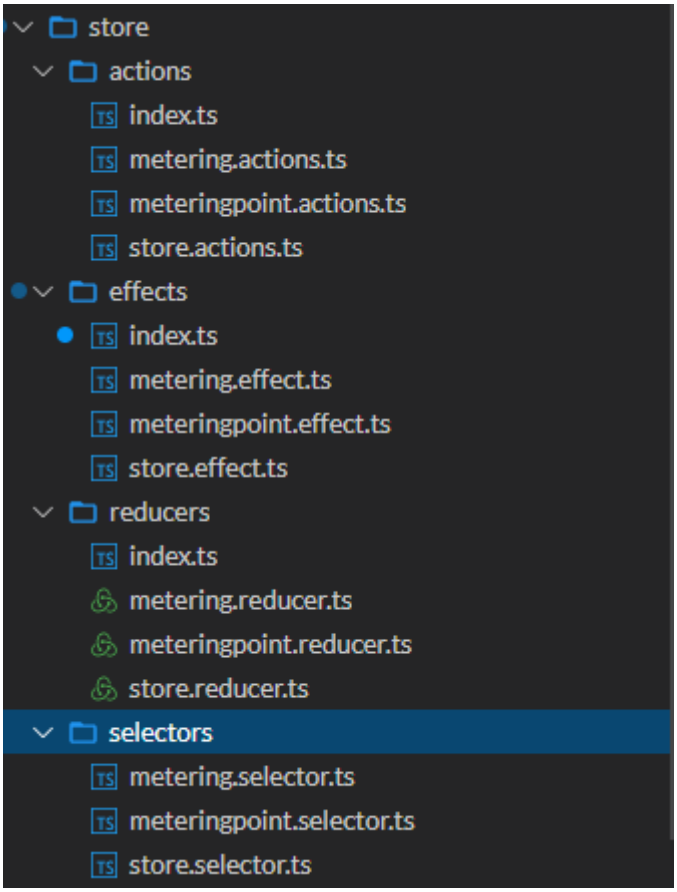
Jos sovellus käyttää efektejä, ne on tuotava sovellukseen ja moduuliin erikseen.

Komento *"npm install @ngrx/effects –save"* installoi efektit projektiin. Kun kaikki efektit ovat määritetty, ne kerätään yhteen efektikansion index-tiedostossa. Tiedostossa määritetään effects-taulukko, johon kaikki efektiluokat importoidaan. Tämä effects-taulukko injektoidaan moduuliin samaan tapaan kuin reducerit, mutta SoreModulen sijasta ne kuuluvat EffectsModulelille. *EffectsModule.forRoot(effects)* tuo moduulille efektit. Se injektoidaan reducerien tapaan moduulin imports-taulukkoon.

3.3.2 Kansiorakenne

Osa hyvin hallittua store-rakennetta on kaikkien sen osien tehokas ja selkeä järjestely. Jokainen storen osa on järjestelty niitä vastaaviin kansioihin. Kansioista niiden importointi komponentteihin, palveluihin tai muihin sovelluksen niitä tarvitseviin osiin on selkeää.

Selkeässä NgRx-toteutuksessa kaikki storen toiminnot on kerätty yhden kansion ”store” alle (ks. kuvio23). Kansioon on kerätty erillisiin kansioihinsa tapahtumat, reducerit, efektit ja selektorit. Näiden kansioiden tiedostojen sisällöt on kerätty kussakin kansiossa index.ts-tiedostoon. Tapahtumien index-tiedostossa kerätään tapahtumat yhteen, jotta niitä voidaan käyttää selkeästi muissa sovelluksen osissa. Näin tapahtumia voidaan tuoda komponenteille importoimatta kaikkia tapahtumia niitä tarvitseville tahoille erikseen.



Kuvio 23. Kansio rakenne.

Kun tapahtumat tuodaan komponenteille tällä tavalla, on myös komponenttien tasolla selkeämpää, mihin storen osaan kyseinen tapahtuma kuuluu (ks. kuviot 24, 25).

```
import * as MeteringsActions from './metering.actions'
import * as MeteringpointActions from './meteringpoint.actions'
import * as StoreActions from './store.actions'

export {MeteringsActions, MeteringpointActions, StoreActions}
```

Kuvio 24. Tapahtumien importointi ja exportointi yhtenä kokonaisuutena.

```
this.store.dispatch(MeteringpointActions.loadMeteringpoints())
```

Kuvio 25. Tyypillinen tapahtuman lähetys actions-kansion index-tiedostosta importoidulla tapahtumalla.

Reducerien index-tiedosto kerää reducerit yhteen, jotta storen eri osien tilat voidaan kerätä yhteen (ks. kuvio 26). Kerätyistä tiloista voidaan näin rakentaa kokonaisuus, joka yhdessä kattaa koko moduulille annetun tilan. Tila voidaan lähettää osasta vastaavalle moduulille storen alustusta varten. Pienemmissä sovelluksissa tämä voi tarkoittaa jo koko sovelluksen tilaa, mutta suuremmissa sovelluksissa yhden suuremman kokonaisuuden tilaa. Suuremmat kokonaisuudet voidaan taas tuoda yhteen samalla periaatteella kansiorakenteen ylemmillä tasoilla.

```
import { ActionReducerMap, createFeatureSelector, createSelector } from '@ngrx/store';
import * as fromMetering from './metering.reducer'
import * as fromMeteringpoint from './meteringpoint.reducer'
import * as fromStore from './store.reducer'


export interface MeteringPointsFeatureState {
  meteringpoints: fromMeteringpoint.MeteringPointState,
  meterings: fromMetering.MeteringState,
  store: fromStore.StoreState
}

export const reducers: ActionReducerMap<MeteringPointsFeatureState> = {
    meteringpoints: fromMeteringpoint.reducer,
    meterings: fromMetering.reducer,
    store: fromStore.reducer
};
```

Kuvio 26. Reducer kansion index-tiedosto.

Efektien index-tiedosto kerää kaikki moduulin storen efektit yhteen. Näin ne voidaan tuoda yhtenä kokonaisuutena storea tarvitsevaan moduuliin ja efektit voidaan alustaa yhtenä kokonaisuutena osaksi storea.

3.4 Työkalut

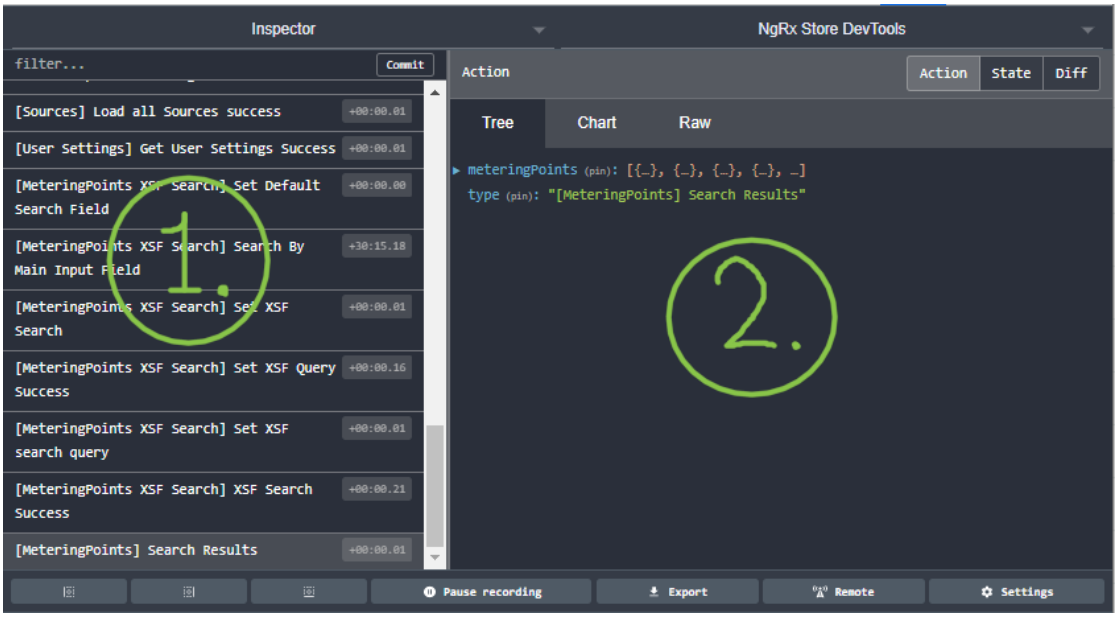
NgRx:n toiminnan seuraamiseen löytyy erinomaisia työkaluja. Selaimella storen tilaa ja tapahtumia voi seurata reaaliajassa Redux DevTools-selainlaajennuksen avulla. Laajennuksen voi ladata Chrome-selaimella home web storesta tai vaihtoehtoisesti Firefox-selaimella add-ons Managerilla. Redux Devtools käyttö vaatii myös StoreDevtoolsModule.instrument-osan importoinnin sovelluksen AppModule moduuliin.

Redux DevTools avulla kehittäjä näkee ne kaikki tapahtumat, joita sovelluksessa suoritetaan. Käyttäjä voi myös ”aikamatkustaa” tapahtumien välillä ja tarkastella tarkemmin tapahtumien tilamuutoksia ja tilaa kokonaisuudessaan sovelluksen elinkaaren eri vaiheissa.

Redux DevTools löytyy sen asentamisen jälkeen selaimen kehittäjäikkunan vasemmasta laidasta, Redux-otsikon alta (ks. kuvio 27).

Kuvio 27. Redux DevTools-työkalu kehittäjä ikkunan valitsimen vasemmassa laidassa

Itse työkalun näkymä rakentuu kahdesta osasta: tapahtuma virrasta vasemmalla (ks. kuvio 28, nro 1) ja tilan ja tapahtuman tarkempaan tarkasteluun tarkoitetusta ikkunasta oikealla (ks. kuvio 28 nro 2)



Kuvio 28. Redux DevTools-työkalun ikkuna.

Tapahtumavirtaosassa (ks. kuvio 28, nro 1) eritellään sovelluksessa tapahtuvat tapahtumat reaaliajassa ja kronologisessa järjestyksessä ylhäältä alas. Rivit sisältävät tapahtumien määritetyt tyypit ja niiden kestot. Tapahtuman voi valita tarkempaan tarkasteluun valitsemalla sen taulukosta.

Tarkemman tarkastelun osassa (ks. kuvio 28 nro 2) käyttäjä voi tarkastella valittua tapahtumaa, tapahtuman aikaista tilaa tai sen aiheuttamia eroja tarkemmin. Näiden kolmen vaihtoehdon välillä voi navigoida oikeassa yläreunassa sijaitsevan valitsimen avulla ja niiden formaattia voi vaihtaa tarkasteluosan vasemmassa yläreunassa olevalla valitsimella.

Tarkasteltaessa yksittäistä tapahtumaa on mahdollista nähdä tyypin lisäksi sen mukanaan kuljettama metadata kokonaisuudessaan. Myös sovelluksen tapahtuman ai-

kaista storen tilaa voi tarkkailla kokonaisuudessaan. Näkymä näyttää storen tilan valitun tapahtuman suorituksen jälkeen ja näin käytännössä mahdollistaa tilan seuraamisen sovelluksen koko elinkaaren jokaisessa vaiheessa. Myös tapahtuman aiheuttamia tilamuutoksia voi tarkastella tarkemmin.

Redux DevTools onkin korvaamaton sovelluksen debuggaus-prosessissa. Se tuo esiin tapahtumien tyyppien johdonmukaisen nimeämisen tärkeyden. Sovelluksessa saattaakin olla lähes identtisiä tapahtumia, jotka tapahtuvat eri ominaisuuksien välillä. Kun eritellään tapahtuman vastaava ominaisuus ja itse tapahtuma mahdollisimman tarkasti, estetään myöhemmässä vaiheessa aiheutuvat väärinymmärrykset.

3.5 NgRx-käytänteet

Tehokas NgRx-kirjaston käyttö vaatii tiettyjen käytänteiden noudattamista.

3.5.1 Tyhmät ja viisaat komponentit

Hyvin jäsennetyn Angularin komponenttipohjaisen arkkitehtuurin voi käytännössä jakaa kahteen eri komponenttityyppiin: niin sanottuihin container-komponentteihin ja tavallisiin komponentteihin. Container-komponentit kokoavat suurempien kokonaisuuksien pienemmät komponentit yhden ison komponentin alle. Container-komponentteja voisi näin kuvailla isompien kokonaisuuksien lennonjohtotorneiksi. Tavalliset komponentit taas hoitavat yksittäisiä niille annettuja yksinkertaisia tehtäviä. Container-komponentteja voisi siis tässä yhteydessä kutsua ”viisaiksi” komponenteiksi, jotka vastaavat kysymykseen ”Miten asiat toimivat?”. Tavallisia komponentteja voisi taas kutsua ”tyhmiksi” komponenteiksi, jotka vastaavat kysymykseen ”Miltä asiat näyttävät?”.

Kun NgRx Storen otetaan käyttöö, halutaan seurata samaa käytännettä ja yksinkertaistaa rakennetta erottamalla storen kanssa kommunikoivat komponentit ”viisaiden” ja ”tyhmien” -komponenttien välille.

Container komponentit toimivat storen arvojen vastaanottajina. Tietyn kokonaisuuden container-komponentti ottaa koko kokonaisuuden tarvitsevat storen arvot vastaan käyttäen selektoreja. Storelta saadut arvot lähetetään niitä tarvitseville komponenteille alaspäin käyttäen komponettien @Input-ominaisuutta. Näin tavalliset komponentit pysyvät mahdollisimman yksinkertaisina ja kokonaisuuksien valitsijat löytyvät aina yhdestä ja samasta paikasta.

Tällä periaatteella vältetään turhien kuuntelijoiden pystytykset ja mahdollistetaan ymmärrettävien ja mahdollisesti uudelleenkäytettävien komponenttien luonnin.

3.5.2 NgRx Storen käyttäminen toteutuksessa

NgRx Storea tilanhallintaan käyttävään sovellukseen tulisi implementoida ominaisuuksia niin, että mahdollisimman suuri osa logiikasta olisi siirretty storen käsiteltäväksi. Implementaatio tulisi siis tehdä storeen käytännössä aina, kun siihen on mahdollisuus. Ominaisuuksia suunnitellessa tulisikin suunnittelu toteuttaa mielellään ensin miettien sitä, kuinka ominaisuus toimii storen toteutusperiaatteeseen pohjaten. Jos ominaisuus typistettäisiin vain lähetettyihin tapahtumiin, mitä tapahtumia tarvittaisiin ja missä järjestyksessä? Kun storen kannalta tarvittavat tapahtumat on ensiksi eritelty, suunnittelua on helppo jatkaa. Kaikki sovelluksen tilamuutokset pitäisi toteuttaa storen avulla. Hyvä sääntö on se, että jokaisen sovelluksen eksaktin tilan pitäisi olla replikoitavissa ainoastaan storen tilaobjektin pohjalta.

Kun kaikkia sovelluksen toimintoja ja tilanmuutoksia käytetään storen kautta, se helpottaa sekä debuggausprosesseja että kehittäjien yhteistyötä. Selkeä store-toteutus

nopeuttaa työskentelyä ja mahdollistaa eri ominaisuuksien välillä työskentelyn ja ymmärtämisen nopealla aikataululla

4 Työn toteutus

4.1 Tavoite

Opinnäytetyössä oli tarkoitus laatia NgRx Storen käytöstä kattava wikisivusto Landis+Gyrin työntekijöille. Selkeän ja kaiken oleellisen sisältävän wikisivuston tarkoitus oli helpottaa tulevien kehittäjien työtä ja mahdollisesti myös madaltaa NgRx-storen oppimiskynnystä ja sen käytön harjoitteluun käytettävää aikaa. Wikisivuston tuli myös asettaa yhteiset toimintamallit ja raamit sovelluskehitykselle.

4.2 Wikisivuston kirjoitusprosessi ja sisältö

Ensimmäinen tehtävä kirjoitusprosessissa oli jaotella ja eritellä NgRx Storen oppimisen kannalta tärkeät aiheet. Minkä asioiden ymmärtäminen olisi lukijan oppimisen kannalta kriittisiä, ja mistä näin jälkikäteen miettien olisi ollut apua omassa oppimisprosessissa? Seuraavaksi luettelen wikisivuston osat, niiden sisällön ja miksi ne ovat oleellisia osia sivustossa ja aiheen syvemmässä ymmärtämisessä.

4.2.1 NgRx Store

Koko wikisivusto on rakennettu tämän pääotsikon NgRx alle. Sivulla selvitetään lukijalle lyhyesti, mistä on kysymys ja mikä oli motivaationa koko wikisivuston laatimiseen. Tältä sivulta lukija voi myös navigoida tehokkaasti minne tahansa sivustolla.

4.2.2 State management

Lukijalle selitetään aluksi se, mitä sovelluksen tilalla tarkoitetaan ja miksi sen hallinta on selkeän sovelluksen ja tehokkaan sovelluskehityksen kannalta olennaista.

NgRx store pohjautuu Facebookin kehittämään Reduxiin, joka taas pohjautuu Facebookin kehittäjätiimin aiempaan flux-ajattelumalliin. Koska koko nykyaikainen tilanhallinta pohjaakin Facebookin toteutuksiin, olen todennut hyväksi käytännöksi selittää ensin, mihin tarpeeseen sen kehitys alun perin pohjautui. Facebookissa oli bugi, joka ratkaistiin uudella tilanhallintatoteutuksella (Ngrx Store - An Architecture Guide. 2020). Myös moni ihminen on aikoinaan itsekin törmännyt Facebookin bugiin. Tämä antaakin hieman tarttumapintaa ja hyvän lähtökohdan itse ongelman tarkempaan selittämiseen. Kun lukija ensin ymmärtää, miksi koko ajattelumalli alun perin otettiin käyttöön, on hänen helpompi ymmärtää, miksi NgRx-kirjastoa käytetään sovelluksissa.

Facebookin sovelluksen oikeassa yläkulmassa sijaitsee painike, joka avaa sovelluksen viestiosion. Painikkeen tehtävä oli myös näyttää se, kuinka monta lukematonta viestiä käyttäjällä kulloinkin oli. Käyttäjä sai kuitenkin systemaattisesti väärää tietoa: painike näytti, että viesti, jonka käyttäjä oli juuri lukenut, olisi vielä ollutkin lukematta. Facebookin kehittäjätiimi kykeni korjaamaan bugin ensimmäisellä kerralla vaivattomasti. Bugi kuitenkin ilmestyi uudelleen esiin joko eri variaatioissa tai uuden ominaisuuden implementoinnin mukana. (Ngrx Store - An Architecture Guide. 2020)

Facebooksin esimerkki on hyvä malliesimerkki sellaisista tilanteista, joita keskitetty tilanhallinta on luotu ratkaisemaan. Yksinkertaistettuna Facebookin ongelmana oli se, että identtistä jatkuvasti muuttuvaa dataa pyrittiin näyttämään eri sovelluksen osissa samanaikaisesti eri mallien toimesta. Kun muuttuvaa tilaa ei kyetty noutamaan

eri instansseihin kestävällä ja keskitetyllä tavalla, kasvava koodipohja loi jatkuvasti uusiutuvia ongelmia. (Ngrx Store - An Architecture Guide. 2020)

4.2.3 Structure

NgRx:n rakennetta tarkastellaan sivulla ensiksi kokonaisuutena, josta lukijalla on mahdollisuus navigoida yksittäisten osien tarkempaan tarkasteluun.

Jokainen NgRx-arkkitehtuurin osa käydään sivuilla läpi käyttäen hyväksi esimerkkejä Landis+Gyrin omista sovelluksista.

Lähes kaikkien NgRx:n osien kirjoitusmuoto päivitettiin Angularin versiossa 7.4.0 uuteen selkeämpään ja tehokkaampaan muotoon (NgRx 2020, Actions). Landis+Gyrin NgRx-toteutuksista löytyy vielä vanhaa kirjoitusmuotoa, vaikka suurin osa sovellusten koodista on päivitetty uuteen muotoon. Tästä syystä wikisivustolla käydään jokaisen storen osan kohdalla läpi molemmat toteutusmuodot, jotta sovellusten ja sivuston väliltä ei löytyisi epäjohdonmukaisuuksia

Vanha toteutus muoto on piilotettu alustavasti, mutta se on nähtävissä klikkaamalla ”old way” tekstiä (ks. kuvio 29). Vanha muoto on alustavasti piilotettu siksi, ettei se veisi huomiota nykyiseltä toteutustavalta.

Kuvio 29. Esimerkki “old way”

4.2.4 Practises

Practises-kohtaan on kerätty joitakin yleisiä ohjeita NgRx:n käyttöön: milloin käyttää NgRx-storea ja kuinka tyhmät ja viisaat -komponentit rakenne toimii.

4.2.5 Why do we use NgRx Store

Wikisivustolla on tärkeää myös selittää, miksi NgRx-toteutus on valikoitunut Landis+Gyrin tilanhallintamalliksi. Syiden ymmärtäminen vahvistaa myös yleistä aiheen ymmärtämistä. Angularille löytyy muitakin tilanhallintaratkaisuja, mutta miksi NgRx on juuri valikoitunut käytettäväksi Landis+Gyr:in sovelluksissa?

4.3 Esimerkkisovelluksen rakentaminen

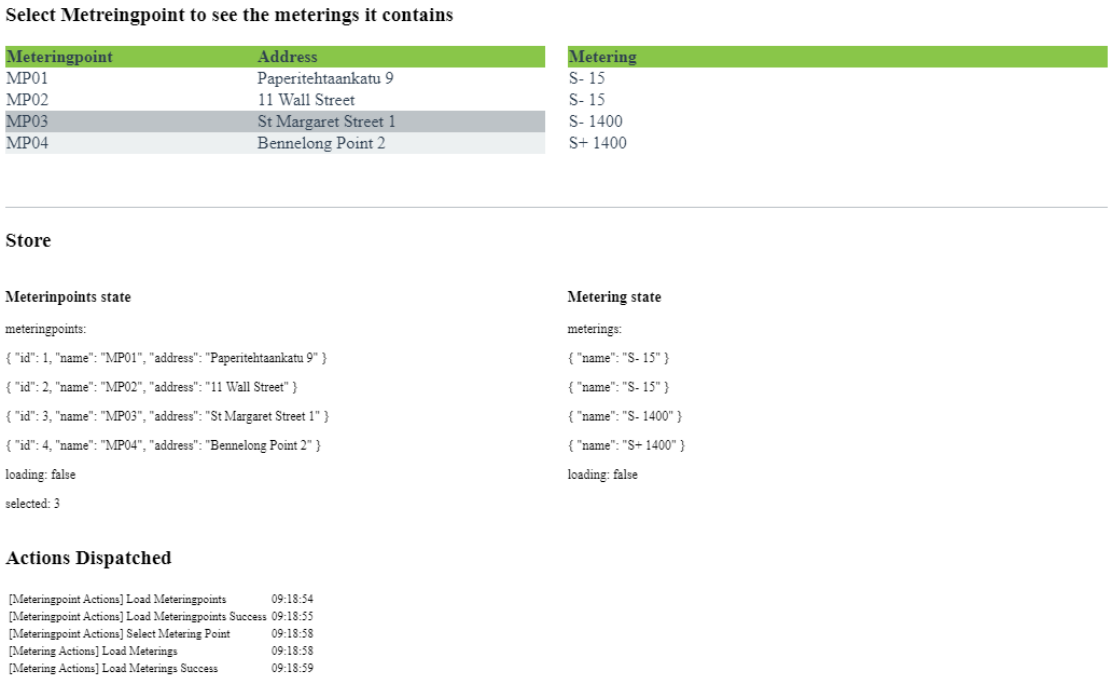
Wikinsivuston rinnalle on tarkoitus rakentaa myös esimerkkisovellus. Esimerkkisovellus toimii yksinkertaistettuna versiona Landis+Gyrin omista sovelluksista. Se on rakennettu pohjaten samoihin käytäntöihin ja sen toiminta on rakennettu NgRx-storen

ympärille. Sovellus rakennettiin Stackbliz-alustalle, joka on selainpohjainen sovelluskehitysalusta, joka tuo reaktiivisen kehitysympäristön selaimeen kaikkien ulottuville. Se mahdollistaa pienien sovellusten jakamisen nopeasti ja tehokkaasti. Alusta mahdollistaa sovelluksen toiminnan ja rakenteen tarkan seuraamisen. Koodiin voi myös tehdä käyttöliittymän kautta muutoksia, jotka välittömästi heijastuvat itse sovellukseen. Jos käyttäjällä ei ole lupaa tehdä muutoksia sovellukseen, muutokset eivät kuitenkaan tallennu sessioiden yli.

Stackbliz mahdollistaakin nopean sovelluksen tarkastelun ilman vaivalloisia ympäristöjen pystytyksiä. Sen avulla voidaan myös antaa pintakosketus - niin tuleville sovelluskehittäjille kuin ulkopuolisillekin – altistamatta kuitenkaan Landis+Gyrin omia sovelluksia ja omaisuutta ulkopuolisien saataville. Näin se mahdollistaa myös sovelluksesta otettujen esimerkkien käytön tässä opinnäytetyössä.

Itse sovellus on erittäin yksinkertainen, mutta hyvä esimerkki storen tärkeimpien osien toiminnasta (ks. kuvio 30). Se antaa hyvän ensikosketuksen NgRx:n peruskäsitteisiin, kuten tapahtumapohjaiseen ajatteluun ja storen rakenteen perusteisiin. Se myös antaa hyvän esimerkin siitä, kuinka efektit keskustelevat ulkopuolisten APIen kanssa ja kuinka ne integroidaan sovellukseen.

Sovelluksen käyttöliittymä rakentuu kolmesta osasta ja perustuu löyhästi Landis+Gyrin sovellusten teemaan. Ensimmäinen osa on mittapistetaulukko, jossa on listattuna yksinkertaisia mittapisteitä. Toinen osa on mittaritaulukko, jossa voidaan tarkastella valittuun mittapisteeseen kuuluvia mittareita. Kolmanneksi sovelluksen alaosassa olevasta osiosta voidaan seurata tapahtuvia tapahtumia ja storen ajankohtaista tilaa. Näin käyttäjä saa heti kosketuksen storen toimintaan - niin suoraan graafisesti kuin rakenteellisestikin - Stackblizin käyttöliittymän koodi-ikkunasta.



Kuvio 30. Stackbliz esimerkkisovellus.

Esimerkkisovellusta alustaessa laukaistaan Load Meteringpoints -tapahtuma, joka käsitellään efektissä. Efekti kutsuu meteringpoint -palvelua, joka vuorostaan simuloi API-kutsua odottaen ensiksi sekunnin ja palauttaen sitten jo ennalta määritetyn taulukon mittapisteitä. Load Metreingpoints -tapahtuma tekee jo alustavan muutoksen storen meteringpoints-haaraan ja vaihtaa meteringpoints-reducerissa loading-arvon todeksi. Käyttöliittymä reagoi storen loading-arvoon vaihtaen meteringpoints-taulukon tekstiin ”loading” näyttääkseen latauksen olevan käynnissä. Pitkittämällä meteringpoints-palvelun taulukon palautusta voidaan simuloida oikeaa API-kutsua ja nähdä latausindikaattori käyttöliittymässä. Kun palvelu on palauttanut mittapistetaulukon, se annetaan tietosisältönä Load Meteringpoints Success -tapahtumalle, joka laukaistaan efektin lopussa. Storen Meteringpoints-tilahaarasta vastaava reducer ottaa tapahtuman vastaan ja lisää mittapistetaulukon tilaansa. Tilamuutos laukaisee mittapistetilasta vastaavan selektorin, joka puolestaan toimittaa uusimman tilan metreingpoints.container-komponentille. Tämä komponentti toimittaa taulukon

asynkronisesti jälleen eteenpäin meteringpoints-table-komponentille, joka vihdoin tuo arvot käyttöliittymään tarkasteltaviksi.

Mittapistetaulukosta voidaan valita mittapisteitä, valinta laukaisee Select Metreingpoint -tapahtuman. Tapahtuman tietosisällöksi annetaan valitun mittapisteen id. Tapahtuman vastaanottaa efekti, joka puolestaan laukaisee Load Meterings -tapahtuman, jonka tietosisällöksi välitetään annettu id. Tästä eteenpäin toiminta onkin lähes identtinen mittapisteiden noudon kanssa. Erona on se, että meterings-palvelu sisältää avainparitaulukon, josta palautetaan annetun aikamääreen jälkeen avainta vastaava mittaritaulukko. Lopulta valittua mittapistettä vastaavat mittareiden arvot löytävät tiensä mittaritaulukkoon käyttöliittymässä.

Mittapiste valintaesimerkin lisäksi sovelluksen käyttöliittymään kuuluu vielä storen tilaa seuraava osio. Osio koostuu kahdesta osasta. Toisessa seurataan storen ajankohtaista tilaa ja toisessa listataan kaikki sovelluksessa lähetetyt tapahtumat reaaliajassa. Storessa on mittapiste- ja mittarihaaran lisäksi vielä yksi haara, joka listaa kaikki tapahtumat. Haarasta vastaava efekti kuuntelee kaikkia sovelluksessa lähetettyjä tapahtumia ja lähettää jokaisen tapahtuman jälkeen Add Action -tapahtuman. Add Action -tapahtuman tietosisällöksi annetaan kuunnellun tapahtuman tyyppi ja siitä vastaavassa reducerissa se lisätään objektina tapahtumataulukkoon. Jokaiselle tapahtumaobjektille lisätään myös sen lähetysajankohdan kellonaika. Käyttöliittymässä näytetään taulukko, jossa ovat listattuina laukaistujen tapahtumien tyypit ja kunkin tapahtuman laukaisuhetken kellonaika. Sovelluksen store-osio antaa näin käyttäjälle eheän kuvan storen toiminnasta ja sen tilan sisällöstä, kuten myös siitä, miten storen tila ja tapahtumat heijastuvat käyttöliittymään.

5 Työn tulokset

5.1 Työn tavoite

Työn tavoitteena oli koota Landis+Gyrin käyttöön kattava dokumentaatio NgRx-store-kirjaston käytön perusteista ja myös sen käyttämisestä Landis+Gyrin sovelluksissa ja niiden kehitystyössä. Opinnäytetyönä laadittiin tähän tarkoitukseen wikisivusto, joka antaa kattavan katsauksen NgRx-kirjaston käytöstä. Wikisivusto on suunnattu niin uusille Landis+Gyrin kehittäjille NgRx:n alkeiden oppimiseen ja heidän oppimisprosessinsa tukemiseen kuin myös niille kokeneemmille kehittäjille, jotka tarvitsevat tietoa erityisesti juuri Landis+Gyrin käytänteistä.

NgRx-kirjaston toimintaa käydään läpi käyttäen hyväksi oikeita esimerkkejä Landis+Gyrin sovelluksista. Tämä tukee oppimisprosessia ja kasvavaa ymmärtämistä sekä myös myöhemmin ymmärryksen syventämistä.

Wkisivuston rinnalle luotu esimerkkisovellus antaa hyvän ja yksinkertaisen pohjan aiheeseen tutustuville. Se antaa esimerkkipohjan käyttäen juuri Landis+Gyrin käytänteitä. Stackbliz mahdollistaa koodin ja reaaliaikaisen sovelluksen tarkastelun yhdessä kätevässä ikkunassa. Käyttäjä voi myös tehdä vapaasti muutoksia esimerkkisovellukseen kokeillen, miten eri muutokset vaikuttavat itse sovelluksen toimintaan

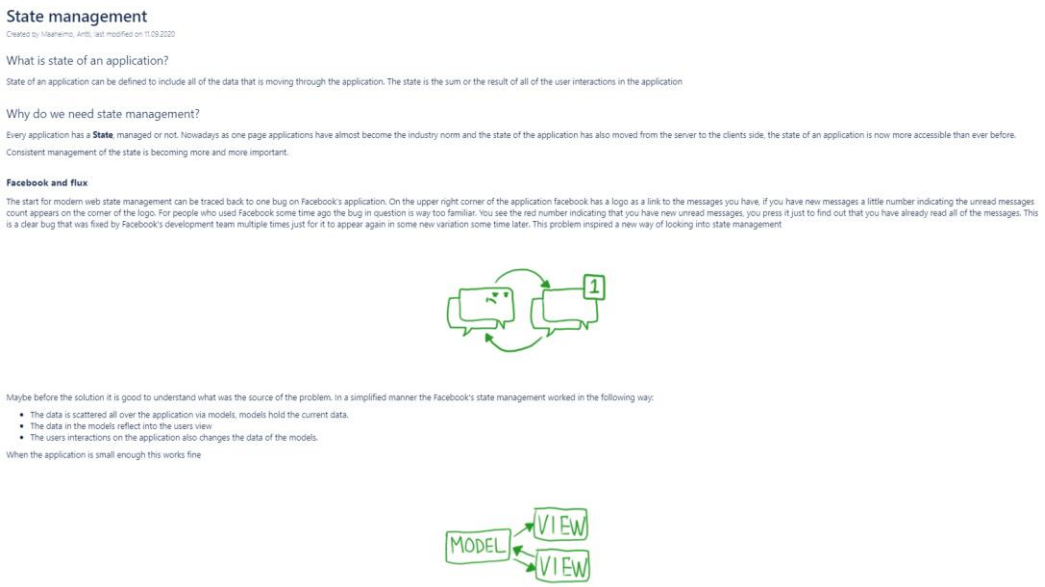
5.2 Wikisivusto

Lopullisen työn tuloksena syntynyt wikisivusto koostuu kymmenestä osasta (ks. kuvio 31). Osien välillä voi navigoida niin Landis+Gyrin wikin omasta wikirakenteesta sivun vasemmalta puolelta tai vaihtoehtoisesti wikin hierarkiassa ylimmältä sivulta ”NgRx Store”.

- State management
- Structure
  - Actions
  - Reducers
  - Selectors
  - Effects
- NgRx-devtools
- Practises
- Why do we use NgRx store
- Example application

Kuvio 31. Sisällysluettelo.

Wikisivuston eri osissa käytetään - niin sisällön selkeyttämiseksi kuin sen keventämiseksikin - vihreällä teemalla piirrettyjä kuvia (ks. kuvio 32, 33). Kuvat on laatinut opinnäytteen tekijä itse wiki sivustoa varten ja niiden tehtävänä on konkretisoida sivustolla esitettyjä asioita lukijalle.



State management

Created by Maaheimo, Antti, last modified on 11.09.2020

What is state of an application?

State of an application can be defined to include all of the data that is moving through the application. The state is the sum or the result of all of the user interactions in the application

Why do we need state management?

Every application has a **State**, managed or not. Nowadays as one page applications have almost become the industry norm and the state of the application has also moved from the server to the clients side, the state of an application is now more accessible than ever before.

Consistent management of the state is becoming more and more important.

Facebook and flux

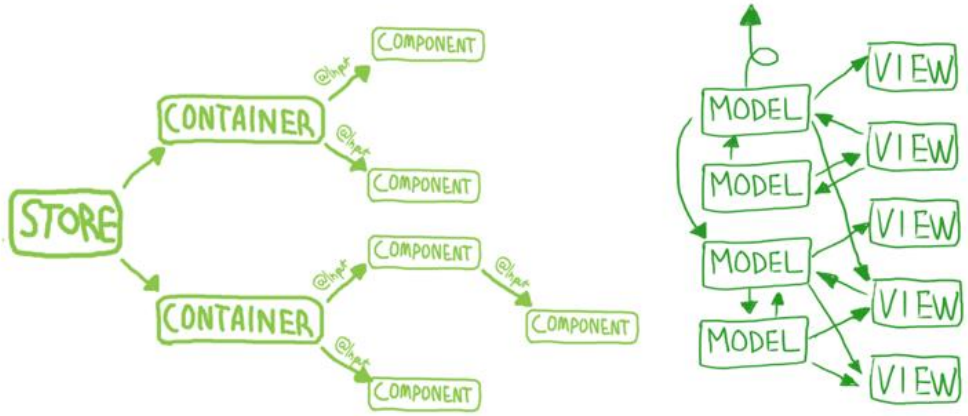
The start for modern web state management can be traced back to one bug on Facebook's application. On the upper right corner of the application facebook has a logo as a link to the messages you have, if you have new messages a little number indicating the unread messages count appears on the corner of the logo. For people who used Facebook some time ago the bug in question is way too familiar. You see the red number indicating that you have new unread messages, you press it just to find out that you have already read all of the messages. This is a clear bug that was fixed by Facebook's development team multiple times just for it to appear again in some new variation some time later. This problem inspired a new way of looking into state management

Maybe before the solution it is good to understand what was the source of the problem. In a simplified manner the Facebook's state management worked in the following way:

- The data is scattered all over the application via models, models hold the current data.
- The data in the models reflect into the users view
- The users interactions on the application also changes the data of the models.

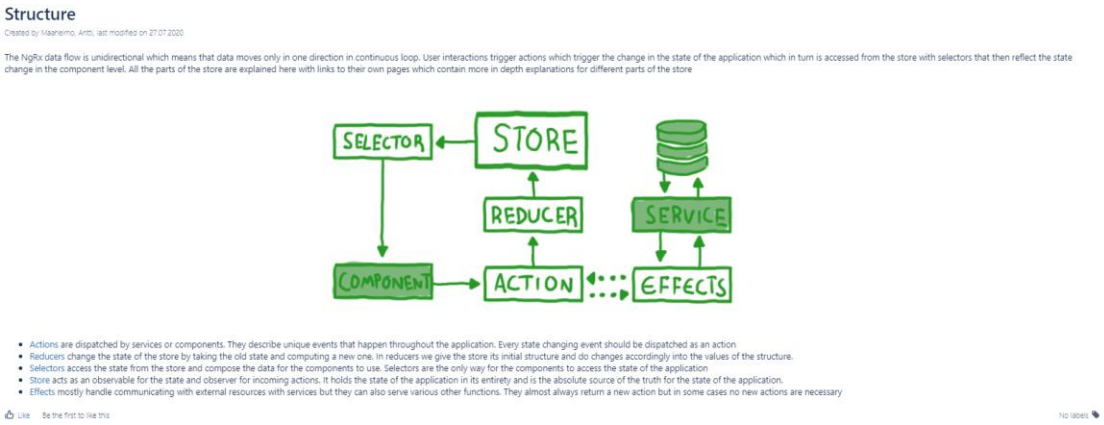
When the application is small enough this works fine

Kuvio 32. Wikin state management -luku.



Kuvio 33. Esimerkki käytetyistä kuvioista

Structure-sivu (ks. kuvio 34) sisältää paitsi kaavion storen toiminnasta myös yksinkertaistetut selitykset sen eri osien toiminnasta. Sivulta voi myös navigoida jokaisen osan spesifiseen dokumentaatioon, joissa jokainen osa käsitellään yksityiskohtaisemmin.



Kuvio 34. Structure.

Jokainen storen osa käydään läpi yksityiskohtaisesti. Eri osien toimintaperiaatteita käydään läpi käyttäen hyväksi Landis+Gyrin sovelluksista otettuja esimerkkejä. Esimerkit puretaan pienempiin osiin ja käydään ne läpi pala palalta (ks. kuvio 35, 36).

Effects

Created by Maaheimo, Antti, last modified on 11.09.2020

In service based Angular applications, components handle interacting with external resources trough services. In NgRx the effects are a way to isolate these interactions from the components and make them part of the NgRx architecture and data flow. Usually in our application their main purpose is to call for external API's trough services and pass the gotten data forward with a new action. Effects can also be used in many other ways but usually they isolate effects to other actions that **return new actions** as a result.

- Effects isolate side effects from components allowing us to have more pure components
- Effects are listeners of observable streams that continue until an error or completion occurs
- They listen for specific actions and
- Effects return new actions at the end of their task (most of the time)

As ngrx is based on RxJs you have most likely already seen rxjs pipes. In the effects the whole effect operations are based on the rxjs pipes which can be quite hard to understand on the first glimpse. RxJs pipes let you combine multiple functions into a single function making working with observables a little clearer and more efficient. More on the RxJs Pipes: https://rxjs.dev/guide/v6/pipeable-operators

Also a good article about the operators: https://medium.com/@luukgruijs/understanding-rxjs-map-mergemap-switchmap-and-concatmap-833fc1fb09ff

Examples

Here's loadMeteringPointMeterings$. The task of this effects is to isolate the external API call into this action where we call it trough a service. When we do the API calls like this We can isolate the whole event to two parts which makes the whole operation clearer. Like this we can isolate it to actions: *load* and then *load success* or load fail. The load is triggered by the user interactions on the UI. The load action is then caught by the effect where we call the a service to fetch the data. The service fetches the data from the external API and returns it to the effect where we return new action. The success action with the fetched data or the fail action if something in the API call went wrong. Fetching data is the most common use case of the effects in our application but they can be used in various cases where we need to trigger new actions as a result of some specific ones.

```
loadMeteringPointMeterings$ = createEffect(() =>
  this.actions$.pipe(
    ofType(MeteringsActions.LoadMeteringPointMeterings),
    map(action => action.meteringPointId),
    mergeMap(meteringPointId => {
      return this.meteringsService.getMeteringPointMeterings(meteringPointId).pipe(
        map(meterings =>
          MeteringsActions.LoadMeteringPointMeteringsSuccess({
            meteringPointId: meteringPointId,
            meterings: meterings
          })
        ),
        catchError(error => of(MeteringsActions.LoadMeteringPointMeteringsFail({ meteringPointId })))
      );
    })
  )
);
```

1. We define the name of the effect loadMeteringPointMeterings$.
2. We construct our effect inside createEffect() function
3. We use the rxjs's pipe function on the actions$ variable which we have defined in the constructor: `constructor(private actions$: Actions, private meteringsService: MeteringsService) {}` With this we can listen to the action stream and act accordingly on specific type of actions.
4. And we isolate this effect to be triggered from only this specific type of an action
5. Now that we have the action we isolate the meteringPointId from it
6. And with mergemap we take that id
7. We call for meteringsService that handles the external API interaction and return us the data gotten from the API
8. With the returned data we then call for action: LoadMeteringPointMeteringsSuccess giving the gotten data and the meteringId as its parameters
9. If something in the process goes wrong the cathError() dispatches LoadMeteringPointMeteringsFail action

Kuvio 35. Ote Effects-sivulta.

Selectors

Created by Maaheimo, Antti, last modified on 03.08.2020

Selectors are the only way to access the store data. As the raw store state might not be in suitable form for the components to use, we can refine the data to a more suitable form and then send it to the components already in a usable form. All this is done inside the selectors which are just pure functions which can access the store. Selectors return **observable**s, so thanks to **RxJs** the data gotten from the selectors in the components is always up to date with the store, as as soon as the store value changes the selectors get notified and send the component the latest state.

Selectors structure

In the selector below:

1. We first specify the name of the selector
2. We use the createSelector() function to compile it
3. Inside we first specify the part of the state or the selector(s) we want to use
4. We specify the names that we want to use on them and their types
5. And do basically what ever we want with them, returning the result

```
export const getMeteringPointsState = createSelector(
  getMeteringPointsFeatureState,
  (state: MeteringPointsFeatureState) => state.meteringpoints
);
```

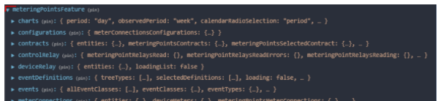
Again as for all of the store parts selector files have been divided by the topics in the application.

In all of the selectors the first thing that we do is accessing the sate branch in question. In the picture above we are in the **meteringpoints.selectors.ts** file so we want to access the **meteringpoints** branch that is a child of the **meteringPointsFeature** state. The feature state is imported from the reducers.

After this we can access the child states of the meteringPointsState as below:

```
export const getMeteringPointsEntities = createSelector(getMeteringPointsState, state => state.entities);
export const getMeteringPointsChecked = createSelector(getMeteringPointsState, state => state.checked);
export const getMeteringpointDetails = createSelector(getMeteringPointsState, state => state.details);
export const getMeteringPointsSelected = createSelector(getMeteringPointsState, state => state.selected);
export const getMeteringPointsSearchResults = createSelector(getMeteringPointsState, state => state.searchResults);
export const getMeteringPointsCustomProperty = createSelector(getMeteringPointsState, state => state.customPropertyEntities);
```

The corresponding sates in the state tree:

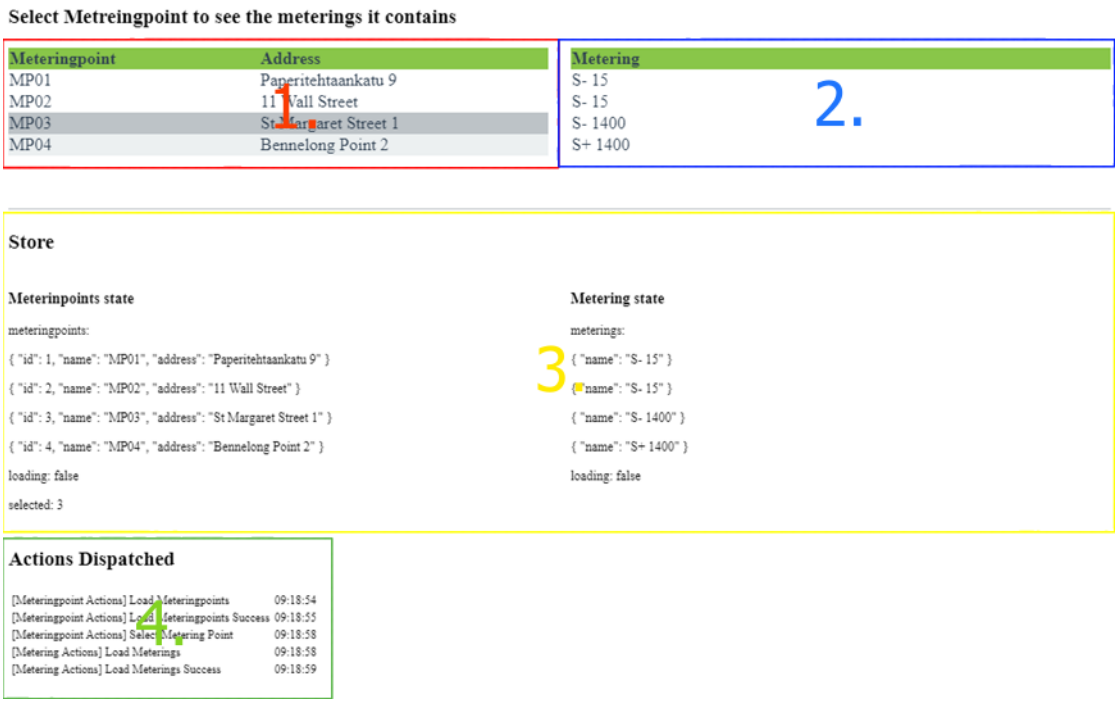


Kuvio 36. Ote Selectors-sivulta.

5.3 Esimerkkisovellus

Lopullinen Stackbliz-sivulle toteutettu NgRx:n perustoiminnallisuutta esittelevän sovdemosovellus on toteutettu Landis+Gyrin hengessä sähkömittariteemalla. Sovelluksen käyttöliittymä koostuu seuraavista neljästä osasta (ks. kuvio 37):

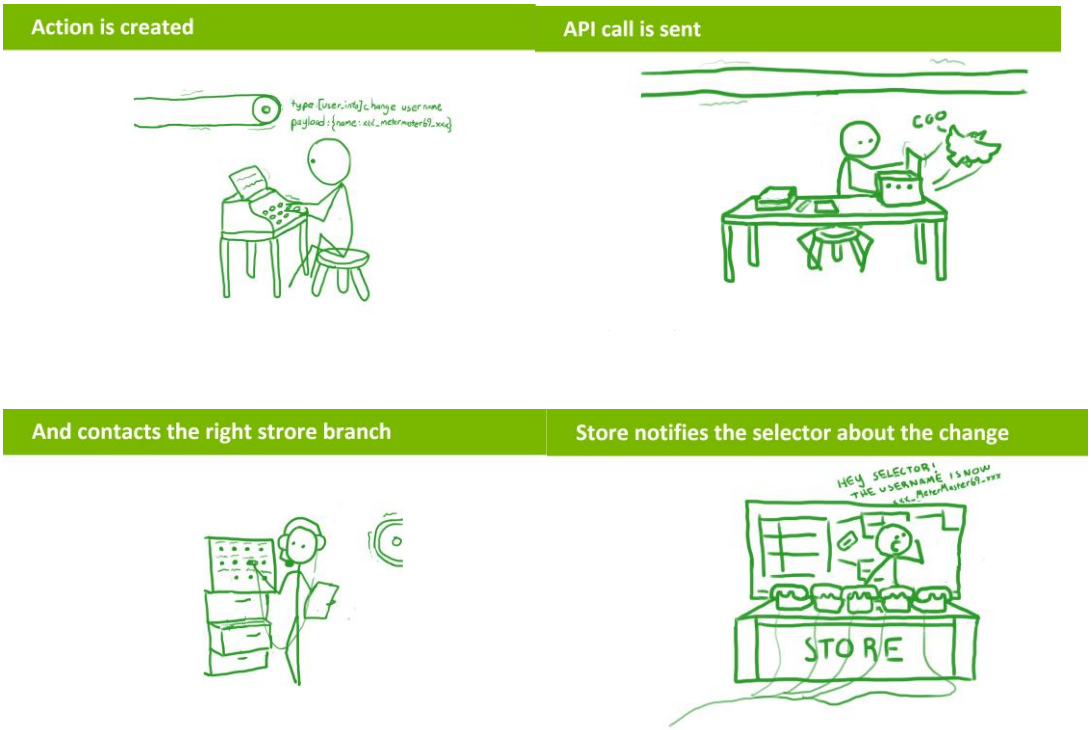
1. Mittapistetaulukosta käyttäjä voi valita mittapisteitä.
2. Jokaisella mittapisteellä on mittareita. Valitun mittapisteen mittarit näytetään mittaritaulukossa.
3. Storen ajankohtaista tilaa voi seurata käyttöliittymän store-osiosta. Osiossa näytetään sovelluksen kaksi tilaobjektia: mittapistetila ja mittaritila. Käyttäjän interaktiot taulukoissa heijastuvat suoraan sovelluksen tilaan ja näin myös näihin objekteihin.
4. Kaikki sovelluksen NgRx-elinkaaren tapahtumat listataan sovelluksessa alimpana. Käyttäjä voi seurata, mitä tapahtumia käyttöliittymän muutokset laukaisevat. Taulukossa näytetään jokaisen tapahtuman tyyppi ja laukaisuajankohta.



Kuvio 37. Esimerkkisovellus paloiteltuna.

5.4 NgRx datavirta animaatio

Lisäksi wikiin on lisätty animaatio, jonka tarkoituksena on konkretisoida NgRx:n datavirtaa. Storen toimintaa kuvaamaan on luotu sen eri osia imitoivat hahmot. Lisäksi jokaiselle storen osalle on luotu sitä vastaava hahmo. Animaatiohahmot käyvät esimerkkitapauksena läpi käyttäjän käyttäjänimen vaihtotapahtuman (ks. kuvio 36).



Kuvio 38. Otteita animaatiosta.

Animaatio koostuu 27 kuvasta, jotka alkavat tapahtuman lähetyksestä ja päättyvät selektorin datan välitykseen komponentille.

6 Johtopäätökset ja pohdinta

6.1 Tavoitteet ja johtopäätökset

NgRx on tehokas, mutta kompleksinen tilanhallinta ratkaisu suuren kokoluokan web-sovelluksiin. Se yhtenäistää ja yksinkertaistaa sovellusten tilanhallintaa. Tehokkaan tilanhallinnan hyödyt sovelluskehityksessä ovat kiistattomat. NgRx tuo valtavasti hyötyä Landis+Gyrin sovellsukehitykseen.

Tämän opinnäytetyön laatimisen tavoitteena oli laatia wikisivusto, joka auttaisi NgRx-kirjaston ja Landis+Gyrin sovellusten toimintaperiaatteita ja käytäntöjä opettelevia sovelluskehittäjiä. Lopputuloksena syntynyt wikisivusto antaa hyvän tuen kirjaston kanssa aloittaville sovelluskehittäjille. Se luo yhtenäisen paikan Landis+Gyrin sovelluskehityskäytänteiden oppimiseen ja dokumentointiin.

Samoin kuin NgRx keskitettynä tilanhallintaratkaisuna tuo koko sovelluksen tilan yhteen ja ainoaan paikkaan, luodun dokumentaation tehtävä on tuoda NgRx:n käyttöä ja käytänteitä keräävä dokumentaatio yhteen ja ainoaan paikkaan. Tapoja kirjaston implementointiin ja käyttöön on käytännössä rajattomasti, mutta yhtenäinen dokumentaatio luo pohjan kestävälle ja tehokkaalle sovelluskehitykselle.

6.2 Lähteiden käyttö

NgRx-kirjastoa koskevan tiedonkeruu perustui tässä opinnäytetyössä lähes yksinomaan NgRx:n omaan dokumentaatioon. Lähes kaikki aiheesta kirjoitettu tieto ja tehokkaiden käytänteiden jakaminen perustuu erilaisiin epävirallisiin blogikirjoituksiin ja artikkeleihin. Landis+Gyrin sovellukset käyttävät käytänteitä, jotka on kerätty monista blogeista, artikkeleista ja kursseista. Monet käytänteistä ovat myös uniikkeja

Landis+Gyrin sovelluskehitykseen. Niiden toimivuus on todennettu käytännön kautta. Usein kehitystavat ovatkin rakentuneet kehityksessä kehittäjien erheiden ja käytännön kokemusten kautta. Käytänteet ovat yhdistelmä kaikkea mainittua ja lopullinen lähteiden löytäminen kaikille käytänteille on käytännössä jälkikäteen mahdotonta. Käytänteet elävät jatkuvasti ja niitä pyritään parantamaan mahdollisuuksien mukaan. Täten vaikka itse opinnäytetyössä tuodaankin esiin joitakin hyväksi todettuja käytänteitä, selitetään niitä tarkemmalla tasolla itse wikisivustolla.

6.3 Sisällön kriittinen pohdinta

Opinnäytetyön ja tuotetun wikisivuston sisältö on kirjoitettu niille sovelluskehittäjille, jotka haluavat tietää, miten NgRx:ää käytetään. Wikisivusto vastaa kysymyksiinn kattavasti ja antaa tukea kehittäjien oppimisprosessiin. Wikisivusto ei kuitenkaan syvenny siihen, miten NgRx toimii syvemmällä tasolla. Esimerkiksi koko storen toiminta pohjautuu RxJS-kirjastoon ja sen asynkronisiin metodeihin, mutta lukija saa kirjastosta toiminnasta vain sen käytön kannalta olennaista informaatiota. Wikisivusto sivuuttaakin lähes kaiken, mitä ei olla todettu välttämättömäksi NgRx:n oppimisessa tai käytössä.

Auttaisiko käsitteiden teknisempi läpikäynti - vai aiheuttaisiko se hänelle jo valmiiksi kompleksisen kokonaisuuden hahmottamisessa ongelmia? Tarvitsisiko store-toteutuksen kehittäjän tuntea teknologiaa syvemmällä tasolla, vaikka tietämys ei vaikuttaisikaan sovelluskehitysprosessiin? Wikisivusto on tarkoituksella laadittu siten, että asiat on käsitelty mahdollisimman selkeästi. Voidaan tietysti pohtia sitä, tarvitsisiko teknologioiden taustoja käydä wikisivustossa tarkemmin läpi, mutta tähän ei mielestäni ole tarvetta. NgRx-storeen tutustuva kehittäjä saa wikisivustosta kaiken tarvitsemansa sovelluskehitykseen NgRx-kirjastolla toteutetun tilanhallinnan omaavaan sovellukseen.

6.4 Jatkokehitys

Wikisivusto antaa hyvän pohjan sekä NgRx:ään tutustuville, että Landis+Gyrin sovelluksien toteutukseen tutustuville. Sivusto on ensisijaisesti kuitenkin laadittu uusien työntekijöiden käyttöön tai sellaisille sovellusten kanssa työskentelemään siirtyville, joilla ei ole aiempaa kokemusta tilanhallinnasta NgRx-kirjastoa käyttäen.

Vaikka kompleksisimpia asioita on jätetty pois syystä, se ei kuitenkaan tarkoita, ettei niiden lisääminen toisi lisäarvoa sen lukijoille. Wiki vain itsessään sisältää jo käsitteitä ja uusia ajatusmalleja, joiden sisäistäminen ja ymmärtäminen yksinkertaisesti on aikaa kuluttavaa. Kompleksisimman kerroksen lisääminen wikiin voisi hyvinkin olla suotavaa tulevaisuudessa, jos sen tuominen wikiin onnistuisi kestävällä ja ymmärrettävällä tavalla.

Luonnollisesti Landis+Gyrin sovellukset, kuten kaikki jatkuvassa kehityksessä olevat sovelluksetkin, tulevat muuttumaan. Uusia käytänteitä ja toteutustapoja tullaan varmastikin jatkuvasti hyödyntämään sovelluskehityksessä. NgRx Store-tilanratkaisutoteutus ja sen käytänteet tulevat varmasti myös elämään ajan kuluessa. Myös itse NgRx-kirjasto on koko elinkaarensa aikana muuttunut huomattavasti ensimmäisestä implementaatiosta ja tulee myös muuttumaan jatkuvan kehitystyön seurauksena.

Vaikka wikisivusto on nyt ajan tasalla toteutustavoista ja kirjaston käytöstä, tulee se varmasti vaatimaan jatkuvaa ylläpitoa aiheen ja sovellusten eläessä.

Lähteet

Actions. NgRx dokumentaatio. Viitattu 20.9.2020. https://ngrx.io/guide/store/actions

Angular Application Architecture - Building Flux Apps with Redux and Immutable.js Artikkeli Angular university sivustolla. Viitattu 10.9.2020. https://blog.angular-university.io/angular-2-application-architecture-building-flux-like-apps-using-redux-and-immutable-js-js/

AsyncPipe. Angular dokumentaatio. Viitattu 4.9.2020. https://angular.io/api/common/AsyncPipe

Duffy, J. Operators 2004. State-Management. Artikkeli CoDe magasine lehden sivustolla. Viitattu 10.9.2020. https://www.codemag.com/Article/0409061/State-Management

Freeman, A. 2020. Pro Angular 9: Build Powerful and Dynamic Web Apps, Fourth Edition. Apress. Viitattu 28.9.2020. https://janet.finna.fi, books24x7

Gackenheimer, C. 2015. Introduction to React. Aprress. Viitattu 29.10.2020. https://janet.finna.fi, books24x7

@ngrx/effects. NgRx dokumentaatio. Viitattu 20.9.2020. https://ngrx.io/guide/effects

Ngrx Store - An Architecture Guide. Artikkeli Angular university sivustolla. Viitattu 1.9.2020. https://blog.angular-university.io/angular-ngrx-store-and-effects-crash-course/

Observable. RxJS-dokumentaatio. Viitattu 29.10.2020. https://rxjs-dev.firebaseapp.com/guide/operators

Operators. RxJS-dokumentaatio. Viitattu 29.10.2020. https://rxjs-dev.firebaseapp.com/guide/operators

Reducers. NgRx dokumentaatio. Viitattu 20.9.2020. https://ngrx.io/guide/store/reducers

Update on Adobe Flash Player End of Support. Windows Blogs. Blogi kirjoitus Microsoftin sivuilla. Viitattu 21.9.2020. https://blogs.windows.com/msedgedev/2020/09/04/update-adobe-flash-end-support/

Why Use React Redux?. React Redux. React Redux dokumentaatio. Viitattu 5.9.2020. https://react-redux.js.org/introduction/why-use-react-redux