jamk.fi

Angular-sovelluksen keskitetty tilanhallinta NgRx-kirjastolla

Santtu Sarlin

Opinnäytetyö
Helmikuu 2021
Luonnontieteiden ala
Tradenomi (AMK), tietojenkäsittelyn tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Sarlin, Santtu	Opinnäytetyö, AMK	Helmikuu 2021
	Sivumäärä 41	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: X

Työn nimi

Angular-sovelluksen keskitetty tilanhallinta NgRx-kirjastolla

Tutkinto-ohjelma

Tietojenkäsittelyn tutkinto-ohjelma

Työn ohjaaja(t)

Niko Kiviaho

Toimeksiantaja(t)

Skillwell Oy

Tiivistelmä

Modernit verkkosovellukset ovat ajan kanssa kasvaneet yhä monipuolisemmiksi niiden korvatessa vanhat työpöytäsovellukset. Tutkimuksen toimeksiantaja Skillwell Oy kehittää asiakkailleen monipuolisia web-käyttöliittymiä sisältäviä SaaS-palveluita, käyttäen Angular-sovelluskehystä sekä NgRx-kirjastoa. NgRx-kirjasto ei ole ollut toimeksiantajalla käytössä pitkään, joten toimeksiantaja halusi lisää oppia ja ymmärrystä sen käytöstä.

Tutkimuksen tavoitteena oli tutkia verkkosovellusten tilanhallintaa yleisellä tasolla, sekä keskitetyn tilanhallinnan toteuttamista Angular-verkkosovelluksiin NgRx-kirjastoa hyödyntäen, sekä saavuttaa syvempää ymmärtämistä aihetta kohtaan. Tutkimuksessa tutkittiin toimeksiantajan kehittämiä verkkosovelluksia ja analysoitiin, kuinka niissä on NgRx:llä ratkaistu erilaisia tilanhallinnallisia ongelmia. Analysoitavat sovellukset ja kohteet päätettiin yhdessä toimeksiantajan kanssa.

Tutkimus toteutettiin kvalitatiivisena tutkimuksena. Siinä pyrittiin selvittämään, mistä Angular-verkkosovelluksen tila kokonaisuudessaan koostuu, sekä pyrittiin tutkimaan oikeita menetelmiä ja käytänteitä keskitetyn tilanhallinnan toteuttamisessa NgRx-kirjastolla.

Tutkimuksen tuloksena syntyi paljon tietoa verkkosovellusten tilanhallinnasta, sekä varmennus siitä että toimeksiantajan kehitysympäristössä on noudatettu oikeita käytänteitä tilanhallinnan kehityksessä.

Johtopäätös tutkimuksesta on, että globaali keskitetty tilanhallinta on laajoihin ja monimutkaisiin verkkosovelluskokonaisuuksiin hyvä ratkaisu. Sillä voidaan välttää kehitykseen liittyviä ongelmia ja helpottaa monipuolisten sovellusten kehitystä ja ylläpitoa.

Avainsanat (asiasanat)

Angular, NgRx, tilanhallinta, verkkosovellus, sovelluskehitys

Muut tiedot (Salassa pidettävät liitteet)

jamk.fi

Description

Author(s)	Type of publication	Date
Sarlin, Santtu	Bachelor's thesis	February 2021
		Language of publication: Finnish
	Number of pages: 41	Permission for web publication: X

Title of publication

Angular-application state management with NgRx-library

Degree programme

Business Information Technology

Supervisor(s)

Kiviaho, Niko

Assigned by

Skillwell Oy

Abstract

As time has gone by, modern web applications have grown to be complicated entities as they are replacing old desktop applications. Skillwell Oy, which is the assignor of this research develops diverse SaaS-services that contain web interfaces for web applications, using Angular framework and NgRx library. The assignor wanted to deepen the knowledge of NgRx and get a better understanding on the usage of the technology.

The goal of this research is to study state management in web applications on a general level and research the development of a centralized global state management store in Angular web applications utilizing the NgRx library, and also to achieve a deeper understanding and better knowledge on the subject. This research analyzed solutions of handling several state management problems and used web applications developed by the assignor as it's analyzation material. Subjects to be analyzed was selected together with the assignor.

The research was completed as a qualitative research. The goal was to study, what makes up the web application state in an Angular web application, and to study the correct methods in developing a central state management with the NgRx library.

The outcome from this research developed a lot of new understanding and knowledge regarding state management in web applications, and a confirmation that the assignor has been following correct methods in development with NgRx.

The conclusion of the research is that for diverse and scaled projects a centralized global state management is a good solution. It can help avoid a lot of problems in web application development and make it easier to develop and maintain complicated applications.

Keywords/tags (subjects)

Angular, NgRx, state management, web application, web development

Miscellaneous (Confidential information)

Sisältö

Kuviot

Käsitteet

Angular: Googlen kehittämä ohjelmistokehys verkkosovellusten kehittämiseen.

API (Application Program Interface: Ohjelmointirajapinta, joka mahdollistaa sovelluksen kommunikoinnin ulkoisten palveluiden kanssa.

HTML (Hypertext Markup Language): Verkkosivun rakennetta kuvaava ohjelmointikieli.

JavaScript: Selaimessa toimiva ohjelmointikieli, jonka avulla voidaan kehittää verkkosovelluksiin toiminnalisuuksia.

NgRx: RxJs pohjainen kokoelma kirjastoja keskitetyn tilanhallinan toteuttamista varten, joka perustuu Redux-arkkitehtuuriin

REST API (Representational State Transfer): Arkkitehtuuri, joka mahdollistaa tiedon lukemisen ja kirjoittamisen palvelimelta HTTP-pyynnöillä.

RxJs: JavaScript kirjasto, jolla mahdollistetaan reaktiivinen ohjelmointi.

Typescript: JavaScript ekstensio joka lisää ohjelmointikieleen tyyppimääritykset.

Verkkosovellus: Selaimessa toimiva sovellus.

WebSocket: Muodostaa selaimen ja palvelimen välille jatkuvan kaksisuuntaisen yhteyden.

1 Johdanto

Yhä enemmän maailma on siirtynyt siihen suuntaan, että verkkoselaimilla toimivat verkkosovellukset korvaavat vanhat työpöytäsovellukset. Työpöytäsovellukset ovat tavanomaisesti perinteisiä verkkosivuja ja verkkosovelluksia monimutkaisempia kokonaisuuksia. Monimutkaisempien verkkosovellusratkaisujen kehittämiseksi voidaan hyödyntää erilaisia sovelluskehyksiä ja kirjastoja.

Angular on Googlen kehittämä sovelluskehys modernien monipuolisten verkkosovellusten kehittämistä varten. Sovellusten kuitenkin kasvaessa entistä isommiksi kokonaisuuksiksi, ja kehittyessä entistä dynaamisemmiksi sekä reaktiivisemmiksi, tulee Angularilla itsessään haasteita tilanhallinnan kehityksessä ja sen ylläpidossa. Tätä varten on olemassa NgRx-kirjasto, jolla voidaan toteuttaa reaktiivinen Redux-mallin tyyppinen tilanhallinta Angular-verkkosovellukseen. Keskitetyllä tilanhallinnalla voidaan helpottaa monimutkaisten sovelluskokonaisuuksien kehittämistä eristämällä kaikki sovelluksen tilan logiikkaan liittyvän toiminnan erilliseksi osaksi muusta sovelluksen toimintalogiikasta.

Tämän opinnäytetyön toimeksiantaja kehittää web-käyttöliittymiä sisältäviä monipuolisia SaaS-palveluita (engl. Software as a Service) asiakkailleen. Tässä työssä tutkitaan ja syvennytään verkkosovellusten tilanhallintaan ja sen toteuttamiseen Angular-verkkosovelluksissa NgRx-kirjastolla.

2 Tutkimusasetelma

Tässä luvussa kerrotaan, mitä ilmiötä työssä tutkitaan, millä tutkimusmenetelmällä ilmiötä tutkitaan sekä mikä on tutkimuksen tavoite.

2.1 Opinnäytetyön toimeksiantaja ja tavoitteet

Tämän opinnäytetyön toimeksiantaja on Skillwell Oy. Skillwell Oy on verkko-ohjelmistoja, liiketoiminnan kehittämisen ratkaisuja sekä muita digitaalisia palveluita kehittävä yritys. Skillwell Oy on myös virallinen Amazon Web Services (AWS) -sertifioitu konsultti.

Toimeksiantaja kehittää erinäisiä verkkosovelluksia AWS-ympäristössä joiden kehittämisessä käytetään Angular-sovelluskehystä. Näissä verkkosovelluksissa hyödynnetään NgRx-kirjastoa sovelluksen sisäiseen globaalin tilanhallintaan. Tämän opinnäytetyön tavoittena on tutkia tilanhallintaa verkko-sovelluksissa sekä sen toteuttamista Angular-sovelluksessa käyttäen NgRx-kirjastoa. Tutkitun tiedon ja sen analysoinnin perusteella tavoitteena on kehittää syvempää ymmärrystä aihetta kohtaan ja havaita mahdollisia parannuksia nykyisiin toimenpiteisiin ja käytäntöihin NgRx-tilanhallinnan toteutuksessa.

2.2 Tutkimusongelma ja -kysymykset

Tutkimusongelma on, miten Angular sovellukseen voidaan toteuttaa tehokas keskitetty tilanhallinta käyttäen NgRx-kirjastoa.

Tutkimuskysymykset ovat:

- Mitä tilanhallinta verkkosovelluksessa tarkoittaa?
- Mikä on NGRX?
- Kuinka keskitetty tilanhallinta voidaan toteuttaa toimeksiantajan teknologiaympäristössä?

2.3 Tutkimusmenetelmä ja aineisto

Tutkimuksessa käytetään kvalitatiivista eli laadullista tutkimusmenetelmää. Laadullisessa tutkimuksessa pyritään johtopäätöksiin ilman tilastollisia menetelmiä tai muita määrällisiä keinoja. Tällä menetelmällä pyritään ymmärtämään ilmiö

syvällisesti ja tuottaa kerätystä aineistosta syvällinen ja rikas tulkinta tutkimuksen tuloksissa. (Kananen 2008, 24.)

Tämän opinnäytetyön tavoitteena on syventyä tilanhallintaan teoriatasolla, ja analysoida toimeksiantajan kehittämiä verkkosovelluksia peilaten tämänhetkisiä toteutuksia uuten syvempään teoreettiseen oppiin. Kvalitatiivinen tutkimus sopii tämän opinnäytetyön tutkimusmentelmäksi, koska työn tavoitteena on kehittää syvällisempää ymmärrystä tilanhallintaa kohtaan ilmiönä sekä havaita mahdollisia laadullisia puutteita nykyisissä verkkosovellusten toteutuksissa.

Aineistoa tutkimukseen kerätään myös haastattelulla toimeksiantajan kanssa. Haastattelun tavoitteena on kerätä aineistoa ja määritellä tutkimuksen analysoitavat kohteet. Haastattelu toteutetaan palaverin yhteydessä ja siitä kerätään muistiinpanot ylös.

3 Tilanhallinta verkkosovelluksissa

3.1 Mitä on tilanhallinta?

Internet-aikakauden alkuvaiheilla verkkosivustot olivat hyvin yksinkertaisia. Ne koostuivat staattisista komponenteista eivätkä ne juurikaan sisältäneet käyttäjäinteraktioita. Web 2.0:n nousun myötä on kuitenkin staattiset verkkosivut alkaneet siirtyä enemmän toiminnallisuuksia sisältäviksi dynaamisiksi verkkosovelluksiksi. Osa sovelluksen liiketoimintalogiikasta on teknologioiden kehityttyä siirtynyt palvelimelta suoraan selaimen suorittamaksi. (Cheng 2018)

Modernit verkkosovellukset ovat yleensä toteutettu usean sivun sijaan yhdelle sivulle. Näitä sovelluksia kutsutaan SPA-sovelluksiksi (Single Page Application). SPA-sovellukset pitävät sisällään monimutkaisia reitityksiä sekä tarjoavat usein laajaa käyttäjäinteraktiota. Näiden käyttäjätoimintojen, reititysten ja liiketoimintalogiikan hallinta ovat tilanhallintaa. (Cheng 2018)

3.2 Mitä verkkosovelluksen tila pitää sisällään?

Tyypillisen SPA-verkkosovelluksen tilan voi pilkkoa kolmeen osaan, joita tarvitsee hallita.

Sovelluksen globaali tila

Globaali tila pitää sisällään globaalia dataa, joka jaetaan korkealla tasolla koko verkkosovelluksen kaikkien sivujen välillä. Se voi sisältää järjestelmäkonfiguraatioita sekä sovelluksen metadataa. (Cheng 2018)

Sivujen ja komponenttien yhteinen tila

Sivujen ja komponenttien yhteistä tilaa tarvitaan, jotta erinäiset sivut ja komponentit voivat kommunikoida keskenään ja toimia sulavasti yhdessä. Tiloja voidaan välittää eri sivujen välillä reitittimellä. Komponenttien tilaa voidaan jakaa ja välittää lapsikomponentteja ja tapahtumavirtoja hyödyntäen. (Cheng 2018)

Komponenttien sisäinen tila

Komponentit sisältävät myös oman sisäisen tilan, jota ei välitetä ulospäin. Komponenttien sisäinen tila voi sisältää monimutkaisia käyttäjäinteraktioita tai kommunikointia palvelinpuolelle. Esimerkkinä voi toimia lomake, jonka sarakkeita täytyy hallita validaatiologiikan tai käyttäjän syötteen perusteella. (Cheng 2018)

Näitä tiloja hyödyntäen voidaan sovelluksen tilaa hallita yksinkertaisemmissa projekteissa hyvin. Kun sovellukset muuttuvat laajemmiksi ja vaikeammaksi hallita, alkaa tämä toimintamalli olemaan kehittäjille vaikea havaita datavirtauksia komponenttien välillä. Dataa voidaan myös muokata monessa eri sijainnissa, jonka takia saattaa ilmetä virheitä ajan kanssa. Vikadiagnosointi ja bugien korjaus voi muuttua erittäin haastavaksi sovellukselle tulevaisuudessa näitä menetelmiä käyttäen. (Cheng 2018)

Nykyaikana parhaana tilanhallinnan arkkitehtuurimallina pidetään globaalia ”single source of truth” tietovarastoa, jossa koko verkkosovelluksen tiedonlähteenä toimii yksi globaali tietovarasto. Tämän mallin on kehittänyt ja nostanut suosioon Facebook, kun he kehittivät oman Flux-arkkitehtuurimallinsa. Flux-arkkitehtuurin pohjautuvia kirjastoja on monia, yleisimpinä ovat Redux- ja NgRx-kirjastot. (Cheng 2018)

4 Angular-ohjelmistokehys

4.1 Mikä on Angular-ohjelmistokehys?

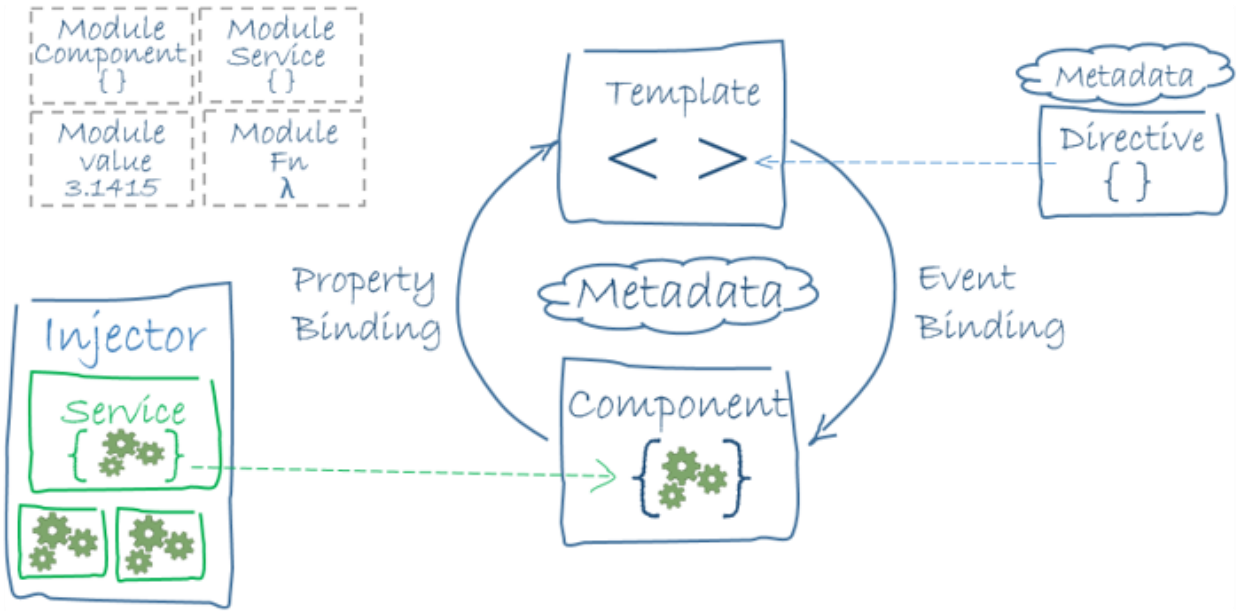
Angular on Googlen kehittämä, laajaan käyttöön levinnyt avoimen lähdekoodin alusta ja viitekehys yksisivuisten SPA-verkkosovellusten kehittämistä varten. Se on saanut alkunsa AngularJS nimisestä sovelluskehyksestä. Angularin ydin on TypeScript ohjelmointikieli, jota Angular sovelluskehityksessä käytetään. Angular implementoi sen päätoiminnallisuudet TypeScript-kirjastoina ja hyödyntää HTML-tiedostoja templaattien luomiseen. Angular tarjoaa sovelluskehittäjille hyvät puitteet dynaamisten verkkosovellusten kehittämiseen. (Kumar 2019)

Angular on helposti ylläpidettävä alusta, sillä se on komponentti ja luokkapohjainen, tarjoaa mahdollisuuden modulaarisiin arkkitehtuureihin, sisältää hierarkisen komponenttirakenteen, sekä yksinkertaiset deklaratiiviset templaatit. (Kumar 2019)

4.2 Angular-sovelluksen arkkitehtuuri

Angular-verkkosovelluksen arkkitehtuuri perustuu muutamaan olennaiseen konseptiin. Angular-verkkosovelluskehyksen vakiot rakenteellisen osat ovat Angular komponentteja, jotka järjestellään NgModuuleiksi. NgModuulit kasaavat siihen liittyvän koodin ja järjestää ne toiminnalliseksi kokoelmaksi. Angular-sovellus on määritetty kokoelma NgModuuleita. Angular-sovellus sisältää aina ainakin yhden juurimoduulin (engl. Root module) ja yleensä useampia ominaisuusmoduleita (engl.

Feature module). Komponenttien ja moduulien lisäksi Angular-sovellukset käyttävät sovelluksen ulkopuolisiin kutsuihin palveluita, joita kutsutaan komponenteista. Kuvio 1:ssä on selkeistetty sovelluksen arkkitehtuurimallia. (Architecture overview n.d.)



Kuvio 1. Angularin komponenttirakenne (Architecture overview n.d.)

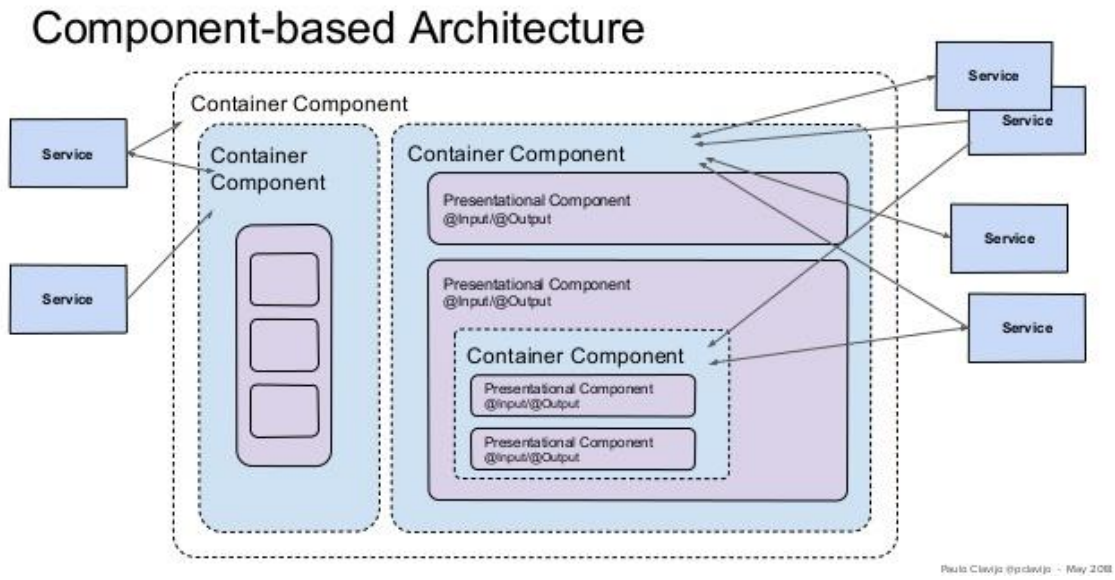
Angular noudattaa perinteistä malli-näkymä-käsittelijä (engl. model-view-controller) ohjelmistosuunnittelumallia, jossa pyritään erottamaan sovelluksen käyttöliittymä sovelluksen toimintalogiikasta. (Noring 2018, 8.)

Angular sovellus koostuu moduleista (engl. module), komponenteista (engl. component) sekä palveluista (engl. service). Komponentit määrittävät sovelluksen näkymät (engl. view), joita Angular voi valita sekä muokata sovellukseen ohjelmoidun logiikan sekä datan perusteella. Moduulit, komponentit ja palvelut ovat luokkia (engl. class), joiden alkuun on määtitetty @-alkuinen dekoraattorifunktio, joilla määritetään luokan tyyppi ja välitetään metadataa, joka kertoo Angularille miten luokan kuuluu toimia. (Architecture overview n.d.)

4.3 Komponentit

Angularissa kannustetaan kehittäjää luomaan useita komponentteja, jotka hoitavat tiettyä asiaa sovelluksen toiminnan kannalta sekä ovat toisistaan mahdollisimman riippumattomia. Malli-näkymä-käsittelijä ohjelmistosuunnittelumallin kontekstissa Angular sovelluksen komponentti sisältää käsittelijäluokan sekä näkymätemplaatin. Komponenttitiedostoon voi sisältää näkymän avoimena templaattina tai eristää sen omaan templaatti-tiedostoon, joka määritellään komponentin käytettäväksi. Komponentti itsessään on taas käsittelijäluokka, joka suorittaa käyttäjäinteraktiota vaativaa logiikkaa sekä noutaa dataa, jota komponentissa halutaan käytettävän. (Noring 2018, 19.)

Hyvin yleinen arkkitehtuurimalli on toteuttaa kahdenlaisia komponentteja. Älykkäitä säiliökomponentteja (engl. smart container component), sekä tyhmiä esityskomponentteja (engl. dumb presentational component). Säiliökomponentit vastaavat kaikesta kommunikoinnista ja toimintojen lähettämisestä palveluille. Kaikki komponenttitason datan käsittely suoritetaan säiliökomponentin sisällä. Esityskomponentin tehtävä on vain näyttää haluttua tietoa käyttöliittymään. Kaikki tieto mitä esityskomponentti saa otetaan vastaan säiliökomponentilta hyödyntäen @Input-direktiiviä. Jos esityskomponentilta täytyy lähettää dataa ulos, esimerkiksi käyttäjän täyttämän lomakkeen sisältämä data, tulee se lähettää säiliökomponentille @Output-direktiivillä, käyttäen hyväksi Angularin EventEmitter luokkaa. (Farhi 2017)



Kuvio 2. Angular-sovelluksen komponenttiarkkitehtuuri. (Clavijo 2018)

4.4 Moduulit

Vaikka Angularin komponentit yleensä hoitavat vain yhtä tehtävää, voidaan ne kategorisoida yleensä rykelmiksi, jotka kuuluvat joihinkin samoihin teemoihin. Angular sovellus vaatii aina vähintään yhden juurimoduulin (engl. root module) joka mahdollistaa sovelluksen käynnistämisen. Moduuleita voidaan ajatella kirjastona, johon sisältyy aina johonkin tiettyyn aiheeseen liittyvät komponentit ja palvelut (engl. services). (Architecture overview n.d.)

Sovelluksen koodin jakaminen erillisiin moduleihin auttaa monimutkaisten sovellusten kehittämistä sekä komponenttien suunnittelemista uudelleenkäytettäviksi. Modulit antavat myös mahdollisuuden laiskaan-lataamiseen (engl. lazy-loading), jonka avulla moduleita voidaan ladata vain tarvittaessa, täten vähentäen sovelluksen käynnistyksen kuormaa. (Architecture overview n.d.)

4.5 Palvelut ja riippuvuusinjektio

Palvelu on laaja käsite, joka voi kattaa mitä vaan arvoja, funktioita tai ominaisuuksia, joita sovellus tarvitsee. Palvelut ovat luokkia, joilla on yleensä tarkkaan määritelty tarkoitus. Angular erottaa komponentit ja palvelut kasvattaakseen modulaarisuutta ja uudelleenkäytettävyyttä. Komponenttien tulisi mahdollistaa käyttäjäkokemus. Yleensä palvelut hoitavat palvelinpuolelle tapahtuvien kutsujen käsittelyn, käyttäjäsyötteen validointia ja datan kirjaamista konsoliin. (Introduction to services and dependency injection n.d.)

Palvelut ovat luokkia, joita voidaan jakaa usealle eri komponentille. Niillä suoritetaan datakutsuja ja sovelluksen toimintalogiikkaa, jotka eivät välttämättä ole sidoksissa mihinkään tiettyyn näkymään. Palveluluokkaan määritellään @Injectable()-dekoraattori, jonka avulla palvelu voidaan sisällyttää komponenttiin riippuvuutena (engl. dependecy). (Introduction to services and dependency injection n.d.)

4.6 Reititin

Angularin reititin (engl. router) mahdollistaa navigaatiopolkujen määrittämisen sovelluksen eri tiloihin ja näkymä hierarkioihin. Reititin liittää URL-reitit näkymiin omien sivujen sijaan, jolla mahdollistetaan SPA-sovelluksen toiminnallisuus. Kun käyttäjä suorittaa toiminnon, joka aktivoi reitin, reititin katkaisee toiminnan, joka avaisi linkin uudessa selaimen välilehdessä ja vaihtaa sen sijaan vain Angular-sovelluksen sisäistä tilaa. Tämän avulla, kun sovelluksen sivu vaihtuu, päivittyy tieto käyttäjälle suoraan, eikä selaimen tarvitse päivittää selainikkunaa. (Architecture overview n.d.)

5 Reaktiivinen ohjelmointi

5.1 Mitä on reaktiivinen ohjelmointi?

Reaktiivinen ohjelmointi on ohjelmointiparadigma, joka perustuu arvojen jatkuvaan päivittämiseen ja muutokseen. Reaktiivisella ohjelmoinnilla mahdollistetaan tapahtumavirta pohjaisten sovellusten kehittäminen, jossa kehittäjä kertoo ohjelmistolle mitä tehdä ja ohjelmointikieli hoitaa itse sen, milloin tehdään. Reaktiivisessa ohjelmoinnissa hallitaan ja käsitellään asynkronisia datavirtoja. Asynkronisessa datavirrassa on virtaus dataa, jonka arvot lähetetään yksi toisensa jälkeen. (Bainomugisha, Carreton, Cutsem, Mostinckx & Meuter 2012)

5.2 Observer ohjelmistosuunnittelumalli

Datavirrat voivat sisältää dataa monesta eri asiasta kuten muuttujista, käyttäjän syötteestä, ominaisuuksista, välimuistista, datarakenteista ja monesta muusta eri lähteestä. Reaktiivinen ohjelmointi antaa kehittäjälle laajan työkalupakin erilaisia funktioita, joilla hallita, yhdistää, luoda ja suodattaa datavirtoja. Datavirta on aikajärjestyksessä oleva virta tapahtuvia tapahtumia, joka lähettää kolme eri asiaa, jotain tyyppiä olevan arvon, virheen tai ”valmis”-viestin, jolla virta lopetetaan. Datavirrat ovat asynkronisia ja niitä kuunnellaan erinäisillä funktioilla. Datavirran kuuntelemista kutsutaan myös tilaamiseksi (engl. Subscription). Nämä funktiot ovat niin sanottuja observereita. Datavirta on subject,tai observable jota tilataan ja tarkkaillaan. (Medeiros 2014)

5.3 RxJs-kirjasto

RxJs (Reactive Extensions for JavaScript) on JavaScript-kirjasto, jota voidaan käyttää Angular-verkkosovelluksissa reaktiivisen ohjelmoinnin ratkaisujen kehittämiseen.

Observable ja observer

Observable-muuttujat ovat Angularin suosittelema tekniikka tapahtumien käsittelyyn, asynkroniseen ohjelmointiin sekä useiden arvojen käsittelyyn. Observer-malli on ohjelmistosuunnittelumalli, jossa subjectiksi nimetty olio ylläpitää listaa riippuvaisuuksista, joita kutsutaan observervereiksi. Subject ilmoittaa tilan muutoksista automaattisesti kaikille sen observereille. Observable-muuttujat ovat deklaratiivisia, joten niille määritellään funktio arvojen julkaisuun, mutta sitä ei suoriteta ennen kuin siihen tilataan subscription. Subscriber saa ilmoituksia kohteelta, kunnes funktio on suoritettu loppuun tai subscription päätetään. (Observables overview n.d.)

Subscription

Havaittava instanssi alkaa julkaisemaan arvojaan vasta kun sen lähettämä tietovirta subscribetaan johonkin. Tietovirta voidaan subscribea kutsumalla observable-instanssin subscribe()-metodia, joka välittää observerin vastaanottamaan arvoja. Angular-sovelluksen sisällä usein suositaan tilaamaan observable- muuttuja suoraan komponentin templaatissa käyttäen Angularin Async-putkea. (Observables overview n.d.)

Operaattorit

Operaattorit ovat funktioita, joiden avulla mahdollistetaan asynkroninen ohjelmointi Angular-sovelluksen sisällä deklaratiivisin menetelmin. Operaattoreita on olemassa kahta eri tyyppiä, putkitettavia operaattoreita (engl. pipeable operators) sekä luovia operaattoreita (engl. creation operators). Putkitettavat operaattorit ovat operaattoreita, joita voidaan putkittaa observable-instanssiin ja kutsua useita peräkkäin käyttäen syntaksia observableInstance.pipe(operator()). Ne eivät kutsuessaan muuta observable-instanssia vaan palauttavan uuden observable-muuttujan, jonka tilauslogiikka määrittyy ensimmäisen observablen pohjalta riippuen siitä, minkälaista operaattorifunktiota on käytetty. Putkitettavat operaattorit ovat käytännössä puhtaita funktioita (engl. pure functions), joka ottaa sisääntulona (engl.

input) yhden observable- muuttujan ja se generoi ulostulona (engl. output) uuden observable-instanssin. (Operators n.d.)

Luovat operaattorit ovat toisenlaisia operaattoreita, joita voidaan käyttää uusien observable-instanssien luomiseen jollain tietyillä ennalta määritellyllä käytöksellä. Esimerkiksi of(1, 2, 3)-funktio loisi observable-instanssin, joka yksi toisensa jälkeen lähettäisi arvot 1, 2 ja 3. (Operators n.d.)

6 NgRx-kirjasto

NgRx on Angular-ohjelmistokehykselle kehitetty kokoelma reaktiivisia kirjastoja Redux-ohjelmistokehitysmallin mukaista tilanhallinnan implementointia varten. Se sisältää useita kirjastoja eri tarpeisiin.

6.1 Keskitetty tilanhallinta

Reduxilla tavoitellaan ennakoitavuutta. Ennakoitavuudella koitetaan ylläpitää ja helpottaa tietoa siitä, kuka tekee mitä ja mihin eri osiin sovelluksen tilanhallinnan näkökulmasta. Yksi totuuden lähde (engl. Single source of truth) on yksi Reduxin ydinkonsepteista, siinä tavoitteena on yksi lähde kaikelle sovelluksen datalle, ja tässä tapauksessa sillä viitataan storeen. Reduxilla tavoitellaan myös helpottamaan tiedon ylläpitämistä siitä, ketkä ovat oikeutettuja muuttamaan varaston tilaa ja sisältöä. (Noring 2018)

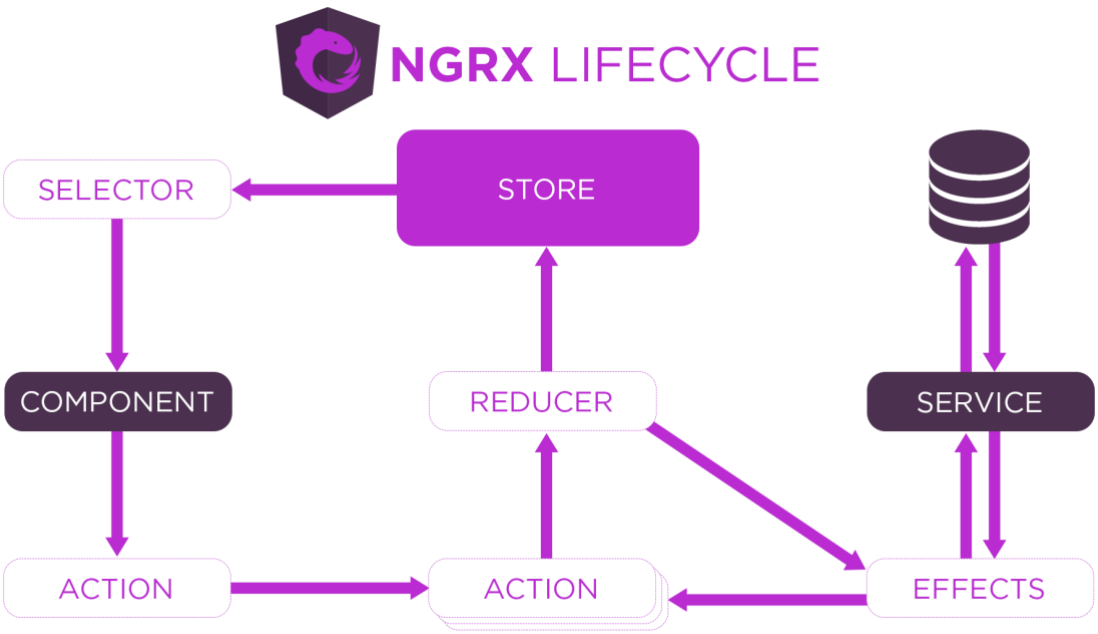
Angular-sovelluksen kasvaessa kontrollereiden sekä näkymien hallinta menee helposti monimutkaiseksi ja kehittäjä kadottaa helposti yleisnäkymän siitä mikä koodi vaikuttaa mihinkin sovelluksen tilaan. Redux helpottaa tätä niin että näkymä ei koskaan suoraan muokkaa tilaa, vaan vaatii actionien laukaisua, jotka esittävät aietta, kuinka tilaa tulee muokata. Reduxissa ei myöskään koskaan mutatoida tilaa, vaan hyödynnetään puhtaita funktioita (engl. Pure functions), joilla vanhasta tilasta luodaan aina kopio, johon on tehty kutsutun actionin määrittämät muutokset, ja

korvataan vanha tila uudella kopiolla. Tämä helpottaa myös tarvittaessa vanhaan tilaan palaamista. (Noring 2018)

Keskitetty tilanhallintavarasto mahdollistaa saman tilan jakamisen usean eri komponentin välillä. Kaikkea sovelluksen tilaa ei kuitenkaan välttämättä aina tarvitse tallentaa varastoon, jotain asioita ei tarvitse viedä jonkun tietyn komponentin lokaalin tilan ulkopuolelle. Yleensä sellaista tilatietoa, jonka menettäminen ei haittaa, on oivallista olla tallentamatta varastoon ja säilyttää se komponentin omassa tilassa, esimerkiksi jonkun pudotusvalikon (engl. Dropdown menu) valinta. (Noring 2018)

6.2 Store-kirjasto (ngrx/store)

Ngrx/store on käytännössä Redux-arkkitehtuuriin perustuva tilanhallinta Angularille. Se pohjautuu RxJs-kirjastoon ja implementoi Redux tyyppisen rajapinnan hyödyntäen RxJs rajapintaa. Tällä mahdollistetaan storeen alusta asti asynkroninen tuki. Reduxin tavoin, NgRx-store pohjautuu yksisuuntaisiin tietovirtoihin. (ks. kuvio 3) (Farhi 2017)



Kuvio 3. NgRx tilanhallinnan elinkaari. (Store n.d.)

Store on säiliönä toimiva yksiö olio (engl. singleton objectI), joka pitää sisällään sovelluksen tilan. Sitä tulee pitää ainoana yksittäisen totuuden lähteenä, joka toimittaa dataa koko sovellukselle. Storen toiminta on suunniteltu kolmen avainobjektin ympärille, jotka ovat store, reducerit sekä actionit. Yksi suurimpia NgRx-storen hyötyjä on se, että sen avulla saadaan eristettyä kaikki sovelluksen dataan liittyvä logiikka ulos Angularin komponenttitasolta. (Farhi 2017)

NgRx:n StoreModule otetaan käyttöön implementoimalla se sovelluksen juurimoduuliin. StoreModulen forRoot()-metodissa määritellään parametrina vähentäjät.

6.3 Actionit

Actionit ovat yksi NgRx:n perusrakenteista. Actionit toimivat ilmaisuina uniikeista tapahtumista, jotka tapahtuvat ympäri sovellusta. Ne toimivat sisään- (engl. input) ja ulostuloina (engl. output) useassa eri järjestelmässä ympäri NgRx:ää. Actioneilla autetaan ymmärtämään, kuinka sovelluksen tapahtumat tulee käsitellä. (Actions n.d.)

Actionit ovat olioita jotka luodaan const-muuttujiin hyödyntäen NgRx:n createAction-metodia, joka vaatii parametrikseen tyypin, sekä mahdollisen payloadin, joka voidaan määritellä createActionin props()-metodin avulla. Toimintamuuttujille määriteltävä tyyppi nimetään hyvien tapojen mukaisesti toiminnan lähteen mukaan ja sisällytetään mukaan teksti, jossa kuvataan määrättyä toimintoa. (Actions n.d.)

Sovelluksen tilaa päivitetään lähettämällä (engl. dispatch) toimintoja. Vähentäjä sekä efektit kuuntelevat toimintoja ja reagoivat halutulla tavalla, kun ne lähetetään. Toiminnat lähetetään käyttämällä varaston dispatch-metodia. Kun toiminto on lähetetty, varasto suorittaa vähennysprosessin kaikilla toimintoa kuuntelevilla vähentäjillä, tuottaa uuden tilan, ja päivittää mahdollisille tilaajille uudet muutokset. (Farhi 2017)

6.4 Reducerit

NgRx:n reducerit käsittelevät siirtymät yhdestä tilasta seuraavaan tilaan. Reducer-funktiot hoitavat nämä siirtymät määrittelemällä, miten toimia lähetettyjen actionien tyyppien perusteella. Reducerit ovat puhtaita funktioita, ne tuottavat saman ulostulon välittämättä sisääntulosta. Niillä ei ole mitään sivuefektejä ja ne hoitavat tilamuutokset synkronisesti. Reducerit ottavat viimeisimmän lähetetyn actionin, nykyisen tilan, ja päättää palauttaako se uuden actionin perusteella muokatun tilan vai alkuperäisen tai vanhan tilan. Jos reducer joutuu reagoimaan johonkin actioniin, on lopputuloksena aina uusi muuttunut tila. (Reducers n.d.)

Sovelluksen kokonainen keskitetty tila jakaantuu todellisuudessa pienempiin osiin eri ominaisuuksien mukaan, joista jokaiselle osalle luodaan oma reducer. Luomalla useita reducereita eri tarpeisiin mahdollistetaan sovelluksen organisoitu laajennettavuus.

Reducerit kuuntelevat actioneita. Kun jokin action lähetetään, se käydään läpi kaikissa sovelluksen reducereissa, mutta vain ne reducerit reagoivat mitkä ovat ohjattu reagoimaan kyseiseen actioniin.

6.5 Selectorit

Selectorit ovat puhtaita funktioita, joita käytetään palauttamaan storesta haluttu osa tilasta. Selectoreiden iso hyöty on se, että niillä saadaan eristettyä datan käsittelyyn liittyvä toimintalogiikka erilleen komponenteista. Komponenttien tarvitsee vain suorittaa kutsu selectorille ja saa vastauksena haluttua oikeamuotoista dataa. Komponentit voivat siis keskittyä vain actioneiden kutsuun, ja ne pystyvät luottamaan siihen, että actionit palauttavat niille oikeaa haluttua dataa. Komponenttien sisällä ei muokata dataa lainkaan.

Selectorit luodaan hyödyntäen NgRx:n tarjoamia apufunktioita. Funktio createFeatureSelector on metodi, jonka avulla palautetaan ylemmän tason ominaisuustila (engl. feature state). Se palauttaa tyypitetyn selector-funktion jollekin

tietylle tilan ominaisuusosiolle. Kun ominaisuusvalitsin on luotu, voidaan luoda createSelector-metodia hyödyntäen tavallinen valitsin, jonka avulla saadaan tilasta haluttu osa. (Selectors n.d.)

FeatureSelector-metodia hyödyntäen voidaan sovelluksen tilapuuta paloitella järkevästi eri vastuualueille ominaisuuksien mukaan. Tämä auttaa pitämään sovelluksen kokonaistilan selkeämpänä, ja helpottaa sovelluksen laajennettavuutta, kun tilaa jaetaan pienempiin osuuksiin.

6.6 Effects

Kun actioneja lähetetään, actionit voivat vaikuttaa suoraan käyttöliittymän tilaan. Aina actionit eivät kuitenkaan päivitä tilaa heti, vaan niillä voi olla sivuvaikutuksia, jotka yleisimmin ovat API-kutsuja. Aina kun NgRx:ssä lähetetään actioneja, store käy läpi ensin reducerit ja sen jälkeen efektit tarkistaakseen missä kyseistä lähetettyä actionia kuunnellaan.

Efektit (engl. effects) ovat sivuvaikutuksia. Näillä sivuvaikutuksilla viitataan operaatioihin, jotka kohdentuvat sovelluksen ulkopuolelle, kuten tiedon hakemiseen verkon yli sovelluksen ulkopuolisista resursseista, ja ne eivät ole suoraan kytköksissä sovelluksen tilaan. Koska actionit lähetetään synkronisesti ja muutokset tulevat tilaan välittömästi, ei sovelluksen ulkopuolelle tehtäviä kutsuja haluta suorittaa reducerissa. Tätä varten on @ngrx/effects-kirjasto. Actionit, joiden suorittaminen voi mahdollisesti kestää pitempään, suoritetaan efekteissä. Efekti on injektoitava palvelu, joka kuuntelee jotain tiettyä actionia. Efekti voi suorittaa useita operaatioita ja muunnoksia käsiteltävään payloadiin. Efekti ottaa vallan, kun sen kuuntelema action laukaistaan ja se lopettaa hallinnan lähettämällä uuden actionin suoritettavien toimintojen jälkeen. (Noring 2018)

Palvelupohjaisissa Angular-sovelluksissa komponentit ovat vastuussa kommunikoinnista palveluiden kanssa päästäkseen käsiksi sovelluksen ulkopuolisiin resursseihin. Sen sijaan efektit eristävät palvelut täysin muista komponenteista ja

mahdollistavat niiden kanssa kommunikoinnin NgRx:n toimintamallin mukaisesti. (Effects n.d.)

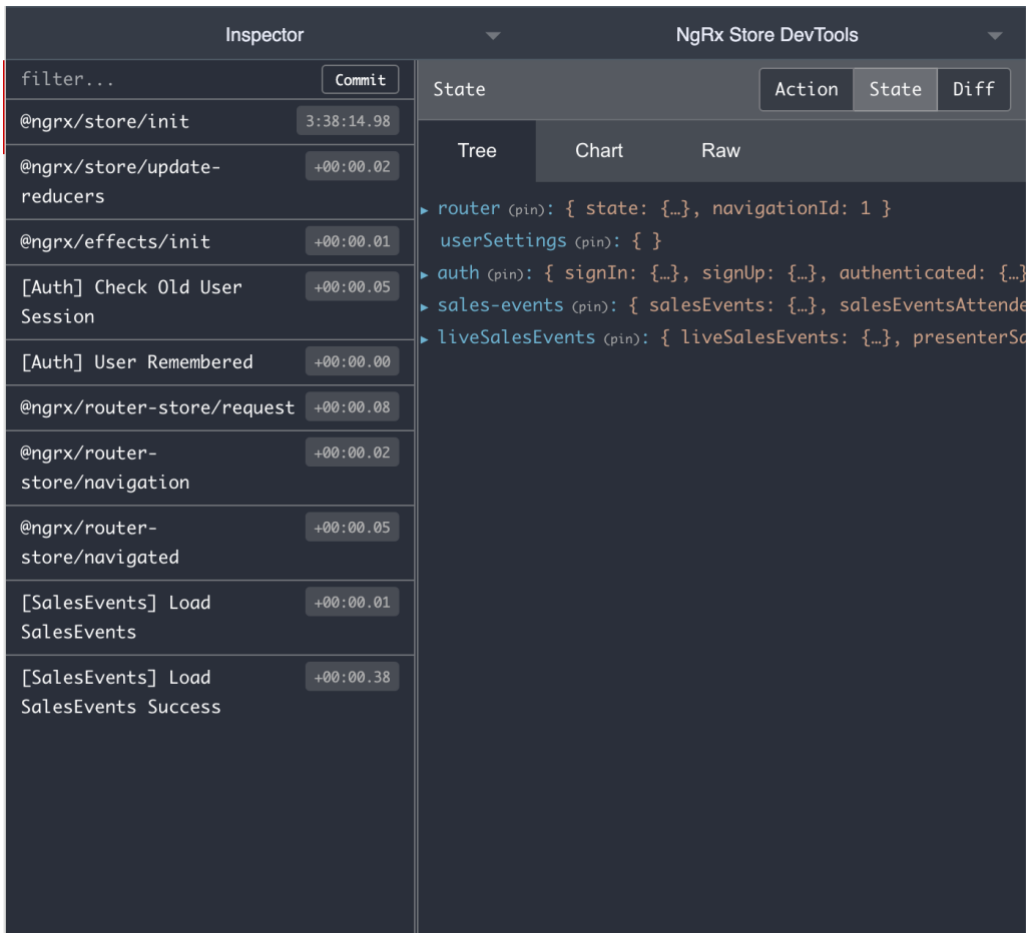
6.7 NgRx/Router-store

Tiedon tallentaminen siitä missä käyttäjä sovelluksessa sijaitsee milläkin hetkellä voi olla erittäin hyödyllistä. Sovelluksen sijaintitiedot määrittää reitti, reittiparametrit, sekä kyselyparametrit (engl. query parameters). (Noring 2018)

Angularin reititin voidaan kytkeä NgRx-storeen @ngrx/router-store kirjaston avulla. Router-store kirjaston avulla jokaisen reitittimen navigaatiosyklin yhteydessä lähetetään useita actioneja, jotka mahdollistavat muutoksien kuuntelun reitittimen tilassa. Reitittimen tilaa voidaan täten ylläpitää storessa, ja sen dataa voidaan hyödyntää sovelluksessa. (@ngrx/router-store n.d.)

6.8 Kehittämistyökalut

Kehittäjien tueksi NgRx tarjoaa @ngrx/store-devtools kirjaston, joka pitää sisällään kehitystyökaluja sekä instrumentaatiota varastolle. Store Devtoolsin avulla sovelluksen toiminnasta saadaan pidettyä kirjaa kaikista lähetetyistä actioneista sekä tilasta ja siihen tapahtuvista muutoksista sovelluksen elinkaaren aikana. Tämä kirjasto toteuttaa tarvittavat toiminnalisuudet kehitystä varten, mutta jotta voidaan visuaalisesti tarkkailla muutoksia, tulee kehittäjän myös asentaa selaimeen Redux DevTools niminen lisäosa. (ks. kuvio 4)



Kuvio 4. Redux DevTools -kehitystyökalu.

Redux DevToolsin vasemmassa osiossa listataan kaikki lähetetyt actionit aikajanan mukaan. Actioneiden listauksessa niminä toimivat actioneiden tyypit. Hyvät nimeämiskäytänteet pitävät tapahtumavirran selkeänä ja auttavat virheiden etsinnässä. Kehittäjä voi myös valita minkä tahansa lähetetyn actionin klikkaamalla sitä, tarkkailla sen parametrejä, ja havainnoida miltä sovelluksen kokonaistila näytti kyseisen actionin lähettämisen jälkeen. (ks. kuvio 4)

Työkalun oikeassa osiossa voidaan tarkastella sovelluksen tilaa kolmessa mahdollisessa eri näkymässä: puu- kaavio- tai raakanäkymässä. Osiossa voidaan myös tarkastella actioneiden lähettämää payloadia sekä tarkistaa miten actionin suorittaminen muutti tilaa. (ks. kuvio 4)

7 Tilanhallinnan käytännön ratkaisut

Tässä luvussa tutkitaan toimeksiantajan toteuttamia Angular-sovelluksia ja analysoidaan NgRx-tilanhallinnan toteutuksia. Analyysin kohteeksi on toimeksiantajan kanssa valittu mahdollisimman monipuolisesti erityyppisiä tilanhallinnan ongelmia ja tutkittu kuinka ne ovat ratkaistu NgRx-kirjaston käytöllä.

7.1 Datan lataaminen storesta ja listaaminen komponentin näkymään

Tässä luvussa käydään kokonaisvaltaisesti läpi tilanhallinnan näkökulmasta NgRx-storen alustus, kuinka saadaan haettua dataa jostain ulkopuolisesta verkkorajapinnasta, tallennettua sitä NgRx-storeen, ja miten se saadaan näytettyä käyttäjälle Angular-komponentin näkymässä.

Globaalin storen alustamiseksi täytyy ensin määrittää sovelluksen juurimoduulissa eli app.module.ts tiedoston @NgModule-dekoraattorin imports-taulukkoon StoreModule.forRoot({}) ja EffectsModule.forRoot({}). Ja koska tässä toteutuksessa hyödynnetään myös NgRx-router-storea, tulee sekin alustaa StoreRouterConnectingModulen määrittämisellä. (ks. liite 1)

Toimeksiantajan toteutuksessa store-kansio löytyy ominaisuusmoduulin juuritasolta. Jokaiselle storen osalle on luotu oma kansio, jotka pitävät sisällään index.ts tiedoston sekä ominaisuuskohtaiset muut NgRx:n tiedostot. (ks. liite 2) Index tiedostoihin otetaan importteina vastaan kaikki toiminnallisuudet ja niiden käyttö muissa komponenteissa mahdollistetaan exporteilla. (ks. kuvio 5)

4-Angular - index.ts

```
1  import * as SalesEventActions from './sales-event.actions';
2  import * as SalesEventAttendeesActions from './sales-event-attendees.actions';
3  import * as SalesEventImageActions from './sales-event-image.actions';
4  import * as SalesEventEditActions from './sales-event-edit.actions';
5  import * as SalesEventPresentersActions from './sales-event-presenters.actions';
6  import * as SalesEventPublishActions from './sales-event-publish.actions';
7  import * as ProductSearchActions from './product-search.actions';
8  import * as SalesEventProductsActions from './sales-event-products.actions';
9
10 export {
11   SalesEventActions,
12   SalesEventAttendeesActions,
13   SalesEventImageActions,
14   SalesEventEditActions,
15   SalesEventPresentersActions,
16   SalesEventPublishActions,
17   ProductSearchActions,
18   SalesEventProductsActions,
19 };
20
```

Kuvio 5. Actioneiden Index-tiedoston importit ja exportit.

Kun on tarve hakea dataa jostain, tulee määrittää kolme eri actionia, jotka kuvaavat toiminnon eri vaiheita, eli käskyä suorittaa toiminto, toiminnon suorittamisen onnistumista ja toiminnon suorittamisen epäonnistumista. Tämä pätee myös usein muihinkin toimintoihin kuten tiedon lisäämiseen, päivittämiseen ja poistamiseen. (ks. kuvio 6)

4-Angular - sales-event.actions.ts

```
2  import { SalesEvent } from '@sales-events/models/sales-event.model';
3  import { SalesEventInfo } from '../../models/sales-event-info.model';
4
5  export const loadSalesEvents = createAction('[SalesEvents] Load SalesEvents');
6
7  export const loadSalesEventsSuccess = createAction(
8    '[SalesEvents] Load SalesEvents Success',
9    props<{ salesEvents: SalesEvent[] }>()
10 );
11
12 export const loadSalesEventsFail = createAction(
13   '[SalesEvents] Load SalesEvents Fail',
14   props<{ error: string }>()
15 );
```

Kuvio 6. Actionit tiedon lataamiselle.

Ensimmäinen laukaistava action toimii käskynä, jota efekti ja reducer kuuntelee. LoadSalesEvents-action laukaisee putken, jolla efektin kautta haetaan dataa verkkorajapinnasta. Kun efekti kuulee, että loadSalesEvents action on lähetetty, se kutsuu palvelua joka suoritta API-kutsun. Jos API-kutsu onnistuu, efekti lähettää eteenpäin uuden lataamisen onnistumista kuvaavan loadSalesEventsSuccess actionin, jonka payloadina annetaan API-kutsun palauttama data observable datavirtana. (ks. kuvio 7)

4-Angular - sales-events.service.ts

```
21  getSalesEvents(): Observable<any[]> {
22    return this.authHttp.get(`${environment.apiBase}/private/sales-events`);
23  }
```

Kuvio 7. Palvelun funktio, joka suorittaa http-pyynnön API:lle.

Jos API-kutsu palauttaakin jotain vääränlaista dataa tai se epäonnistuu, efektissä voidaan havaita poikkeus ja lähettää eteenpäin epäonnistumista kuvaava loadSalesEventsFail action. (ks. kuvio 8)

4-Angular - sales-events.effects.ts

```
14  @Injectable()
15  export class SalesEventsEffects {
16    loadSalesEvents$ = createEffect(() =>
17      this.actions$.pipe(
18        ofType(SalesEventActions.loadSalesEvents),
19        switchMap(() =>
20          this.salesEventsService.getSalesEvents().pipe(
21            map((salesEvents) => SalesEventActions.loadSalesEventsSuccess({ salesEvents })),
22            catchError((error: string) => {
23              console.log(error);
24              return of(SalesEventActions.loadSalesEventsFail({ error }));
25            })
26          )
27        )
28      )
29    );
```

Kuvio 8. Efekti datan lataamiselle.

Actioneita kuuntelee myös reducerit. Tilan tuottamiseksi reducerissa vaaditaan tiettyjä määrityksiä. Ensin on hyvä määrittää kyseiselle tilalle rajapintamalli. Mallissa määritetään useita eri boolean-tilamuuttujia sekä tilan sisältämät entiteetit, jotka ovat tässä tapauksessa SalesEvent-mallin mukaisia olioita. (ks. kuvio 9)

4-Angular - sales-

```
12  export interface SalesEventsState {
13    entities: { [id: string]: SalesEvent };
14    loadingSalesEvent: string | undefined;
15    adding: boolean;
16    deleting: boolean;
17    loading: boolean;
18    updating: boolean;
19    publishing: boolean;
20    editing: boolean;
21  }
```

Kuvio 9. Reducerin rajapintamalli.

Kun on saatu hahmotettua ja mallinnettua tila kokonaisuutena, tulee luoda lähtökohta tilalle. Alustava tila määritetään initialState const-muuttujaan ja sen tyypiksi asetetaan aiemmin mallinnettu SalesEventsState-malli. Oletuksena pidämme tilan tyhjänä ja boolean-tilamuuttujat false tilassa. (ks. kuvio 10)

4-Angular - sales-events.reducer.ts

```
23  export const initialState: SalesEventsState = {
24    entities: {},
25    loadingSalesEvent: undefined,
26    adding: false,
27    deleting: false,
28    loading: false,
29    updating: false,
30    publishing: false,
31    editing: false,
32  };
```

Kuvio 10. Alustetun tilan määritykset.

Const-muuttujaan on luotu createReducer()-metodilla reducer, joka kuuntelee actioneita ja toimii määrätyllä tavalla kun actionit laukaistaan. Kun reducer kuulee loadSalesEvents-actionin, se päivittää tilaan vain boolean-tilamuuttujan loading, tilaan true, joka pitää tilassa yllä sitä tietoa, että sovellus on lataamassa jostain dataa. Funktion sisällä välitetään putkessa eteenpäin ”…” spread operaattorilla vanha tila, jota ei muokata ollenkaan. Jos efektin puolella datan lataus suoriutuu onnistuneesti, lähetetään sieltä onnistumista kuvaava loadSalesEventsSuccess-action. Onnistumisen funktion parametriksi välitetään payloadissa oleva salesEvents olio, joka tallennetaan entities-muuttujaan ja käsitellään JavaScriptin reduce-metodia hyödyntäen. Ladatut tapahtumat käydään läpi ja asetetaan storeen avaimenaan kunkin tapahtuman uniikki tunniste id, sekä palautetaan loading boolean-tilamuuttuja tilaan false. Jos efekti palauttaakin onnistumisen sijasta virheen, reducer ei päivitä entiteettejä ja asettaa storeen sen sijaan actionin payloadissa olevan errorMessage virheviestin. (ks. kuvio 11)

4-Angular - sales-events.reducer.ts

```
34  const salesEventsReducer = createReducer(
35    initialState,
36    on(SalesEventActions.loadSalesEvents, (state) => ({
37      ...state,
38      loading: true,
39    })),
40    on(SalesEventActions.loadSalesEventsSuccess, (state, { salesEvents }) => {
41      const entities = salesEvents.reduce(
42        (salesEventEntities: { [id: string]: any }, salesEvent: any) => ({
43          ...salesEventEntities,
44          [salesEvent.id]: salesEvent,
45        }),
46        {});
47      return {
48        ...state,
49        entities,
50        loading: false,
51      };
52    }),
53    on(SalesEventActions.loadSalesEventsFail, (state, action) => ({
54      ...state,
55      loading: false,
56      errorMessage: action.error,
57    })),
```

Kuvio 11. Reducer kuuntelee lataus actioneita.

Kun sovelluksessa siirrytään reittiin, joka aktivoi salesEvents-ominaisuusmoduulin (engl. feature module), moduulin ominaisuussäiliö-komponentissa (engl. feature container component) alustuksen yhteydessä lähetetään action, joka lataa ja asettaa tiedon storeen. (ks. kuvio 12)

4-Angular - sales-events-feature.container.ts

```
10  export class SalesEventsFeatureContainerComponent implements OnInit {
11    constructor(private store: Store<AppState>) {}
12
13    ngOnInit() {
14      this.store.dispatch(SalesEventActions.loadSalesEvents());
15    }
16  }
```

Kuvio 12. Ominaisuussäiliön alustuksessa laukaistaan action.

Jotta data saadaan storesta komponentille käsiteltäväksi, tulee luoda selector, jota komponentti voi kutsua. Selectorissa haetaan halutusta reducerista tarvittava tila, ja

luodaan createSelector-apufunktiolla selector. Ensin haetaan kaikki entiteetit, ja sitten putkitetaan toinen getSalesEventsList muuttujaan tallennettu selector siihen, joka ottaa entiteetit ja järjestää ne halutulla tavalla. Exportoitua selectoria voidaan kutsua komponentissa. (ks. kuvio 13)

4-Angular - sales-events.selectors.ts

```
5  import { getSalesEventsState } from '../reducers';
6
7  const getSalesEventEntities = createSelector(getSalesEventsState, (state) => state.entities);
8
9  export const getSalesEventsList = createSelector(getSalesEventEntities, (entities) =>
10   sortBy(Object.values(entities), ['eventDate', 'startTime']).reverse()
11 );
```

Kuvio 13. Selector, joka hakee storesta entiteetit sisältävän tilan.

Oletuksena storen latauksen jälkeen reitti siirtyy sales-events komponenttiin, jonka säiliökomponentissa kutsutaan valitsinta. Jotta valitsinta voi kutsua, tulee ensin komponentin konstruktorissa implementoida store komponentin käytettäväksi. Selector palauttaa komponentille observablena halutun datan, ja se tallennetaan salesEvents$ muuttujaan. (ks. kuvio 14)

4-Angular - sales-events.container.ts

```
10 export class SalesEventsContainerComponent {
11   salesEvents$ = this.store.select(SalesEventsSelectors.getSalesEventsList);
12   constructor(private store: Store<State>) {}
13 }
```

Kuvio 14. Selectorin data otetaan komponenttiin sisään observablena.

Koska säiliökomponentti toimii vain älykkäänä äitikomponenttina, tulee data välittää tyhmälle lapsikomponentille, joka hoitaa tiedon näyttämisen. Observable datavirtana käsiteltävä data välitetään lapsikomponentille inputtina säiliön templaatissa, ja se käsitellään Angularin async-putken avulla, jotta sitä voidaan lapsikomponentissa tulostaa näkymään. (ks. kuvio 15)

4-Angular - sales-

```
1  <main>
2    <sales-events-list
3      [salesEvents]="salesEvents$ | async"
4    ></sales-events-list>
5  </main>
```

Kuvio 15. Datan välittäminen async-putken läpi lapsikomponentille.

Selectorin avulla voidaan myös päästä helposti käsiksi johonkin yhteen yksilölliseen entiteettin storessa. Esimerkissä (ks. kuvio 16) hyödynnetään NgRx:n router-store kirjastoa, jonka avulla sovelluksen reitistä haetaan käyttäjän valitseman entiteetin uniikki id, ja palautetaan sen perusteella oikea entiteetti.

4-Angular - sales-events.selectors.ts

```
29  export const getSelectedSalesEventEntity = createSelector(
30    RouterSelectors.getSelectedSalesEventId,
31    getSalesEventEntities,
32    (salesEventId, entities) => entities[salesEventId]
33  );
```

Kuvio 16. Yhden entiteetin selector.

7.2 Datan lisääminen storeen

Komponentista voidaan myös lähettää ulos dataa storeen. Esimerkissä (ks. kuvio 17), jossa näkymäkomponentilta lähetetään @Output-koristelijan ja EventEmitterin avulla ylöspäin säiliökomponentille.

4-Angular - add-sales-event.component.ts

```
15  @Output()
16  addSalesEventDataSubmitted = new EventEmitter<SalesEventInfo>();
17
18  eventDateFormControl = new FormControl('', Validators.required);
19
20  addSalesEventForm: FormGroup = new FormGroup({
21    eventName: new FormControl('', Validators.required),
22    eventDescription: new FormControl(''),
23    eventDate: this.eventDateFormControl,
24    startTime: new FormControl('', Validators.required),
25  });
```

Kuvio 17. Datan lähetys komponentilta @Output()-dekoraattorilla.

Säiliökomponentin templaatissa otetaan vastaan lapsikomponentin lähettämä data, ja annetaan se parametriksi funktiokutsulle joka laukaisee addSalesEvent()-actionin ja antaa vastaanotetun datan sen payloadiksi. (ks. kuvio 18)

4-Angular - add-sales-event.container.ts

```
12  export class AddSalesEventContainerComponent {
13    isAdding$ = this.store.select(SalesEventsSelectors.isAdding);
14
15    constructor(private store: Store<State>) {}
16
17    onSalesEventSubmit(salesEventInfo: SalesEventInfo): void {
18      this.store.dispatch(SalesEventActions.addSalesEvent({ salesEventInfo }));
19    }
20  }
```

Kuvio 18. Säiliökomponentti laukaisee actionin payloadin kanssa.

Actionin lähetettyä reducer ensin kuulee, että uutta tietoa ollaan lisäämässä ja sen jälkeen efektille tulee tieto actionista. Efekti suorittaa palvelun avulla API-kutsun ja lähettää datan tietokantaan. Onnistuneen API-kutsun jälkeen laukaistaan onnistunutta suoriutumista kuvaava action, jota kuuntelee uusi efekti, joka router-storea hyödyntäen navigoi käyttäjän sovelluksessa eteenpäin, sekä antaa käyttöliittymän tiedon onnistuneesta toiminnosta. (ks. kuvio 19)

4-Angular - sales-events.effects.ts

```
addSalesEvents$ = createEffect(() =>
  this.actions$.pipe(
    ofType(SalesEventActions.addSalesEvent),
    switchMap((action) =>
      this.salesEventsService.addSalesEvent(action.salesEventInfo).pipe(
        map((salesEvent) => SalesEventActions.addSalesEventSuccess(salesEvent)),
        catchError((error: string) => {
          console.log(error);
          return of(SalesEventActions.addSalesEventFail({ error }));
        })
      )
    )
  )
);

addSalesEventSuccess$ = createEffect(() =>
  this.actions$.pipe(
    ofType(SalesEventActions.addSalesEventSuccess),
    switchMap((action) => [
      RouterActions.navigate({ commands: ['sales-events', action.id] }),
      UIActions.openSnackbar({
        message: 'Tapahtuman lisääminen onnistui',
        action: 'OK',
        panelClass: 'successSnack',
      }),
    ])
  )
);
```

Kuvio 19. Efekti joka kuuntelee addSalesEvent-actionia.

Reducer kuuntelee onnistunutta actionia, jonka payloadissa välitetään lähetetty data. Reducer hakee storesta kaikki vanhat entiteetit, ja lisää olioon payloadin mukana tulleen uuden entiteetin, jonka jälkeen se korvaa storen vanhan tilan uudella tilalla. (ks. kuvio 20)

4-Angular - sales-events.reducer.ts

```
81  on(SalesEventActions.addSalesEvent, (state) => ({ ...state, adding: true })),
82
83  on(SalesEventActions.addSalesEventSuccess, (state, salesEvent) => {
84    const entities = { ...state.entities, [salesEvent.id]: salesEvent };
85    return {
86      ...state,
87      entities,
88      adding: false,
89    };
90  }),
91
92  on(SalesEventActions.addSalesEventFail, (state, action) => ({
93    ...state,
94    adding: false,
95    errorMessage: action.error,
96  })),
```

Kuvio 20. Reducer kuuntelee addSalesEvent()-toimintoja.

7.3 Actioneiden laukaisut sovelluksen ulkoisesta lähteestä

Useimmiten actioneita lähetetään joko komponenttien alustuksen yhteydessä tai käyttäjäinteraktioiden seuraamuksena, mutta niitä voidaan myös laukaista esimerkiksi jonkun ulkoisen lähteen kautta. Tässä esimerkkitapauksessa sovellus kuuntelee WebSocket-palvelinta, jonka lähettämien muutostapahtumaviestien perusteella voidaan laukaista tilaa päivittäviä actioneita, joilla saadaan reaktiivisesti päivitettyä käyttöliittymään tieto tapahtuneesta muutoksesta.

Kun sovellukseen kirjaudutaan sisään, lähettää sovellus toiminnon, johon reagoidaan luomalla yhteys WebSocket-palvelimelle. WebSocket palvelussa on socketSubscription niminen reaktiivinen putki, joka kuuntelee palvelimelle lähetettyjä viestejä. Aina kun palvelimella havaitaan jokin tapahtuma, putki lähettää viestin eteenpäin onMessage()-funktiolle joka laukaisee yleisen actionin ja antaa viestin sille payloadiksi. (ks. kuvio 21)

5-AngularAPP - websocket.service.ts

```
53  this.socketSubscription = this.socket$.subscribe(
54        (message) => this.onMessage(message),
55        (error) => this.onError(error),
56        () => this.onComplete()
57      );
58
59      return of('Connected.');
60    }
61
62    private onMessage(message: any) {
63      this.store.dispatch(WebSocketActions.receivedMessage({ message }));
64    }
```

Kuvio 21. Palvelu, joka reagoi WebSocket-palvelimen tapahtumiin.

WebSocket-palvelu lähettää yleisen actionin eteenpäin, jota kuunnellaan useassa eri ominaisuusmoduulissa, ja niistä kukin pystyy reagoimaan toimintoon haluamallaan tavalla. Viestit pitävät sisällään tyyppiarvon, jonka mukaan moduulien efektissä suodatetaan viesti. Jos viesti pitää sisällään moduulin haluaman tyypin, laukaistaan uusi action, jonka avulla haetaan uutta dataa, ja päivitetään tila reaktiivisesti. (ks. kuvio 22)

5-AngularAPP - coach-websocket.effects.ts

```
8   @Injectable()
9   export class CoachWebSocketEffects {
10    receivedMessage$ = createEffect(() =>
11      this.actions$.pipe(
12        ofType(WebSocketActions.receivedMessage),
13        map((action) => action.message),
14        filter((message) => message.type === 'EXERCISE_FILE_CONVERT_COMPLETE'),
15        map((message) =>
16          CoachCoachingExerciseActions.getCoachingExerciseFiles({
17            exerciseId: message.exerciseId
18          })
19        )
20      )
21    );
22    constructor(private actions$: Actions) {}
23  }
```

Kuvio 22. Efekti, joka reagoi WebSocket muutokseen.

8 Tutkimustulokset ja johtopäätökset

Tässä tutkimuksessa perehdyttiin verkkosovellusten tilaan ja sen hallintaan. Tutkimuksen tavoitteena oli saavuttaa syvempi ymmärrys verkkosovellusten tilanhallinnasta, sekä sen toteuttamisesta Angular-sovelluksissa NgRx-kirjastoa hyödyntäen. Tässä luvussa käydään läpi vastaukset tutkimuskysymyksiin ja tuloksista saadut johtopäätökset.

Mitä tilanhallinta verkkosovelluksissa tarkoittaa?

Modernit verkkosovellukset ovat muodostuneet kokonaisuuksiksi, jotka sisältävät paljon käyttäjäinteraktiota sekä dynaamisia toimintoja. Staattisten verkkosivujen sijaan verkkosovelluksia kehitetään sovellusaluksilla tehokkaiksi yhden sivun SPA-sovelluksiksi, jotka pitävät sisällään monimutkaisia reitityksiä ja laajan komponenttiarkkitehtuurin.

Kaiken verkkosovelluksessa tapahtuvan tilan tarkkailu, ylläpito ja muutosten hallinta on tilanhallintaa. Koska modernit verkkosovellukset voivat olla todella isoja sovelluskokonaisuuksia, on kehitystyön ja sovelluksen toiminnan kannalta erittäin tärkeä ottaa huomioon millä teknologialla sovelluksen tilanhallinta tulee ratkaista.

Mikä on NgRx?

NgRx on RxJs:ään pohjautuva kokoelma kirjastoja keskitetyn tilanhallinan toteuttamista varten. NgRx:n arkkitehtuuri perustuu Redux arkkitehtuuriin, jonka tavoitteena on keskittää sovelluksen tila yhteen lähteeseen.

Perinteisen Angular-kehitysmallin mukaan sovelluksen logiikkaa hoitavat komponentit ja palvelut. Sen sijaan NgRx:llä saadaan eristettyä sovelluksen koko tilanhallinnan logiikka omaksi entiteetiksi. Tämä tarkoittaa sitä, että kehitysvaiheessa, ja kun johonkin sovellusprojektiin tulee tarve tehdä muutoksia, on helppo löytää toimenpiteitä vaativat kohteet. Etenkin pienien muutosten implementointi helpottuu kehittäjille.

NgRx:llä pystytään toteuttamaan monipuolisesti sovelluksen vaatimaa tilanhallintaa, se ei rajoitu vaan tiedon ylläpitämiseen, vaan sen avulla voidaan suorittaa ja hallinnoida myös käyttöliittymän toimintoja, sovelluksen reitityksiä, sekä sovelluksen taustalla tapahtuvaa logiikkaa.

NgRx:n käyttö tuo mukanaan myös omia haasteitaan. Sen käyttöönotto voi olla aluksi haastaavaa, sillä se vaatii syvää ymmärrystä Angularista sekä RxJs:stä. NgRx:n mukana myös sovelluksen kansiorakenne kasvaa ja se voi aluksi sekaannuttaa etenkin kehittäjiä, joille NgRx on uusi tuttavuus.

Redux DevToolsin kehitystyökalujen ansiosta NgRx:llä on myös kehittäjän helppo suorittaa vianmääritystä. Kehittäjä pystyy seuraamaan mistä päin sovelluksesta mikäkin data kulkee, ja täten havaitsemaan mahdollisia ongelmakohteita. Kehityksen tueksi NgRx:stä on vapaasti saatavilla myös laajat viralliset ohjedokumentaatiot.

Kuinka keskitetty tilanhallinta voidaan toteuttaa toimeksiantajan teknologiaympäristössä?

NgRx-kirjasto tarjoaa toimeksiantajalle Angular-verkkosovellusten kehitykseen tehokkaan ratkaisun tilanhallinnan toteuttamiseen. Toimeksiantajan kehitysympäristössä NgRx on toimiva valinta sen skaalautuvuuden sekä joustavuuden ansiosta. NgRx:n avulla kyetään toteuttamaan monipuolisesti haastaviakin tilanhallinnallisia ratkaisuja vaativia dynaamisia, sekä reaktiivisia verkkosovelluksia.

Toimeksiantajan toimintaympäristössä kehitetään samanaikaisesti useampia verkkosovelluksia. Angular sekä NgRx ovat tehokkaita teknologioita uudelleenkäytetävyyden näkökulmasta. Kun jotain tilanhallinnallisia ongelmia on kehitetty Angularilla sekä NgRx:llä yhdessä projektissa, niitä voidaan joustavasti hyödyntää ja mukauttaa muihinkin projekteihin mukaan, säästäen kehitysaikaa. NgRx myös helpottaa eri sovellusten kehityksen yhtenäistämistä, sillä tilanhallinnan arkkitehtuuri on samanlainen jokaisessa sovelluksessa.

NgRx on toimiva valinta tilanhallinnan kehitykseen, mutta vastaan voi tulla myös tilanteita, jolloin sen käyttöönottoa kannattaa harkita. Etenkin jos kehityksen kohteena on jokin hieman pienempi sovellus, voi NgRx:n käyttöönotto loppujen lopuksi syödä enemmän aikaa kuin sovelluksen kehitys ilman sitä. Eli kun on tarve kehittää yksinkertaisia asioita nopeasti, ei NgRx ole tehokkain ratkaisu.

9 Pohdinta

Tutkimuksen onnistuminen ja luotettavuus

Tutkimuksen toteuttaminen onnistui hyvin. Tutkimuksen teoria pohjautui artikkeleihin, kirjallisuuteen, tutkimuksiin sekä hyvin paljon Angularin, RxJs:n sekä NgRx:n virallisiin ohjesivustoihin ja niiden sisältämään dokumentaatioon. NgRx on teknologiana suhteellisen tuore, mutta siihen löytyi hyvin tietoa eri lähteistä.

Tutkimuksen tuloksena on saavutettu syvempi ymmärtäminen verkkosovellusten tilanhallintaa sekä NgRx:ää kohtaan. Tutkimuksen tavotteista kommunikointiin toimeksiantajan kanssa sen suorittamisen aikana paljon, ja tulokset vastaavat toimeksiantajan toiveita.

Tutkitun aineiston perusteella voidaan todeta että NgRx-kirjaston käyttö toimeksiantajan kehitysympäristössä on toimiva sekä tehokas valinta. Sovelluskehityksessä on kuitenkin aina mietittävä tapauskohtaisesti teknologisia ratkaisuja ja etenkin pienemmissä, nopeaa toteutusta vaativissa tehtävissä NgRx ei välttämättä ole oikea valinta.

Tämän tutkimuksen aikana Angular 11 on sovelluskehyksen uusin versio, ja NgRx:n tämänhetkinen versio on 10. Sovelluskehityksessä tekniikat päivittyvät ja muuttuvat jatkuvasti joten teknologioiden tulevissa versioissa kaikki tämän työn tulokset eivät välttämättä enää pidä paikkaansa.

Tutkimustyön mahdollinen laajentaminen

Tutkimuksen aikana tuli esiin paljon muitakin teknologioita verkkosovellusten tilanhallinan kehitystä varten. Hyvä idea lähteä laajentamaan tätä tutkimustyötä olisi perehtyä muihin Angularille saataville oleviin tilanhallinnan teknologiaratkaisuhuin kuten NgXs:ään sekä Akitaan. Myös erittäin laajasti käytössä olevat muut sovelluskehykset kuten React sekä Vue.js voisivat olla hyviä kohteita tilanhallinnan vertailuun.

Lähteet

Actions. N.d. NgRx virallinen ohjesivusto. Viitattu 12.1.2021. https://ngrx.io/guide/store/actions

Architecture overview. N.d. Angularin virallinen ohjesivusto. Viitattu 8.12.2020. https://angular.io/guide/architecture

Bainomugisha, E., Carreton, A. L., van Cutsem, T., Mostinckx, S. & de Meuter, W. 2012. A survey on reactive programming. ACM Computing Surveys, 45, 4, 1-34. Viitattu 7.12.2020.

Cheng, F. 2018. Build Mobile Apps with Ionic 4 and Firebase: Hybrid Mobile App Development, Second Edition, Chapter 6 – State Management with NgRx. Apress. Viitattu 4.12.2020.

Clavijo, P. 2018. Managing the state of your application with the Redux pattern. Viitattu 14.1.2021. https://www.slideshare.net/paucls/angular-and-redux

Effects. N.d. NgRx virallinen ohjesivusto. Viitattu 13.1.2021. https://ngrx.io/guide/effects

Farhi, O. 2017. Reactive Programming with Angular and ngrx: Learn to Harness the Power of Reactive Programming with RxJs and ngrx extensions. Appress. Viitattu 7.1.2021.

Kananen, J. 2008. Kvali: Kvalitatiivisen tutkimuksen teoria ja käytänteet. Jyväskylä: Jyväskylän ammattikorkeakoulu. Viitattu 4.12.2020.

Kumar, D. 2019. Angular Essentials: The Essential Guide to Learn Angular. BPB Publications. Viitattu 8.12.2020.

Medeiros, A. 2014. The introduction to Reactive Programming you've been missing. Viitattu 7.12.2020. https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

Noring, C. 2018. Architecting Angular Applications with Redux, RxJS, and Ngrx : Learn to Build Redux Style High-Performing Applications with Angular 6. Birmingham: Packt Publishing. Viitattu 10.12.2020.

Operators. N.d. RxJs virallinen ohjesivusto. Viitattu 15.12.2020. https://rxjs.dev/guide/operators

Reducers. N.d. NgRx virallinen ohjesivusto. Viitattu 12.1.2021. https://ngrx.io/guide/store/reducers

Selectors. N.d. NgRx virallinen ohjesivusto. Viitattu 13.1.2021. https://ngrx.io/guide/store/selectors

Liitteet

Liite 1. App.module.ts tiedoston @NgModule-määritykset

4-Angular - app.module.ts

```
34  @NgModule({
35    declarations: [
36      AppComponent,
37      HomeContainerComponent,
38      NotFoundPageContainerComponent,
39      TopToolbarComponent,
40      ...components,
41    ],
42    imports: [
43      BrowserModule,
44      CommonModule,
45      MaterialModule,
46      MatTableModule,
47      FlexLayoutModule,
48      HttpClientModule,
49      StoreModule.forRoot(reducers, {
50        runtimeChecks: {
51          strictStateImmutability: true,
52          strictActionImmutability: true,
53          strictStateSerializability: true,
54          strictActionSerializability: true,
55        },
56      }),
57      StoreRouterConnectingModule.forRoot({
58        routerState: RouterState.Minimal,
59        serializer: RouterStateSerializer,
60      }),
61      environment.production ? [] : StoreDevtoolsModule.instrument(),
62      EffectsModule.forRoot(effects),
63      BrowserAnimationsModule,
64      SharedModule,
65      AuthModule,
66      SalesEventsModule,
67      LiveSalesEventModule,
68      AppRoutingModule,
69      ServiceWorkerModule.register('ngsw-worker.js', {
70        enabled: false,
71      }),
72    ],
```

Liite 2. Store-kansiorakenne