ANGULAR-SOVELLUKSEN TILANHALLINTA NGRX-KIRJASTOLLA

Ville Haapavaara

Opinnäytetyö
Marraskuu 2018
Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

HAAPAVAARA, VILLE:
Angular-sovelluksen tilanhallinta NgRx-kirjastolla

Opinnäytetyö 42 sivua
Marraskuu 2018

Opinnäytetyössä käsitellään Angular-sovellusten tilanhallintaa NgRx-kirjastolla. Opinnäytetyön toimeksiantaja oli tamperelainen DiCode Oy. DiCoden projekteissa oli ollut haasteita sovelluksen tilanhallinnassa. Eri puolilla sovellusta esitettävän saman tilan tuli olla käyttäjän toimista riippumatta aina ajan tasalla. Tämän lisäksi tilanhallinnan tuli olla mahdollisimman optimoitua. Ratkaisuksi tällaisiin haasteisiin mietittiin tilanhallintakirjastoa.

Tilanhallintakirjastojen käytöstä oli kuultu paljon hyvää. Tämän vuoksi opinnäytetyössä haluttiin selvittää, pitävätkö nämä huhut paikkansa ja millaisia ratkaisuja kirjasto tarjosi sovellusten tilan hallitsemiseksi. Koska DiCodella kehitettyjen sovellusten käyttöliittymät oli toteutettu Angular-kehyksellä, päädyttiin selvitystyössä tutkimaan Angular-sovelluksille suunnattua NgRx-tilanhallintakirjastoa. Tavoitteena oli löytää ratkaisuja sovelluksen tilanhallinnallisiin haasteisiin nimenomaan NgRx-kirjastoa hyödyntämällä.

NgRx-kirjaston tarpeellisuuden ja toimivuuden selvittämiseksi ohjelmoitiin pieni esimerkkisovellus. Esimerkkisovelluksen tilanhallinta toteutettiin käyttämällä NgRx-kirjastoa. Sovelluksessa käytettiin NgRx:n tehtäviä, vähentäjiä, varastoa ja valitsimia. Tilanhallinnan toteuttaminen NgRx:llä vaatii yleisen käsityksen mukaan paljon pohjakoodia, mutta sen määrää ei opinnäytetyön edetessä pidetty kuitenkaan liiallisena. Lisäksi esimerkkisovelluksen kehittämisen aikana miellyttiin ajatukseen, jossa sovelluksen tilaa hallitaan sovelluksessa lähetettävien tehtävien kautta.

NgRx on monipuolinen kirjasto Angular-sovelluksen tilanhallintaan. Sen avulla voidaan sovelluksen tilanhallinta eriyttää helposti omaksi sovellusosakseen. NgRx myös tukee Angularin moduuliajattelua, ja sen käyttäminen ohjaa yleisesti hyvänä pidettyyn Angular-sovelluksen sovellusarkkitehtuuriin. NgRx:n käyttäminen edellyttää kuitenkin erinomaista RxJS-kirjaston tuntemusta. Selvitystyöhön tehty esimerkkisovellus on pieni sovellus. Tämän vuoksi tarkka kuva NgRx:n tarpeellisuudesta ja toimivuudesta saadaan, kun kirjasto otetaan käyttöön monimutkaisessa sovelluksessa.

Asiasanat: Angular, NgRx, tilanhallinta

ABSTRACT

Tampere University of Applied Sciences
Business Information Systems
Software Development

HAAPAVAARA, VILLE:
Managing State in an Angular Application with NgRx

Bachelor's thesis 42 pages
November 2018

This thesis deals with managing application state in an Angular application with NgRx. The topic of this thesis comes from a customer. The commission was given by DiCode Oy which is a company located in Tampere. Some of the projects at DiCode had had application's state related challenges. This was due to the fact that usually the state of the application should be up-to-date, no matter what the user's actions have been. The state management should also be as optimized as possible. Studying and testing the pros and cons of using NgRx in Angular applications seemed to provide possible solutions to these challenges.

Using an application state management library in programming has recently been addressed as a functional and a useful tool in the field of software development. One of the goals of this thesis was to examine how accurate these assumptions were, and how useful an application's state management library would really be. Because the front end applications produced in DiCode Oy were mainly Angular-based, the focus of this thesis was to study NgRx.

A small exemplary application was programmed to determine the necessity and functionality of the NgRx library. The state management of the exemplary application was implemented using the NgRx library. The application used NgRx's actions, reducers, store and selectors. It is known that NgRx requires plenty of boilerplate code, but its amount was not considered excessive. The idea of controlling the application state in actions transmitted through-out the application was considered powerful yet clear.

NgRx is a versatile library for managing state in Angular applications. It can be used to easily separate the application's state management into own section of the application. NgRx also supports Angular's modular thinking. It guides the developer to Angular's commonly preferred application architecture. However, using NgRx requires an excellent knowledge of the RxJS library. Additionally, the exemplary application developed for this thesis was a relatively small application. Therefore, an exact picture of the necessity and functionality of NgRx will be obtained when the library is used in a more complex application.

Key words: Angular, NgRx, state management

SISÄLLYS

1 JOHDANTO........6
1.1 Nykyiset web-sovellukset........6
1.2 Keskitetty tilanhallinta NgRx-kirjastolla........6
1.3 Opinnäytetyön tausta, tavoite ja tarkoitus........7
1.4 Opinnäytetyön sisältö ja eteneminen........8
2 ANGULAR-SOVELLUSKEHITYS........10
2.1 Angular........10
2.2 Moduulit, komponentit ja palvelut........11
2.3 Tila ja tilanhallinta........14
2.4 Node package manager – npm........16
3 TILANHALLINTA NGRX-KIRJASTOLLA........18
3.1 Keskitetty tilanhallinta........18
3.2 Tehtävät........20
3.3 Vähentäjät........22
3.4 Varasto........24
3.5 Tilan valitseminen ja tehtävien lähettäminen........26
3.6 Efektit........28
3.7 Reitit........31
4 VIRHEENETSINTÄ TILASTA........33
4.1 NgRx-sovelluksen tilan kehitystyökalu........33
4.2 Kehitystyökalun käyttöönotto........34
5 JOHTOPÄÄTÖKSET JA POHDINTA........36
5.1 Opinnäytetyön toteutus ja onnistuminen........36
5.2 Päätelmät NgRx:n käytöstä........36
5.3 Ajatuksia selvitystyön laajentamisesta........39
LÄHTEET........41

ERITYISSANASTO

Detalji-sivu – esimerkkisovelluksen sivu, jolla näytetään taskin tiedot; oma käsite
Front-end – web-sovelluksen selainpuoli
HTML – *Hypertext Markup Language*; kieli web-dokumenttien laatimiseen
JavaScript – dynaaminen web-ohjelmointikieli
NGXS – NgRx:n kaltainen tilanhallintakirjasto
Puhdas funktio – *pure function*; funktio, joka palauttaa samoilla argumenteilla aina saman tuloksen ja jolla ei ole sivuvaikutuksia
Redux – NgRx:n kaltainen tilanhallintakirjasto
RxJS – *Reactive Extensions for JavaScript*; kirjasto reaktiiviseen ohjelmointiin
SPA-sovellus – engl. *single page application*; yhden sivun sovellus
Taski – esimerkkisovelluksen tehtävä
TypeScript – ohjelmointikieli, joka laajentaa JavaScriptia

1 JOHDANTO

1.1 Nykyiset web-sovellukset

Internet-selaimilla käytettävät web-sovellukset ovat nykyään yhä monimutkaisempia. Tämä johtuu siitä, että sovelluksiin pyritään saamaan sekä työpöytäsovelluksien monimutkaisuus että käytön ja toimivuuden sulavuus. Jotta nämä molemmat ominaisuudet saadaan toteutettua mahdollisimman helposti, käytetään apuna usein erilaisia sovelluskehyksiä. Yksi esimerkki tällaisista kehyksistä on Angular, joka on kehys selainpuolen (front end) sovelluksien kehittämiseen. Angular tarjoaa monipuoliset työkalut niin kutsuttujen yhden sivun sovellusten (single page application, myöh. SPA) rakentamiseen.

Monipuolisuudestaan huolimatta Angular ei kuitenkaan pysty kaikkiin tilanhallinnan haasteisiin, ainakaan kovin ketterästi. Erityisesti SPA-sovelluksissa on paljon tilaa, jota tulee hallita. Kun sovellusta laajennetaan, tilanhallinta pelkästään Angular-sovelluksen komponenteissa (components) ja palveluissa (services) käy monimutkaiseksi: ajantasaisen tilan tulee olla saatavilla eri puolilla sovellusta, jota monien komponenttien on kyettävä muuttamaan. Yksi vaihtoehto tällöin on keskittää sovelluksen tila yhteen paikkaan esimerkiksi NgRx-kirjaston avulla.

1.2 Keskitetty tilanhallinta NgRx-kirjastolla

NgRx on kirjasto, joka tarjoaa työkalut keskitetyn tilanhallinnan toteuttamiseen. Se on suunniteltu ja luotu nimenomaan Angular-sovelluksille. NgRx käyttää hyödykseen RxJS:n ominaisuuksia, aivan kuten Angularkin. Sovellukseen voidaan helposti lisätä NgRx:n tarjoamia monipuolisia ominaisuuksia kattamaan sovelluksen tilanhallinnalliset tarpeet. NgRx:ssä on monipuolisesti erilaisia ominaisuuksia, joita voidaan lisätä sovellukseen sitä mukaa kun niitä tarvitaan.

Keskitetyssä tilanhallinnassa ajatuksena on, että koko sovelluksen tila on keskitetty yhteen paikkaan, varastoon (store). Varastossa oleva tila on kuin yksi iso JavaScript-objekti, jolla on puumainen rakenne. Objekti haarautuu ominaisuuskohtaisiin tilamuuttujiin.

Ohjelmoija voi itse määrittää, mitä dataa hän haluaa tilamuuttujiin tallentaa. Kun tila on tällä tavalla yhdessä varastossa, on siitä helppo ottaa palasia käytettäväksi eri puolilla sovellusta.

NgRx:n kaltainen keskitetyn tilanhallinnan ratkaisu voidaan toteuttaa ohjelmoimalla se itse, mutta usein sen luomiseen käytetään jotain tilanhallintaan tarkoitettua kirjastoa. Kirjastoja voidaan ajatella kokonaisuuksina, jotka nivovat toisiinsa keskeisesti liittyvät toiminnallisuudet yhdeksi paketiksi. Esimerkiksi NgRx-kirjastossa nämä toiminnallisuudet liittyvät sovelluksen tilanhallintaan. Pääajatus kirjastoissa on, että niiden tarjoamat toiminnallisuudet ovat uudelleen käytettäviä. Ohjelmoija saa käyttöönsä kirjaston tarjoamat toiminnallisuudet kutsumalla kirjaston rajapinnassa (interface) olevia funktioita.

Tilanhallinnan keskittäminen varastoon ei ole aivan ongelmaton asia. Keskitetyn tilanhallinnan toteuttaminen on huomattavasti monimutkaisempaa kuin tilan hallitseminen komponenteissa tai palveluissa. Muutokset keskitettyyn tilanhallintaan voivat myös vaatia usean tiedoston muuttamista ja ylläpitämistä. Angularilla toteutetuissa SPA-sovelluksissa tilaa voidaan hallita sovelluksen komponentti- ja palvelutasolla keskitetyn tilanhallinnan sijaan. Yhdessä nämä tasot luovat oivan tavan hallita sovelluksen tilaa. Keskitetty tilanhallinta ei tällöin ole välttämättömyys. Abramov (2016), Reduxin (NgRx:ää vastaava kirjasto) luoja, onkin todennut, että Reduxin lisäämistä projektiin on tarkoin harkittava eikä sitä välttämättä edes tarvita. Tämä pätee myös NgRx-kirjastoon.

1.3 Opinnäytetyön tausta, tavoite ja tarkoitus

Työelämässä on tullut vastaan erilaisia sovelluksen tilanhallinnallisia haasteita, erityisesti Angular-sovellusten kohdalla. On ollut esimerkiksi tarve pitää ajan tasalla sovelluksessa eri puolilla näytettävä sama data. Näiden ongelmien ratkaisut ovat tuntuneet jääneen vajaiksi, ja niitä on ollut jälkeenpäin hankala ymmärtää tai muuttaa. Keskitetty tilanhallinta ja Redux olivat termeinä tuttuja. Niiden kuitenkin ajateltiin olevan monimutkaisia ja vaativan paljon pohjakoodia. Tässä työssä haluttiin selvittää, pitivätkö tilanhallintaan liittyvät oletukset paikkansa ja voidaan sovelluksen tilanhallintaa parantaa keskittämällä se.

On olemassa useita NgRx:n kaltaisia kirjastoja, joista tunnetuimmat ovat NgRx, Redux ja NGXS. Selvitystyön alussa huomattiin, että NGXS on vielä melko tuore kirjasto. Tämän vuoksi lopullinen valinta tehtiin Reduxin ja NgRx:n välillä. Redux oli joukon tunnetuin kirjasto, ja se olisi ollut varmasti myös hyvä valinta. Lopulta kuitenkin päädyttiin NgRx:ään, koska se on suunniteltu nimenomaan Angular-sovelluksille. NgRx:ää on myös päivitetty tasaisin väliajoin. Lisäksi NgRx on ollut saatavilla jo jonkin aikaa. Tästä voitiin olettaa, että kirjaston alkuvaikeudet ovat jo takanapäin. Työssä haluttiin myös tutkia NgRx:n tarjoamaa erityistä kehitystyökalua, joka helpottaa sellaisen sovelluksen kehittämistä, jossa on käytössä NgRx.

Opinnäytetyö on ennen kaikkea selvitystyö NgRx:n sopivuudesta Angulariin ja kirjaston tarpeellisuudesta. Työn tavoitteena on selvittää, millaisiin sovelluksiin keskitetty tilanhallinta erityisesti sopii. Työssä tutkittiin myös, onko sovelluksen kehittäminen merkittävästi työläämpää keskitetyn tilanhallinnan monimutkaisuuden ja sen vaatiman pohjakoodin määrän vuoksi. Tarkoitus oli löytää ratkaisuja työelämässä eteen tulleisiin, sovellusten tilanhallinnallisiin haasteisiin.

1.4 Opinnäytetyön sisältö ja eteneminen

Opinnäytetyö etenee teoriasta käytäntöön. Jotta NgRx:ää pystyy ymmärtämään, täytyy ensin saada peruskäsitys Angularista. Johdannon jälkeen käsitellään sovelluskehitystä nimenomaan Angular-sovellusten näkökulmasta. Teorian selventämiseksi käsittely rajattiin kattamaan vain Angular. AngularJS:n käsittely jätettiin selvityksestä kokonaan pois. Luvussa kaksi kerrotaan, mikä Angular on ja millaisista osasista sillä toteutetut SPA-sovellukset rakentuvat. Luvun loppupuolella käsitellään myös sitä, miten riippuvuuksia muihin sovelluskirjastoihin ja sovelluskehyksiin Angular-sovelluksessa hallitaan. Lisäksi tilan hallinta Angular-sovelluksissa, joissa ei ole käytetty keskitettyä tilanhallintaan, selitetään tämän luvun aikana.

Angular-taustoituksen jälkeen kolmannessa luvussa luodaan lyhyt katsanto keskitettyyn tilanhallintaan ja siirrytään tarkastelemaan itse NgRx:ää. NgRx:n perusteita käydään läpi ohjelmoidun esimerkkisovelluksen kautta ja siitä valittujen koodiesimerkkien avul-

la. Samalla selvitetään NgRx:n tarpeellisuutta ja toimivuutta käytännössä. Esimerkkisovellus on tehtävien hallintasovellus, jossa voidaan luoda tehtäviä ja kommentoida niitä.

Luvussa neljä tutkitaan vielä NgRx:n tarjoamaa kehitystyökalua. Siinä selvitetään, onko se tarpeellinen, kuinka se otetaan käyttöön ja millaiset ominaisuudet se tarjoaa niiden sovellusten kehittämiseen, joissa se on käytössä. Kehitystyökalua ja sen tarpeellisuutta käsitellään koodiesimerkkien kautta. Lopuksi viidennessä luvussa tehdään yhteenveto selvityksestä. Siinä kootaan yhteen tärkeimmät huomiot NgRx-kirjaston hyvistä ja huonoista puolista ja tehdään loppupäätelmä kirjaston tarpeellisuudesta.

Opinnäytetyössä pyritään käyttämään tärkeimpien käsitteiden suomennoksia. Koska ohjelmointiala on kuitenkin vahvasti sidoksissa englannin kieleen, kerrotaan käsittelyssä aina myös alkuperäiset englanninkieliset termit. Termi ilmoitetaan sulkeissa suomenkielisen termin jälkeen. Työstä on koostettu myös sanasto, johon on koottu keskeisiä työssä esiintyviä käsitteitä ja niiden mahdolliset englanninkieliset vastineet selityksineen.

2 ANGULAR-SOVELLUSKEHITYS

2.1 Angular

Angular on sovelluskehys (application framework) web-sovellusten rakentamiseen. Toisin kuin kirjastot, kehykset, kuten siis Angularkin, tarjoavat raamit, joihin sovellus rakennetaan. Kehyksiä on erilaisia eri tarkoituksiin ja eri ohjelmointikielillä. Angular on web-sovelluksen selainpuolen JavaScript-kehys (front end framework). Karkeasti voidaan ajatella, että kehys kutsuu ohjelmoijan kirjoittamaa koodia, kun taas ohjelmoija voi tarvittaessa kutsua kirjastojen tarjoamia toiminnallisuuksia.

Angularilla toteutetut sovellukset ovat niin sanottuja yhden sivun sovelluksia (single page application, myöh. SPA). SPA-sovellusten ajatuksena on, että käyttäjä lataa vain yhden html-dokumentin. Dokumentti sisältää hyvin vähän HTML:lää, usein vain *head*- ja *body*-elementit. SPA-sovellus ladataan dokumentin sisältämään säiliö-elementtiin (container). Säiliöelementti on tyypillisesti joko dokumentin *body*-elementti tai jokin *body*-elementin lapsi-elementeistä. *Body*-elementti voi sisältää säiliö-elementin lisäksi myös muita elementtejä, joita ei tarvitse muuttaa käyttäjän toimintojen mukaan. Tällaisia elementtejä voivat olla esimerkiksi ylä- ja alaviite-elementit. (Klauzinski & Moore 2016, Methods of presenting an SPA container)

Aina uuden HTML-dokumentin lataamisen sijaan SPA-sovelluksen säiliö-elementtiä muokataan JavaScriptin keinoin. Dokumenttia muokataan esimerkiksi käyttäjän toimien tai palvelimen palauttaman datan mukaan dynaamisesti. Klauzinskin ja Mooren mukaan SPA-sovellukset ovat jouheita käyttää, koska jo ladattua dokumenttia vain muokataan. SPA-sovellukset voivat myös tarvita toimiakseen vähemmän verkkoliikennettä, ja ne kuormittavat palvelimia vähemmän. Tämä johtuu siitä, ettei sovelluksen käyttö vaadi eri html-dokumenttien lataamista palvelimelta sivulta toiselle navigoidessa. (Klauzinski & Moore 2016, Preface)

Angular-sovellukset ohjelmoidaan tyypillisesti käyttämällä TypeScriptia. TypeScript on ohjemointikieli, joka laajentaa JavaScriptia. TypeScript tuo mukanaan muun muassa koodin tyypittämisen sekä rajapinnat. Tyypittäminen tarkoittaa, että ohjelmoija voi mää-

rittää minkä tyypin dataa kuhunkin muuttujaan voidaan tallentaa tai funktio voi ottaa vastaan. Rajapintojen avulla ohjelmoija voi määrittää, mitä dataa mihinkin objektiin voidaan tallentaa. Käyttämällä rajapintoja, ohjelmoija voi luoda käytännössä omia tyyppejä. Niiden käyttäminen helpottaa muun koodin tyypittämistä. TypeScript auttaa vain sovellusta ohjelmoitaessa, sillä se käännetään JavaScriptiksi (Gabe de Wolff, Jansen & Vane 2016). JavaScript itse ei tue esimerkiksi muuttujien tyypitystä eikä rajapintoja.

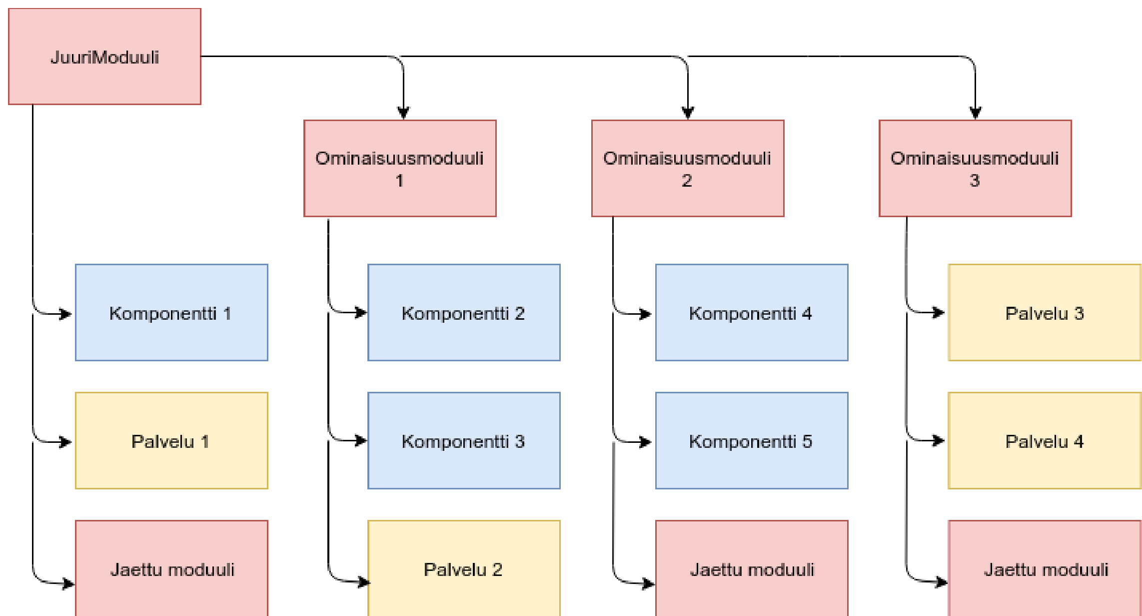
Angular-sovellukset pääosaset ovat moduulit (module), komponentit (component) sekä palvelut (service). Näistä osasasista kerrotaan tarkemmin seuraavassa luvussa. Edellä mainittujen sovellusosasten lisäksi myös erilaiset vartijat (guards), direktiivit (directive) ja putket (pipes) ovat keskeisiä Angular-sovellusten osasia. Niitä ei kuitenkaan käsitellä tässä opinnäytetyössä.

2.2 Moduulit, komponentit ja palvelut

Angular-sovellusten voidaan ajatella koostuvan moduuleista. Jokainen Angular-sovellus koostuukin vähintään yhdestä moduulista, juurimoduulista (root module) (Arora & Hennessy 2018, Angular Modules). Moduuleissa ilmoitetaan sovellukseen kuuluvat komponentit, palvelut sekä muut Angular-sovelluksen osaset.

Usein Angular-sovellukset haarautetaan juurimoduulista ominaisuuksiensa mukaan niin sanottuihin ominaisuusmoduuleihin (feature module). Ominaisuusmoduuleissa voidaan ilmoittaa vain moduulia vastaavaan ominaisuteen liittyvät komponentit ja palvelut. Moduulien avulla sovelluksen osat voidaan organisoida yhteenkuuluviksi kokonaisuuksiksi.

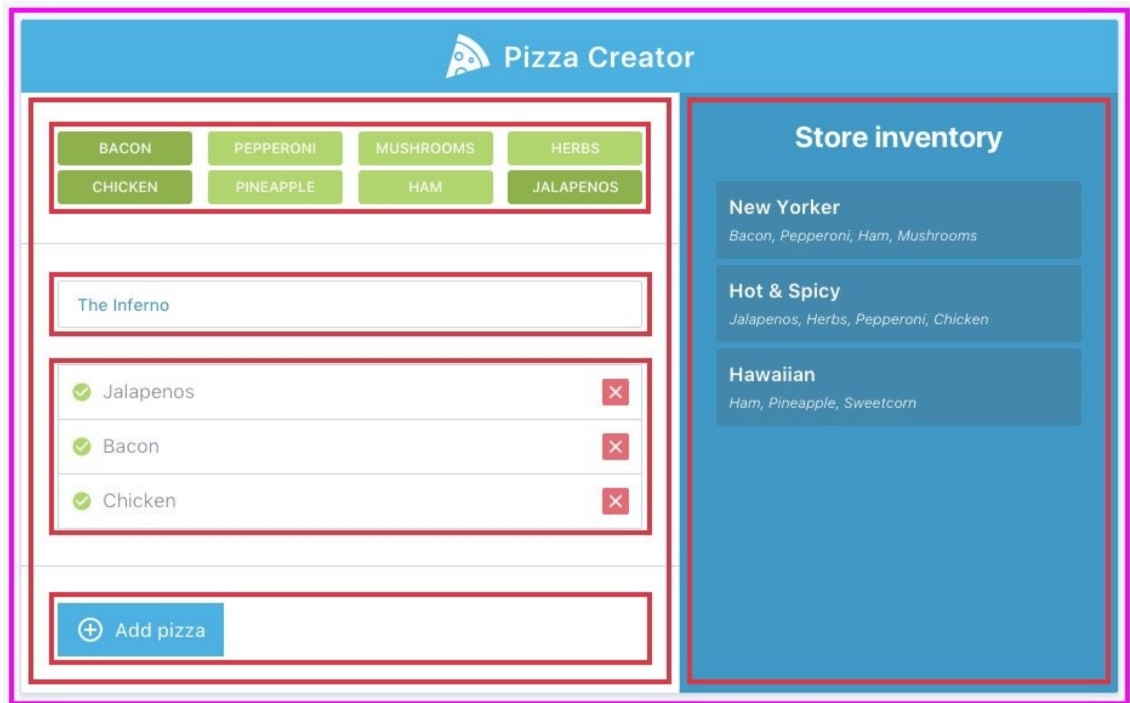
Moduulit voivat ottaa käyttöönsä myös muita moduuleita. Niistä voidaan myös viedä moduuliin kuuluvia osasia muiden moduulien käytettäväksi. Angular-sovellukset usein sisältävätkin osasia, jotka tarvitsevat toimiakseen samoja moduuleita. Tällöin on tapana tehdä jaettu moduuli (shared module), joka pitää sisällään sovelluksessa yleisesti käytettäviä moduuleita, komponentteja, palveluita sekä muita Angular-sovellusten osasia. Kuvassa 1 on hahmoteltu Angular-sovelluksen organisointia moduulien avulla.



Kuva 1: Angular-sovelluksen jakaminen moduuleiksi (Haapavaara 2018)

Komponentit vastaavat sovelluksen näkymistä. Ne luodaan käyttämällä Angularin *Component*-koristelijaa (decorator). *Component*-koristelijaan voidaan sisällyttää metadataa, kuten esimerkiksi komponentin malli (template) sekä sen tyylit. Näiden lisäksi komponentille luodaan myös TypeScript-luokka. Siihen sisällytetään komponentin ominaisuudet ja luokan toimintaa ohjaavat funktiot. Luokkaan sisällytettyjen muuttujien arvoja voidaan esittää komponentin mallissa. Malli on HTML:ää, ja se määrittelee komponentin näkymän rakenteen.

Murray, Coury, Lerner ja Taborda kuvaavat Angular-sovellusta komponenttipuuna. Ylin komponenttipuun komponentti on usein nimetty *AppComponentiksi*. Se ladataan ensimmäisenä, kun Angular-sovellus käynnistetään. Ylin komponentti tulee ilmoittaa sovelluksen juurimoduulissa, joka on nimetty tyypillisesti *AppModuleksi*. Sovelluksen ylin komponentti koostuu muista komponenteista. Komponentit ovat siis koostettavia, eli ne voivat koostua useista erikokoisista komponenteista. Komponentti, joka koostuu toisista komponenteista, kutsutaan isäntä- tai vanhempikomponentiksi. Vanhempikomponentti koostuu komponenteista, joita kutsutaan lapsikomponenteiksi. (Murray ym. 2018, 81) Kuvassa 2 on jaettu rajaviivoin sovellus omiksi komponenteikseen. Purppura alue vastaa komponenttipuun ylintä komponenttia. Muut sovelluksen osat on jaettu omiksi komponenteikseen punaisilla rajaviivoilla.



Kuva 2: Sovelluksnäkymän jakaminen komponenteiksi (Motto 2017)

Angular-sovellukset rakennetaan tyypillisesti siten, että osa komponenteista vastaa näkymän tilasta ja näkymään liittyvistä toiminnallisuuksista. Tällaisia komponentteja kutsutaan sisältökomponenteiksi (container component). Sisältökomponentit pitävät sisällään itse sisältökomponentin näkymästä vastaavia esityskomponentteja (presentational component). Esityskomponentit pitävät sisällään esimerkiksi lomakkeita tai taulukoita. Yksi hyvä tapa rakentaa Angular-sovelluksia on siten, että lomakkeen lähettäminen tai taulukon rivin poistaminen ei tapahdu suoraan esityskomponentissa, vaan esityskomponentti ohjaa tehtävän sisältökomponentille. Sisältökomponentti suorittaa tehtävän ja muuttaa esityskomponenttien näkymään sen mukaan.

Moduulien ja komponenttien lisäksi Angular-sovellukset koostuvat usein myös palveluista. Myös ne ovat TypeScript-luokkia, ja ne koristellaan *Injectable*-koristelijalla. Palvelut ovat yksiöitä (singleton), jotka yhdistetään komponentteihin tai toisiin palveluihin käyttämällä Angularin riippuvuusinjektointi-ominaisuutta (dependency injection). Injektoinnin avulla komponenttien tai toisen palvelun on helppo käyttää injektoidun palvelun ominaisuuksia.

Koska palvelut ovat TypeScript-luokkia, voidaan myös niille määritellä omat ominaisuudet ja funktiot. Palveluita käyttämällä voidaan vaikka osa komponentin logiikasta

siirtää omaan tiedostoonsa. Täten palvelut tarjoavat mainion paikan ohjelmoida sellaista sovelluslogiikkaa, jonka tulee olla usean sovellusosasen käytettävissä. Tällaista logiikkaa ovat esimerkiksi sovellustila tai api-kutsut (ohjelmarajapinta, application program interface), joiden tulee olla usein usean eri komponentin ja palvelun käytettävissä.

2.3 Tila ja tilanhallinta

Web-sovellukset voidaan jakaa staattisiin web-sivuihin ja dynaamisiin web-sovelluksiin. Duffyn (2004) mukaan staattiset web-sivut toimivat siten, että palvelin palauttaa html-dokumentteja käyttäjän lähettämien http-pyyntöjen mukaan. Keskusteluyhteys palvelimen ja käyttäjän välillä päättyy sillä hetkellä, kun käyttäjä vastaanottaa dokumentin. Palvelimen vastaanottaessa seuraavan http-pyynnön se ei tiedä kuka pyynnön lähetti. Jotta palvelin tietää pyynnön lähettäjän, tietoa käyttäjästä tulee säilyttää joko selaimessa tai palvelimella. Tällaiset Duffyn määrittelyyn sopivat web-sivut eivät kuitenkaan sisällä tilaa, jota tulisi hallita.

Angularilla luodut SPA-sovellukset ovat monimutkaisia ja dynaamisia sovelluksia: niiden sisältö muuttuu käyttäjän toimien mukaan. Tällaiset käyttäjäkohtaiset toimet edellyttävät, että sovellus sisältää tilaa. Tilaa voidaan ajatella sovelluksen sisältämänä tietona. Siihen voi sisältyä esimerkiksi tietoa kirjautuneesta käyttäjästä, tietokannasta saapunutta tietoa tai käyttäjän syöttämää lomaketietoa. Tilanhallinta tarkoittaa yksinkertaisesti tämän tiedon hallitsemista. Kun sovelluksen tilaa hallitaan, voidaan luoda monipuolisia ja käyttäjäkokemukseltaan parempia sovelluksia.

Yksinkertaisimmillaan tilaa hallitaan Angular-sovelluksessa komponenttitasolla. Angularin komponenteille voidaan luoda muuttujia, joihin voidaan tallentaa komponenttia koskevaa tila. Komponentin näkymä voidaan asettaa riippumaan näistä muuttujista. Tila ja näkymä muuttuu sitä mukaan, kun muuttujien arvot muuttuvat tai niihin sijoitetaan uusia arvoja.

Komponentti voi myös saada tilansa yläkomponentiltaan. Tämä tapahtuu Angularin *Input*-koristelijan kautta. Yläkomponentti voi *Input*-koristelijan kautta yhdistää oman muuttujansa alakomponenttiinsa. Yläkomponentista alakomponenttiin yhdistetty muut-

tuja ei tällöin voi olla yksityinen yläkomponentille. Muuttujan arvon muuttaminen toisessa komponentissa muuttaa sen arvoa myös toisessa, koska muuttuja on yhdistetty kahden komponentin välillä.

Alakomponentista voidaan myös kutsua yläkomponentin funktioita. Alakomponenttiin on tällöin määriteltävä erityinen *EventEmitter*-muuttuja, jonka lähettämiä arvoja kuunnellaan yläkomponentissa. Luotu *EventEmitter*-muuttuja koristellaan *Output*-koristelijalla. Kun muuttuja lähettää arvon, voidaan sitä käsitellä muuttujaa kuuntelevassa yläkomponentin funktiossa. Funktioiden kuunteleminen tapahtuu siis alhaalta ylöspäin, päin vastoin kuin muuttujien yhdistäminen yläkomponentilta alakomponentille.

Tilanhallinta komponenttitasolla muuttuu haastavaksi, kun muuttujia ja funktioita pitäisi yhdistää *Input*- ja *Output*-koristelijoiden avulla sisältökomponentilta usean näkymäkomponentin läpi jollekin tietylle näkymäkomponentille. Tästä käytetään nimitystä porauttaminen (prop drilling). Porauttamisessa attribuutti yhdistetään ylhäältä alaspäin eri komponenttien läpi komponenttipuussa. Porauttamalla *Input*- ja *Output*-muuttujia komponenttipuun läpi muuttujien määrä voi kasvaa kohtuuttomaksi. Doddsin (2018) mukaan tämä voi vaikeuttaa koodin lukemista ja ymmärtämistä. Komponenttien siirtäminen tai poistaminen komponenttipuun keskeltä voi myös olla jälkeen päin haastavaa (Dodds 2018).

Porauttamiselta voidaan välttyä, mikäli komponenteista tehtäisiin hyvin suuria. On kuitenkin suositeltavaa pilkkoa sovellus mahdollisimman pieniin ja rajattuihin palasiin komponenttien avulla (Noring 2018, Components from an architectural standpoint). Tällöin porauttaminen on välttämätöntä. Yksi ratkaisu porauttamisen aiheuttamiin ongelmiin on siirtää osa sovelluksen tilanhallinnasta Angularin palveluihin.

Palvelut jo itsessään tarjoavat toimivan tavan jakaa eri puolilla sovellusta tarvittavaa sovelluslogiikkaa. Tavallisten muuttujien ja funktioiden sijaan tilanhallintaratkaisu palvelussa voidaan toteuttaa käyttämällä esimerkiksi RxJS-kirjaston tarjoamia ominaisuuksia. RxJS-kirjasto tarjoaa monia hyödyllisiä ominaisuuksia, joita voidaan hyödyntää myös tilanhallinnassa. Yksi näistä ominaisuuksista on subjektit (subject). Subjektit ovat kuin säilöjä, jotka pitävät sisällään jonkin arvon, jota voidaan muuttaa. Subjekti voidaan tila-

ta (subscribe), esimerkiksi komponentissa tai palvelussa, jolloin subjektin arvon muuttuessa se lähettää uusimman arvonsa subjektin tilanneille.

RxJS sisältää erityisen subjektin, *BehaviorSubjectin*. Se poikkeaa tavallisesta subjektista siten, että se myös lähettää nykyisen arvonsa kun se tilataan. *BehaviorSubjektin* avulla voidaan luoda keskitetyn tilanhallinnan perusominaisuus, jossa viimeisin tieto on saatavilla eri puolilla sovellusta. Voidaan luoda esimerkiksi ominaisuus, jossa rajapintakutsun vastauksen arvo asetetaankin subjektiin. Kun subjekti saa rajapintakutsun vastauksen kautta uuden arvo, kaikki subjektin tilanneet saavat päivitetyn tiedon samaan aikaan. Näin tieto pysyy ajan tasalla eri puolilla sovellusta.

Tilanhallinnan toteuttamiseen palveluissa ei ole yhtä ja oikeaa tapaa, minkä vuoksi ratkaisut voivat vaihdella paljonkin ohjelmoijasta ja projektista riippuen. Koodista voi tulla hankalasti luettavaa, mikäli komponentissa joudutaan tekemään useita tilauksia muiden palvelukutsujen lisäksi. Tilanhallinta voi käydä myös kohtuuttoman monimutkaiseksi, kun tilaa hallitaan sekaisin sekä komponentti- että palvelutasolla. Myös sitä, mikä sovellusosanen muutti yhteisessä käytössä olevaa tilaa milläkin hetkellä, voi olla hankala seurata. Tällöin virheenetsintä sovelluksesta vaikeutuu.

2.4 Node package manager – npm

Web-tekniikoin, kuten myös Angular-kehyksellä, toteutetuissa sovelluksissa kirjastoriippuvuuksia hallitaan npm-paketinhallintatyökalu avulla. Web-sovelluksissa kirjastoja nimitetäänkin usein moduuleiksi (module) tai paketeiksi (package). Npm:llä voidaan asentaa sekä projektikohtaisia kirjastoja että globaaleja kirjastoja. Projektikohtaisia kirjastoja käytetään suoraan sovelluksissa, kun taas globaalit kirjastot tarjoavat ohjelmoijan käyttöön esimerkiksi erilaisia komentorivityökaluja (command line interface, cli).

Eri kehyksillä tai kirjastoilla luoduille projekteille on omat projektin- tai paketinhallintatyökalut. Useimpiin niistä liittyy keskeisesti jonkinlainen projektikohtainen konfiguraatiotiedosto, joka sijoitetaan projektin juureen. Npm:llä hallittavien sovellusten kohdalla konfiguraatiotiedosto on nimeltään *package.json*, johon tallennetaan projektin kirjasto- ja kehysriippuvuudet. Kirjasto- ja kehysriippuvuuksien lisäksi konfiguraatiotiedosto voi

pitää sisällään tietoa projektin keskeisistä asioista, kuten esimerkiksi nimestä, tekijöistä ja versiosta.

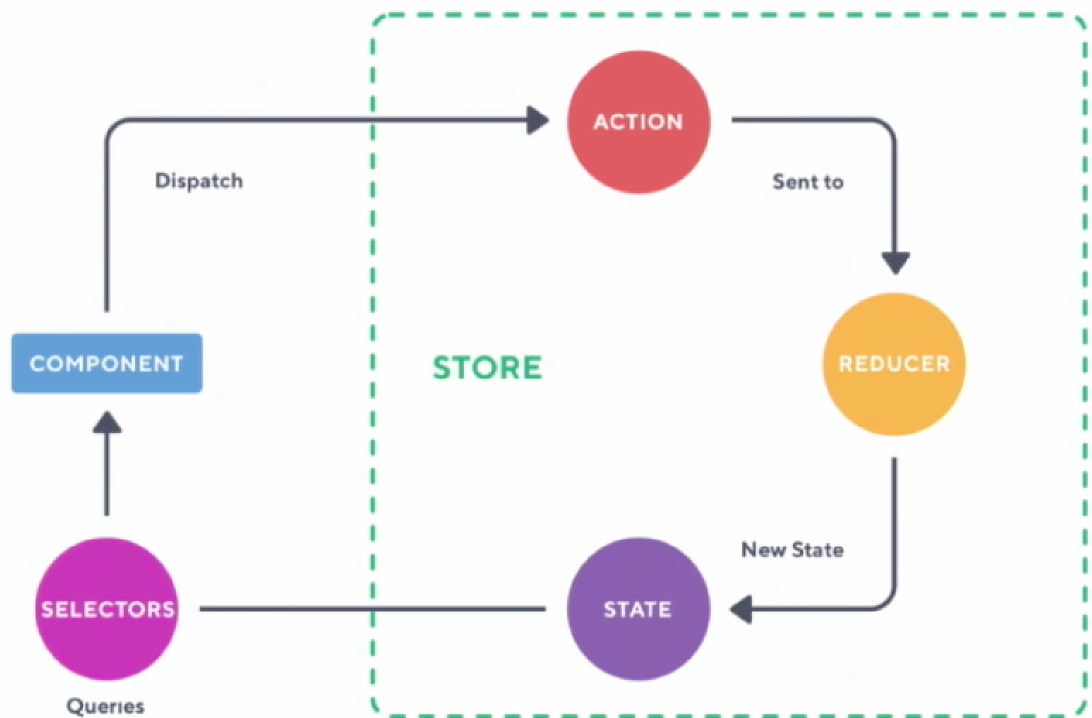
Konfiguraatiotiedoston avulla sovellus osaa ladata, asentaa ja ottaa käyttöönsä tarvitsemansa riippuvuudet. Kun kirjasto liitetään projektiin, *package.json*-tiedostoon tallennetaan kirjaston nimi sekä versio. Kirjastot haetaan projektiin npm-rekisteristä kirjaston nimen ja version mukaan (Juzer 2014, 15). Myös kirjastot, joita npm:llä lisätään projektiin, sisältävät oman *package.json*-tiedostonsa omien riippuvuuksiensa hallitsemiseen. Sovellukseen riippuvuutena tallennetut paketit ladataan *node_modules*-hakemistoon. Se sijaitsee tyypillisesti samassa hakemistossa projektin *package.json*-tiedoston kanssa.

3 TILANHALLINTA NGRX-KIRJASTOLLA

3.1 Keskitetty tilanhallinta

Angular tarjoaa komponenttien ja palveluiden kautta hyviä tapoja sovelluksen tilan hallitsemiseen. Sovelluksen monimutkaistuessa tilanhallinta voi kuitenkin käydä haastavaksi, kun tilamuutoksia voi tulla useista paikoista eri puolilta sovellusta. Tilan päivittyessä tulisi päivitetyn tilan olla nopeasti saatavilla kyseistä tilaa tarvitsevissa sovelluksen osissa. Tämän varmistamiseksi sovelluksessa voidaan joutua tekemään ylimääräisiä palvelinkutsuja. Kun tilanhallinta käy haastavaksi komponentti ja palvelutasolla, on yksi ratkaisu keskittää sovelluksen tilanhallinta.

Tälle päivällä tyypillisten keskitetyn tilanhallintaratkaisujen, kuten Reduxin ja NgRx:n perusosaset ovat tehtävät (actions), vähentäjät (reducers) sekä varasto (store). Pääajatus ratkaisuissa on, että sovelluksessa lähetetään tehtäviä, jotka vähentäjä-funktiot ottavat vastaan. Tehtävän mukaan vähentäjä-funktio muuttaa sovelluksen varastossa olevaa tilaa. Kuvassa 3 on havainnollistettu keskitetyn tilanhallinnan osasten suhteita toisiinsa sekä tiedonkulun suunta sovelluksessa.



Kuva 3: NgRx:n osien suhteet sekä yksisuuntainen tiedonkulku (Motto & East 2018, NgRx Selectors)

Kuvassa 3 on lisäksi hahmoteltu myös niin sanottujen valitsimien (selectors) sijainti suhteessa muihin NgRx:n perusosasiin. Valitsijat ovatkin keskeinen osa NgRx:llä toteutettua sovellusta. Valitsijoiden avulla voidaan sovelluksen varastossa olevasta tilasta valita halutut palaset niitä tarvitseviin sovellusosasiin.

Noringin (2018, Principles) mukaan keskitetyn tilanhallinnan kirjastot pohjautuvat kolmelle perusolettamukselle:

- tila sijaitsee vain yhdessä paikassa (single source of truth)
- tila on kirjoitussuojattu (read-only)
- tila muutetaan puhtaiden funktioiden kautta

NgRx:ssä tila on tallessa kirjaston tarjoamassa varastossa. Koska tila sijaitsee vain yhdessä paikassa, on sovelluksen kehittäminen ja virheiden etsiminen helpompaa (Noring 2018, Single source of truth). Tämän perusteella voidaan myös olla varmoja siitä, että varastossa oleva tila on aina sama eri puolilla sovellusta.

Tila on lisäksi kirjoitussuojattu. Tämä tarkoittaa, että sovelluksen tilaa ei voida muuttaa sijoittamalla uusia arvoja varaston tilassa oleviin muuttujiin. On kuitenkin välttämätöntä, että tila on muutettavissa. Tilamuutokset tapahtuvat vähentäjien kautta. Vähentäjät eivät muuta tilassa olevia muuttujia. Sen sijaan ne tekevät tilasta kopion ja muuttavat kopion muuttujia tehtävän mukaan. Tämän jälkeen ne sijoittavat varastoon sovelluksen tilaksi päivitetyn kopion sovelluksen vanhasta tilasta. Tilassa sijaitsevia muuttujia ei siis muuteta suoraan, vaan tilaksi sijoitetaan aina uusi, päivitetty tila. (Noring 2018, Changing states with pure functions)

Keskitetyn tilanhallinnan tarpeellisuuden ja hyötyjen selvittämiseksi ohjelmoitiin pieni esimerkkisovellus. Sovelluksessa voi luoda, sekä selata ja poistaa jo luotuja tehtäviä. Tehtäviä voi myös kommentoida. Sovelluksen tilaa hallittiin NgRx:n ominaisuuksien avulla. Seuraavissa luvuissa esitetyt koodiesimerkit on poimittu esimerkkisovelluksesta. Esimerkkisovelluksessa luotavista ja kommentoitavista tehtävistä on käytetty termiä *task*, jotta niitä ei sekoitettaisi NgRx:n tehtäviin.

3.2 Tehtävät

Yksi NgRx:n perusosa on tehtävät (actions). Varastossa olevaa tilaa päivitetään tehtävien mukaan. Tehtävistä luodaan tyypillisesti kaksi muuttujaa: tehtäväluokka ja tehtävävakio (action constant). Jokaista tehtäväluokkaa kohden luodaan sitä vastaava tehtävävakio. Tehtävävakio on tehtävää kuvaava merkkijonomuuttuja (string). On tärkeää huomata, että vaikka tehtävävakiot ovat merkkijonomuuttujia, ei niitä kuitenkaan tule tyypittää vahvasti. Tyypitys aiheuttaa ongelmia myöhemmin käsiteltävissä vähentäjä-funktioissa.

On hyvien tapojen mukaista antaa tehtävävakiolle etuliitteeksi hakasulkeissa se ominaisuus, johon tehtävä kuuluu. Tämä auttaa varastossa olevan tilan päivittymisen seuraamista myöhemmin. Kuvassa 4 on kuvattuna kolme erilaista tehtävävakiota.

```
export const LIST_TASKS = '[Task] List tasks';
export const LIST_TASKS_SUCCESS = '[Task] List tasks success';
export const LIST_TASKS_FAIL = '[Task] List tasks failed';
```

Kuva 4: Tehtävävakioita

Pelkästään tehtävävakioiden mukaan varaston tilaa ei voida päivittää. Jokaista tehtävävakiota kohden luodaan oma tehtäväluokkansa (action creator). Tehtäväluokkia kutsutaan yleisesti tehtäviksi. Kuvassa 5 on kuvassa x esitettyjä tehtävävakioita vastaavat tehtävät.

```
import { Action } from '@ngrx/store';

export class ListTasks implements Action {
    readonly type = LIST_TASKS;
}

export class ListTasksSuccess implements Action {
    readonly type = LIST_TASKS_SUCCESS;
    constructor(public payload: Task[]) {}
}

export class ListTasksFail implements Action {
    readonly type = LIST_TASKS_FAIL;
}
```

Kuva 5: Tehtäväluokkia

Tehtävät ovat TypeScript-luokkia, jonka ominaisuudeksi asetetaan tehtävän tyyppi (type) sekä vapaavalintainen tehtäväsisältö (payload). Tehtävän tyyppi on tehtävää vastaava tehtävävakio. Tehtävän tehtävätyyppi tulee kirjoitussuojata. Tehtäväsisältö on muuttuja, joka sisältää tehtävän suorittamiseen liittyvää dataa. Tehtävä sisältö ilmoitetaan tehtävän konstruktori-funktiossa. Kuvassa 5 oleva *ListTasksSuccess*-tehtävä saa tehtäväsisällökseen taulukon *Task*-mallin mukaisia objekteja.

Tehtäviä on myös syytä olla käyttämät uudestaan, vaikka kyseessä olisi hyvin samantapainen tehtävä. On esimerkiksi todennäköistä, että esimerkkisovelluksessa halutaan olevan mahdollisuus lisätä *taskeja* eri sivuilta *task*-listaan. Ryan (2018, Good Action Hygiene With NgRx) suosittelee, että tällaisissa tilanteissa luotaisiin eri sivuille uusi tehtävä, vaikka se toteuttaa saman asian. Tehtävälle annettaisiin oma tehtävävakio. Tehtävävakiossa voitaisiin tarkasti ilmaista, mistä päin sovellusta tehtävä lähetettiin. Tämä helpottaa virheenetsintää. Uuden tehtävän luominen esimerkiksi sivukohtaisesti ei merkittävästi lisää tarvittavan koodin määrää vähentäjä-funktioissa tai efekteissä.

On tärkeää muistaa, että tehtävät eivät itsessään tee mitään. Ne vain kertovat sovellukselle, kuinka toimia kunkin tehtävän kohdalla. Tämän lisäksi niiden avulla siirretään da-

taa. Tehtävävakioiden ja itse tehtävien lisäksi tehtävistä luodaan usein tehtäväliitto (union). *Union* on yksi TypeScriptin erityisistä muuttujatyypeistä. Tehtäväliitto helpottaa tehtävien käsittelemistä vähentäjä-funktiossa. Kuvassa 6 on esitetty esimerkkisovelluksessa tehtäville luotu tehtäväliitto.

```
export type TaskActionsUnion =
    | ListTasks
    | ListTasksSuccess
    | ListTasksFail
    | RemoveTask
    | RemoveTaskSuccess
    | RemoveTaskFail
    | CreateTask
    | CreateTaskSuccess
    | CreateTaskFail;
```

Kuva 6: Tehtäväliitto

3.3 Vähentäjät

Vähentäjät ovat puhtaita funktioita. Ne kuvaavat varastossa olevan tilan palasta sekä kertovat, kuinka juuri sitä palasta muutetaan (Farhi 2017, Reducers). NgRx:llä toteutetussa sovelluksessa on aina vähintään yksi vähentäjä.

Vähentäjät saavat parametreinaan tilan sekä tehtävän. Vähentäjiin voidaan tehdä esimerkiksi *switch-case*-valintarakenne. Rakenteen testattavaksi lausekkeeksi asetetaan vähentäjään parametrina saapuneen tehtävän tyyppi. Tyypin mukaan vähentäjä muuttaa parametrina saanuttaan tilaa tehtävän mukaan. Kuvassa 7 on esitetty vähentäjä-funktio.

```
export function reducer(
    state: State = initialState,
    action: fromActions.TaskActionsUnion
    ): State {
    switch (action.type) {
        case fromActions.LIST_TASKS: {
            return { ...state, loading: true, loaded: false };
        }

        case fromActions.LIST_TASKS_SUCCESS: {
            const entities: Task[] = action.payload;
            return { ...state, loading: false, loaded: true, entities };
        }

        case fromActions.LIST_TASKS_FAIL: {
            return { ...state, loading: false, loaded: false};
        }

        default: {
            return state;
        }
    }
}
```

Kuva 7: Vähentäjä-funktio

Ensimmäisenä parametrinaan vähentäjä saa vähentäjää koskevan tilan. Toisena parametrina vähentäjään saapuu tehtävä, joka koostuu tyypistä sekä vapaavalintaisesta sisällöstä. Koska tehtävät viedään TypeScriptin Union-tyyppinä tehtävävakioiden ja tehtävien lisäksi, on tehtävien vertailu *switch-case*-rakenteessa helppoa. Joidenkin tehtävien lopputulema voi olla sama. Tällaisessa tilanteessa *switch-case*-rakenteessa on helppo määrittää useita tehtäviä suorittamaan sama koodilohko.

Koska vähentäjät ovat puhtaita funktioita, ei niissä tule muuttaa vähentäjää koskevaa tilaa suoraan. Sen sijaan vähentäjä palauttaa aina uuden tilaobjektin riippumatta siitä, muuttuiko tila vai ei. (Farhi 2017, Reducers) On tärkeää palauttaa vähentäjässä tila myös valintarakenteen ulkopuolella. Mikäli näin ei toimita, eivät tilan tilanneet muuttujat vastaanota alustettua tilaa.

Yhtä vähentäjää koskee aina yksi tilamuuttuja. Tilamuuttuja alustetaan samassa tiedostossa vähentäjän kanssa. Alustava tila koostuu tilaa koskevista muuttujista arvoineen ennen tilamuutoksia. Tilalle tehdään usein myös rajapinta, jonka avulla tila saadaan vahvasti tyypitettyä. Kuvassa 8 on esitetty tilamuuttujan rajapinta *State* sekä rajapinnan toteuttava tilamuuttuja *initialState*.

```
import { Task } from './../../models/task.model';

export interface State {
    entities: Task[];
    loading: boolean;
    loaded: boolean;
}

export const initialState: State = {
    entities: [],
    loading: false,
    loaded: false,
}
```

Kuva 8: Tilan rajapinta ja alustava tila

3.4 Varasto

Varasto on säilö, joka pitää sisällään sovelluksen tilan. Varastot ovat yksiöitä, eli niitä on vain yksi sovellusta kohden. Koska varastoja on vain yksi, voidaan olla varmoja, että sovelluksen tila on sama eri puolilla sovellusta. (Farhi 2017, ngrx/store)

Sovelluksen varastossa oleva tila koostuu aina yhdestä tai useammasta vähentäjästä ja sitä koskevasta tilamuuttujasta. Tilalla on puumainen rakenne. Se saa muotonsa vähentäjä-funktioiden ja niitä koskevien tilamuuttujien mukaan (Farhi 2017, Reducers). Tila koostetaan jokaista vähentäjäfunktiota koskevasta tilamuuttujasta.

Varasto luodaan NgRx:n *StoreModule*-luokan ja sen funktion *forRoot* avulla. Funktio *forRoot* saa parametrinaan objektin, jolla on arvoinaan vähentäjä-funktioita. Kuvassa 9 on esitettynä varaston luominen juurimoduulille. Esimerkissä *forRoot*-funktio saa parametrinaan objektin, jolla on arvonaan reitityksestä vastaava vähentäjä-funktio.

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

@NgModule({
  imports: [
    BrowserModule,
    StoreModule.forRoot(fromStore.reducers),
```

Kuva 9: Varaston luominen juurimoduulille

Ominaisuusmoduuleille luodaan varasto lähes samalla tavalla, kuin sovelluksen juurimoduulillekin. Poikkeuksena on, että ominaisuusmoduulien kohdalla käytetään *Store-*

Modulen forFeature-funktiota. Funktio *forFeature* saa arvonaan vähentäjä-funktioiden lisäksi myös ominaisuutta kuvaavan nimen. Kuvassa 10 on esitetty tilan luominen tehtävä-moduulille. Tehtävämoduulin tila koostuu tehtäviin ja kommentteihin liittyvistä vähentäjä-funktioista.

```
import { StoreModule } from '@ngrx/store';
import * as fromTasks from './store/reducers/task.reducer';
import * as fromComments from './store/reducers/comment.reducer';

@NgModule({
  imports: [
    SharedModule,
    RouterModule.forChild(ROUTES),
    StoreModule.forFeature('tasks', {
      tasks: fromTasks.reducer,
      comments: fromComments.reducer
    }),
```

Kuva 10: Varaston luominen ominaisuusmoduulille

Mikäli sovellus käynnistettäisiin nyt, koostuisi sovelluksen tila *route*- ja *tasks*-muuttujista. Muuttajat ovat samanarvoisia, vaikka *route*-muuttuja ilmoitettiinkin sovelluksen juurimoduulissa. Kuvassa 11 on hahmoteltu sovelluksen tilan rakennetta.

```
{
  route: {
    state: {
      url: null,
      params: [],
      queryParams: []
    },
    navigationId: null
  },
  tasks: {
    tasks: {
      entities: [],
      loading: false,
      loaded: false
    },
    comments: {
      entities: []
    }
  }
}
```

Kuva 11: Tilan rakenne sovelluksen käynnistyessä

Tasks- ja *comments*-tilojen sisältö riippuu siis vähentäjä-funktioihin liittyvistä tilamuuttujista ja niiden tyypeistä ja arvoista. Mikäli tilaan halutaan lisätä muuttujia, tulee muuttuja lisätä tilaa koskevan vähentäjän tilamuuttujaan.

Tehtävien, vähentäjä-funktioiden ja varaston asettamisen jälkeen tilanhallinta on nyt valmis käytettäväksi komponenteissa ja palveluissa. Keskitetyn tilanhallinnan asettaminen vaati paljon pohjakoodia: tulee luoda lukuisia tiedostoja, yhdistää ne toisiinsa ja ottaa käyttöön NgRx:n eri ominaisuudet moduuleissa. Esimerkkisovellusta ohjelmoidessa pohjakoodin määrää ei kuitenkaan koettu ongelmalliseksi, vaikka näin oltiin aluksi luultu. Toisaalta, tehty esimerkkisovellus oli pieni sovellus. Laajemmissa ja monimutkaisemmissa sovelluksissa pohjakoodin hallittavuus voi käydä haastavaksi.

3.5 Tilan valitseminen ja tehtävien lähettäminen

Jotta sovelluksen tilaa voidaan valita ja muuttaa, tulee NgRx:n *Store*-luokka injektoida tilaa hyödyntävään sovellusosaseen. Injektoidun *Store*-luokan avulla osanen voi erityisten valitsimien avulla valita tarvitsemansa tilan. Valitsimet ovat funktioita, jotka palauttavat tilasta halutun arvon.

NgRx tarjoaa apufunktioita tilavalitsimien luomiseen. Funktiolla *createFeatureSelector* voidaan luoda valitsija tilan valitsemiseksi tilaobjektin ylätasolta. Se saa parametrinaan vain ominaisuutta vastaavan tilaobjektin nimen. Kuvassa 12 on esitetty ominaisuuteen liittyvän tilan valitseminen.

```
import { createFeatureSelector } from '@ngrx/store';

export const getTaskModuleState =
    createFeatureSelector<State>('tasks');
```

Kuva 12: Ominaisuuteen liittyvän tilan valitseminen

Valitsimilla valitaan tilaa tilapuusta aina yhtä tasoa alempaa. Ominaisuustila *tasks* koostuu samannimisestä tilasta *tasks* ja tilasta *comments*. Kuvassa 13 on esitetty *tasks*-tilan valitsija-funktion luominen. Tämän jälkeen äsken luodun funktion avulla on luotu tilasta *entities*-muuttujan valitseva valitsija-funktio.

```
import { createSelector } from '@ngrx/store';

export const getTaskState = createSelector(
    fromFeature.getTaskModuleState,
    (state: fromFeature.State) => state.tasks
);

export const getTasks = createSelector(
    getTaskState,
    (state: fromTasks.State) => state.entities
);
```

Kuva 13: Tilan valitseminen ominaisuustilasta

Valitsimia käyttämällä voidaan välttyä ylimääräisiltä tilamuuttujilta. Esimerkiksi esimerkkisovellus sisälsi sivun, jolla näytetään valitun *taskin* tiedot. Tilaan olisi voitu asettaa *selectedTask*-niminen muuttuja. Tähän muuttujaan olisi asetettu se *taski*, joka *task*-listasta oltiin valittu. Valittu *taski* kuuluu kuitenkin aina sovelluksen tilassa olevien *task*-entiteettien joukkoon. Tämän vuoksi luotiin valitsin tilamuuttujan sijaan. Luotu valitsin valitsee tilassa olevien *task*-entiteettien joukosta *taskin* sovelluksen polusta löytyvän tunnisteen mukaan. Käyttämällä valitsinta tilaan ei tarvinnut tallentaa tilassa jo olevasta tiedosta johdettua tietoa.

Varastossa olevaa tilaa muutetaan vähentäjien ja tehtävien avulla. Vähentäjälle voidaan lähettää (dispatch) tehtävä käyttämällä varaston *dispatch*-funktiota. Tämä funktio saadaan käyttöön sovellusosasessa sen jälkeen, kun siihen on injektoitu *ngrx*:n *Store*-luokka. Kuvassa 14 on esitetty tehtävän lähettäminen vähentäjälle käyttämällä varaston *dispatch*-funktiota.

```
import { Store, select } from '@ngrx/store';
import { Observable } from 'rxjs';  36.6K (gzipped: 8.7K)

export class TaskListComponent {
  private tasks$: Observable<Task[]>;
  public newTask: Task = emptyTask();

  constructor(private store: Store<fromStore.State>) {
    this.tasks$ = this.store.pipe(select(fromStore.getTasks));
    this.store.dispatch(new fromActions.ListTasks());
  }
```

Kuva 14: Tehtävän lähettäminen vähentäjälle

Mikäli kyseessä olisi tehtävä, johon kuuluu myös tehtäväsisältö, voitaisiin tehtäväsisältö lähettää tehtävän mukana antamalla se parametrina *ListTasks*-luokan konstruktorifunktiolle. Tehtävän lähettämisen lisäksi kuvassa 14 on nähtävissä tilan valitseminen varastosta valitsimen avulla.

3.6 Efektit

Efektit (effects) ovat ikään kuin vähentäjiä. Ajatus efekteissä on, että niillä eristetään sovelluksen sivuvaikutukset (side effects) muusta sovelluslogiikasta. Termi sivuvaikutus tulee funktionaalisesta ohjelmoinnista. Noring kuvaa sivuvaikutuksia operaatioiksi, jotka muuttavat esimerkiksi tiedostoa tai riviä tietokannassa. Ne eivät siis suoranaisesti liity sovelluksen tilaan, vaikka sovelluksen tila riippuu niiden kautta saatavissa olevasta datasta. (Noring 2018, @ngrx/effects – working with side effects) Voidaan ajatella, että vähentäjät kuuntelevat sovelluksen sisäistä tilaa muuttavia tehtäviä, kun taas efektit sovelluksen ulkopuolisia asioita, kuten tietokantaa, muuttavia tehtäviä.

Efektit ovat eri paketissa, kuin NgRx. Jotta sovelluksessa voidaan käyttää efektejä, tulee siihen asentaa *@ngrx/effects*-niminen paketti. Efekteille luodaan Angularin palveluiden tapainen injektoitava luokka *Injectable*-koristelijalla. Luokka saa konstruktorifunktiossaan parametrina *@ngrx/effects*-paketin *Actions*-luokan. Kuvassa 15 on esitetty tehtäviin liittyville efekteille luotu luokka *TaskEffects*.

```
import { Effect, Actions, ofType } from '@ngrx/effects';
import { mergeMap, map, catchError } from "rxjs/operators";

@Injectable()
export class TaskEffects {
    constructor(
        private actions$: Actions,
        private taskService: TaskService) {}

    @Effect()
    listTasks$: Observable<Action> = this.actions$.pipe(
        ofType(fromActions.LIST_TASKS),
        mergeMap((action: fromActions.ListTasks) =>
            this.taskService.listTasks().pipe(
                map((tasks: Task[]) =>
                    (new fromActions.ListTasksSuccess(tasks))),
                catchError(() => of(new fromActions.ListTasksFail()))
            )
        )
    );
```

Kuva 15: Efekti tehtävien listaamiseen

Itse efektit luodaan luokan sisälle *@ngrx/effects*-paketin *Effect*-koristelijan avulla. Samasta paketista löytyvällä *ofType*-operaattorilla asetetaan efekti kuuntelemaan jonkin tietyn tehtävän lähettämistä. Operaattori *ofType* voi ottaa useita tehtävätyyppejä parametrinaan, jolloin tehtävien, jotka tekevät saman asian, käsittely on helppoa (Ryan 2018, Good Action Hygiene With NgRx). Kuvan 15 efekti *listTasks$* on asetettu kuuntelemaan tehtävää, jonka tyyppi on *LIST_TASK*. Kun tehtävä lähetetään, suoritetaan *mergeMap*-operaattorin sisällä oleva funktio. Funktio kutsuu *taskServicen* funktiota *listTasks*, joka kutsuu palvelimen rajapintaa. Rajapinta palauttaa tietokannasta kutsun parametrien mukaan löytyvät *taskit*. Onnistuessaan kutsu palauttaa *ListTasksSuccess*-tehtävän, jonka vähentäjä funktio ottaa vastaan. Epäonnistuessaan palautuu *ListTasksFail*-funktio.

Vaikka kuvassa 15 esitetty efekti *listTasks$* on asetettu kuuntelemaan tehtävää, jonka tyyppi on *LIST_TASKS*, voidaan samaa tehtävää kuunnella myös vähentäjä-funktiossa. Efekti suorittaa tehtävän, johon liittyy palvelinkutsun tekeminen. Vähentäjä-funktiossa puolestaan muutetaan sovelluksen sisäistä tilaa.

Efektin ei ole välttämätöntä lähettää toista tehtävää. Tällöin efektille tulee koristelijassa antaa parametrina objekti, joka sisältää *dispatch*-nimisen boolean-attribuutin arvoltaan *false*. Mikäli näin ei toimita, jää sovellus efektiä vastaavan tehtävän vastaanottaessaan

ikuiseen silmukkaan, joka kaataa sovelluksen. Tämä tapahtuu helposti etsiessä virheitä sovelluksesta, esimerkiksi RxJS:n *tap*-operaattorilla.

Jotta efektit voivat kuunnella sovelluksessa lähetettyjä tehtäviä, tulee ne ilmoittaa Angularin moduulissa. Efektit ilmoitetaan siinä moduulissa, mihin ominaisuuteen ne liittyvät. Kuvassa 16 on esitetty *tasks*-ominaisuuteen liittyvien efektien ilmoittaminen.

```
import { EffectsModule } from '@ngrx/effects';
import { TaskEffects } from "./store/effects/task.effects";
import { CommentEffects } from './store/effects/comment.effects';

@NgModule({
  imports: [
    RouterModule.forChild(ROUTES),
    StoreModule.forFeature('tasks', {
      tasks: fromTasks.reducer,
      comments: fromComments.reducer
    }),
    EffectsModule.forFeature([
      TaskEffects,
      CommentEffects
    ]),
```

Kuva 16: Efektien ilmoittaminen

Kuvassa 16 on nähtävissä, kuinka moduuli on asetettu kuuntelemaan efektejä *@ngrx/effects*-paketin *EffectsModulen forFeature*-funktion avulla. Funktiolle on annettu parametrina ominaisuuteen liittyvät efektit taulukossa. Juurimoduulissa tulee puolestaan käyttää *EffectsModulen forRoot*-funktiota, jolle voidaan antaa parametrina esimerkiksi tyhjä taulukko, mikäli moduulissa ei kuunnella efektejä. Vaikka efektejä kuunneltaisiin vain ominaisuusmoduulissa, tulee juurimoduulissa silti ilmoittaa *EffectsModule forRoot*-funktion avulla. Ilman tätä ei sovellus kuuntele ominaisuusmoduuleissa niihin liittyviä efektejä.

Efektit vaativat paljon RxJS-osaamista. Efekteihin liittyvä ajattelutapa saattaa aluksi tuntua vieraalta, sillä esimerkiksi kuvassa 15 *taskien* listaaminen vaatii kahden eri tehtävän lähettämistä *task*-listan saamiseksi tietokannasta varastoon. Esimerkkisovellusta tehdessäni kuitenkin tykästyin ajatukseen, jossa sovelluksen sivuvaikutukset rajataan selkeästi omiksi funktioikseen.

3.7 Reitit

Myös selaimen polkua voidaan käyttää osana tilanhallintaa. NgRx tarjoaa paketin *@ngrx/router-store*, jonka avulla erilaista dataa sovelluksen reitistä voidaan tallentaa sovelluksen varastossa olevaan tilaan. Paketin käyttöönotto on yksinkertaista. Kuvassa 17 on esitetty, kuinka paketti otetaan käyttöön sen jälkeen, kun se on lisätty projektiin.

```
import {
  StoreRouterConnectingModule,
  routerReducer
} from '@ngrx/router-store';

@NgModule({
  imports: [
    BrowserModule,
    StoreModule.forRoot({
      route: routerReducer
    }),
    StoreRouterConnectingModule.forRoot(),
```

Kuva 17: Paketin *@ngrx/router-store* lisääminen projektiin

Reitille luodaan oma tilaobjekti ja oma vähentäjä paketin tarjoamien ominaisuuksien avulla. Tämän jälkeen luotu vähentäjä ilmoitetaan projektin juurimoduulissa samaan tapaan kuin muutkin vähentäjäfunktiot. Vähentäjäfunktion ilmoittamisen lisäksi tulee *RouterModule* yhdistää *StoreModuleen StoreRouterConnectingModule.forRoot()*-funktion avulla. Reitti on nyt osa sovelluksen tilaa ja valmis käytettäväksi. Paketti tallentaa tilaan kuitenkin todella paljon dataa, jota voi olla haastavaa hyödyntää sellaisenaan.

Haluttu data reitistä saadaan tallennettua varastoon apuluokan avulla. Luokasta tehdään Angularin *Injectable*-koristelijan avulla Angularin palvelun kaltainen luokka. Se asetetaan toteuttamaan *RouterStateSerializer*-luokan *@ngrx/router-store*-paketista. Luokalle luodaan *serialize*-niminen funktio, joka ohjelmoidaan palauttamaan haluttu data polusta. Halutulle datalle luotiin esimerkkisovelluksessa oma rajapinta *RouterState* samalla tavalla kuin muillekin sovelluksen tiloille. Kuvassa 18 on esitetty esimerkkisovellukselle luotu apuluokka.

```
import { RouterStateSnapshot, Params } from '@angular/router';
import * as fromRouter from '@ngrx/router-store';

export class CustomRouteSerializer
  implements fromRouter.RouterStateSerializer<RouterState> {

  serialize(routerState: RouterStateSnapshot): RouterState {
    let route = routerState.root;

    while (route.firstChild) {
      route = route.firstChild;
    }

    const {
      url,
      root: { queryParams },
    } = routerState;
    const { params } = route;

    return { url, params, queryParams };
  }
}
```

Kuva 18: Apuluokka reitin valitsemiseen sovelluksen tilaan

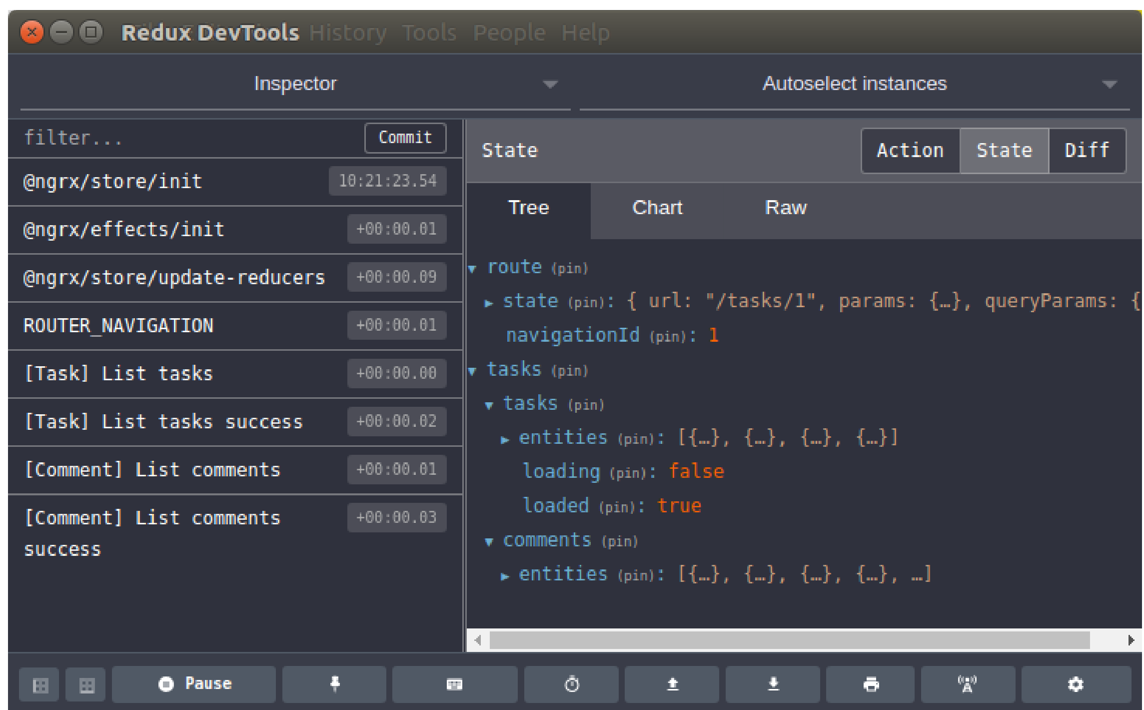
Tämän jälkeen vain haluttu reittidata tallennetaan sovelluksen tilaan. Koska reitti on sovelluksen tilassa, on sitä helppo valikoida valitsimien avulla käytettäväksi eri puolille sovellusta. Esimerkiksi esimerkkisovelluksessa on detalji-sivu, josta voidaan tarkastella listanäkymää tarkemmin tietyn *taskin* tietoja. Reitistä on tallennettu sovelluksen tilaan *taskin* tunniste. Tallennetun tunnisteen avulla valittu *taski* voidaan valikoida valitsinta käyttäen kaikkien *taskien* joukosta, ja esittää detalji-sivulla.

Kun sovelluksessa tallennettiin reittidataa sovelluksen tilaan, vältyttiin useilta reitteihin liittyvien moduulien injektoimisilta eri sovellusosasiin. Muun muassa komponentteihin ei tarvinnut tämän jälkeen ohjelmoida logiikkaa tunnisteiden poimimiseksi reitistä. Muita käyttökohteita *@ngrx/router-store* paketille ei esimerkkisovelluksessa löydetty, eikä sitä täten koettu yhtä välttämättömäksi kuin esimerkiksi efektejä. Toisaalta, paketin lisääminen ja käyttöönotto oli yksinkertaista ja nopeaa. Kun paketti oli kerran otettu käyttöön, ei siihen liittyvää ohjelmakoodia tarvinnut enää myöhemmin muuttaa.

4 VIRHEENETSINTÄ TILASTA

4.1 NgRx-sovelluksen tilan kehitystyökalu

Sovelluksen kehittämisen tueksi NgRx tarjoaa erityisen paketin nimeltään *@ngrx/store-devtools*. Paketin työkalujen avulla sovellus voi pitää kirjaa keskitetyn tilanhallinnan tehtävistä, joita sovelluksessa sen käytön aikana on lähetetty. Kehitystyökalulla itsellään ei vielä voida tarkastella sovelluksen tilaa. Tätä varten tulee asentaa Redux DevTools -työkalu esimerkiksi Chrome- tai Firefox-selaimeen selaimen lisäosana. Kuvassa 19 on esitetty Redux DevToolsin näkymä sen jälkeen, kun sekä *ngrx/store-devtools* että Redux DevTools on asennettu ja esimerkkisovellus on käynnistetty.



Kuva 19: Redux DevToolsin näkymä

Vasemmassa lohkossa ovat listattuna sovelluksessa lähetettyjen tehtävien tyyppi. Kun tehtävävakiot nimetään tarkasti, nähdään nopeasti, mistä päin sovellusta tehtäviä lähetettiin. Tämä nopeuttaa virheenetsintää.

Pelkän listauksen lisäksi tehtävälistasta voidaan valita tehtäviä. Tehtävää klikkaamalla voidaan siirtyä sovelluksen tilaan, jossa se oli heti kyseisen tehtävän suorittamisen jäl-

keen. Tästä käytetään myös nimitystä time-travel debugging (Noring 2018, @ngrx/store-devtools - debugging). Tämän ominaisuuden avulla on helppo palata haluttuun tilan hetkeen. Tiettyyn hetkeen palaamiseksi ei myöskään tarvitse ladata sovellusta uudestaan.

Näkymän oikeassa lohkossa on nähtävillä sovelluksen tila. Tilaa voidaan tarkastella kolmessa eri näkymässä: puu-, kaavio- tai raakanäkymänä. Lohkossa voidaan myös valita näytettäväksi lähetetyn tehtävän tehtäväsisältö. Lisäksi voidaan tarkastella sitä, kuinka sovelluksen tila muuttui tehtävän mukaan.

4.2 Kehitystyökalun käyttöönotto

Kehitystyökalu *@ngrx/store-devtools* kuuluu eri pakettiin *@ngrx/storen* kanssa. Tämän vuoksi se tulee erikseen ladata ja asentaa projektiin npm-työkalulla. Asentamisen jälkeen työkalu on helppo ottaa käyttöön projektissa. Kuvassa 20 voidaan nähdä, kuinka kehitystyökalu on lisätty Angular-projektiin sisällyttämällä se juurimoduulissa.

```
import { StoreDevtoolsModule } from '@ngrx/store-devtools';

@NgModule({
  imports: [
    BrowserModule,
    StoreModule.forRoot(fromStore.reducers),
    StoreRouterConnectingModule.forRoot(),
    StoreDevtoolsModule.instrument({
      maxAge: 25,
      logOnly: environment.production
    }),
```

Kuva 20: Kehitystyökalun käyttöönotto juurimoduulissa

Käyttöönottoa varten juurimoduuliin tulee sisällyttää kehitystyökalun asennuksen mukana tullut *@ngrx/store-devtools*-kirjasto. Kirjasto tarjoaa *StoreDevToolsModule*-luokan, joka lisätään moduulin imports-listaan käyttämällä sisällytetyn luokan *instruments*-funktiota. Tämän jälkeen työkalu on valmis käytettäväksi. Kirjastoa ei tarvitse ottaa käyttöön juurimoduulin lisäksi muissa moduuleissa.

Kun *StoreDevtoolsModule*-luokka lisätään projektiin *instruments*-funktion avulla, voidaan sille antaa samalla objekti, joka sisältää erilaisia ominaisuuksia. Kuvasta 20 voidaan havaita, että esimerkkisovelluksessa *instruments*-funktiolle annettiin avaimet *maxAge* ja *logOnly*. Avain *maxAge* tarkoittaa niiden tehtävien määrä, joita työkalu pitää muistissaan. Kun funktiolle annetaan ominaisuutena *logOnly*, tarjoaa sovellus tuotantoversiossaan vain lokitiedon sovelluksessa liikkuvista tehtävistä. Avain-arvo-pareja voidaan antaa myös useita muita. Yksi näistä on *predicate*-funktio, joka toimii välikäsittelijän (middleware) omaisesti. Tätä funktiota kutsutaan ennen jokaisen tehtävän lähettämistä. Funktion *predicate* avulla voidaan esimerkiksi helposti tulostaa tehtäviä tehtäväsisältöineen konsoliin.

Työkalun käyttöön ottaminen oli nopeaa ja sen käyttäminen helppoa. Työkalulla näki virhetilanteissa helposti, missä tilassa sovelluksen tila oli erilaisten virheiden sattuessa. Tehtävien, jotka eivät muuttaneet tilaa oikein, jäljittäminen oli myöskin vaivatonta. Työkalua käyttämällä ei tilaa koskevaa virheenetsintää tarvinnut suorittaa esimerkiksi selaimen omassa kehitystyökalussa. Tilaa ei myöskään tarvinnut tulostaa eri toimintojen välillä selaimen konsoliin, sillä tila oli kokoajan tarkasteltavissa kehitystyökalun avulla.

5 JOHTOPÄÄTÖKSET JA POHDINTA

5.1 Opinnäytetyön toteutus ja onnistuminen

Opinnäytetyö toteutettiin Dicode Oy:lle selvitystyönä. Opinnäytetyössä tutkittiin, miten hyvin NgRx-tilanhallintakirjasto soveltuu Angular-sovellusten kehittämiseen. Työ aloitettiin luomalla teoreettinen pohja Angularista ja tutustumalla siihen liittyviin tärkeimpiin käsitteisiin. Angularista käytiin läpi moduulit, komponentit ja palvelut. Lisäksi tarkasteltiin, miten tilaa hallitaan Angular-sovelluksessa ilman NgRx-kirjastoa.

Tämän jälkeen tutkittiin NgRx-kirjaston käyttämistä erilaisten itse ohjelmoitujen käytännön esimerkkien avulla. Näin kirjastosta saatiin mahdollisimman kokonaisvaltainen kuva. Esimerkkisovellukseen luotiin keskitetty tilanhallinta käyttämällä *@ngrx/store*-paketin tarjoamia ominaisuuksia. Sovelluksen sivuvaikutukset eristettiin omaksi osakseen *@ngrx/store-effects*-paketin avulla. Lisäksi sovelluksen reitti lisättiin tilaan käyttämällä *@ngrx/router-state*-pakettia.

Lisäksi opinnäytetyössä tutustuttiin NgRx-kirjaston tarjoamaan kehitystyökaluun, joka helpottaa virheiden etsimistä sovelluksen tilasta. Kehitystyökalu lisättiin omaan esimerkkisovellukseen asentamalla siihen työkalun sisältävä *@ngrx/store-dev-tool*-paketti. Työkalun käyttöön ja sen tarjoamiin ominaisuuksiin tutustuttiin käytännön kautta hyödyntämällä sitä esimerkkisovelluksessa.

5.2 Päätelmät NgRx:n käytöstä

Selvitystyössä huomattiin NgRx:n sopivan Angular-sovellusten tilanhallintaan varsin hyvin, vaikka omat monimutkaisuutensa kirjastolla toki onkin. NgRx-kirjaston soveltuvuutta Angulariin arvioitiin erityisesti kolmesta näkökulmasta: kirjaston vaikutus sovelluksen rakenteeseen, kirjaston dokumentaatio ja päivitykset sekä kirjaston käyttäminen käytännössä.

Aiemmin sovellusten tilaa hallittiin ilman NgRx:ää joko suoraan komponentissa, palvelussa tai näiden yhdistelmällä. Ratkaisut tähän vaihtelivat ohjelmoijasta riippuen. Yksi selvitystyössä esiin nousseista NgRx:n hyvistä puolista on se, että NgRx jakaa tilanhallinnan omaksi osakseen sovellusta. Tila on siis yhdessä paikassa ja sitä käytetään aina samalla tavalla, ohjelmoijasta riippumatta. Tilaa valitaan käyttöön ja tehtäviä lähetetään haluttuihin sovellusosasiin injektoidun varaston kautta. Tilanhallintalogiikka ei tällöin huku osaksi muuta sovelluslogiikkaa, eikä tilaa tarvitse porauttaa komponenteilta toisille. Koodi pysyy täten luettavana ja helpommin ymmärrettävänä.

NgRx edellyttää, että sitä käytetään tietyllä, hyväksi havaitulla tavalla. Tällöin se myös tukee ja ohjaa ohjelmoijaa hyvin toimivaa sovellusarkkitehtuuria kohden, kun sovellusominaisuudet jaetaan moduulien avulla omiksi kokonaisuuksikseen. Esimerkiksi ominaisuuteen liittyvät vähentäjä-funktiot sekä efektit määritellään sovelluksen juuri- ja ominaisuusmoduuleissa.

Vaikka NgRx ohjaa ohjelmoijaa, ratkaisut NgRx:nkin kanssa voivat kuitenkin vaihdella. Toinen ohjelmoija voi esimerkiksi hyödyntää index-tiedostoja kun taas toinen ei. Nämä eroavaisuudet kuitenkin liittyvät vain riippuvuuksien sisäistämiseen ja viemiseen, eivät ohjelman toimintaan. Käytännöt NgRx:n käytöstä tarkentuvat kirjaston käyttökokemuksen karttuessa ja koodikatselmointien avulla.

NgRx on julkinen, avoimen lähdekoodin kirjasto. Tämän takia siitä on saatavilla dokumentaatio ja esimerkkejä, joita voidaan käyttää ohjelmoinnin tukena. Kun ohjelmoija ohjelmoi sovellukseen sovelluskohtaisen tilanhallinnan ilman keskitetyn tilanhallinnan kirjastoa, kuten esimerkiksi NgRx:ää, on sen toiminta vain ohjelmoijan itsensä tiedossa. Ratkaisusta ei tällöin ole saatavilla esimerkkejä, eikä sitä välttämättä ole dokumentoitu. Tästä johtuen sovelluskohtaisten tai työkaverin ratkaisujen ymmärtäminen ja käyttäminen voi olla muista ohjelmoijista haastavaa.

NgRx jakaa päivityksiään pieninä (minor) ja isoina (major) päivityksinä. Sovellusta, johon on asennettu NgRx, on nopea muuttaa vastaamaan pienten päivitysten muutoksia. Esimerkiksi tilan valinta on muuttunut NgRx:ssä siten, että ennen tilaa valittiin varastosta varaston omalla *select*-funktiolla. Nykyään *select* on oma operaattorinsa, jota voidaan putkittaa muiden RxJS-operaattorien tapaan. Niederbergerin (2017) mukaan isojen

päivitysten kohdalla sovelluksen saaminen ajan tasalle voi olla työläämpää. NgRx:n iso päivitys voi myös edellyttää sovelluksen Angular-version päivittämistä, joka voi omalta osaltaan lisätä päivitystyötä merkittävästi.

Jotta NgRx:stä saa parhaan hyödyn irti, on sekä itse Angular että RxJS tunnettava erityisen hyvin. Erityisesti NgRx:n efektit vaativat erinomaista RxJS-kirjaston tuntemista, mikä tuo omat haasteensa kirjaston käyttöön. Mikäli NgRx:ää halutaan hyödyntää Angular-sovelluksen eri osasissa, kuten esimerkiksi vartijoissa, tulee kyseessä olevan osasen toiminta tuntea hyvin. Erityisesti sovelluksen reitittäminen NgRx:ää hyödyntämällä vaatii erinomaista Angularin vartijoiden sekä RxJS:n tuntemista.

NgRx:ään tutustuminen voi aluksi tuntua hämmentävältä. Kirjasto on laaja, ja sen käyttäminen edellyttää monen uuden käsitteen ja niiden välisten suhteiden opettelemista. NgRx lisää sovellukseen useita uusia käsitteitä, jotka ohjelmoijan tulee tuntea. Jokaista käsitettä varten luodaan tyypillisesti ominaisuuskohtaisesti oma tiedostonsa. Luotuja tiedostoja tulee paljon, jolloin sovelluksen ylläpito voi hankaloitua. Tiedostojen ja niiden välisten riippuvuuksien hallitsemiseen apua tuo ominaisuuteen liittyvien käsitteiden vieminen erityisen index-tiedoston kautta. Esimerkkisovellusta tehtäessä koodieditori ja paikallinen kehityspalvelin menivät välillä jumiin, sillä ne eivät aina ei ymmärtäneet index-tiedostojen välittämiä riippuvuuksia oikein. Editorin ja kehityspalvelimen käynnistäminen uudelleen auttoi tähän ongelmaan.

Esimerkkisovellusta tehtäessä vei hetken tottua ajatukseen, että osaa tehtävistä kuunneltiin vähentäjä-funktioissa ja osaa efekteinä omissa tiedostoissaan. Esimerkkisovellusta kehittäessä opittiin kuitenkin nopeasti tunnistamaan, milloin tuli luoda uusi vertailurakenne vähentäjään ja milloin uusi efekti-funktio. Virheenetsintä vähentäjä-funktiosta ja efekteistä oli myös aluksi hidasta, sillä ensin tuli selvittää, kummassa osassa virheen aiheuttanutta tehtävää kuunneltiin. Nykypäivän koodieditorit tarjoavat onneksi paljon apuja tällaisiin tilanteisiin. Ne pystyvät nopeasti löytämään sovelluksesta ne tiedostot, joissa tehtävä tai funktio on ilmoitettu ja missä niitä käytetään.

Yksi NgRx:n eduista on, että se voidaan lisätä projektiin pala kerrallaan. Lisäominaisuudet, kuten efektit, reitteihin liittyvä tilanhallinta ja kehitystyökalu, ovat saatavilla omina paketteinaan. Ne voidaan lisätä projektiin joustavasti, kun tarve niiden ominai-

suuksille nousee. Myös ohjelmoija hyötyy tästä siten, että hän voi tutustua NgRx:ään ominaisuus kerrallaan, eikä kaikkea tarvitse omaksua heti.

5.3 Ajatuksia selvitystyön laajentamisesta

Jotta Angular-projekteihin voidaan valita paras tilanhallinnallinen ratkaisu, tulisi myös muista kirjastoista olla hyvä käsitys. Tällaisia kirjastoja ovat muun muassa NGXS ja viime aikoina paljon esillä ollut MobX. Niissä molemmissa on pyritty vähentämään tarvittavan pohjakoodin määrää, ja ajatus varsinkin MobX:n taustalla on hieman erilainen kuin NgRx:ssä tai Reduxissa.

Tässä opinnäytetyössä ei myöskään työn rajatun laajuuden vuoksi käsitelty NgRx:ään tarjolla olevaa *@ngrx-entity*-pakettia. Entiteetit itsessään ovat hyvin laaja kokonaisuus. Entiteetti-paketin lisäksi myös efekteihin jäi asioita, joihin ei esimerkkisovelluksen kehittämisen aikana ehditty tutustumaan. Näitä olivat muun muassa erityiset sovelluksen käynnistämiseen liittyvät efektit. Myöskin vähentäjiin liittyvät välikäsittelijät (middleware), meta-vähentäjät, jäivät huomiotta. Näihin asioihin tutustumalla ja niitä käyttämällä sovelluksen tilanhallinnasta NgRx:n avulla saataisiin vielä tehokkaampi.

Opinnäytetyö ja esimerkkisovellus antavat osviittaa siitä, millaisiin projekteihin NgRx sopii ja mitkä sen edut ja haitat ovat. Esimerkkisovelluksen perusteella voidaan todeta, että erityisesti silloin, kun sovellus laajenee ja kun siihen lisätään ominaisuuksia, kannattaa vakavasti harkita NgRx:n käyttöä. Tällaisissa tapauksissa NgRx pystyy yksinkertaistamaan sovelluksen monimutkaisuutta. Myös NgRx:n kehitystyökalu jo itsessään pidettiin hyvänä syynä lisätä NgRx projektiin.

Kun oppii ajattelemaan sovelluksen tilanhallintaa NgRx:n näkökulmasta, on asioiden toteuttaminen sillä melko yksinkertaista ja suoraviivaista. Pohjakoodin jo ollessa paikallaan, on uusien tehtävien luominen ja lisääminen vähentäjään ja efekteihin helppoa ja nopeaa. Mikäli kuitenkin halutaan täysin tarkka kuva NgRx:n tarpeesta ja toimivuudesta, saadaan se kuitenkin vasta, kun kirjastoa käytetään aidossa ja monimutkaisemmassa projektissa.

Eväkallio (2017) on sanonut hyvin NgRx:n kaltaisesta Reduxista, että se ei sovi yksinkertaisten asioiden tekemiseen nopeasti mutta se on loistava yksinkertaistamaan haastavia asioita. Opinnäytteen teon aikana kertyneen kokemuksen perusteella tämä toteamus pätee myös NgRx:ään: parhaimmillaan NgRx:n hyödyntäminen parantaa projektin rakennetta ja selkeyttää tilan hallintaa.

LÄHTEET

Abramov, D. 2016. Artikkeli: You Might Not Need Redux. Luettu 28.9.2018. https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367

Ali, J. 2013. Instant node package manager: Create your own node modules and publish them on npm registry, automating repetitive tasks in between. Birmingham: Packt Publishing. Luettu 29.9.2018.

Arora, C., Hennessy, K. 2018. Angular 6 by Example. Birmingham: Packt Publishing. Luettu 22.9.2018.

Dodds, K. 2018. Artikkeli: Prop Drilling. Julkaistu 21.5.2018. Luettu 28.9.2018. https://blog.kentcdodds.com/prop-drilling-bb62e02cb691

Duffy, J. 2004. State Management. CODE Magazine 9/2004. Luettu 22.9.2018.

East, D. Motto, T. 2018. Store Selectors. Ng-conf 19.4.2018. Katsottu 17.10.2018. https://www.youtube.com/watch?v=Y4McLi9scfc

Eväkallio, J. 2017. Artikkeli: Introducing Redux Offline: Offline-First Architecture for Progressive Web Applications and React Native. Luettu 9.11.2018. https://hackernoon.com/introducing-redux-offline-offline-first-architecture-for-progressive-web-applications-and-react-68c5167ecfe0

Farhi, O. 2017. Reactive Porgramming with Angular and ngrx: Learn to Harness the Power of Reactive Programming with RxJS and ngrx Extensions. Apress. Luettu 15.10.2018.

Gabe de Wolff, I. Jansen, R. H. Vane, V. 2016. TypeScript: Modern JavaScript Development. Birmingham: Packt Publishing. Luettu 14.11.2018.

Klauzinski, P., Moore, J. 2016. Mastering JavaScript Single Page Application Development. Birmingham: Packt Publishing. Luettu 22.9.2018.

Motto, T. 2017. Artikkeli: Component architecture recipes for Angular’s reactive forms. Luettu 12.11.2018. https://toddmotto.com/component-architecture-reactive-forms-angular

Murray, N., Coury, F., Lerner, A., Taborda, C. 2018. ng-book The Complete Book on Angular 6. San Francisco: Fullstack.io. Luettu 6.10.2018.

Niederberger, D. 2017. Artikkeli: State Management: ngrx/store vs Angular services. Luettu 9.11.2018. https://www.bersling.com/2017/06/05/state-management-ngrxstore-vs-angular-services/

Noring, C. 2018. Architecting Angular Applications with Redux, RxJS, and NgRx. Birmingham: Packt Publishing. Luettu 28.9.2018.

Ryan, M. 2018. Good Action Hygiene With NgRx. Ng-conf 19.4.2018. Katsottu 16.10.2018. https://www.youtube.com/watch?v=JmnsEvoy-gY