TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kristo Naeris 175921IADB

Oleku haldamine Ravimiameti üheleherakenduse näitel

Bakalaureusetöö

Juhendaja: Jaanus Pöial
PhD

Tallinn 2022

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Kristo Naeris

07.05.2022

Annotatsioon

Käesoleva diplomitöö eesmärk on valida oleku haldamise raamistik Ravimiameti üheleherakenduse arendamiseks.

Eesmärgi täitmiseks analüüsitakse erinevaid oleku haldamise lahendusi, võrreldakse neid ja valitakse sobivaim. Seejärel luuakse valitud raamistikku kasutades analüütiku koostatud nõuete põhjal teavituste funktsionaalsus ning kirjeldatakse loodud koodi.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 36 leheküljel, 5 peatükki, 19 joonist, 3 tabelit.

Abstract

State Management in Single Page Applications on the Example of Estonia State Agency of Medicines

The aim of this thesis is to choose a state management framework to develop the Estonia State Agency of Medicines single page application.

Different state management frameworks are analyzed, compared and then the most appropriate one will be selected. Using the selected framework the notifications system will be developed and the code will be provided. The notifications system will be developed according to requirements assembled by the system analyst.

The thesis is in Estonian and contains 36 pages of text, 5 chapters, 19 figures, 3 tables.

Lühendite ja mõistete sõnastik

Akita	Oleku haldamise raamistik
Angular	*JavaScript* raamistik üheleherakenduste arendamiseks
API	*Application Programming Interface*
APP	*Application*
Boilerplate	Korduv kood standardiseerimise eesmärgil
CRUD	*Create, Read, Update, Delete*
FRP	*Functional-reactive programming*
GitHub	Platvorm versioonihalduseks
Google Trends	Veebisait Google otsingupäringute analüüsimiseks
HTML	*HyperText markup language*
MVC	*Model-View-Contoller*
NgRx	Oleku haldamise raamistik Angular rakendustele
NGXS	Oleku haldamise raamistik Angular rakendustele
Npm	*Node Package Manager*
RxJs	*Reactive Extensions for JavaScript*
URL	*Uniform Reseource Locator*
UX/UI	*User Experience/User Interface*

Sisukord

1 Sissejuhatus ... 10
  1.1 Taust ... 10
  1.2 Probleem ... 11
  1.3 Eesmärk ... 12
  1.4 Ülevaade ... 13
2 Metoodika ... 14
  2.1 Oleku haldamise raamistiku valimise metoodika ... 14
    2.1.1 Kontseptsioonide kirjeldus ... 14
    2.1.2 Küpsus ja kogukond ... 14
    2.1.3 Koodi hallatavus ja kirjutamisele kuluv aeg ... 14
    2.1.4 Kõrvalmõjude käitlemise võimalus ... 15
  2.2 Valitud lahenduse juurutamise metoodika ja arendusmetoodika ... 15
3 Oleku haldamise raamistiku valimine ... 16
  3.1 NgRx ... 16
    3.1.1 Kontseptsioonide kirjeldus ... 16
    3.1.2 Küpsus ja kogukond ... 17
    3.1.3 Koodi hallatavus ja kirjutamise aeg ... 17
    3.1.4 Kõrvalmõjude käitlemine ... 18
  3.2 NGXS ... 18
    3.2.1 Kontseptsioonide kirjeldus ... 18
    3.2.2 Küpsus ja kogukond ... 19
    3.2.3 Koodi hallatavus ja kirjutamise aeg ... 19
    3.2.4 Kõrvalmõjude käitlemine ... 19
  3.3 Akita ... 19
    3.3.1 Kontseptsioonide kirjeldus ... 20
    3.3.2 Küpsus ja kogukond ... 20
    3.3.3 Koodi hallatavus ja kirjutamise aeg ... 21
    3.3.4 Kõrvalmõjude käitlemine ... 21

3.4 Lahenduste võrdlus 21
3.4.1 Küpsus ja kogukond 21
3.4.2 Koodi hallatavus ja kirjutamise aeg 22
3.4.3 Kõrvalmõjude käitlemine 23
4 *NgRx* juurutamine ja rakendamine 24
4.1 Teekide valik 24
4.2 Oleku haldamise printsiibid 24
4.3 Koodistiil 25
4.4 Parimad praktikad 25
4.5 Juuroleku loomine 26
4.6 Teavituste oleku haldamine ja arendus 28
4.6.1 Teavituste UX/UI disain ja nõuded 28
4.6.2 Teavituse mudel 31
4.6.3 Tegevuste kirjeldamine 32
4.6.4 Teavituste oleku loomine 36
4.6.5 Oleku uuendamine vastavalt tegevusele 37
4.6.6 Tegevuste kõrvalmõjud 38
4.6.7 Vaatemudelite kaardistamine 40
4.6.8 Olekute valimine 43
4.6.9 Angular komponendid 44
4.7 Hinnang tulemusele 44
5 Kokkuvõte 46
Kasutatud kirjandus 47
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks 50
Lisa 2 – Teavituste päise konteinerkomponendi kood 51
Lisa 3 – Teavituste haldamise lehe konteinerkomponendi kood 52
Lisa 4 – Tegevuste kood 53
Lisa 5 – Kõrvalmõjude kood 55
Lisa 6 – Selektorite kood 56

Jooniste loetelu

Joonis 1. NgRx oleku haldamise elutsükkel [9] 17
Joonis 2. NGXS oleku haldamise elutsükkel [12] 19
Joonis 3. Akita oleku haldamise elutsükkel [15] 20
Joonis 4. Lahenduste otsitavus 2021 aastal Google Trends andmetel 21
Joonis 5. Juuroleku objekt 26
Joonis 6. Juuroleku reduktor 27
Joonis 7. Koodi silumisel abistav metareduktor 27
Joonis 8. Juuroleku lao moodulis objektide kokku sidumine 28
Joonis 9. UX/UI disaineri nägemus teavituste haldamise lehest 29
Joonis 10. UX/UI disaineri nägemus päises asuvatest teavitusetest 29
Joonis 11. Serverist päritud teavituse mudel 31
Joonis 12. Tegevuste kategooriad 33
Joonis 13. Teavituste oleku objekt 36
Joonis 14. Teavituste olemite kollektsiooni haldamise adapter 36
Joonis 15. Teavituste algolek 37
Joonis 16. Reduktor funktsioon teavituste oleku asendamiseks 38
Joonis 17. Teavituse vaatemudel 42
Joonis 18. Teavituste haldamise lehe vaatemudel 42
Joonis 19. Teavituste päise komponendi vaatemudel 42

Tabelite loetelu

Tabel 1. Raamistike populaarsuse võrdlus 22
Tabel 2. Raamistike arendamise ajalugu 22
Tabel 3. Serverist päritud teavituse mudeli väljade selgitused...................................... 31

1 Sissejuhatus

Käesolevas peatükis kirjeldatakse lahti töö taust, probleem, eesmärk ja antakse ülevaade tehtud tööst.

1.1 Taust

Ravimiamet on Eesti Sotsiaalministeeriumi haldusalas tegutsev riigiamet, mille ülesandeks on Eesti rahva tervise kaitsmine. Ravimiameti tegevusvaldkonnaks on nii meditsiinis kui veterinaarias kasutatavate ravimite üle järelevalve teostamine.

Samtrack II projekti eesmärk on pakkuda ravimite müügilubade menetlusele tehnilist tuge, kiirendades seeläbi protsesse ja võimaldaks edastada kaasajastatud teavet ravimite ohutuse, kvaliteedi ja kättesaadavuse kohta.

Samtrack II projekti käigus arendatav infosüsteem on valmimisjärgus. *Samtrack II* infosüsteemi arendusprojekti teostatakse mitmes etapis. Tööde I etapi tulemusena valmivad üldised funktsionaalsused ja ravimite müügiloa andmise teenus.

Arendatav rakendus järgib klient-server mudelit ning luuakse üheleherakendusena. Rakenduse arendamiseks kasutatakse *Angular* raamistikku.

Angular on komponendi põhine raamistik, mis leiab enamasti rakendust suuremate ja keerukamate üheleherakenduste arendamisel. *Angular* raamistik on suunab arendajat koodi kirjutama lähtudes *MVC* põhisest arhitektuurilistest suunitlustest. Raamistik pakub arendajale kasutamiseks laia valikut erinevaid tööriistu, mis on suuremõõtmeliste rakenduste loomiseks vajalikud.

Üheleherakenduste puhul on osa äriloogikat kandunud serveri poolelt veebilehitseja poolele. Äriloogika kandumisel veebilehitseja poolele, on tarvis hallata veebilehitseja poolel ka olekut.

Efektiivne oleku haldus saavutatakse õige arendusmustri rakendamisega. Õige arendusmustri valimine on oluline, et erinevate komponentide arendamine ja andmebaasist päritud andmete haldamine oleks kerge vaevaga teostatav [1].

Angular raamistiku arhitektuuriliste suunitluste kohaselt realiseeritakse oleku haldamine teenusklassides.

Teenusklassides oleku haldamine toimub peamiselt kahel erineval viisil:

1. Ajutine olek salvestatakse teenustes defineeritud muutujatesse. Ajutiste olekute salvestamiseks kasutatakse JavaScript kõige levinumat *Push* süsteemi *Promise*.

2. Olek salvestatakse teenustes erilist tüüpi vaadeldavatesse objektidesse ehk *Subject* tüüpi objektidesse. *Subject* on *RxJs* teegis asuv erilist tüüpi vaadeldav objekt, mille abil on võimalik olekuid emiteerida ja kuulata.

Teenuste muster oleku haldamiseks on pigem sobilik rakenduste arendamiseks, kus on väike hulk hallatavat olekut [2].

1.2 Probleem

Kasutajaliidese arendamisel kasutatud arendusmuster ei vastanud vajadustele. Tegemist oli keeruka rakendusega ning rakenduse arhitektuurist tulenevalt oli klientrakenduse poolel tarvis hallata olekut. Oleku haldamine oli keeruline, koodi kirjutamine muutus kiiresti keeruliseks ning kood raskesti hallatavaks. Sellest tulenevalt hakkasid rakenduse töösse tekkima vead, mille parandamine osutus keeruliseks ja aeganõudvaks.

Rakenduse arendamise alguses hoiustati olekut *Promise* põhistes jagatud teenuses. Peagi selgus, et *Promise* põhistes jagatud teenustes oleku haldamine ei ole piisav, kuna *Promise* süsteem ei paku piisavalt funktsionaalsuseid suuremõõtmelise rakenduse oleku haldamiseks ja nende teenuste haldamine muutub kiiresti keeruliseks.

Seejärel võeti kasutusele vaadeldavate objektidega teenused. Ka need teenused muutusid kiiresti hallatava oleku mahukuse tõttu võrdlemisi kiiresti keeruliseks ning raskesti hallatavaks.

Kui tegemist on suurema ja keerulisema rakendusega, kus hallatavat olekut on suurem hulk ning teenused enam oleku haldamiseks ei ole piisavad, siis on soovituslik kasutusele võtta keskse andmelaoga arendusmuster [3]. Keskse andmelaoga arendusmustri kasutamist väiksemate ja lihtsamate rakenduste puhul ei peeta tihti vajalikuks ega mõistlikuks, kuna sellega võib kaasneda liigne keerukus [4].

Kuna seni kasutusel olnud arendusmustrid ei toetanud oleku haldamist, siis otsustati kasutusele võtta keskse andmelaoga oleku haldamise arendusmustrit pakkuv raamistik.

Keskse andmelaoga arendusmustrit on võimalik juurutada ka *Angular* raamistiku poolt pakutava *RxJs* teegi abil, kuid raamistiku kasutamine lahendab arendaja eest probleemid, mis ise lahendust luues esineda võivad [5].

1.3 Eesmärk

Töö eesmärgiks on uurida keskse andmelaoga oleku haldamise raamistikke ja valida sobivaim. Seejärel valitud raamistik juurutada järgides seejuures parimaid praktikaid. Raamistiku abil arendada uued veel arendamata funktsionaalsused. Aja kokkuhoiu mõttes varasemalt valminud olemasolevaid funktsionaalsuseid esialgu raamistiku abil ümber ei kirjutata.

On oluline, et raamistik oleks küps ja raamistikul oleks võimalikult suur ja aktiivne kogukond. Küpse raamistiku puhul esineb vähem vigu ja produktiivsus on kõrgem [6]. Sageli on vaja mingile esinevale probleemile internetist abi leida ning mida rohkem on raamistikul kasutajaid, seda kiiremini leiab internetist lahendusi erinevatele probleemidele [7].

On oluline, et raamistikku kasutades kirjutatud kood oleks lihtsasti mõistetav ja hallatav, kuid samas on oluline ka koodi kirjutamisele kuluv aeg, kuna projekti arendamise aeg on piiratud.

On oluline, et raamistik võimaldaks isoleerida kõrvalmõjud käitlemise loogika muust loogikast. Sageli on tarvis teostada järjestikuseid *API* kutsungeid, kus eelmise *API* kutsungi vastus on sisendiks järgmisele *API* kutsungile. Sellises olukorras toimub kutsung eelmise kutsungi kõrvalmõjuna. Kõrvalmõjude korrektsel käitlemisel on oluline roll rakenduse korrektses toimimises. Vaikimisi *Angular* raamistikuga arendades

käideldakse kõrvalmõjusid komponentides. Kui esineb rohkelt kõrvalmõjude käitlemist, siis võib muutuda komponentides kõrvalmõjude käitlemine keeruliseks ja kood raskesti hallatavaks.

1.4 Ülevaade

Antud töö sisu on jaotatud kolmeks osaks.

Esimeses osas kirjeldatakse metoodika, mille alusel valitakse sobivaim oleku haldamise raamistik, kirjeldatakse valitud lahenduse juurutamise metoodikat ja samuti arendusmetoodikat, mida arendusettevõte kasutab rakenduse arendamisel.

Töö teises osas kirjeldatakse oleku haldamise lahendused, uuritakse nende vastavust eesmärkidele ning valitakse seejärel sobivaim.

Kolmandas osas kirjeldatakse lahenduse kasutamiseks parimad praktikad ja juurutatakse lahendus vastavalt. Lisaks arendatakse valitud lahendust kasutades teavituste funktsionaalsus ja oleku haldamine.

2 Metoodika

Antud peatüki esimeses osas kirjeldab autor metoodikat, mida kasutas oleku haldamise lahenduse valimisel. Teises osas kirjeldab autor metoodikat, mida ettevõte kasutas rakenduse arendamisel ning millest lähtudes arendati ka teavituste funktsionaalsus.

2.1 Oleku haldamise raamistiku valimise metoodika

Eesmärgi täitmiseks on välja valitud kolm oleku haldamise alternatiivi. Alternatiive esmalt kirjeldatakse ning seejärel valitakse sobivaim. Sobivaima alternatiivi leidmiseks võrreldakse järgnevaid omadusi: kogukond, küpsus, koodi hallatavus, koodi kirjutamiseks kuluv aeg ja kõrvalmõjude käitlemine.

2.1.1 Kontseptsioonide kirjeldus

Alternatiive kirjeldatakse lühidalt ning kirjeldatakse nende põhikontseptsioone ja elutsüklit. Kirjeldamisel lähtutakse alternatiivi ametlikust dokumentatsioonist ja internetist leitud artiklitest/kirjandusest.

2.1.2 Küpsus ja kogukond

Raamistike küpsuse ja kogukonna võrdlemiseks uuritakse alternatiivide *GitHub* repositooriumite tärnide arvu, esmast väljalaset, versioone ja *npm* allalaadimisi. Kogukonna aktiivsuse uurimiseks uuritakse otsingu sagedust *Google Trends* põhjal. Suurem *GitHub* tärnide arv, *npm* allalaadimiste arv ja otsingu sagedus viitab suuremale ja aktiivsemale kogukonnale ja ühtlasi aktiivsemale kogukonnale.

2.1.3 Koodi hallatavus ja kirjutamisele kuluv aeg

Koodi hallatavuse ja koodi kirjutamisele kuluva aja võrdlemiseks võrreldakse erinevate lahenduste rakendamisel *boilerplate* koodi kirjutamise hulka. *Boilerplate* kood võimaldab edasi anda rohkem konteksti ja on väljendusrikkam, kuid ühtlasi nõuab rohkem koodi kirjutamist.

Väljendusrikas kood, mis annab edasi rohkem konteksti, on lihtsamini mõistetav. Kood, mis on lihtsamini mõistetav on ühtlasi ka lihtsamini hallatav. Küll aga suurem koodi hulk tähendab ka suuremas mahus koodi ning ühtlasi keerulisemat hallatavust [7]. Samuti rohkem koodi kirjutamist suurendab koodi kirjutamisele kuluvat aega. Kuna tegemist on keeruka rakendusega, siis on autori hinnangul olulisem hallatava koodi lihtsus, kui koodi hulk.

2.1.4 Kõrvalmõjude isoleerimise võimalus

Kõrvalmõjude käitlemise isoleerimise osa uuritakse dokumentatsiooni alusel, kas ja mis tingimustel mingi lahendus võimaldab kõrvalmõjusid isoleerida ja käidelda muust koodist eraldi.

2.2 Valitud lahenduse juurutamise metoodika ja arendusmetoodika

Lahenduse juurutamiseks ja rakendamiseks uuritakse ning kirjeldatakse valitud lahenduse parimaid praktikaid. Seejärel lahendus juurutatakse ning rakendatakse, järgides parimaid praktikaid, uute funktsionaalsuste arendamisel. Varasemalt arendatud funktsionaalsuseid esialgu lahendusele üle ei viida.

Arendamisel kasutati *Scrum* raamistiku põhist arendusmetoodikat. *Scrum* on populaarne raamistik, mis järgib välearenduse põhimõtteid. Erinevalt traditsioonilistest metoodikatest, kus arendus toimub lineaarselt, toimub välearenduse puhul arendus iteratiivselt ja inkrementaalselt.

Antud projektil töötav arendusmeeskond oli 8 liikmeline: *UX/UI* disainer, 2 analüütik-testijat, arhitekt, 3 arendajat ja projektijuht. Arendaja sai arendustöödeks vajalikud spetsifikatsioonid enamasti analüütikult ning kes pärast arendustööde valmimist kontrollis manuaalse testimise käigus töö vastavust nõuetele.

3 Oleku haldamise raamistiku valimine

Käesolevas peatükis kirjeldatakse valitud oleku haldamise raamistike ja nende põhikontseptsioone. Seejärel lahendusi võrreldakse ja valitakse sobivaim.

3.1 NgRx

NgRx on *JavaScript* oleku haldamise raamistik ehitamaks reaktiivseid *Angular* rakendusi ning ühtlasi *RxJS* teegil baseeruv *Redux* mustri lahendus. Tegemist on progressiivse raamistikuga ning see koosneb mitmetest teekidest.

3.1.1 Kontseptsioonide kirjeldus

Peamised kontseptsioonid on:

- Ladu (*Store*) on objekt, milles hoiustatakse rakenduse olekut. See on rakenduses ainus tõeallikas. Laos hoiustatud olekut saab muuta ainult tegevuste vahendusel. *EntityState* on liides, mis hoiustab olemite olekuid. *EntityState* on oma olemuselt nagu andmebaasi tabel, kus iga olem vastab ühele reale tabelis ja igal olemil peab olema unikaalne identifikaator [8].

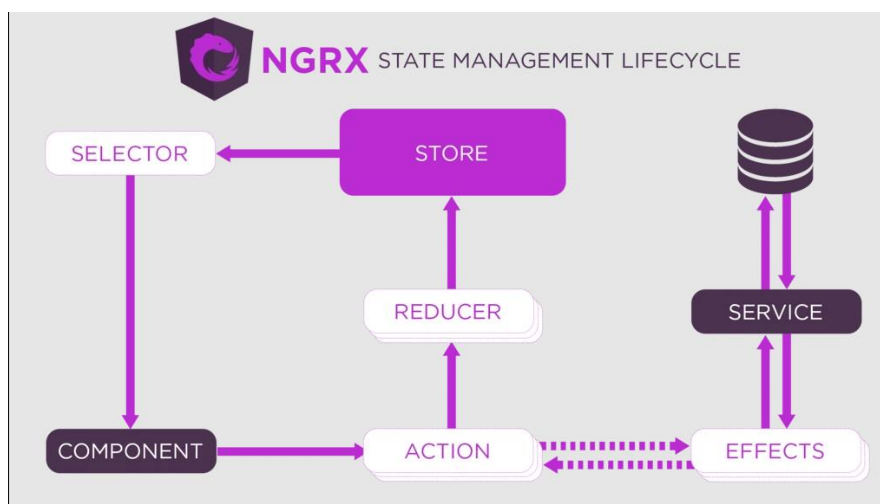
- Tegevused (*Action*) on ühed peamised *NgRx* ehitusplokid. Tegevused tähistavad rakenduses esinevaid unikaalseid sündmuseid. Tegevused võivad tähistada nii kasutaja tehtud tegevusi, *API* kutsungeid jne. Tegevusel on kohustuslik märkida tegevuse liik ja valikuliselt on võimalik anda juurde ka koormus. Koormus võib olla näiteks *API* kutsungi vastuses olnud andmete hulk. Tegevuse koormuse andmetega saab manipuleerida laos hoiustatud olekut. Näiteks, kui tegevuse koormaks on mingi lisandunud kirje, saab selle lisada lattu. Kirje lisamisel lattu lao olek muutub. Järgmise oleku arvutamiseks peab tegevus läbima reduktori funktsiooni.

- Reduktorid (*reducer*) vastutavad rakenduses ühelt olekult järgmisele ülemineku eest. Reduktor funktsioonid kuulavad rakenduses toimuvaid tegevusi ja käitlevad oleku üleminekut vastavalt tegevusele. Reduktorid on puhtad funktsioonid ning tagastavad alati saadud sisendi korral sama väljundi. Reduktorid on ilma kõrvalmõjudeta ja käitlevad oleku üleminekut sünkroonselt. Reduktor funktsioon

võtab sisendina tegevuse ja hetke oleku. Nende põhjal reduktor määrab, kas on tarvis oleku üleminekut või mitte. Reduktori väljund saab uueks olekuks ja saab olema sisendiks järgmisele reduktori kutsungile. Oleku muutumisel käivitatakse komponentides uuenenud andmete kuvamiseks vaadete taas renderdamine.

- Selektorid (*selector*) on puhtad funktsioonid, mille abil on võimalik laos hoiustatud olekuid kombineerida ja edastada vaatekomponentidesse. Selektorid on funktsioonid, mis defineerivad, kuidas olekust andmeid valitakse. Üks oluline selektorite omadus on see, et nad jätavad tulemusi meelde, vältimaks üleliigseid arvutusi [8].

- Efektide (*effects*) eesmärk on tegevuste kõrvalmõjude käitlemise loogika isoleerimine muust loogikast. Kõrvalmõjude käitlemiseks kuulatakse kõiki tegevusi, mis on lattu saadetud [8].



Joonis 1. NgRx oleku haldamise elutsükkel [9]

3.1.2 Küpsus ja kogukond

NgRx GitHub repositoorium omab 7000 tärni ning on *npm* andmetel 2021 aastal allalaaditud üle 2 miljoni korra. *NgRx* esialgne väljalase toimus aastal 2016. Lõputöö kirjutamise hetkel toimus viimane väljalase 7 detsember 2021 ning uusim versioon on 13.0.2

3.1.3 Koodi hallatavus ja kirjutamise aeg

NgRx nõuab võrreldes alternatiividega rohkem *boilerplate* koodi kirjutamist [10].

Teek *ngrx/data* pakub abstraktsioone elimineerimaks *boilerplate* kood. Samuti on saadaval *ngrx/schematics* teek, mis võimaldab poolautomaatselt vastava käsu abil genereerida vajalikud failid ja *boilerplate* kood.

3.1.4 Kõrvalmõjude käitlemine

Kõrvalmõjude isoleerimiseks on *NgRx* puhul nõutud *ngrx/effects* teegi paigaldamine.

3.2 NGXS

NGXS on Angular raamistikule oleku haldamise arendusmustrit pakkuv teek. Loojate sõnul on *NGXS* eesmärgiks vähendada *boilerplate* koodi hulka ja teha koodi kirjutamine lihtsamaks.

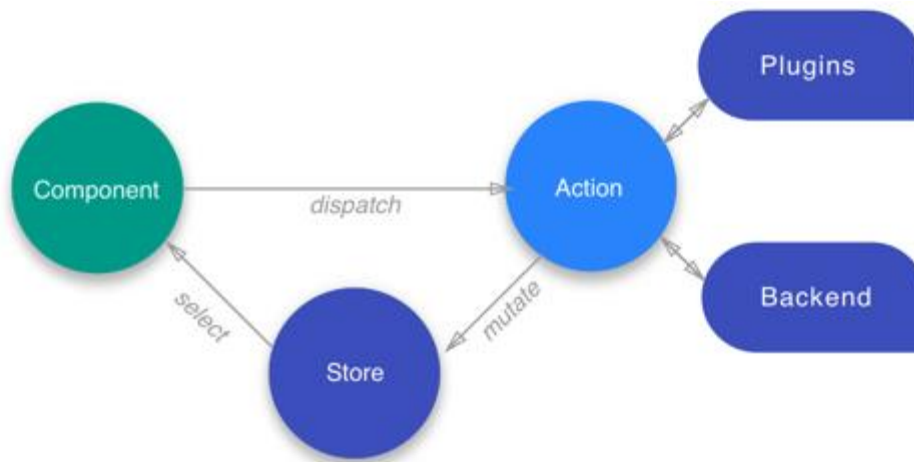
3.2.1 Kontseptsioonide kirjeldus

NGXS peamised kontseptsioonid on:

- Ladu (*Store*) on globaalse oleku konteiner. Sarnaselt *NgRx* laole hoiustab rakenduse olekut ja on ainsaks tõeallikaks. Ladu hõlbustab komponentide ja oleku vahelist vastastikust toimimist. *NGXS* kontekstis on olek (*State*) klass, mis defineerib oleku konteineri [11]. Valimine (*Select*) on funktsioon, mida kasutatakse laost oleku edastamiseks vaatekomponenti. Laos hoiustatud olekut saab muuta ainult tegevuste vahendusel.

- Tegevused (*Action*) sarnaselt *NgRx* tegevustele tähistavad unikaalseid sündmuseid. Kuna *NGXS* ei oma reduktorite kontseptsiooni, siis toimub lao oleku muutmine otse läbi tegevuste [12].

- *Plugin* liides võimaldab arendajatel ehitada abistavaid laiendusi. *NGXS* ametlikult pakub näiteks *Logger* pluginat, mis logib veebilehitseja konsooli tegevuste töötlemise protsessi.



Joonis 2. NGXS oleku haldamise elutsükkel [12]

3.2.2 Küpsus ja kogukond

NGXS GitHub repositoorium omab 3200 tärni ning on *npm* andmetel 2021 aastal allalaaditud napilt alla 360000 korda. *NGXS* esialgne väljalase toimus 28 märts 2018. Lõputöö kirjutamise hetkel oli toimunud viimane väljalase 18 mai 2021 ning uusim versioon on 3.7.2.

3.2.3 Koodi hallatavus ja kirjutamise aeg

NGXS nõuab vähem *boilerplate* koodi kirjutamist, kuna *NGXS* arendajad on püüelnud selle nimel, et vähese koodiga saaks rohkem saavutada. Selle jaoks on kombineeritud tegevused ja reduktorid ühte faili [12]. Samuti kasutatakse *boilerplate* koodi vähendamiseks *TypeScript* funktsionaalsusi nagu dekoraatorid ja klassid [11].

3.2.4 Kõrvalmõjude isoleerimine

Kõrvalmõjude isoleerimist võimaldab *NGXS* puhul selle põhipakett. Kõrvalmõjude haldamine toimub läbi tegevuste käsitlejate (*Action handler*). Kõik tegevused saabuvad enne lattu jõudmist tegevuse käitlejasse [14].

3.3 Akita

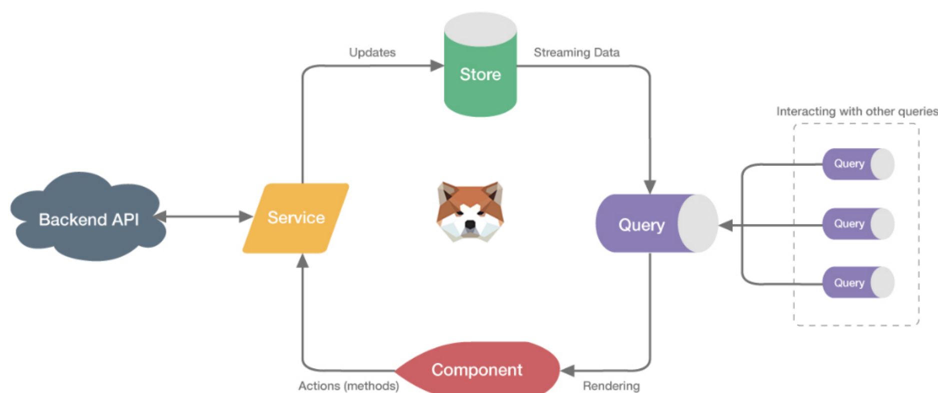
Akita on oleku halduse lahendus, mis baseerub *RxJS*-l ja on inspireeritud *Flux* ja *Redux* mustrist. *Akita* eeliseks peetakse tema lihtsust.

3.3.1 Kontseptsioonide kirjeldus

Akita põhikontseptsioonid on:

- Ladu (*Store*), mis sarnaselt teiste alternatiividega on ainsaks tõeallikaks. Lihtsustamaks töötamist olemitega, pakub Akita ka spetsiifilisemat ladu olemite olekute hoiustamiseks (*EntityStore*). Akita raamistikus üldiselt puudub tegevuse mõiste, mille abil teistes alternatiivides toimus lao oleku muutmine. Lao uuendamiseks pakub ladu selleks vastavaid meetodeid. Antud meetodeid kutsutakse välja üldjuhul *Angular* teenustest [15].

- Päring (*Query*) on klass, mis pakub funktsionaalsusi pärimaks laost olekut, mida edastada komponentidesse. Tegemist on sarnase päringuga nagu andmebaasist andmete päring [15]



Joonis 3. Akita oleku haldamise elutsükkel [15]

3.3.2 Küpsus ja kogukond

Akita GitHub repositoorium omab 3300 tärni ning on *npm* andmetel 2021 aastal alla laaditud üle 197000 korda. Akita esialgne väljalase toimus 12 juuni 2018. Lõputöö kirjutamise hetkel oli toimunud viimane väljalase 29 november 2021 ning uusim versioon on 7.0.1.

3.3.3 Koodi hallatavus ja kirjutamise aeg

Akita lahendus nõuab minimaalselt *boilerplate* koodi kirjutamist [10]. *Akita* on madala *boilerplate* koodi tase on saavutatud osaliselt mõningate kontseptsioonide ära kaotamise teel.

3.3.4 Kõrvalmõjude isoleerimine

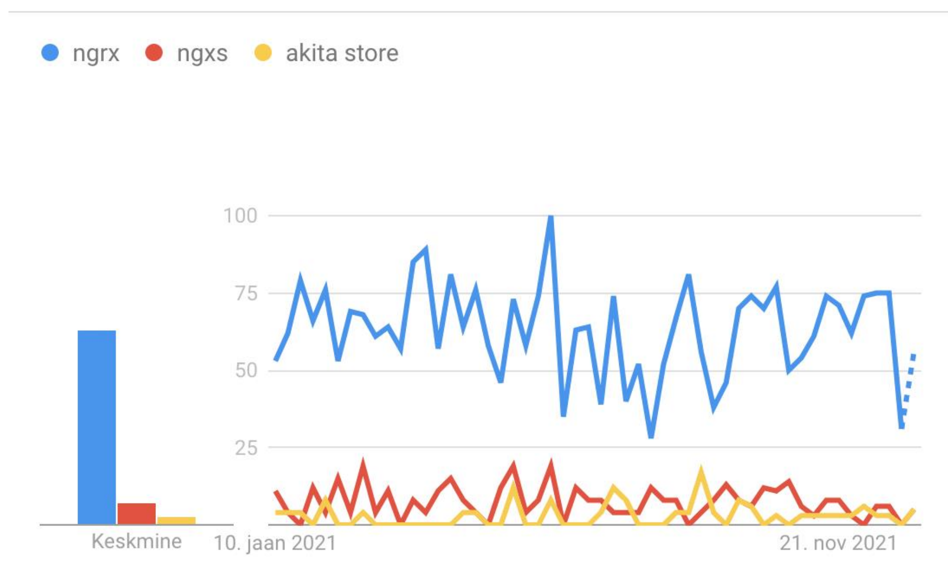
Akita ise kõrvalmõjude isoleerimist ei võimalda. Kõrvalmõjude isoleerimiseks *Akita* raamistikus on selleks vajalik paigaldada kolmanda osapoole teek. Antud teegi kasutamine nõuab ka tegevuste kontseptsiooni juurutamist.

3.4 Lahenduste võrdlus

Järgnevalt analüüsitakse ja võrreldakse raamistikke nende uuritud omaduste põhjal.

3.4.1 Küpsus ja kogukond

Google Trends andmetel on kõige otsitavam oleku haldamise lahendus *NgRx* ning *Akita* ja *NGXS* on kordades vähem otsitavad terminid.



Joonis 4. Lahenduste otsitavus 2021 aastal Google Trends andmetel

Samuti ka *GitHub* tähtede arvult on *NgRx* kõige populaarsem, omades rohkem tärne, kui *Akita* ja *NGXS* kahepeale kokku. Samuti on *NgRx npm* allalaadimiste arvu põhjal 2021 aastal teistest alternatiividest kaugelt ees.

Tabel 1. Raamistike populaarsuse võrdlus

	NgRx	***NGXS***	***Akita***
GitHub tähtede arv	7000	3200	3300
npm allalaadimiste arv 2021 aastal	20082701	3599674	1971305

NgRx arendamist alustati 2 aastat enne *NGXS* ja *Akita* arendamist, mis teeb temast kõige pikema ajalooga raamistiku. Viimase väljalaske kuupäeva ja versiooni numbri põhjal autor järeldab, et *NgRx* on kõige küpsema ja kõige suurema ning aktiivsema kogukonnaga alternatiiv

Tabel 2. Raamistike arendamise ajalugu

	NgRx	***NGXS***	***Akita***
Võrreldav versioon	13.0.2	3.7.2	7.0.1
Esialgne väljalase	2016	28.03.2018	12.06.2018
Viimane väljalase	07.12.2021	18.05.2021	29.11.2021

3.4.2 Koodi hallatavus ja kirjutamise aeg

Akita plussiks on madal *boilerplate* koodi hulk. Küll aga madal *boilerplate* koodi hulk oli saavutatud tänu sellele, et *Akita* on elimineerinud mõned olulised kontseptsioonid nagu näiteks tegevused ja kõrvalmõjud. Selline kontseptsioonide ära kaotamine või abstrakteerimine võib osutuda keerulisemate rakenduste loomisel probleemiks.

NGXS nõuab keskmiselt *boilerplate* koodi kirjutamist, kuid kuna raamistik on *Angular* arendajale alguses lihtsama vaevaga õpitav, siis seetõttu on koodi kirjutamisel kuluv aeg alguses madalam.

NgRx poolt vaikimisi nõutud *boilerplate* koodi kirjutamine suurendab koodi kirjutamiseks kuluvat aega. Standardiseeritud koodi tõttu on lihtsustatud keeruliste funktsioone arendamine. Autori hinnangul keerulisemate funktsionaalsuste arendamisel *boilerplate* koodi kirjutamine ning lihtsamate funktsionaalsuste arendamisel *ngrx/data* teeki kasutamine võimaldab kõige paremat koodi hallatavuse ja kirjutamiseks kuluva aja suhet.

3.4.3 Kõrvalmõjude isoleerimine

Kõrvalmõjude isoleerimiseks oli *Akita* puhul vajalik kolmanda osapoole teegi lisamine. Ühtlasi oli lisaks vajalik juurutada ka tegevuste kontseptsioon, kuna vastasel juhul ei ole võimalik kõrvalmõjusid käidelda muust koodist eraldi. Autori arvates on hea hoida kolmanda osapoole teekide arv võimalikult madal.

NGXS ja *NgRx* eelis *Akita* ees on see, et ei ole vajalik kolmanda osapoole teegi lisamine kõrvalmõjude käitlemiseks.

4 *NgRx* juurutamine ja rakendamine

Järgnevalt juurutatakse *NgRx* raamistik. *NgRx* valiti ning võeti arendajate poolt kasutusele, kuna oli kõige küpsem ja suurima kogukonnaga raamistik. Samuti võimaldab kõige paremat koodi hallatavuse ja koodi kirjutamiseks kuluva aja suhet.

4.1 Teekide valik

NgRx on progressiivne raamistik ja pakub järgnevaid teekisid:

- *ngrx/store* - *NgRx* põhiteek oleku haldamiseks.
- *ngrx/effects* - Teek kõrvalmõjude isoleerimiseks
- *ngrx/route-storer* - Teek ühendamaks *Angular* raamistiku ruuter *ngrx/store* oleku halduse lahendusega.
- *ngrx/entity* - Teek, olemi kogumite halduse lihtsustamiseks.
- *ngrx/schematics* - Teek *boilerplate* koodi genereerimiseks
- *ngrx/store-devtools* - Arendustööriistad raamistiku efektiivsemaks kasutamiseks
- *ngrx/data* - Teek *boilerplate* koodi elimineerimiseks

Parima arendamise kogemuse eesmärgil võeti kasutusele kõik teegid välja arvatud *ngrx/data* teek. Tulevikus, kui on rohkem kogemust *NgRx* lahendusega ja selle toimimise loogika on kinnistunud, siis saab vajadusel lisaks *ngrx/data* teegi kasutusele võtta eesmärgiga kirjutada koodi kiiremini.

4.2 Oleku haldamise printsiibid

NgRx oleku haldamise kolm printsiipi on [16]:

- Üks ja ainus tõe allikas. Olek, mis on oma olemuselt objektide hierarhia, on hoiustatud üksikus lao objektis.

- Olek on ainult loetav. Olemasolevat olekut ei uuendata, vaid asendatakse uue olekuga
- Oleku asendamiseks kasutatakse puhtaid funktsioone. Reduktor funktsioon võtab sisendina tegevuse ja hetke oleku ning tagastab uue oleku.

4.3 Koodistiil

NgRx on tugevalt *FRP* suunitlustega ning suunab arendajat kirjutama deklaratiivses stiilis koodi. *FRP* ehk funktsionaal-reaktiivne programmeerimine on deklaratiivse programmeerimise stiil, mis hõlmab funktsionaalse programmeerimise paradigma kasutamist reageerimaks reaktiivsetele andmevoogudele. Tegemist on funktsionaalse ja reaktiivse programmeerimise paradigma kombineerimisel tuletatud paradigmaga [17].

Paradigmasse on kombineeritud ideed nagu näiteks [18]:

- Muutmatud andmed
- Vaadeldavad objektid
- Puhtad funktsioonid
- Staatiline tüüpimine
- Oleku üleminek ühesuunaliselt

4.4 Parimad praktikad

NgRx kodulehel esile toodud artikli [19] kohaselt üheks parimaks praktikaks peetakse juur lao mooduli ja tunnuste lao moodulite loomist. Juur lao mooduli eesmärk on siduda kokku tunnuste lao moodulid, misjärel on seda hõlpsalt võimalik importida rakenduse peamisesse moodulisse ehk App moodulisse.

NgRx dokumentatsiooni [20] põhjal peetakse parimaks praktikaks tegevuste kirjutamisel järgnevate reeglite järgimist:

- Tegevuste kirjutamine enne muid arendustöid. See võimaldab paremini mõista arendatavat tunnust.

- Tegevuste kategoriseerimine vastavalt sündmuse allikale.
- Kirjutada võimalikult palju tegevusi, kuna tegevuste kirjutamine on soodne ning mida rohkem tegevusi on kirjeldatud, seda paremini on väljendatud rakenduse toimimise voog.
- Tegevus tähistab sündmuse toimumist mitte käske
- Iga tegevus vastab unikaalsele sündmusele ning on informatiivselt kirjeldatud võimaldades paremat koodi silumist.

NgRx arendaja Brandon Roberts on öelnud oma blogipostituses [21], et levinud on vaatemudelite koostamine kombineerides komponentides asuvaid vaadeldavaid objekte kasutades *combineLatest* operaatori. Tema sõnul on parem praktika vaatemudeli koostamine ühe selektorina. Selline lahendus on efektiivsem, kuna teostatakse vähem arvutusi oleku uuenemise korral ning komponendid saavad olema puhtamad.

4.5 Juuroleku loomine

Juuroleku loomiseks defineeritakse `RootState` nimeline liides. Juurolekuga seotakse esialgu *NgRx* ruuteri lao olek, kuna tegemist on *ngrx/router-store* teegi poolt pakutava funktsionaalsusega. *NgRx* ruuteri lao olek on sünkroniseeritud *Angular* ruuteri olekuga ning ruuteri ja juuroleku sidumise eesmärgiks on hõlbustada ruuteri olekus hoiustatud oleku kasutamist selektorites. Ruuteri olekus hoiustatakse näiteks aadressiriba olekut ehk *UR*L.

```
export interface RootState {
   router: RouterReducerState<any>;
}
```

Joonis 5. Juuroleku objekt

Juuroleku muutmise jaoks kaardistatakse rakenduse tunnuste olekute reduktorid ühte muutujasse. Igale juurolekuga seotud tunnuse olekule on vajalik registreerida vastav reduktor.

```
export const reducers: ActionReducerMap<RootState> = {
   router: routerReducer
};
```

Joonis 6. Juuroleku reduktor

Angular ruuteri oleku ja ja *NgRx* ruuteri lao oleku sünkroonis hoidmiseks on lisatud kaardistatud reduktorite objekti *ngrx/router-stor*e implementatsioon ruuteri reduktorist ehk `routerReducer`.

Tagamaks, et olek on ainult loetav ning vältimaks olekuga esinevaid probleeme, mis võivad tuleneda otsesest oleku muutmisest, kasutatakse *storeFreeze* nimelist metareduktorit. Antud metareduktor on *NgRx* arendajate poolt arendatud teek, mis külmutab lao oleku ning takistab laos hoiustatud oleku otsest muutmist, andes sellest arendajale teada konsooli prinditud veateate näol.

Koodi mugavaks silumiseks lõi töö autor `debug` nimelise metareduktori. Tegemist on puhta funktsiooniga, mis logib konsooli rakenduse hetkeoleku ning tegevuse, mis käivitas metareduktori funktsiooni. Kuna *NgRx* oleku haldamise muster jõustab ühesuunalist andmevoogu, siis autori hinnangul on selline koodi silumise lahendus arendamisel kasulik. Lahendus võimaldab väga täpselt jälgida rakenduses hoiustatud olekute muutumist tegevuste korral.

```
export function debug(reducer: ActionReducer<any>):
ActionReducer<any> {
 return function(state, action) {
   console.log('state', state);
   console.log('action', action);

   return reducer(state, action);
 };
}
```

Joonis 7. Koodi silumisel abistav metareduktor

Vigade silumise ja lao külmutamise metareduktoreid kasutatakse ainult arendus- ja testkeskkonnas ning tootekeskkonda see kood ei lähe.

Sidumaks kokku juuroleku juures olulised objektid, luuakse *Angular* moodul ehk *NgModule* ning registreeritakse need mooduli `imports` massiivis.

```
@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    StoreModule.forRoot(reducers, {metaReducers}),
    EffectsModule.forRoot([]),
    StoreRouterConnectingModule.forRoot({
      stateKey: 'router'
    }),
  ],
})
export class RootStoreModule { }
```

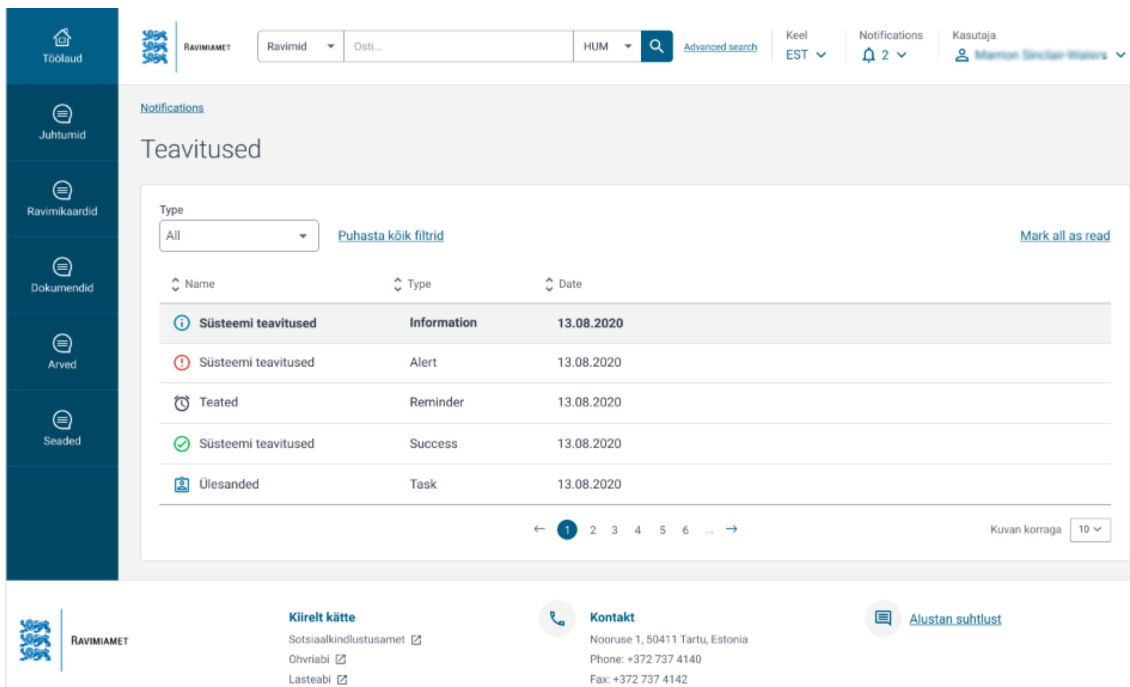
Joonis 8. Juuroleku lao moodulis objektide kokku sidumine

4.6 Teavituste oleku haldamine ja arendus

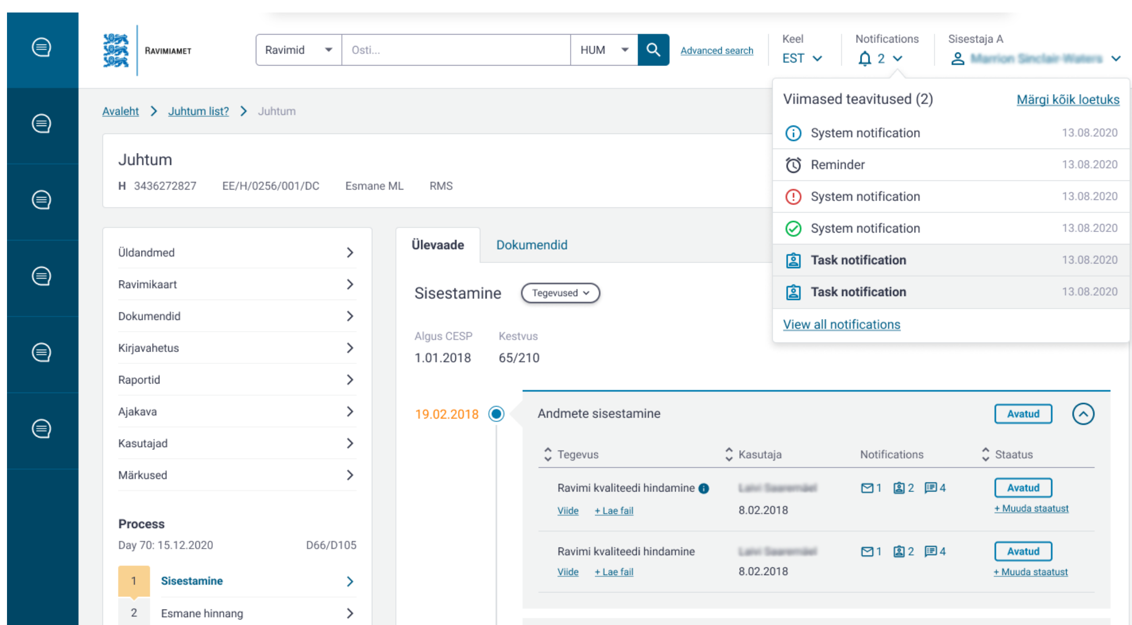
Järgnevalt arendatakse teavituste funktsionaalsus ning teostatakse ka oleku haldamine. Funktsionaalsuse arendamisel järgitakse analüütiku koostatud nõudeid ja *UX/UI* disaineri koostatud disaine. Teavituste funktsionaalsuse arenduse kajastamine antud töös sai valitud seetõttu, kuna tegemist on sarnase probleemiga, mis pani aluse ühele esimesele keskse andmelaoga üheleherakenduse oleku haldamise mustrile ehk *Flux* arendusmustrile [22].

4.6.1 Teavituste UX/UI disain ja nõuded

Järgnevalt on esitatud *UX/UI* disaineri koostatud disainide joonised. Joonised on arendajatele üldiseks järgimiseks ning ei peegelda reaalset arendamise tulemit. Näiteks joonistel kujutatud andmed ei ole reaalsed ning osad tekstid (nt. tabeli tulpade päised jne.) on inglise keeles. Arendamise käigus luuakse teavituste funktsionaalsus siiski peamiselt analüütiku koostatud nõuete alusel, kuid mis suuresti kattuvad disaini joonistel kujutatuga.



Joonis 9. UX/UI disaineri nägemus teavituste haldamise lehest



Joonis 10. UX/UI disaineri nägemus päises asuvatest teavitusetest

Järgnevalt on kirjeldatud analüütiku koostatud nõuded, millest arendaja lähtub teavituste arendamisel. Osa nõudeid olid oma olemuselt sarnased. Nõuded hõlmasid teavitusega seotud olemite pärimist ning nimetuse kuvamist vastavalt seotud olemile. Autor otsustas valida välja ühe sarnase nõude, mida töös kajastada ning jätta ülejäänud sarnase olemusega nõuete täitmise töö skoobist välja. Otsustati kajastada nõude täitmist, mis

hõlmas endas kasutaja teavitustega seotud tööülesannete pärimist ja teavituse nimes kuvamist (Nõue nr 5). Peamiselt saavad kajastamist nõuete täitmised, mis on seotud oleku haldamisega. Samuti kasutaja teavituste seadistamise osa kajastatakse põgusalt ning mitte täielikult.

Teavituste üldised nõuded:

1. Kasutaja logimisel süsteemi tehakse alglaadimine, mis pärib sisse loginud kasutaja kõik teavitused.
2. Mingi aja tagant teostatakse päringud, mis pärib vahepeal lisandunud uusi teavitusi.
3. Teavitusi kuvatakse nii päises kui ka teavituste haldamise lehel
4. Teavituste nimetused ja ikoonid kuvatakse vastavalt teavituse tüübile.
5. Kui tegemist on tööülesannetega seotud teavitusega, siis pärida seotud tööülesanded ja teavituse nimetusele lisada juurde tööülesande nimetus.
6. Teavitus võib olla link ja kui sellele peale vajutada, siis navigeeritakse kasutaja vastavale lehele.
7. Teavitused on vaikimisi järjestatud aja järgi kahanevas järjestuses.
8. Lugemata teavitused on halli taustaga.
9. Teavitusi saab märkida loetuks
10. Peab olema võimalik korraga ka kõigi teavituste loetuks märkimine.

Teavituste nõuded päises:

1. Päises kuvatakse kellukese ikoon.
2. Kellukese ikoon kuvatakse vastavalt sellele, kas on lugemata teavitusi või mitte.
3. Kellukese ikooni juures kuvatav lugemata teavituste arv.
4. Kuvatakse korraga maksimaalselt 10 teavitust.
5. Peab olema viide teavituste haldamise lehele

Teavituste nõuded teavituste haldamise lehel:

1. Teavitused kuvatakse tabelis. Tabelil on veerud: teavituse liik ja aeg
2. Teavitusi saab aja alusel järjestada
3. Teavitusi saab filtreerida järgnevatel alustel: teavituse liik, aeg, kas teavitus on loetud või lugemata

4. Teavitusi saab seadistada. Seadistused võimaldavad teavituste saamist sisse ja välja lülitada vastavalt teavituse liigile.

4.6.2 Teavituse mudel

Joonisel on kujutatud serverist päritud teavituse mudel lihtsustatud kujul. Töö skoobist tulenevalt on eemaldatud viited teistele olemitele, millega teavitus võib seotud olla. Ainsate viidetena on sisse jäetud viited seotud tööülesande olemile ja teavituse saajale.

```
export interface Notification {
 id: string;
 taskId: string;
 createdTime: string;
 isRead: boolean;
 notificationType: string;
 personId: string; }
```

Joonis 11. Serverist päritud teavituse mudel

Tabelis 3 on selgitatud serverist päritud mudeli väljasid.

Tabel 3. Serverist päritud teavituse mudeli väljade selgitused

Välja nimetus	Selgitus
id	Teavituse identifikaator
taskId	Viide tööülesande olemile, millega teavitus on seotud ning mille nimetust on tarvis kuvada seotud teavituse pealkirjas.
createdTime	Teavituse loomise aeg
isRead	Kas teavitus on loetud või lugemata
notificationType	Teavituse liik
personId	Viide teavituse saajale

4.6.3 Tegevuste kirjeldamine

Tegevuste kirjeldamisel lähtutakse alamjaotises 4.1.4 välja toodud reeglitest. Järgnevalt autor kirjeldab, kuidas toimub tegevuste kirjutamisel reeglite järgimine.

Kohtlemaks tegevusi sündmustena ning mitte käskudena, kirjutatakse tegevuste muutujate nimed ja kirjeldused mineviku vormis.

Et tegevus oleks informatiivselt kirjeldatud, siis püütakse võimalikult täpselt kirjeldada sündmust, mida tegevus tähistab.

Tagamaks, et iga tegevus tähistab mingit unikaalset sündmust, siis pannakse tegevused teele ainult ühest allikast ja ühe korra ehk välditakse tegevuste taaskasutamist.

Tegevused on kategoriseeritud kolme kategooriasse vastavalt sellele, kust tegevus teele pannakse:

- `NotificationsAPIActions` kategooriasse kuuluvad tegevused, mis on saadetud lattu teavituste *API* kutsungi tulemusena. Nende tegevuste sündmuse allikaks on kandiliste sulgude vahele märgitud '*Notifications API'*.
- `HeaderNotificationsActions` kategooriasse kuuluvad tegevused, mis on saadetud lattu päises asuvast teavituse komponendist. Nende tegevuste sündmuse allikaks on kandiliste sulgude vahele märgitud *'Header Notifications'*.
- `NotificationsPageActions` kategooriasse kuuluvad tegevused, mis on saadetud lattu teavituste haldamise lehelt. Nende tegevuste sündmuse allikaks on kandiliste sulgude vahele märgitud '*Notifications Page'*.

```
import * as NotificationsAPIActions from './notifications-api.actions';
import * as HeaderNotificationsActions from './header-notifications.actions';
import * as NotificationsPageActions from './notifications-page.actions';

export {
	NotificationsAPIActions,
	HeaderNotificationsActions,
	NotificationsPageActions
}
```

Joonis 12. Tegevuste kategooriad

Järgnevalt kaardistatakse ja luuakse nõuete põhjal tegevused.
Tegevuste loomiseks kasutatakse `createAction` funktsiooni. Igal tegevuse loomisel väärtustatakse kohustuslik parameeter, mis on sõne tüüpi ja tähistab tegevuse liiki. Tegevuse liigi väärtustamisel on lähtutud *NgRx* dokumentatsioonis esitatud konventsioonist, mille kohaselt kandilistes sulgudes on sündmuse allikas ja sulgude järel sündmuse kirjeldus. Tegevuste kaardistamise tulemusena loodud kood asub Lisa 3.

Nõue: Kasutaja logimisel süsteemi tehakse alglaadimine, mis pärib sisse loginud kasutaja kõik teavitused.

Loodud tegevused:

- Tegevus `requestedPersonAllNotifications` kirjeldab sündmust, mis tähistab kasutaja logimise järel nõude esitamist kõikide teavituste alglaadimiseks. Sündmuse allikaks on päise teavituse komponent ning tegevus saadetakse teele päise teavituse komponendi algväärtustamisel kasutaja sisse logimise järgselt.
- Tegevus `requestedPersonAllNotificationsSuccess` kirjeldab sündmust, mis tähistab sisse loginud kasutaja kõikide teavituste pärimiseks teostatud *API* kutsungi õnnestumist. Sündmuse allikaks on teavituste *API*. Tegevuse koormaks on *API* kutsungi tulemusena päritud teavituste massiiv.
- Tegevus `requestedPersonAllNotificationsFailure` kirjeldab sündmust, mis tähistab sisse loginud kasutaja kõikide teavituste pärimiseks teostatud *API* kutsungi ebaõnnestumist. Sündmuse allikaks on teavituste *API*. Tegevuse koormaks on *API* kutsungi vastuses sisalduv veateade.

Nõue: Mingi aja tagant teostatakse päringud, mis pärib vahepeal lisandunud uusi teavitusi.

Loodud tegevused:

- Tegevus `startedPollingForNewNotifications` kirjeldab sündmust, mis tähistab nõudmist alustamaks sisse loginud kasutaja uute teavituste pärimist intervalli tagant. Sündmuse allikaks on päise teavituse vaatekomponent ning see toimub päise teavituse vaatekomponendi algväärtustamisel rakenduse käivitamisel.
- Tegevus `receivedNewNotificationsSuccess` kirjeldab sündmust, mis tähistab uute teavituste päringu õnnestumist. Sündmuse allikaks on teavituste *API*. Tegevuse koormaks on *API* kutsungi tulemusena päritud uute teavituste massiiv.
- Tegevus `receivedNewNotificationsFailure` kirjeldab sündmust, mis tähistab uute teavituste päringu ebaõnnestumist. Sündmuse allikaks on teavituste *API*.

Nõue: Peab olema võimalik korraga kõigi teavituste loetuks märkimine.

Loodud tegevused:

- Tegevus `markedAllAsReadInHeader` kirjeldab sündmust, mille allikaks on päise teavituse komponent ning mis tähistab päises asuva "Märgi kõik loetuks" nupule vajutamist.
- Tegevus `markedAllAsRead` kirjeldab sündmust, mis tähistab teavituste haldamise lehel asuva "Märgi kõik loetuks" nupule vajutamist.
- Tegevus `markedAllAsReadSuccess` kirjeldab sündmust, mis tähistab kõikide teavituste loetuks märkimise eesmärgil teostatud *API* kutsungi õnnestumist. Tegevuse koormaks on *API* kutsungi tulemusena uuenenud (väli `isRead` on märgitud tõeseks) teavituste massiiv.
- Tegevus `markedAllAsReadFailure` kirjeldab sündmust, mis tähistab kõikide teavituste loetuks märkimise eesmärgil teostatud *API* kutsungi ebaõnnestumist. Tegevuse koormaks on *API* kutsungi vastuses sisalduv veateade

Nõue: Teavitust saab märkida loetuks.

Loodud tegevused:

- Tegevus `markedAsReadInHeader` kirjeldab sündmust, mis tähistab rakenduse päises asuva teavituste aknas vajutamist mõnele lugemata teavitusele. Tegevuse koormaks on loetuks märgitud teavituse identifikaator.
- Tegevus `markedAsRead` kirjeldab sündmust, mis tähistab teavituste haldamise lehel vajutamist mõnele lugemata teavitusele. Tegevuse koormaks on loetuks märgitud teavituse identifikaator.
- Tegevus `markedAsReadSuccess` kirjeldab sündmust, mis tähistab teavituse loetuks märkimise eesmärgil teostatud *API* kutsungi õnnestumist. Tegevuse koormaks on *API* kutsungi vastuses sisalduv uuenenud teavitus, mille omadus `isRead` on märgitud tõeseks.
- Tegevus `markedAsReadFailure` kirjeldab sündmust, mis tähistab kõikide teavituste loetuks märkimise eesmärgil teostatud *API* kutsungi ebaõnnestumist. Tegevuse koormaks on *API* kutsungi vastuses sisalduv veateade

Nõue: Kui tegemist on tööülesannetega seotud teavitusega, siis teavituse nimetusele lisada juurde tööülesande nimetus.
Loodud tegevused:

- Tegevus `requestedNotificationsTasksSuccess` kirjeldab sündmust, mis tähistab teavitusega seotud tööülesannete pärimise eesmärgil teostatud *API* kutsungi õnnestumist. Tegevuse koormaks on *API* kutsungi tulemusena päritud tööülesannete massiiv.
- Tegevus `requestedNotificationsTasksFailure` kirjeldab sündmust, mis tähistab teavitusega seotud tööülesannete pärimise eesmärgil teostatud *API* kutsungi ebaõnnestumist.

4.6.4 Teavituste oleku loomine

Teavituse oleku jaoks loodi `NotificationsState` nimeline liides, mis laiendab `EntityState` liidest. `EntityState` liidest on soovitatav laiendada, kui hallatakse olemite kollektsioone. Teavituste olekule hetkel väljasid juurde ei lisata ning keskendutakse peamiselt teavituste kollektsiooni haldamisele. Tulevikus on plaanitud lisada juurde veel väljasid, mis on seotud andmete laadimise staatusega ja veateadetega.

```
export interface NotificationsState extends EntityState<Notification> {
}
```

Joonis 13. Teavituste oleku objekt

Olekus hoiustatud teavituste kollektsiooni haldamiseks on loodud `createEntityAdapter` funktsiooni kasutades adapter, mis pakub kollektsiooni haldamiseks *CRUD* operatsioone. Kollektsiooni sorteerimise eesmärgil on võimalik funktsioonile anda sisendina sorteerimise funktsioon `sortComparer`. Kuna üks nõuetest oli, et teavitused peavad olema sorteeritud ajalises järjestuses, siis `sortComparer` väljale edastatakse vastav järjestamise funktsioon. Funktsioon võtab sisendina kaks parameetrit: esimene ja teine element, mida omavahel võrreldakse aja järgi. Kui esimese elemendi aeg on varasem, kui teise elemendi aeg, siis tagastatakse arv 1 ehk esimene element lükatakse järjestuses ühe võrra tahapoole. Kui esimese elemendi aeg on suurem ehk hilisem, kui teise elemendi aeg, siis tagastatakse arv -1 ehk esimene element liigub järjestuses ühe võrra ettepoole. Muudel juhtudel tagastatakse arv 0 ehk elementide järjestus ei muutu.

```
export const notificationsAdapter = createEntityAdapter<Notification>({
 sortComparer: (first, second) => {
   if (first.createdTime < second.createdTime) {
       return 1;
     }
     if (first.createdTime > second.createdTime) {
       return -1;
     }
     return 0;
 }
});
```

Joonis 14. Teavituste olemite kollektsiooni haldamise adapter

Olemasoleva oleku asendamine uue olekuga toimub reduktor funktsioonis. Reduktor funktsioon võtab sisendina kaks parameetrit, mille põhjal tagastab uue oleku: hetke olek ja käsitletav tegevus. Kui reduktor ei tea, kuidas tegevust peaks käsitlema, tagastab ta hetke oleku. Reduktori esmaseks käivitamiseks tuleb defineerida algolek.

Algoleku väärtustamiseks kasutatakse `getInitialState` meetodit, millele antakse sisendiks tühi objekt, kuna hetkel hallatakse ainult teavituste kollektsiooni ning muid väljasid olekus defineeritud ei ole.

```
export const initialState: NotificationsState =
notificationsAdapter.getInitialState({});
```

Joonis 15. Teavituste algolek

Teavituste oleku haldamise kontseptsioonide kokku sidumiseks luuakse vastav lao moodul ning seejärel seotakse see kokku juurolekuga.

4.6.5 Oleku uuendamine vastavalt tegevusele

Järgnevalt on esitatud teavituste oleku asendamist teostav reduktor funktsioon. Reduktoris on registreeritud hetkel ainult need tegevused, mis on seotud teavituste kollektsiooni haldamisega. Olekut, mis on seotud veateadete ja alglaadimise staatusega, hetkel ei hallata ning nendega seotud tegevusi seetõttu ei registreerita. See osa lisatakse hiljem.

Tegevuse `requestedPersonAllNotificationsSuccess` korral kasutatakse oleku muutmiseks meetodi `setAll`, mis seab tegevuse koormaks olnud teavituste massiivi teavituste olekus hoiustatud teavituste kollektsiooni väärtuseks.

Tegevuse `markedAllAsReadSuccess` korral kasutatakse oleku muutmiseks meetodi `updateMany`, mis uuendab teavituste olekus hoiustatud teavituste kollektsiooni neid kirjeid, mille identifikaatorid kattuvad tegevuse koormaks olnud kirjete identifikaatoritega.

Tegevuse `receivedNewNotificationsSuccess` korral kasutatakse oleku muutmiseks meetodi `addMany`, mis lisab tegevuse koormaks olnud teavituste massiivi kirjed teavituste olekus hoiustatud teavituste kollektsiooni.

Tegevuse `markedAsReadSuccess` korral kasutatakse oleku muutmiseks meetodi `updateOne`, mis uuendab teavituste olekus hoiustatud kollektsiooni kirjet, mille identifikaator kattub tegevuse koormaks olnud kirje identifikaatoriga.

```
export const notificationsReducer = createReducer<NotificationsState>(
  initialState,
  on(NotificationsAPIActions.requestedPersonAllNotificationsSuccess, (state, action) => {
    return notificationsAdapter.setAll(action.payload, state)
  }),
  on(NotificationsAPIActions.markedAllAsReadSuccess, (state, action) => {
    return notificationsAdapter.updateMany(action.payload, state)
  }),
  on(NotificationsAPIActions.receivedNewNotificationsSuccess, (state, action) => {
    return notificationsAdapter.addMany(action.payload, state)
  }),
  on(NotificationsAPIActions.markedAsReadSuccess, (state, action) => {
  return notificationsAdapter.updateOne(action.payload, state)
})
);
```

Joonis 16. Reduktor funktsioon teavituste oleku asendamiseks

Teavitustega seotud tööülesannete kollektsiooni haldamiseks on loodud samuti vastav olek, mille uuendamine toimub vastava reduktor funktsiooni abil. Teavitustega seotud tööülesannete pärimiseks teostatud *API* kutsungi järgselt teele pandud `requestedNotificationsTasksSuccess` tegevuse korral kasutatakse tööülesannete reduktoris meetodit `upsertMany`, Antud meetod lisab tegevuse koormaks olnud tööülesannete massiivi kirjed kollektsiooni, kui neid seal juba ei eksisteerinud. Kui kirjed eksisteerisid, siis asendab need.

4.6.6 Tegevuste kõrvalmõjud

Järgnevalt kirjeldatakse kõrvalmõjusid, mis esinevad mingi tegevuse/sündmuse korral. Kõrvalmõju vaatleb lattu saadetud tegevusi. Tegevustega opereerimiseks kasutatakse `pipe` funktsiooni. Kõrvalmõjude loomiseks kasutatakse `createEffect` funktsiooni. Operaator `ofType` kasutatakse filtreerimaks kõikide tegevuste seast välja tegevus, mis on vastavas kõrvalmõjus registreeritud. Üks kõrvalmõju võib esineda mitme tegevuse korral. Kõrvalmõjude realiseerimise käigus loodud kood asub Lisa 4.

Tegevuse `requestedPersonAllNotifications` kõrvalmõjuna teostatakse *API* kutsung pärimaks sisse loginud kasutaja kõik teavitused. *API* kutsungi teostamiseks valitakse laost sisse loginud kasutaja identifikaator ja edastatakse *API* kutsungi meetodisse. Eduka *API* kutsungi järgselt saadetakse lattu `requestedPersonAllNotificationsSuccess` tegevus, mille koormaks on *API* kutsungi vastusest loetud teavituste massiiv. *API* kutsungi nurjumise järgselt saadetakse lattu `requestedPersonAllNotificationsFailure` tegevus, mille koormaks on *API* kutsungi vastuses olnud veateade (vt Lisa 4 `requestedPersonAllNotifications$` nimelist kõrvalmõju).

Tegevuse `requestedPersonAllNotificationsSuccess` kõrvalmõjuna teostatakse järjestikune *API* kutsung pärimaks tööülesandeid, mis on seotud eelmise *API* kutsungi tulemusena päritud teavitustega. *API* kutsungi teostamiseks on kaardistatud teavituste mudelite põhjal tööülesannete identifikaatorite massiiv ja seejärel edastatud sisendina *API* kutsungi meetodisse. Eduka *API* kutsungi järgselt saadetakse lattu `requestedPersonAllNotificationsTaskSuccess` tegevus, mille koormaks on *API* kutsungi vastusest loetud tööülesannete massiiv. *API* kutsungi nurjumise järgselt saadetakse lattu `requestedPersonAllNotificationsTasksFailure` tegevus, mille koormaks on *API* kutsungi vastuses olnud veateade (vt Lisa 4 `requestedPersonAllNotificationsSuccess$` nimelist kõrvalmõju).

Tegevuste `startedPollingForNewNotifications`, `receivedNewNotificationSuccess` ja `receivedNewNotificationFailure` kõrvalmõju eesmärgiks on intervalli tagant API kutsungi teostamine pärimaks kasutaja uusi teavitusi. Kõrvalmõju käivitamise alguses oodatakse 30000 millisekundit. Seejärel valitakse laost sisse loginud kasutaja identifikaator ning edastatakse uute teavituste *API* kutsungi meetodisse parameetrina. Eduka *API* kutsungi järgselt saadetakse lattu `receivedNewNotificationsSuccess` tegevus, mille koormaks on *API* kutsungi vastusest loetud teavituste massiiv. *API* kutsungi nurjumise järgselt saadetakse lattu `receivedNotificationsFailure` tegevus, mille koormaks on *API* kutsungi vastuses olnud veateade. Mõlemad tegevused käivitavad kõrvalmõju uuesti ning kõik kordub nii kaua kuni kasutaja on sisse loginud. Kui saabub tegevus, mis tähistab kasutaja välja logimist, siis tagastatakse tühi tegevus ning kõrvalmõju käivitamise tsükkel katkeb (vt Lisa 4 `startedPollingForNewNotifications$` nimelist kõrvalmõju).

Tegevuste `markedAllAsRead` ja `markedAllAsReadInHeader` kõrvalmõjuna teostatakse *API* kutsungid märkimaks kasutaja kõikide teavituste `isRead` väli tõeseks. Laost küsitakse kasutaja kõik teavitused ning igal teavitusel märgitakse väli `isRead` väli tõeseks ning seejärel teostatakse teavituse salvestamise *API* kutsung. Kuna *API* kutsung teostatakse iga teavituse kohta eraldi, siis on *API* kutsungite vastuste koondamiseks massiivi kasutatud `forkJoin` operaatorit. Kui kõik teavitused on edukalt salvestatud, siis saadetakse lattu `markedAllAsReadSuccess` tegevus, mille koormaks on uuendatud teavituste massiiv. Kui teavituste salvestamine ebaõnnestub, siis saadetakse lattu `markedAllAsReadFailure` tegevus, mille koormaks on veateade (vt Lisa 4 `markedAllAsRead$` nimelist kõrvalmõju).

Tegevuste `markedAsRead` ja `markedAsReadInHeader` kõrvalmõjuna teostatakse *API* kutsung märkimaks teavituse `isRead` väli tõeseks. *API* kutsungi teostamiseks küsitakse laost kõik teavitused ning valitakse seejärel identifikaatori alusel vastav teavituse mudel ning märgitakse sellel `isRead` väli tõeseks. Seejärel teostatakse kirje salvestamise *API* kutsung. Kui kutsung õnnestus, saadetakse lattu `markedAsReadSuccess` tegevus, mille koormaks on uuendatud teavitus. Kui kutsung ebaõnnestus, saadetakse lattu `markedAsReadFailure` tegevus, mille koormaks on veateade (vt Lisa 4 `markedAsRead$` nimelist kõrvalmõju).

4.6.7 Vaatemudelite kaardistamine

Mida kasutajale ekraanil kuvatakse ei ole mudel, vaid tegemist on vaatemudeliga. Mõnikord vaatemudel ja serverist päritud mudel ühtivad - kuid enamasti on tegemist kahe erineva mudeliga [22]. Vaatemudelisse kaardistatakse kokku andmed erinevatest, kuid omavahel seotud mudelitest, mis on vajalikud kasutajale kuvamiseks. Järelliide *VM* tähistab *ViewModel* ehk vaatemudelit.

Teavituse vaatemudeli kaardistamine

Serverist päritud teavituse mudel ei sisaldanud kogu teavet, mida on kasutajale tarvis teavituse kohta kuvada. Seetõttu loodi `NotificationVM` nimeline vaatemudel, kuhu kogutakse kuvamiseks vajalikud andmed kokku nii teavituse kui ka sellega seotud tööülesande mudelist. Kuna teavituste kuvamise alused on samad nii teavituste haldamise lehel ja päises asuvatel teavitusel, siis antud `NotificationVM` massiivi hakatakse

kasutama tüübina teavituse haldamise lehe ja teavituse päise vaatemudelites tähistamaks kuvatavaid teavitusi.

Tulenevalt üldisest nõudest, mille kohaselt peab saama teavitust märkida loetuks, lisatakse teavituse vaatemudelisse väli id ehk vastava teavituse mudeli kirje identifikaator. Teavituse loetuks märkimisel päritakse olekust identifikaatori alusel vastav mudel, mis on sisendiks *API* kutsungi meetodile märkimaks teavitus loetuks.

Tulenevalt üldisest nõudest, mille kohaselt kuvatakse kasutajale teavituse pealkiri vastavalt teavituse tüübile ja teavitusega seotud olemitele, lisatakse vaatemudelisse väli `title` ehk teavituse pealkiri.

Tulenevalt üldisest nõudest, mille kohaselt kuvatakse kasutajale teavituse aeg kahanevas järjestuses, lisatakse vaatemudelisse väli `createdTime` ehk vastava teavituse loomise aeg. Lisaks on nõutud `createdTime` väli teavituste haldamise lehel teavituste järjestamiseks aja alusel.

Tulenevalt üldisest nõudest, mille kohaselt lugemata teavitused kuvatakse hallina, lisatakse vaatemudelisse väli `isRead`. Lisaks on nõutud teavituste haldamise lehel teavituste filtreerimist antud tunnuse alusel.

Tulenevalt üldisest nõudest, mille kohaselt teavitusele vajutades kasutaja navigeeritakse vastavasse vaatesse, on lisatud väli `routerLink` ehk asukoht, kuhu kasutaja navigeeritakse teavitusele vajutamisel.

Tulenevalt üldisest nõudest, mille kohaselt ikoon kuvatakse vastavalt teavituse tüübile, on lisatud väli `icon`. Väli sisaldab teavituse ikooni koodi, mis on pärit *Material Icons* leheküljelt.

Tulenevalt teavituse haldamise lehe nõudest, mille kohaselt teavitusi saab filtreerida tüübi alusel, on lisatud väli `type` ehk teavituse liik.

```
export interface NotificationVM {
 id: string;
 title: string
 createdTime: string;
 isRead: boolean;
 routerLink: string[];
 icon: string;
 type: string;
}
```

Joonis 17. Teavituse vaatemudel

Teavituse haldamise lehe vaatemudeli kaardistamine

Teavituste haldamise lehe nõuete kohaselt peab kasutajale kuvama tabelis teavitusi, mida on võimalik ajaliselt järjestada ja filtreerida. Kuvatavaid teavitusi tähistab vaatemudelis `notifications` nimeline väli. Järjestamise ja filtreerimise olekut laos ei hoiustata ja seetõttu vaatemudelisse vastavaid väljasid ka ei lisata.

Lisaks on nõuete kohaselt teavituse haldamise lehel nõutud kasutaja teavituste seadistamise funktsionaalsuse olemasolu ning selle jaoks on lisatud vaatemudelisse `notificationSettings` nimeline väli.

```
export interface NotificationsPageViewModel {
  notifications: NotificationVM[];
  notificationSettings: NotificationSetting[];
}
```

Joonis 18. Teavituste haldamise lehe vaatemudel

Päises asuvate teavituste komponendi vaatemudeli kaardistamine

Päises kuvatavate teavituste nõuete kohaselt on kasutajale vaja kuvada 10 viimast teavitust ja lugemata teavituste arv. Teavituste lisamiseks vaatemudelisse lisatakse `headerNotifications` nimeline väli ning lugemata teavituste arvu jaoks lisatakse `notReadNotificationsLength` väli. Ülejäänud päise teavitustega seotud nõuded on realiseeritud täielikult esitluskomponentides ning neid vaatemudelisse ei lisata.

```
export interface HeaderNotificationsViewModel {
  headerNotifications: NotificationVM[];
  notReadNotificationsLength: number;
}
```

Joonis 19. Teavituste päise komponendi vaatemudel

4.6.8 Olekute valimine

Järgnevalt kirjeldatakse vaatemudelite ja olekute valimist selektorite abil. Selektorite realiseerimise käigus loodud kood asub Lisa 6.

Teavituste oleku valimine

Teavituste oleku valimiseks juurolekust luuakse `createFeatureSelector` funktsiooni abil vastav selektor. Funktsioonile edastatakse parameetrina vastav võti, millega tunnus on juur lao moodulis registreeritud (vt Lisa 6 `selectNotificationsState` nimelist selektorit).

Teavituste kollektsiooni kõikide kirjete valimine

Teavituste kollektsiooni kõikide kirjete ehk sisse loginud kasutaja kõikide teavituste valimiseks teavituste olekust on loodud `getSelectors` meetodi abil `selectAll` nimeline selektor. Meetodi sisendiks on teavituste oleku selektor (vt Lisa 6 `selectAll` nimelist selektorit).

Teavituste vaatemudeli valimine

Teavituste vaatemudelite valimiseks valitakse olekust kõik teavitused ning tööülesanded. Antud töös on vajalik teavituste pealkirjas kuvada teavitusega seotud tööülesande nimetust. Selle jaoks kaardistatakse kõik valitud teavituste mudelid vaatemudeliteks kasutades selleks `map` funktsiooni. Iga teavituse mudeli kohta otsitakse selles sisalduva tööülesande identitifikaatori alusel vastav tööülesande mudel ja edastatakse mõlemad `getTitle` funktsiooni. Antud funktsioon tagastab seejärel teavituse pealkirja, mis sisaldab tööülesande nimetust. Väljade `routerLink` ja `icon` väärtustamiseks on samuti loodud vastavad funktsioonid, mis võtavad sisendiks teavituse mudeli ja tagastavad vastavad väärtused (vt Lisa 6 `selectNotificationsVMs` nimelist selektorit).

Teavituste haldamise lehe vaatemudeli valimine

Teavituste haldamise lehe vaatemudeli valimiseks valiti teavituste vaatemudelid ja kõik teavituste sätted ning tagastati need kombineeritult üheks objektiks (vt Lisa 6 `selectNotificationsPageViewModel` nimelist selektorit).

Päise teavituste vaate vaatemudeli valimine

Päise teavituste vaate vaatemudel sisaldas viimast kümmet teavitust ning lugemata teavituste arvu. Viimase kümne teavituse saamiseks rakendati teavituste vaatemudelite massiivile `splice` funktsiooni, mille sisendiks olid algus- ja lõppindeks. Antud funktsioon lõikas massiivist välja need kirjed, mis ei mahtunud indeksite vahemikku ning kuna massiiv on juba järjestatud kahanevas järjestuses, siis on tulemuseks maksimaalselt 10 viimast teavitust.

Lugemata teavituste arvu jaoks loodi muutuja ning selle väärtuse saamiseks filtreeriti teavitustest välja teavitused, mille `isRead` välja väärtus on väär ja võeti seejärel tulemusena saadud massiivi pikkus. Seejärel tagastati saadud tulemused kombineeritult üheks objektiks (vt Lisa 6 `selectHeaderNotificationsViewModel` nimelist selektorit).

4.6.9 Angular komponendid

Iga Angulari komponent koosneb *HTML* mallist, mis deklareerib lehe sisu ja *TypeScript* klassist, mis defineerib käitumise. Komponendid jagatakse kahte kategooriasse: konteinerkomponendid ja esitluskomponendid. Esitluskomponentide eesmärk on andmete kuvamine ja kasutajalt sisendi võtmine. Konteinerkomponent või vaatekomponent sisaldab koosneb esitluskomponentidest ja selle eesmärk on keskenduda sellele, kuidas rakendus töötab. Konteinerkomponentidesse on sõltuvusena süstitud ladu, mida kasutatakse oleku valimiseks ja edastamiseks esitluskomponentidesse. Samuti toimub konteinerkomponentidest tegevuste saatmine lattu. Päise teavituste konteinerkomponendi kood asub Lisa 1 ja teavituste haldamise lehe konterinerkomponendi kood asub Lisa 2.

4.7 Hinnang tulemusele

Käesolevas peatükis kirjeldati kasutuselevõetud *NgRx* raamistiku teekisid, oleku haldamise printsiipe ja koodistiili. Lisaks kirjeldati parimad praktikad *NgRx* raamistiku rakendamiseks. Ühe parima praktika kohaselt loodi juur olek, milles seotakse kokku rakenduse tunnuste olekud.

Rakenduse üheks tunnuseks oli teavituste funktsionaalsuse olemasolu. Arendati analüütiku nõuete kohaselt teavituste funktsionaalsus ja oleku haldamine.

NgRx kasutamise tulemusena saavutati ühesuunaline andmevoog, mis võimaldas saada oleku muutumisest hea ülevaate ning võimaldades seeläbi lihtsamat koodi silumist. Samuti standardiseeritud kood võimaldas arendada keerulisi funktsionaalsuseid kergema vaevaga, kuid nõudis rohkem koodi kirjutamist.

Arendatud teavituste funktsionaalsuse eesmärk on vähendada ravimiameti töötajal erinevate asjade meeles hoidmist ning seeläbi vähendada inimvea tekkimise tõenäosust. Kuna lõputöö kirjutamisel hetkel on rakendus veel arendamisjärgus ning Ravimiamet rakendust veel ei kasuta, siis antud funktsionaalsuse kasutegurit mõõta ei olnud võimalik.

5 Kokkuvõte

Selle töö eesmärgiks oli uurida, valida ja juurutada Ravimiameti üheleherakenduse arendamiseks sobiv oleku haldamise lahendus ning seejärel valitud lahendust rakendada teavituste funktsionaalsuse arendamisel.

Eesmärgi saavutamiseks valiti välja kolm oleku haldamise lahendust: *Akita*, *NgRx* ja *NGXS*. Kirjeldati lahenduste peamiseid kontseptsioon, uuriti lahenduste kogukonda, küpsust, koodi hallatavust, koodi kirjutamiseks kuluvat aega ning kõrvalmõjude isoleerimise võimalusi.

Uurimuse ja võrdluse tulemusena valiti oleku haldamise lahenduseks *NgRx* raamistik, kuna antud raamistik oli võrreldes alternatiividega kõige küpsem, suurema ja aktiivsema kogukonnaga, ei nõudnud kolmanda osapoolte teekide kasutamist kõrvalmõjude käitlemiseks ning võimaldas kõige paremat koodi hallatavuse ja selle kirjutamiseks kuluva aja suhet.

Valitud oleku haldamise lahenduse juurutamiseks kirjeldati parimad praktikad ning lähtuti juurutamisel nendest. Lisaks rakendati lahendust teavituste funktsionaalsuse arendamiseks. Teavituste funktsionaalsuse arendamisel lähtuti süsteemianalüütiku poolt koostatud nõuetest.

Kasutatud kirjandus

[1] A. R. Khalid, „What is State-Management and Why You Should Learn It?,“ DEV Community, 3 September 2020. [Võrgumaterjal]. Available: https://dev.to/abdurrkhalid333/what-is-state-management-and-why-you-should-learn-it-3kai. [Kasutatud 2022].

[2] 2muchcoffee, „State Management: don't shoot yourself in the foot before you start an Angular application,“ Medium, 14 November 2019. [Võrgumaterjal]. Available: https://medium.com/@2muchcoffee/angular-state-management-a-must-have-for-large-scale-angular-apps-8b98e5a761c7. [Kasutatud 2022].

[3] „NgRx documentation: Why store,“ [Võrgumaterjal]. Available: https://ngrx.io/guide/store/why. [Kasutatud 2022].

[4] A.University, „Angular NgRx Store and Redux - When to use a Store and Why?,“ Angular University, 21 November 2016. [Võrgumaterjal]. Available: https://blog.angular-university.io/angular-2-redux-ngrx-rxjs/. [Kasutatud 2022].

[5] „Introduction to client-side frameworks - Learn web development,“ MDN, [Võrgumaterjal]. Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction. [Kasutatud 2022].

[6] Z. Khalifa, „How to choose your development framework and libraries?,“ Dev Genius, 2020 Juuni 27. [Võrgumaterjal]. Available: https://blog.devgenius.io/how-to-choose-your-development-framework-and-libraries-d36230960342. [Kasutatud 2022].

[7] „How to choose a framework,“ [Võrgumaterjal]. Available: https://hackernoon.com/how-to-choose-a-framework-ea8b5b1e1f44. [Kasutatud 2022].

[8] F.Cheng, %1 *Build Mobile Apps with Ionic 4 and Firebase: Hybrid Mobile App Development*.

[9] „NgRx documentation: Getting Started," [Võrgumaterjal]. Available: https://ngrx.io/guide/store. [Kasutatud 2022].

[10] O. D. Smet, „NGRX vs. NGXS vs. Akita vs. RxJS: Fight!," Ordina JWorks Tech Blog, [Võrgumaterjal]. Available: https://ordina-jworks.github.io/angular/2018/10/08/angular-state-management-comparison.html. [Kasutatud 2022].

[11] Dženopoljac, „State Management in Angular Applications – NGXS vs. NGRX," Serengeti, 16 November 2021. [Võrgumaterjal]. Available: https://serengetitech.com/tech/state-management-in-angular-applications-ngxs-vs-ngrx/. [Kasutatud 2022].

[12] Poveda, „Choosing A State Management Library for Angular Enterprise Applications," Perficient Blogs, 21 Juuli 2020. [Võrgumaterjal]. Available: https://blogs.perficient.com/2020/07/21/choosing-a-state-management-library-for-angular-enterprise-applications/. [Kasutatud 2022].

[13] „NGXS documentation: Introduction," [Võrgumaterjal]. Available: https://www.ngxs.io/concepts/intro. [Kasutatud 2020].

[14] „NGXS documentation: Action Handlers," [Võrgumaterjal]. Available: https://www.ngxs.io/advanced/action-handlers. [Kasutatud 2022].

[15] Udagepala, „Angular: State Management with Akita," Medium, 02 Mai 2020. [Võrgumaterjal]. Available: https://medium.com/swlh/angular-state-management-with-akita-b4c5439c1ab5. [Kasutatud 2022].

[16] F. Y ja M. A, %1 *Angular Development with TypeScript*.

[17] N. Singh, „Functional Reactive Programming(FRP) — Imperative vs Declarative vs Reactive style," [Võrgumaterjal]. Available: https://www.linkedin.com/pulse/functional-reactive-programmingfrp-imperative-vs-style-navdeep-singh. [Kasutatud 2022].

[18] J. Eames, „Why is Functional Reactive Programming (FRP) Showing Up in React and Angular," CODE Magazine, 2021. [Võrgumaterjal]. Available: https://www.codemag.com/article/1601071/How-Functional-Reactive-Programming-FRP-is-Changing-the-Face-of-Web-Development. [Kasutatud 2022].

[19] W. Grimes, „NgRx — Best Practices for Enterprise Angular Applications,“ 30 Mai 2018. [Võrgumaterjal]. Available: https://wesleygrimes.com/angular/2018/05/30/ngrx-best-practices-for-enterprise-angular-applications. [Kasutatud 2022].

[20] „NgRx documentation: Actions,“ [Võrgumaterjal]. Available: https://ngrx.io/guide/store/actions. [Kasutatud 2022].

[21] B. Roberts, „Maximizing and Simplifying Component Views with NgRx Selectors,“ [Võrgumaterjal]. Available: https://brandonroberts.dev/blog/posts/2020-12-14-maximizing-simplifying-component-views-ngrx-selectors/. [Kasutatud 2022].

[22] A. University, „Ngrx Store - An Architecture Guide,“ Angular University, 7 Märts 2017. [Võrgumaterjal]. Available: https://blog.angular-university.io/angular-ngrx-store-and-effects-crash-course/. [Kasutatud 2022].

[23] L. S. Sterling, The Art of Agent-Oriented Modeling, London: The MIT Press, 2009.

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Kristo Naeris

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Oleku haldamine Ravimiameti üheleherakenduse näitel" , mille juhendaja on Jaanus Pöial.
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

08.05.2022

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 – Teavituste päise konteinerkomponendi kood

```
@Component({
 selector: '<app-header-notifications *ngIf="headerNotificationsViewModel$ | async as vm"
[headerNotifications]="vm.headerNotifications"
[notReadNotificationsLength]="vm.notReadNotificationsLength"
(markAsRead)="onMarkedNotificationAsRead($event)"
(markAllAsRead)="onMarkedAllNotificationsAsRead()">
</app-header-notifications>'
})
export class HeaderNotificationsContainer {

 headerNotificationsViewModel$: Observable<HeaderNotificationsViewModel> =
this.store$.select(HeaderNotificationsSelectors.selectHeaderNotificationsViewModel);

 constructor(private store$: Store<RootStoreState.RootState>) {
   this.store$.dispatch(HeaderNotificationsActions.startedPollingForNewNotifications());
   this.store$.dispatch(HeaderNotificationsActions.requestedPersonAllNotifications());
}


 onMarkedNotificationAsRead(id: string) {
   this.store$.dispatch(HeaderNotificationsActions.markedAsReadInHeader({payload: id}))
 }


 onMarkedAllNotificationsAsRead() {
   this.store$.dispatch(HeaderNotificationsActions.markedAllAsReadInHeader());
 }
}
```

Lisa 3 – Teavituste haldamise lehe konteinerkomponendi kood

```
Component({
 template: '<app-notifications-page
*ngIf="notificationsPageViewModel$ | async as vm"
[notifications]="vm.notifications"
[notificationSettings]="vm.notificationSettings"
(markAllRead)="onMarkedAllNotificaitonsAsRead()"
(markRead)="onMarkedNotificationAsRead($event)"
(savedNotificationSettings)="onSavedNotificationSettings($event)">
</app-notifications-page>'
})
export class NotificationsPageContainer {

 notificationsPageViewModel$: Observable<NotificationsPageViewModel> =
this.store$.select(NotificationsPageSelectors.selectNotificationsPageViewModel;

 constructor(private store$: Store<RootStoreState.RootState>) {
   this.store$.dispatch(NotificationsPageActions.requestedPersonAllNotificationsSettings());
 }

 onMarkedAllNotificationsAsRead() {
   this.store$.dispatch(NotificationsPageActions.markedAllAsRead());
 }

 onMarkedNotificationAsRead(id: string){
   this.store$.dispatch(NotificationsPageActions.markedAsRead({payload: id}));
 }

 onSavedNotificationSettings(notificationSettings: NotificationSetting[]){
   this.store$.dispatch(NotificationsPageActions.requestedSaveNotificationSettings({payload:
notificationSettings}));
 }
}
```

Lisa 4 – Tegevuste kood

```
export const requestedPersonAllNotificationsSuccess =
createAction(
  '[Notifications API] requested person all notifications
success',
  props<{payload: Notification[]}>()
);

export const requestedPersonAllNotificationsFailure =
createAction(
  '[Notifications API] requested person all notifications
failure',
  props<{error: Error}>()
);

export const receivedNewNotificationsSuccess = createAction(
  '[Notifications API] received new notification success',
  props<{payload: Notification[]}>()
);

export const receivedNewNotificationsFailure = createAction(
  '[Notifications API] received new notification failure
);

export const markedAllAsReadSuccess = createAction(
  '[Notifications API] marked all notifications as read
success',
  props<{payload: Notification[]}>()
);

export const markedAllAsReadFailure = createAction(
  '[Notifications API] marked all notifications as read
failure',
  props<{error: Error}>()
);

export const markedAsReadSuccess = createAction(
  '[Notification API] marked notification as read success',
  props<{payload: Notification}>()
);

export const markedAsReadFailure = createAction(
  '[Notification API] marked notification as read failure',
  props<{error: Error}>()
);
```

```
export const startedPollingForNewNotifications = createAction(
  '[Header Notifications] started polling for new notifications'
);

export const requestedPersonAllNotifications = createAction(
  '[Header Notifications] requested person all notifications'
);

export const markedAllAsReadInHeader = createAction(
  '[Header Notifications] marked all notifications as read'
);

export const markedAsReadInHeader = createAction(
  '[Header Notifications] marked notification as read',
  props<{payload: string}>()
);

export const markedAllAsRead = createAction(
  '[Notifications Page] marked all notifications as read'
);

export const markedAsRead = createAction(
  '[Notifications Page] marked notification as read',
  props<{payload: string}>()
);

export const requestedNotificationsTasksSuccess = createAction(
  '[Tasks API] requested notifications tasks success',
  props<{payload: Task[]}>()
);

export const requestedNotificationsTasksFailure = createAction(
  '[Tasks API] requested notifications tasks failure',
  props<{error: Error}>()
);
```

Lisa 5 – Kõrvalmõjude kood

```
requestedPersonAllNotifications$ = createEffect(() => {
     return this.actions$.pipe(
       ofType(HeaderNotificationsActions.requestedPersonAllNotifications),
withLatestFrom(this.store$.select(AuthStoreSelectors.selectLoggedInPersonId)),
       mergeMap(([action, personId]) => {
         return this.notificationsService.getNotifications({personId} as Notification)
             .pipe(map((data: Notification[]) => NotificationsAPIActions.requestedPersonAllNotificationsSuccess({payload:
data})),
               catchError(error => of(NotificationsAPIActions.requestedPersonAllNotificationsFailure({error}))))
         }
       )
     );
   });
```

```
 requestedPersonAllNotificationsSuccess$ = createEffect(() => {
    return this.actions$.pipe(
      ofType(requestedPersonAllNotificationsSuccess),
      mergeMap(action => {
        const tasksIds: string[] = action.payload?.map(notification => notification.taskId).filter(id => !!id);
        return this.tasksService.getTasksByIds(tasksIds).pipe(
          map((data: Task[]) => requestedPersonAllNotificationsTasksSuccess({payload: data})),
          catchError(error => of(requestedPersonAllNotificationsTasksFailure({error})))
        )
      })
    )
  });
```

```
 startedPollingForNewNotifications$ = createEffect(() => {
   return this.actions$.pipe(
     ofType(HeaderNotificationsActions.startedPollingForNewNotifications,
           NotificationsAPIActions.receivedNewNotificationsSuccess,
NotificationsAPIActions.receivedNewNotificationsFailure),
     delay(30000),
     withLatestFrom(this.store$.select(AuthStoreSelectors.selectLoggedInPersonId)),
     mergeMap(([action, personId]) => {
       return this.notificationsService.getNotifications({personId} as Notification).pipe(
         map((response: Notification[]) => NotificationsAPIActions.receivedNewNotificationsSuccess({payload :response})),
         catchError(error => of(NotificationsAPIActions.receivedNewNotificationsFailure()))
       )
     })
   )
 })
```

```
markedAllAsRead$ = createEffect(() => {
   return this.actions$.pipe(
     ofType(HeaderNotificationsActions.markedAllAsReadInHeader, NotificationsPageActions.markedAllAsRead),
     withLatestFrom(this.store$.select(NotificationsStoreSelectors.selectAll)),
     mergeMap(([action, notifications]) => {
       return forkJoin(notifications.map(notification => this.notificationsService.saveNotification({...notification, isRead:
true}))).pipe(
         map(notifications => NotificationsAPIActions.markedAllAsReadSuccess({payload: notifications})),
         catchError(error => of(NotificationsAPIActions.markedAllAsReadFailure({error})))
       )
     })
   )
 });
```

```
markedAsRead$ = createEffect(() => {
   return this.actions$.pipe(
     ofType(HeaderNotificationsActions.markedAsReadInHeader, NotificationsPageActions.markedAsRead),
withLatestFrom(this.store$.select(NotificationsStoreSelectors.selectEntities)),
     mergeMap(([action, notifications]) => {
       const notification: Notification = notifications[action.id];
       return this.notificationsService.saveNotification({...notification, isRead: true}).pipe(
         map(notification => NotificationsAPIActions.markedAsReadSuccess({payload: notification})),
         catchError(error => of(NotificationsAPIActions.markedAsReadFailure({error})))
       )
     })
   )
 });
```

Lisa 6 – Selektorite kood

```
export const selectNotificationsState =
createFeatureSelector<NotificationsState>(
  'notifications'
);
```

```
export const {
  selectAll
} = notificationsAdapter.getSelectors(selectNotificationsState);
```

```
export const selectNotificationsVMs = createSelector(
  selectAll,
  TasksStoreSelectors.selectEntities,
  (allNotifications: Notification[], allTasks: Dictionary<Task>) => {
    const notificationVMs: NotificationVM[] = allNotifications
      .map(notification => {
        return {...notification,
          title: getTitle(notification,
            allTasks[notification.taskId]),
          routerLink: getRouterLink(notification),
          icon: getIcon(notification)
        } as NotificationVM;
      });
    return notificationVMs;
  }
);
```

```
export const selectNotificationsPageViewModel = createSelector(
  selectNotificationsVMs,
  NotificationSettingStoreSelectors.selectAll,
  (notifications: NotificationVM[], notificationSettings:
NotificationSetting[]) => {
       return {notifications, notificationSettings} as
NotificationsPageViewModel
  }
);
```

```
export const selectHeaderNotificationsViewModel = createSelector(
 NotificationsStoreSelectors.selectNotificationsVMs,
  (notifications: NotificationVM[]) => {
   const headerNotifications = notifications.splice(0, 10);
   const notReadNotificationsLength = notifications.filter(notification =>
notification.isRead == false).length;
    return {headerNotifications, notReadNotificationsLength}
  });
```