Flutter-sovelluksen tilanhallintaratkaisujen vertailu ja sovelluksen jatkokehitys

Case Järvisuunnistus

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutus
kevät, 2023

Waltteri Grek

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutus Tiivistelmä

Tekijä Waltteri Grek Vuosi 2023

Työn nimi Flutter-sovelluksen tilanhallintaratkaisujen vertailu ja sovelluksen jatkokehitys

Ohjaaja Tommi Lahti

TIIVISTELMÄ

Opinnäytetyön aiheena on Flutter-sovelluksen tilanhallintaratkaisujen vertailu sekä tilanhallintaratkaisun toteutus Järvisuunnistus-sovellukseen. Työn tavoitteena oli ensin tutkia erilaisia Flutter-kehyksen tilanhallintaratkaisuja ja saatujen tietojen perusteella vertailla niitä keskenään, jotta lopuksi voidaan valita Järvisuunnistus-sovellukseen paras mahdollinen ratkaisu. Opinnäytetyön toimeksiantaja oli HAMK Smart.

Opinnäytetyön teoriaosassa käsitellään Flutter-kehykseen, tilanhallintaratkaisuihin sekä itse Järvisuunnistus-sovellukseen liittyviä käsitteitä, rakenteita ja isompia asiayhteyksiä. Tietopohja koostuu Flutter-kehyksen ja tilanhallintaratkaisujen osalta lähes täysin niiden omista dokumentaatioista sekä verkkosivuista. Opinnäytetyö on samaan aikaan sekä tutkimuksellinen että toiminnallinen.

Tutkimuksessa havaittiin, että on olemassa paljon erilaisia tilanhallintaratkaisuja, joista jokainen on hieman erilainen. Järvisuunnistus-sovellusta varten Riverpod-tilanhallintaratkaisu osoittautui parhaaksi vaihtoehdoksi, joten kyseinen ratkaisu valittiin ja toteutettiin sovellukseen. Toimeksiantaja oli tyytyväinen tuloksiin ja opinnäytetyön alussa asetetut tavoitteet saavutettiin onnistuneesti.

Avainsanat Flutter, tilanhallinta, Riverpod, http-pyyntö, state

Sivut 31 sivua ja liitteitä 1 sivu

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology Abstract

Author Waltteri Grek Year 2023

Subject Comparing state management solutions and further development of a Flutter app

Supervisor Tommi Lahti

ABSTRACT

The purpose of the thesis was to explore different kinds of state management solutions and then to select and implement the most suitable solution for this app. This thesis was commissioned by HAMK Smart.

The theory part of the thesis explores the concepts and broad contexts of Flutter, its state management solutions and the app itself. The sources of Flutter and state management solutions are almost completely from their own documentations and websites. The thesis is both practical and theoretical.

The analysis indicates that there are multiple state management solutions available, each of them differing from each other. Riverpod turned out to be the most suitable solution for Järvisuunnistus, therefore it was selected and implemented to the app. The client was pleased with the result and the goals set in the beginning of the thesis were achieved.

Keywords Flutter, state management, Riverpod, http request, state

Pages 31 pages and appendices 1 pages

Sanasto

http-pyyntö	Hypertext Transfer Protocol, tiedonsiirto WWW-palvelimen ja asiakasohjelman välillä
tila, state	ohjelman tila, joka kerää välimuistiin tietoa esimerkiksi käyttöliittymästä tai käyttäjän tekemistä valinnoista
API	Application Programming Interface, rajapinta palvelimen ja asiakasohjelman välillä
REST	Representational state transfer, rajapintatyökalu
widget	Flutter-kehyksen työkalu, jolla sovelluksen käyttöliittymä rakennetaan

Sisällys

Kuvat, ohjelmakoodit ja taulukot

Liitteet

Liite 1 Aineistonhallintasuunnitelma

1 Johdanto

Hämeen ammattikorkeakoulu sekä Vanajavesikeskus ovat toteuttajia Suunta luontoon -hankkeessa, jonka ideana on tukea koronapandemiasta elpymistä Kanta-Hämeen alueella. Hankkeen ideana on kehittää digitaalisia ja vihreitä palveluja, joista yksi on Järvisuunnistus-mobiilisovellus. Järvisuunnistus on melontasuunnistussovellus, joka sisältää vesisuunnistuskohteita Kanta-Hämeen vesistöissä. Opinnäytetyö on tilattu Hämeen ammattikorkeakoulun tutkimusyksikön HAMK Smartin toimesta. Työn ideana on parantaa sovelluksen laatua ja käyttömukavuutta tilanhallintaratkaisun rakentamisen avulla.

Opinnäytetyö keskittyy erityisesti sovelluksen tietopyyntöjen lähettämiseen ja tallentamiseen. Tällä hetkellä sovellus lähettää lukuisia http-tietopyyntöjä aina uuteen sovellusikkunaan siirtyessä, ja tämä olisi tarkoitus korvata tilanhallintaratkaisulla siten, että suunnistuspaikkojen- ja reittien paikkatiedot haettaisiin jo sovelluksen käynnistyessä. Tämä ratkaisu parantaisi sovelluksen suoritusnopeutta ja täten parantaisi saavutettavuutta.

Hämeen ammattikorkeakoulun tutkimusyksikkö HAMK Smart haluaisi kehittää sovellusta tilanhallintaratkaisun avulla, joka mahdollistaisi useiden http-tietopyyntöjen lähettämisestä luopumisen ja täten nopeuttaisi sovelluksen toimintaa. Tämän opinnäytetyön teoriaosuudessa tutkitaan ja vertaillaan sopivinta tilanhallintaratkaisua kyseistä sovellusta varten ja käytännön osuudessa valittu ratkaisu kehitetään sovellukseen. Sovellus on toteutettu Flutter-kehyksellä, johon myös tutustutaan työn teoriaosuudessa.

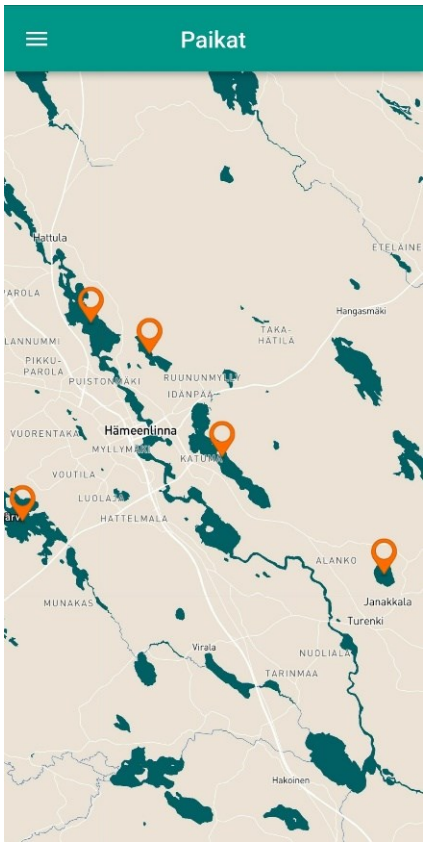
Tutkimuskysymykset ovat:

- Mitä tilanhallintaratkaisuja on tarjolla?
- Mikä tilanhallintaratkaisu sopii parhaiten tähän sovellukseen?
- Miten tilanhallintaratkaisu toteutetaan?

2 Järvisuunnistus-sovelluksen kuvaus

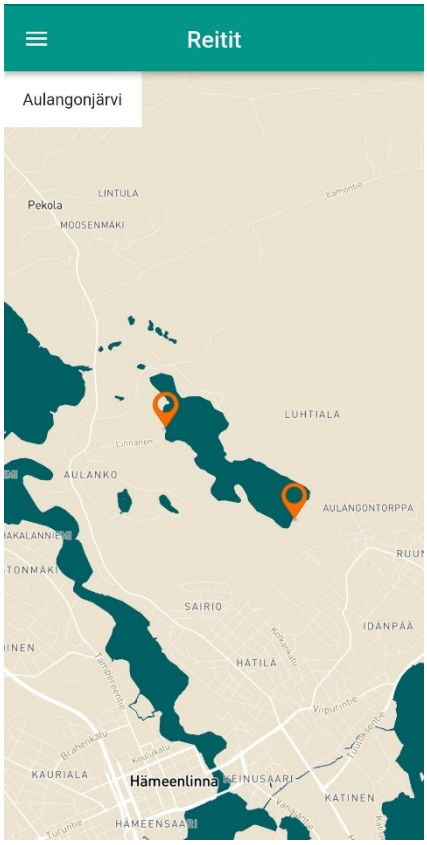
Järvisuunnistus on melontasuunnistukseen tarkoitettu mobiilikarttasovellus, jossa on 11 eri suunnistuskohdetta Kanta-Hämeen alueella. Sovelluksessa on muutama eri ikkuna, joista kahta käytetään pääasiallisesti suunnistuksessa. Kuvassa 1 näkyvä Paikat-ikkuna näyttää suunnistusreittejä sisältävät järvet.

Kuva 1 Sovelluksen Paikat-ikkuna



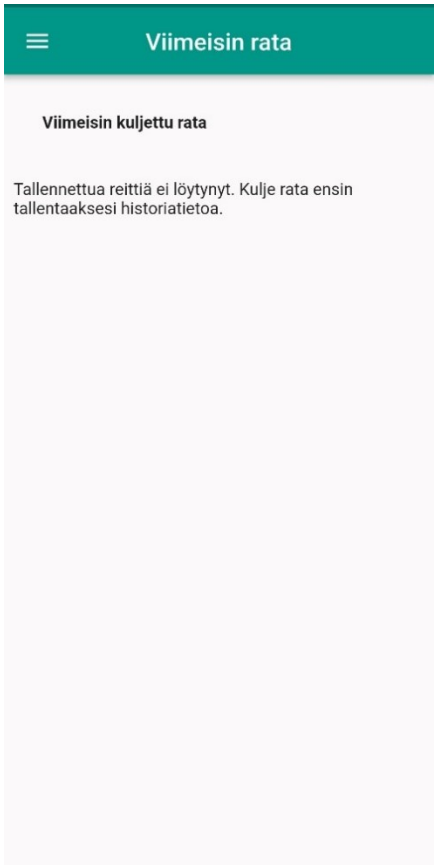
Järvien päällä olevia oransseja ikoneita painamalla aukeaa Reitit-ikkuna. Reitin aloituspaikan voi valita osalla järvistä tarjotuista vaihtoehdoista, jotka ilmenee Kuvassa 2 näkyvillä oransseilla ikoneilla.

Kuva 2 Sovelluksen Reitit-ikkuna, Aulangonjärvi valittuna



Suunnistus aloitetaan painamalla Aloita-painiketta, ja sovellus kerää rastin automaattisesti 20 metrin säteellä mobiililaitteen GPS-paikanninta käyttäen. Kun suunnistus halutaan lopettaa, painetaan Lopeta-painiketta. Mikäli rasteja on kerätty, reitti tallennetaan Kuvassa 3 näkyvälle Viimeisin rata -ikkunalle.

Kuva 3 Sovelluksen Viimeisin rata -ikkuna



2.1 REST

REST (tunnetaan myös nimellä RESTful API) on Roy Fieldingin kehittämä rajapinta, joka välittää pyyntöjä asiakasohjelmalta palvelimelle ja palauttaa tietyn resurssin tilan takaisin asiakasohjelmalle. Tieto välitetään jonain http:n tyyppinä: JSON-, HTML-, XLT-, Python-, PHP- tai tekstimuotona. Näistä vaihtoehdoista JSON-muoto on käytetyin sen luettavuuden takia. (Red Hat, 2020.)

Sovelluksessa näkyvien paikkojen, reittien ja rastien koordinaatit sijaitsevat Azuren palvelimella, ja nämä tiedot haetaan sovellukseen http-pyynnöillä REST-rajapinnan kautta. Rajapinnan visualisointia ja ylläpitoa varten on otettu käyttöön käyttöliittymätyökalu Swagger. HTTP-metodit, joita REST-rajapinnan kanssa käytetään ovat GET tiedon hakua varten, PUT esimerkiksi tiedon päivittämistä varten, POST tiedon luomista varten sekä DELETE tiedon poistoa varten (Gillis, n.d.). Järvisuunnistus-sovelluksessa ja tässä opinnäytetyössä oleellisin metodi on GET, jolla haetaan koordinaatit Azuren palvelimelta.

2.2 Flutter

Järvisuunnistus on kehitetty käyttäen Flutter-kehystä, jonka pääasiallisena kielenä toimii Dart. Flutter on Googlen kehittämä avoimen lähdekoodin kehysympäristö mobiilisovellusten rakentamista varten iOS- sekä Android-järjestelmiin. (Google Developers, 2018.) Kehysympäristön voi asentaa Windows-, macOS- sekä Linux-järjestelmille, joille kirjoitushetkellä uusin versio 3.7.0 ilmeistyi 24.1.2023 (Flutter, n.d.-a).

Dart on avoimen lähdekoodin dynaaminen, olio-orientoitunut ja luokkapohjainen ohjelmointikieli ja se on suunniteltu helposti käyttöönotettavaksi ohjelmoijille, joille esimerkiksi C#- ja Java-ohjelmointikielet ovat jo tuttuja. (Google Developers, 2012.)

Flutter-kehyksellä rakennettujen sovellusten käyttöliittymä muodostuu widgeteistä. Widget kertoo ohjelmalle ohjeistuksen, miltä käyttöliittymän pitäisi näyttää. Sen tyyppejä on esimerkiksi Text tekstiä varten, Row horisontaalista asettelua varten tai Container suorakulmaisen elementin luomiseksi. Widget voi olla tilaa hallitsematon StatelessWidget tai tilaa hallitseva StatefulWidget. State ja StatefulWidget eivät kuitenkaan ole Flutterissa sama asia, sillä widget on väliaikainen objekti ja state on pysyvä objekti, joka pystyy muistamaan tietoa. (Flutter, n.d.-b.)

2.3 Olemassaolevan sovelluksen kuvaus

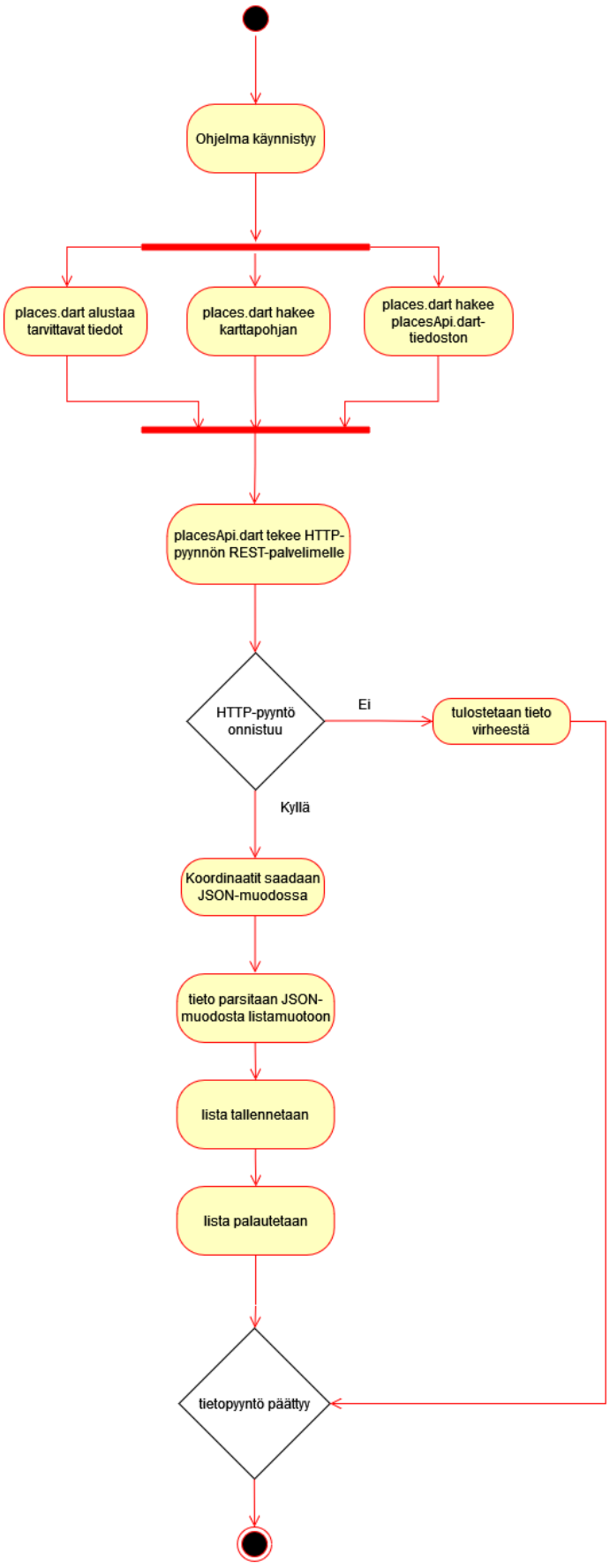
Sovelluksen projektitiedostossa on kolme kansiota, joiden sisältö tulee vaikuttamaan tilanhallintaratkaisun rakentamiseen. Sovelluksen karttanäkymän käyttöliittymästä vastaa kolme dart-tiedostoa pages-kansion alla. Kullekin ikkunalle on oma tiedostonsa, joiden sisällä on aina vähintään yksi widget. Tietoyhteyttä varten on kolme dart-tiedostoa controllers-kansion alla, yksi tiedosto per näkymä. Paikkojen, reittien ja rastien tietojen käsittelystä ja mallintamisesta vastaa tiedostot models-kansiossa. Kuva 4 sisältää havainnollistamisen aktiviteettikaavion avulla Paikat-näkymän tietopyyntöprosessista.

Controllers-kansiossa olevat API-tiedostot haetaan käyttöliittymästä vastaaviin tiedostoihin käyttäen GetX-liitännäistä. Esimerkiksi placesApi.dart-tiedosto saadaan käyttöön places.dart-tiedostossa luomalla siitä uusi instanssi Get.put -komennon avulla. GetX ei vaadi suuria määriä resursseja, joten se tarjoaa tehokkuutta keveytensä ansiosta (pub.dev, 2022-a). Paikkatietojen

haussa HTTP-tietopyyntö lähetetään palvelimelle placesApi.dart-tiedoston fetchAllPlaces-metodissa GET-pyynnön muodossa. Mikäli pyyntö onnistuu, parsitaan pyynnöstä saatu tieto JSON-muodosta listaan. Mikäli tieto saadaan parsittua ja tallennettua listaan, metodi palauttaa lopulta listan places.dart-tiedoston käytettäväksi. Mikäli HTTP-pyyntö tai tiedon parsiminen epäonnistuu, ohjelma tulostaa tiedon virheestä.

Models-kansiossa on muun muassa paikoille ja reiteille tarkoitetut tiedostot, joiden tehtävänä on toimia niihin kuuluvien attribuuttien mallina ja muuttaa http-pyynnöistä saatu tieto json-muodosta yksittäisiin attribuutteihin. Models-kansiossa olevat tiedostot toimivat siis samalla logiikalla kuin constructor-luokat Java-ohjelmoinnissa.

Kuva 4 Aktiviteettikaavio tietopyyntöprosessista



2.4 Tilanhallintaratkaisun tarve sovelluksessa

Jotta sovelluksen karttoihin saadaan oikeat suunnistuspaikat ja -reitit, täytyy niiden tieto (tässä tapauksessa koordinaatit) hakea palvelimelta. Joka kerta, kun Paikat- ja Reitit-ikkuna avataan, sovellus lähettää uuden http-pyynnön palvelimelle tietojen saamiseksi. Tiedot eivät siis jää talteen sovelluksen välimuistiin, vaikka samassa ikkunassa olisi jo käyty samalla käyttökerralla. Tämä voi hidastaa sovelluksessa siirtymistä ikkunoiden välillä.

Tässä opinnäytetyössä tilanhallintaratkaisulla pyritään vähentämään http-pyyntöjen määrää merkittävästi. Sovelluksen käynnistyessä tehtäisiin yksi http-pyyntö, jossa haettaisiin paikka- ja reittitiedot, ja tämä tieto tallennettaisiin tilanhallinnan avulla tilaan eli stateen. Tämän mahdollistaa se, että sovellusta käytettäessä palvelimelta haettavat tiedot pysyvät samana, mikäli paikka- tai rastitietoja ei päivitetä juuri sinä aikana ylläpitäjän toimesta. Kun tiedot ovat valmiina sovelluksen välimuistissa, siirtyminen ikkunoiden välillä on nopeampaa. Kun sovellus sammuu, niin välimuisti (tila) tyhjenee, eli seuraavalla käyttökerralla tehdään uusi http-pyyntö ja siitä saatu tieto tallennetaan uudelleen välimuistiin.

Koska http-pyynnöillä saatu tieto ei lähtökohtaisesti muutu, on tämän sovelluksen tilanhallintatarve hieman tavallisesta poikkeava. Tilanhallintaratkaisut ovat lähtökohtaisesti tarkoitettu muuttuvalle tiedolle, mutta tässä sovelluksessa tilanhallintaa käytetään lähinnä tiedon varastointiin (Flutter, n.d.-c). Sovellusta aiotaan jatkokehittää vielä tulevaisuudessakin esimerkiksi radalla olevan käyttäjän aktiivisella seurannalla, jossa voidaan hyödyntää tässä opinnäytetyössä rakennettua tilanhallintaratkaisua. Tämä on hyvä ottaa huomioon eri tilanhallintaratkaisuja tutkiessa ja vertaillessa.

3 Tilanhallintaratkaisu Flutter-sovelluksessa

Flutter-sovelluksissa tilanhallintaratkaisua kuvataan yleisimmin apuvälineeksi, jolla tietoa voidaan säilöä käyttöliittymän uudelleenrakentamista varten. Kaikkia käyttöliittymään liittyviä asioita ei voi hallita käyttäjän toimesta, vaan Flutter hoitaa esimerkiksi tekstuurien tilanhallinnan. Ne tilat, joihin käyttäjä voi itse vaikuttaa jaetaan kahteen eri tilanhallintatyyppiin: ephemeral state eli lyhytaikainen tila sekä app state eli pitkäkestoisempi tila. (Flutter, n.d.-d).

Ephemeral state on yhteen widgetiin mahtuva tila, joka voisi olla esimerkiksi sen hetkinen valittu välilehti sovelluksen navigaattorissa. Tätä tilanhallintaa käytettäessä tilan sisältävä tieto ei muutu monimutkaisesti, eikä muut widgetit pääse käsittelemään tätä tilaa. Tämä tilanhallintatyyppi tarvitsee toimiakseen vain StatefulWidget-alustuksen, eikä se tarvitse erillisiä tilanhallintatekniikoita. Tila tyhjenee käyttäjän sulkiessa sovelluksen. (Flutter, n.d.-d.)

App state (joissain yhteyksissä shared state) on pitkäkestoisempi tila, joka on käytettävissä sovelluksen monissa osissa. Kyseinen tila voi säilyttää tietoa muistissa käyttökertojen välillä, eli se ei tyhjene käyttäjän sammuttaessa sovelluksen. Koska tilan muistia ei ole pakko tyhjentää, sitä voi käyttää esimerkiksi kirjautumistietojen säilytykseen. App state -tyyppiseen tilanhallintaan on olemassa erilaisia vaihtoehtoja, kukin hieman erilaisia ja eri tarkoituksiin. (Flutter, n.d.-d.)

Valinta näiden kahden tilanhallintatyypin välillä ei ole aina selvä, sillä valinta menee useasti käyttäjän mieltymysten mukaan. Kuitenkin nyrkkisääntönä voi pitää, että ephemeral state -tilanhallinnalla voi käsitellä yhden widgetin yksinkertaista ja muuttumatonta tietoa, kun taas app state -tilanhallinta on suunnattu monimutkaisempiin ja monen widgetin ratkaisuihin. (Flutter, n.d.-d.) Koska sovellusta aiotaan kehittää jatkossa, on järkevintä tutkia ja toteuttaa jokin app state -tyypin tilanhallintaratkaisu, jotta tulevaisuudessa on mahdollista käyttää tilanhallintaa enemmän kuin yhden widgetin tasolla.

3.1 Erilaisia tilanhallintaratkaisuja

Flutter sisältää lukuisia tilanhallintaratkaisuvaihtoehtoja erilaisine käyttötarkoituksineen. Niiden eroina on esimerkiksi käytettävyys, sovelluksen koko, sovelluksen tyyppi sekä tehokkuus. Flutterin omilla nettisivuilla on tarjolla 15 erilaista vaihtoehtoja, jotka kaikki ovat hieman erilaisia toisiinsa verrattuna. (Flutter, n.d.-e.). Tilanhallintaratkaisuista tarkastellaan tässä opinnäytetyössä vain ne vaihtoehdot, jotka mahdollisesti soveltuisivat Järvisuunnistus-sovellukseen.

3.1.1 Provider

Mikäli Flutterin tilanhallinta on uusi käsite, eikä sovelluksen kehittäjällä ole eritystä syytä käyttää jotakin tiettyä tilanhallintaratkaisua, niin Flutter suosittelee tilanhallintaratkaisuksi Provider-pakkausta. Provider on yksinkertainen tilanhallintaratkaisu, mutta se sisältää kaikki samat soveltuvuudet, jotka löytyvät kaikista muista ratkaisuista. (Flutter, n.d.-f.) Provider on ikään kuin pakattu, automatisoitu ja helppokäyttöisempi InheritedWidget, jonka tapaan Provider-ratkaisu noudattaa hierarkiaa sijoittamalla tilaobjektin niiden widgetien yläpuolelle, jotka sitä käyttävät. (pub.dev, 2022-b; Flutter, n.d.-g.)

Providerin ominaisuuksia ovat muun muassa yksinkertaistettu resurssien hävittäminen ja jakaminen, lazy-loading, ChangeNotifier, ChangeNotifierProvider, Consumer sekä Provider.of. Lazy-loading parantaa sovelluksen suorituskykyä viivästyttämällä resurssien alustusta ja käyttöönottoa kunnes niitä tarvitaan (Imperva, n.d.). ChangeNotifier on luokka, joka tarjoaa ilmoituksen tilan muuttuneesta tiedosta. Yksinkertaiseen sovellukseen riittää yksi ChangeNotifier, mutta tarvittaessa ja monimutkaisimmissa sovelluksissa niitä voi olla useampi. Jotta ChangeNotifier-luokalla olisi instanssi widget-hierarkiassa, se tarvitsee ChangeNotifierProvider-widgetin. InheritedWidget-hierarkiaa noudattaen kyseinen luokka sijoitetaan niiden widgetien yläpuolelle, jotka sitä käyttävät. Consumer-widgetin avulla päästään varsinaisesti käyttämään sitä luokkaa, johon ChangeNotifier on yhdistetty. Provider.of mahdollistaa luokan tiedon käyttämistä, ilman että tietoa varsinaisesti luetaan Consumer-widgetin avulla. Tämä voisi olla esimerkiksi jonkin luokan tietoa palauttamattoman metodin kutsu, jolloin widgetin luonti olisi turhaa resurssihukkaa. (Flutter, n.d.-f.)

3.1.2 Riverpod

Riverpod pohjautuu aiemmassa aliluvussa esiteltyyn Provider-ratkaisuun, mutta se sisältää kuitenkin monia tärkeitä eroavaisuuksia. Riverpod-ratkaisu on niin ikään vapaampi versio Provider-ratkaisusta sen tuetessa kaikkia provider-tyyppejä, odottamalla ei-synkronoituja metodikutsuja ja mahdollistamalla provider-objektin lisäämisen mistä tahansa hierarkian kohdasta. Toisin kuin Provider, Riverpod siis odottaa ei-synkronoitujen arvojen latautumista ja sisältää oman virhekäsittelyn, palauttaen aina jonkin arvon. (Riverpod, n.d.-a.)

Riverpod-ratkaisu on saatavilla kolmena eri pakkauksena, aina sovelluksesta ja tarpeesta riippuen: flutter_riverpod (Riverpodin ja Flutterin peruskäyttöön), hooks_riverpod (varustettu flutter_hooks -lisäosalla) sekä riverpod, josta on karsittu kaikki Flutteriin liittyvät asiat pois. (Riverpod, n.d.-b.)

Tilaa kapseloivat provider-objektit ovat tärkeä osa Riverpod-ratkaisua. Provider-objektin avulla ohjelma pystyy seuraamaan kunkin tilan sen hetkistä tilannetta. Se mahdollistaa tilan käsittelyn mistä tahansa ohjelman rakenteesta käsin, mikä korvaa aiemmin mainitun InheritedWidget-rakenteen. Provider-objektit ovat yliajettavissa testauksen ajaksi käyttäytymään eri lailla, mikä parantaa sovelluksen testattavuutta. Provider-ratkaisun tavoin Riverpodin provider-objektit optimoivat suorituskykyä rakentamalla vain sen osan sovelluksesta, mikä varmasti muuttuu jonkin tilan arvon vaihtuessa. Taulukossa 1 on Riverpod-ratkaisun kuusi erilaista provider-tyyppiä palautusarvoineen ja käyttöesimerkkeineen. Yksi provider-tyyppi on Provider-ratkaisusta tuttu ChangeNotifierProvider, jota ei kuitenkaan suositella käytettävän skaalautuville sovelluksille sen muuttuvien tilojen liittyviin ongelmien takia. Kyseinen provider-tyyppi on flutter_riverpod-pakkauksessa kuitenkin mukana, jotta mahdollinen siirtyminen Provider-ratkaisusta Riverpod-ratkaisuun olisi helpompaa. (Riverpod, n.d.-c.)

Taulukko 1 Provider-tyypit (mukaillen Riverpod, n.d.-c)

Provider-tyyppi	Palautusarvo	Käyttöesimerkki
Provider	Mikä tahansa tyyppi	Service-luokka, suodatettu lista
StateProvider	Mikä tahansa tyyppi	Suodatusehto, yksinkertainen tilaobjekti
FutureProvider	Minkä tahansa tyypin Future-arvo	API-kutsun tulos
StreamProvider	Minkä tahansa tyypin Stream-arvo	Jono API-tuloksista
StateNotifierProvider	StateNotifier-aliluokka	Monimutkainen ja muuttumaton tilaobjekti
ChangeNotifierProvider	ChangeNotifier-aliluokka	Monimutkainen ja muuttumista vaativa tilaobjekti

3.1.3 GetX

Koska sovelluksessa oli valmiiksi hyödynnetty GetX-liitännäistä sisäisten tietojen siirtoon, tutkitaan myös GetX-tilanhallintaratkaisua. Get-pakkauksessa on kaksi eri tilanhallintaratkaisua: yksinkertainen tilanhallinta GetBuilder sekä reaktiivinen tilanhallinta GetX/Obx. Tämä ratkaisu eroaa aiemmin esitellyistä tilanhallintaratkaisuista yksinkertaisemman rakenteensa puolesta. Kehittäjän ei tarvitse luoda esimerkiksi luokkaa jokaiselle tilalle, StreamBuilder-objektia attribuuteille tai StreamController-objektia tilan seurantaa varten. Jos var-attribuuttia halutaan seurata jostain widgetistä käsin, attribuutin arvon perään kirjoitetaan ”.obs” ja attribuutista tulee automaattisesti seurattava. Ohjelmakoodissa 1 Var-attribuutin muokkaus state-muotoon on esimerkki seurattavasta var-attribuutista. (pub.dev, 2022-a.)

Ohjelmakoodi 1 Var-attribuutin muokkaus state-muotoon

```
var laji = ’Suunnistus’.obs;
```

Kun jonkin tilan arvo halutaan näyttää käyttöliittymässä, tarvitaan Ohjelmakoodissa 2 esimerkkinä esitetty lyhyt ohjelmakoodi. Tällöin käytetään tilanhallintaratkaisun Obx-osaa, ja attribuutin nimi asetetaan kaarisulkeiden sisään controller-objektin kanssa. (pub.dev, 2022-a.)

Ohjelmakoodi 2 Tila-attribuutin tulostus käyttöliittymään

```
Obx(() => Text("${controller.laji}"));
```

GetX-tilanhallintaratkaisuun ei ole olemassa muita objekteja tai tilanhallintatyyppejä, eli kyseessä on hyvin yksinkertainen tilanhallintaratkaisu (pub.dev, 2022-a).

3.2 Vaihtoehtojen vertailua

Aiemmissa aliluvuissa esitellyt vaihtoehdot valittiin vertailuun muun muassa suosion, käytettävyyden sekä dokumentaation laajuuden perusteella. Vaihtoehtoja olisi ollut enemmänkin, mutta niitä ei otettu vertailuun mukaan esimerkiksi liian vähäisen dokumentaation, käyttötarkoituksen sopimattomuuden tai jatkokehityksen vaikeuden perusteella.
Jotta Järvisuunnistus-sovellukseen valitaan näistä vaihtoehdoista sopivin tilanhallintaratkaisu, on näitä vaihtoehtoja verrattava keskenään.

GetX erottuu valituista vaihtoehdoista eniten yksinkertaisuudellaan. Vaikka GetX-liitännäistä olikin jo käytetty sovelluksen muihin tarkoituksiin, on sen tilanhallintaratkaisu liian suppea tähän sovellukseen ja se voi toimia pullonkaulana tämän sovelluksen jatkokehityksessä. Kyseinen tilanhallintaratkaisu toimisi paremmin pienemmän kokoluokan sovelluksessa, esimerkiksi to do -mobiilisovelluksessa.

Provider- ja Riverpod-ratkaisujen välillä ei suuria eroja ole, sillä Riverpod pohjautuu Provider-ratkaisun rakenteeseen (Riverpod, n.d.-a). Kuitenkin Riverpod sisältää muutaman tärkeän ominaisuuden, jotka vaikuttavat tähän sovellukseen suuresti. Esimerkiksi ei-synkronoitujen provider-objektien tuki on yksi tärkeimmistä ominaisuuksista, sillä sovellukseen rakennettu metodi http-pyyntöjä varten on ei-synkronoitu (async). Sovellus voi siis jatkaa toimintaansa, vaikka http-pyyntö olisi kesken.

Riverpod sisältää myös muita hyödyllisiä ominaisuuksia, mitä Provider-ratkaisussa ole. Riverpod-ratkaisun provider-objekteja voi muokata käyttämällä ”.family”-muunninta, jolla saadaan uniikki provider-objekti ulkoisten parametrien avulla. Ulkoisena parametrina voi käyttää esimerkiksi jonkin objektin ID:tä tai otsikkoa. Esimerkiksi FutureProvider-objekti voidaan yhdistää tähän muuntimeen, jolloin voitaisiin hakea ID:n avulla jokin tietty osa API-kutsun tuloksesta. (Riverpod, n.d.-d.)

Muistin käytön optimoimiseksi Riverpod-ratkaisussa on ”.autoDispose”-toiminto, jolla voidaan poistaa provider-objektin tila, kun sitä ei enää tarvita. Sitä voidaan käyttää myös niin sanottuna nollaustoimintona käyttäjän poistuessa sitä tilaa koskevalta ruudulta. Tätä toimintoa voidaan myös hyödyntää http-pyynnöissä siten, että pyynnön epäonnistuessa tila nollataan ja taas pyynnön onnistuessa tila säilytetään, minkä jälkeen uutta pyyntöä ei tarvitse tehdä. (Riverpod, n.d.-e.)

Riverpod-ratkaisussa on siis monia hyvin tarpeellisia ominaisuuksia tätä sovellusta ajatellen, jotka puuttuvat Provider-ratkaisusta. Riverpod-ratkaisun ominaisuudet palvelevat tämän opinnäytetyön tarkoituksia, mutta on niiden myötä hyvä valinta myös sovelluksen jatkokehitystä varten. Tilanhallintaratkaisuksi valitaan siis Riverpod sovelluksen tilanhallinnan toteuttamista varten.

4 Menetelmät

Opinnäytetyön tiedonsaannissa käytetään lähteitä luotettavilta tahoilta, joista kirjataan tärkeimmät huomiot teoriaosuuteen käytännön osan ymmärtämistä varten. Järvisuunnistus-sovelluksesta tarvittavat tiedot saadaan pitämällä palavereja työn tilaajan kanssa.

Opinnäytetyön kirjoittamisen ohessa täytetään päiväkirjaa, johon merkataan muistiinpanoja sekä tehdyt asiat joka päivältä. Päiväkirja auttaa hahmottamaan työn kulkua ja muistiinpanoista pystyy seuraamaan, että mitä toiminnallisuuksia on koodattu ja millä päivämäärällä.

Työ tehdään vesiputousmallia mukaillen suunnittelemalla ensin sovelluksen tilanhallinta tarpeet huomioiden, jonka jälkeen tilanhallintaratkaisu toteutetaan. Lopuksi suoritetaan tilanhallinnan testaus. Ohjelmointivaiheessa syntyneet muutokset viedään GitHub-versionhallintajärjestelmään, jossa on opinnäytetyötä varten oma haara. Muutokset jäävät haaraan, jotta työn tilannut asiakas pääsee opinnäytetyössä tehtyyn tilanhallintaratkaisuun käsiksi. Myös alkuperäinen versio sovelluksesta jää GitHub-palveluun talteen erilliseen haaraan.

Työn aikana pidetään noin kahden viikon välein palaveri työn tilaajan kanssa, jossa keskustellaan ja tarkastellaan työn sen hetkistä tilannetta sekä tulevia muutoksia. Näin voidaan mahdolliset virheet huomata ajoissa. Kun työ on valmis, käydään se työn tilaajan kanssa läpi ja työn tilaaja antaa opinnäytetyön tekijälle kommentit saadusta lopputuloksesta.

5 Tilanhallintaratkaisun toteuttaminen

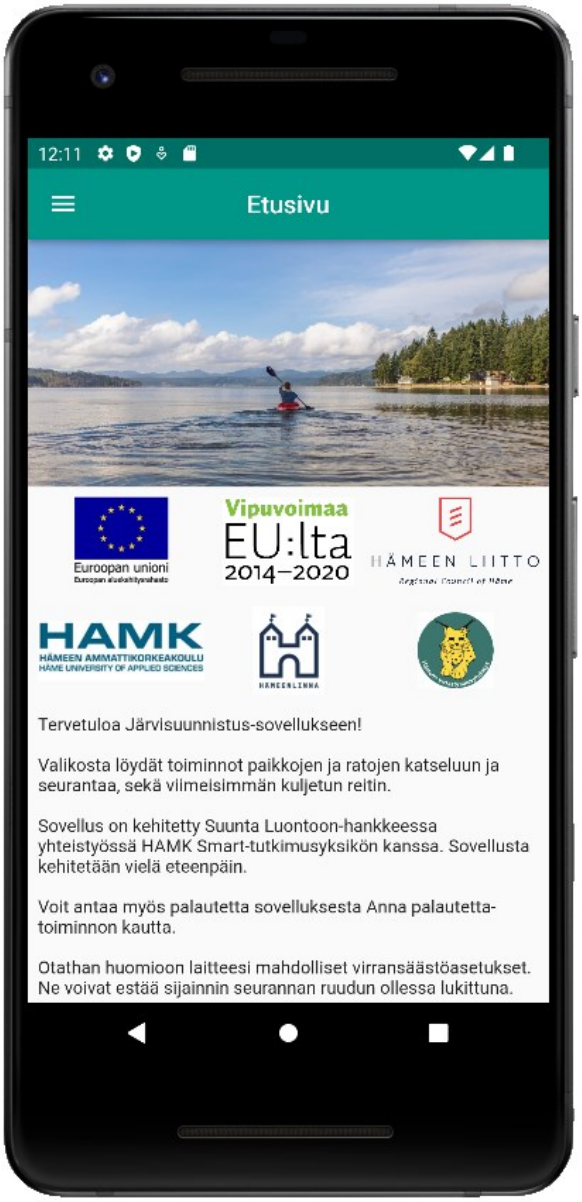
Tässä luvussa toteutetaan Riverpod-tilanhallintaratkaisu Järvisuunnistus-sovellukseen. Sovellukseen asennetaan ensin tarvittavat riippuvuudet, jonka jälkeen luodaan tilanhallintaratkaisu suunnistuspaikkoja sekä -reittejä varten. Lopuksi suoritetaan testaus tilanhallintaratkaisun toimivuudesta.

5.1 Kehitysympäristö

Sovelluksen ajamista varten tarvitaan Visual Studio Code -kehitysympäristö, Android Studio -mobiiliemulaattori sekä Flutter SDK -kehitystyökalu. Työssä käytettävä virtuaalimobiililaite Android Studiossa on Pixel 2, johon on asennettuna Android 12.0 -käyttöjärjestelmä. Flutter SDK -kehitystyökalun versio on 3.3.10.

Sovellus voidaan käynnistää monella tavalla. Tässä työssä käynnistetään ensin virtuaalimobiililaite Android Studiossa, jonka jälkeen avataan Windows PowerShell -komentotulkki. Komentotulkissa siirrytään polkuun, jossa sovelluksen tiedostot ovat. Tässä opinnäytetyössä tiedostot sijaitsevat C-asemalla, ja polku on ”C:\Jarvisuunnistus_VSC\vesisapp”. Kun oikeaan polkuun on siirrytty, kirjoitetaan komentotulkkiin komento ”flutter run”, jonka jälkeen sovellus käynnistyy Kuvassa 5 Sovellus käynnissä Android-emulaattorissanäkyvässä Android-emulaattorissa.

Kuva 5 Sovellus käynnissä Android-emulaattorissa



5.2 Riverpod-ratkaisun yhdistäminen sovellukseen

Ennen kuin Riverpod-ratkaisun ominaisuuksia pääsee käyttämään sovelluksessa, on se asennettava projektiin. Riverpod-tilanhallintaratkaisun versioista asennetaan niin sanottu peruspaketti, ”flutter_riverpod”, sillä Riverpod-ratkaisun lisäosia ei tarvita tämän opinnäytetyön osalta.

5.2.1 Riverpod-paketin asennus ja riippuvuudet

Riverpod-paketin asennus tapahtuu siirtymällä Windows PowerShell -komentotulkilla projektin polkuun ja kirjoittamalla Komento 1 näkyvä lisäyskomento tulkkiin.

Komento 1 Riverpod-paketin lisäys projektiin komentotulkissa

```
flutter pub add flutter_riverpod
```

Tämän jälkeen voidaan tarkistaa Riverpod-riippuvaisuus projektin pubspec.yaml-tiedostosta, joka sisältää metadatan kaikista projektin riippuvaisuuksista. Mikäli asennus onnistui, tiedostosta löytyy Ohjelmakoodissa 3 näkyvä Riverpod-riippuvaisuus.

Ohjelmakoodi 3 Riverpod-riippuvaisuus pubspec.yaml-tiedostossa

```yaml
dependencies:
  flutter:
    sdk: flutter
  permission_handler: ^9.2.0
  oktoast: ^3.1.1
  flutter_map: ^0.14.0
  flutter_map_marker_cluster: ^0.4.0
  flutter_map_marker_popup: ^2.1.2
  shared_preferences: ^2.0.6
  http: ^0.13.4

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.2
  get: ^4.5.1
  path_provider: ^2.0.9
  flutter_ringtone_player: ^3.2.0
  url_launcher: ^6.1.2
  geolocator: ^8.2.1
  sqflite: ^2.0.2+1
  flutter_riverpod: ^2.1.3
```

Muutoksia syntyy myös pubspec.lock-tiedostoon, joka varastoi ja tallentaa kirjastojen ja pakkausten versiot yksityiskohtaisemmin. Kyseinen tiedosto tallentaa ja lukitsee käyttöön sen version pakkauksesta tai kirjastosta, mikä on yhteensopiva muiden riippuvuuksien kanssa. Ohjelmakoodissa 4 näkyy GitHub Desktop -versionhallintatyökalun näkökulmasta kyseiseen tiedostoon syntyneet muutokset.

Ohjelmakoodi 4 pubspec.lock-tiedoston muutokset

pubspec.lock

```
@@ -104,6 +104,13 @@ packages:
      url: "https://pub.dartlang.org"
    source: hosted
    version: "3.2.0"
+ flutter_riverpod:
+   dependency: "direct main"
+   description:
+     name: flutter_riverpod
+     url: "https://pub.dartlang.org"
+   source: hosted
+   version: "2.1.3"
  flutter_test:
    dependency: "direct dev"
    description: flutter
@@ -366,6 +373,13 @@ packages:
      url: "https://pub.dartlang.org"
    source: hosted
    version: "2.1.0"
+ riverpod:
+   dependency: transitive
+   description:
+     name: riverpod
+     url: "https://pub.dartlang.org"
+   source: hosted
+   version: "2.1.3"
  shared_preferences:
    dependency: "direct main"
    description:
@@ -455,6 +469,13 @@ packages:
      url: "https://pub.dartlang.org"
    source: hosted
    version: "1.10.0"
+ state_notifier:
+   dependency: transitive
+   description:
+     name: state_notifier
+     url: "https://pub.dartlang.org"
+   source: hosted
+   version: "0.7.2+1"
  stream_channel:
    dependency: transitive
    description:
```

5.2.2 ProviderScope

Kun riippuvaisuudet ovat kunnossa, voidaan ryhtyä rakentamaan itse tilanhallintaa. Ennen kuin provider-objekteja pääsee hyödyntämään projektissa, on suoritettava pienimuotoinen muutos projektin main.dart-tiedostoon. Kyseisen tiedoston main-funktion sisällä on funktio runApp parametrinaan MyApp-luokka, joka pitää ympäröidä Riverpodin ProviderScope-luokalla Ohjelmakoodin 5 mukaisesti. Provider-objektien tila tallennetaan tähän ProviderScope-luokkaan, ja sen on sijaittava widget-hierarkian juuressa. Ilman tätä luokkaa ohjelma ei toimisi Riverpodilla. (pub.dev, n.d.)

Ohjelmakoodi 5 ProviderScope-luokka main.dart-tiedostossa

```
void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  runApp(
    ProviderScope(
      child: MyApp(),
    ),
  );
}
```

5.3 Tilanhallinnan toteutus suunnistuspaikkoja varten

Aloitetaan luomalla tilanhallinta suunnistuspaikkojen http-pyynnön tulosten säilyttämistä varten. Pyyntö suoritetaan, kun sovelluksesta avataan Paikat-ikkuna, jonka jälkeen kaikki suunnistuspaikat on tarkoitus tallettaa tilaan. Kun tilanhallinta on luotu, käyttäjälle sovellus näyttää täysin samalta.

5.3.1 Provider-objektien luonti

Otetaan ensiksi tarkasteluun placesApi.dart-tiedosto, jossa suoritetaan http-pyyntö sovelluksen suunnistuspaikkojen hakua varten. Riverpod-pakkaus tuodaan tiedostoon lisäämällä tiedoston alkuun import-komento Ohjelmakoodin 6 mukaisesti.

Ohjelmakoodi 6 Riverpod-pakkauksen tuonti tiedostoon

```
import 'package:flutter_riverpod/flutter_riverpod.dart';
```

Tämän jälkeen luodaan tiedostoon uusi provider-objekti PlaceAPI-luokasta. Se kirjoitetaan luokan ulkopuolelle Ohjelmakoodin 7 mukaisesti. Objekti tallennetaan attribuuttiin nimeltä httpProvider,

ja sen tyypiksi asetetaan itse PlaceAPI-luokka. Riverpod-ratkaisuissa funktion kohdalle asetetaan aina ref-objekti parametriksi, jonka kautta provider-objekti voidaan lukea. Lopuksi se palauttaa funktion, joka tässä tapauksessa on PlaceAPI-luokka. Lopputuloksena on siis provider-objekti, johon on injektoitu koko PlaceAPI-luokka.

Ohjelmakoodi 7 PlaceAPI-luokan provider-objekti

```
final httpProvider = Provider<PlaceAPI>((ref) => PlaceAPI());
```

Kun placesApi.dart-tiedosto on injektoitu provider-objektiin, se pitää vielä lisätä toisen provider-objektin sisälle. Koska kyseessä on ei-synkronoitu http-kutsu, käytetään provider-tyyppinä FutureProvider-objektia, joka tukee ei-synkronoitua koodia.

Luodaan projektiin FutureProvider-objekteja varten uusi kansio. Annetaan sille nimi ”data”, ja luodaan sen sisälle place_data.dart-tiedosto. Kun tiedosto on luotu, tuodaan siihen import-komennolla Riverpod-pakkaus, äsken käsitelty placesApi.dart sekä models-kansiosta places.dart. Näistä viimeisin toimii rakentajana placesApi.dart-tiedostolle, joka määrää attribuutteja, kuten esimerkiksi constructor-luokat Java-ohjelmoinnissa.

Tähän luokkaan rakennetaan Ohjelmakoodissa 8 näkyvä FutureProvider-objekti, jonka sisään injektoidaan aiemmin luotu httpProvider-objekti. Annetaan objektille nimeksi placeDataProvider. Sen palautusarvona ref-parametri käyttää watch-metodia, joka saa parametrikseen httpProvider-objektin. Watch-metodi seuraa httpProvider-objektin muutoksia ja kutsuu http-pyynnön suorittavaa fetchAllPlaces-metodia.

Ohjelmakoodi 8 FutureProvider-objekti paikkatiedoille

```
final placeDataProvider = FutureProvider<List<Places?>>((ref) async {
  return ref.watch(httpProvider).fetchAllPlaces();
});
```

5.3.2 Provider-objektin lukeminen Paikat-käyttöliittymässä

Kun paikkatietoja varten on rakennettu provider-objektit, luetaan ne käyttöliittymätiedoston puolella eli pages-kansiossa sijaitsevassa places.dart-tiedostossa. Itse tiedostoa pitää hieman muokata, jotta tilanhallintaa voidaan käyttää.

Käyttöliittymätiedostossa alustetaan kaksi luokkaa: PlacesPage sekä PlacesPageState. Näistä ensimmäinen perii itselleen StatefulWidget-tyypin ja jälkimmäinen State-tyypin. Jotta käyttöliittymäluokat voivat lukea provider-objekteja, on niiden perittävä tyypikseen ConsumerWidget. Muutetaan PlacesPage-luokan tyypiksi siis ConsumerWidget Ohjelmakoodin 9 mukaisesti ja samalla poistetaan PlacesPageState, sillä sitä ei tarvita enää uudessa Riverpod-rakenteessa.

Ohjelmakoodi 9 PlacesPage-luokan uusi tyyppi, ConsumerWidget

```
class PlacesPage extends ConsumerWidget
```

FutureProvider-objekti luetaan yleensä käyttöliittymäluokan widgetissä. Koska sovelluksen käyttöliittymässä käsitellään http-pyynnöstä tullutta tietoa widgetin ulkopuolella, joudutaan tiedonkulun rakennetta hieman muokkaamaan.

Asetetaan build-widgetin parametreihin WidgetRef context-parametrin lisäksi, jotta aiemmin luotu FutureProvider-objekti voidaan lukea widgetin sisällä. Alustetaan objektin sisältämää tietoa varten attribuutti ”_data” ja sen arvoksi placeDataProvider-objektista haettava tieto Ohjelmakoodin 10 mukaisesti.

Ohjelmakoodi 10 Widgetin alustus ja provider-objektin luku

```
Widget build(BuildContext context, WidgetRef ref) {
    final _data = ref.watch(placeDataProvider);
…
}
```

Widgetiin on nyt tuotu aiemmin luotu FutureProvider ja se luetaan käyttäen watch-metodia ref-parametrin kautta. Luettavaa dataa pitää kuitenkin käsitellä ennen kuin sitä voidaan hyödyntää, ja tämä tapahtuu widgetin body-rakenteessa. Body-rakenteen alkuun luodaan ”_data”-attribuutille when-metodi Ohjelmakoodin 11 mukaisesti, jolla on arvot data, error ja loading. Data-arvoksi luodaan funktio, jossa iteroidaan ”_data”-attribuutti ja muunnetaan se luettavaan listamuotoon. Lista tallennetaan finalPlace-listaan, jonka tyyppinä on Places-luokka. Asetetaan tyypin perään kysymysmerkki, jolloin sallitaan luodun listan tyhjä arvo, mikäli tietoa ei jostain syystä saada. Error-arvoksi asetetaan stackTrace-metodi String-muodossa, jotta saadaan tieto viallisesta metodista tai tapahtumasta mahdollisen virheen tapahtuessa. Loading-arvoksi asetetaan Center-widget, ja sen

sisälle CircularProgressIndicator-animaatio, joka visualisoi lataamista ennen tiedon haun valmistumista.

Ohjelmakoodi 11 When-metodi paikkojen widgetissä

```
body: _data.when(
            data: (_data) {
              List<Places?> finalPlace = _data.map((e) => e).toList();

...


error: (error, stackTrace) => Text(error.toString()),
            loading: () => const Center(
                  child: CircularProgressIndicator(),
                )) ...
```

Seuraavaksi lähetetään finalPlace-lista käsiteltäväksi. Widgetissä kutsuttavaan getMarkerLayerOptions-metodiin lisätään parametriksi finalPlace-lista, ja kyseisen metodin try-funktiossa kutsutaan buildPlaceMarkersOnMap-metodia, joka parsii paikkatietoja pienempiin osiin. Lisätään myös tämän metodin parametreihin finalPlace-lista. Paikkatiedot parsitaan buildPlaceMarkersOnMap-metodissa käyttäen forEach-funktiota, jonka sisällä finalPlace-listasta haetaan element-parametrin avulla attribuutit lat (latitude) ja long (longitude), eli yksittäisen suunnistuspaikan leveys- ja pituusaste Ohjelmakoodin 12 mukaisesti. Nämä arvot tallennetaan sijaintityyppiseen (LatLng) attribuuttiin nimeltä coords, ja tietojen käsittely jatkuu tästä eteenpäin sovelluksen alkuperäisen kaavan mukaisesti.

Ohjelmakoodi 12 Metodi paikkatietojen tarkempaa käsittelyä varten

```
Future<List<Marker>> _buildPlaceMarkersOnMap(List<Places?> finalPlace)
async {
    markers.clear();
    //if (finalPlace.length == 0) {
    //listPlaces = await fetchPlaces();

    finalPlace.forEach((element) {
      if (element?.lat == null ||
          element?.lat == 0 ||
          element?.long == null ||
          element?.long == 0) {
        //skip this place cause location is faulty
      } else {
        LatLng coords = LatLng(element!.lat, element.long);
…
```

5.4 Tilanhallinnan toteutus suunnistusreittejä varten

Jatketaan luomalla tilanhallinta suunnistusreittien http-pyynnön tulosten säilyttämistä varten. Pyyntö suoritetaan, kun sovelluksesta avataan Reitit-ikkuna, jonka jälkeen kaikki suunnistusreitit on tarkoitus tallettaa tilaan. Tilanhallinta ei tässäkään ikkunassa tule näkymään loppukäyttäjälle millään tavalla.

5.4.1 Provider-objektien luonti

Reittien http-pyynnöt suorittavassa fetchTrackById-metodissa haetaan rajapinnasta suunnistusreitit valitun suunnistuspaikan mukaan. Suunnistuspaikan tieto on sijoitettu integer-muodossa attribuuttiin placeId. Koska tilanhallinnan ideana on minimoida http-pyyntöjen määrä, poistetaan placeId-attribuutti metodista ja haetaan kaikki suunnistusreitit samalla http-pyynnöllä.

Luodaan TracksAPI-luokkaan uusi provider-objekti nimellä trackProvider. Provider-objektin tyypiksi asetetaan TracksAPI-luokka ja parametriksi ref-objekti. Lopputuloksena on funktio, joka palauttaa koko TracksAPI-luokan. Täten siis koko luokka on injektoitu trackProvider-objektiin Ohjelmakoodin 13 mukaisesti.

Ohjelmakoodi 13 TracksAPI-luokan provider-objekti

```
final trackProvider = Provider<TracksAPI>((ref) => TracksAPI());
```

Tämän jälkeen pitää trackProvider-objekti injektoida FutureProvider-objektin sisään. Luodaan aiemmin luotuun data-kansioon track_data.dart-tiedosto, jonka sisään rakennetaan FutureProvider-objekti Ohjelmakoodin 14 mukaisesti. Annetaan objektille nimeksi trackDataProvider ja tyypiksi Tracks. Parametreihin asetetaan ref-objekti, jolla luetaan trackProvider-objekti ja kutsutaan sen sisältämää fetchTrackById-metodia. Palautusarvona on fetchTrackById-metodin tulos.

Ohjelmakoodi 14 FutureProvider-objekti reittitiedoille

```
final trackDataProvider = FutureProvider<List<Tracks?>>((ref) async {
  return ref.read(trackProvider).fetchTrackById();
});
```

5.4.2 Provider-objektien lukeminen Reitit-käyttöliittymässä

Kun reittitietojen FutureProvider-objekti on luotu, voidaan sitä lukea reittien käyttöliittymässä eli tracks.dart-tiedostossa. Kuten paikkatietojen käyttöliittymässä, pitää käyttöliittymän komponentteja muokata.

Jotta FutureProvider-objekti saa yhteyden käyttöliittymäluokkaan, on TracksPage-luokan perittävä StatefulWidgetin sijaan ConsumerWidget. Alun perin tiedostossa ollut toinen luokka, TracksPageState, tuli tämän muutoksen myötä tarpeettomaksi ja se voidaan poistaa.

Aloitetaan FutureProvider-objektin lukuprosessi build-widgetissä. Lisätään widgetiin parametreiksi WidgetRef, jonka kautta luetaan ja seurataan FutureProvider-objektia. Objekti tallennetaan tracks-attribuuttiin. Widgetin body-komponenttiin luodaan when-metodi Ohjelmakoodin 15 mukaisesti, jonka data-arvossa FutureProvider-objektin sisältävä tracks-attribuutti iteroidaan ja tallennetaan finalTrack-listaan. Error-arvoon luodaan tavallinen stackTrace-metodi ja loading-arvoon CircularProgressIndicator-metodi, kuten suunnistuspaikkojen widgetissä aiemmin tehtiin.

Ohjelmakoodi 15 When-metodi reittien widgetissä

```
tracks.when(
            data: (_tracks) {
              List<Tracks?> finalTrack = _tracks.map((e) => e).toList();
...

},
            error: ((error, stackTrace) => Text(error.toString())),
            loading: () => const Center(
                  child: CircularProgressIndicator(),
                ));
```

Reittien jatkokäsittelyn aloittamiseksi widgetissa kutsutaan getMarkerLayerOptions-metodia, jossa alun perin kutsuttiin buildTrackMarkersOnMap-metodia parametrinaan suunnistuspaikan id-numero. Koska tracks-attribuutti sisältää kaikkien paikkojen reitit, on tässä metodissa suodatettava reitit valitun paikan mukaan ennen tietojen lähetystä eteenpäin seuraavalle metodille. Luodaan uusi lista ja annetaan sille nimeksi filteredList. Sen arvoksi luodaan where-funktio Ohjelmakoodin 16 mukaisesti ja sille etsittäväksi arvoksi placeId-attribuutti (valittu suunnistuspaikka), joka on tuotu TracksPage-luokkaan PlacesPage-luokasta. Mikäli tiedot saadaan

haettua, kutsutaan buildTrackMarkersOnMap-metodia parametrinaan valitun suunnistuspaikan reitit sisältävä filteredList-lista.

Ohjelmakoodi 16 Where-funktio reittien suodattamiseksi

```
Future<MarkerClusterLayerOptions> getMarkerLayerOptions(
      List<Tracks?> finalTrack) async {
    var markers;
    final filteredList = finalTrack.where((e) => e!.placeId == placeId);
```

Kun buildTrackMarkersOnMap-metodia kutsutaan, se käy parametrinaan saadun filteredList-listan läpi ja tallentaa siitä saadut koordinaatit coords-attribuuttiin Ohjelmakoodin 17 mukaisesti.

Ohjelmakoodi 17 Metodi reittitietojen tarkempaa käsittelyä varten

```
Future<List<Marker>> _buildTrackMarkersOnMap(filteredList) async {
    markers.clear();
    _center = LatLng(lat, long);
    filteredList.forEach((element) {
      if (element?.lat == null ||
          element?.lat == 0 ||
          element?.long == null ||
          element?.long == 0) {
        //skip this place cause location is faulty
      } else {
        LatLng coords = LatLng(element!.lat, element.long);
```

5.5 Tilanhallintaratkaisun testaus

Tilanhallinnan luomisen jälkeen sovellusta testatessa voidaan todeta sen toimivan kuten ennenkin. Jotta tilanhallinnan toimivuudesta voidaan olla varmoja, pitää se testata. Koska tilanhallinnan ideana oli tallentaa paikka- ja reittitiedot tilaan http-tietopyynnön jälkeen, on http-tietopyyntöjen määrän tarkastaminen yksi testauskohteista.

Lisätään placesApi.dart- sekä tracksApi.dart-tiedostojen http-metodeihin tulostusmetodit Ohjelmakoodin 18 mukaisesti, joka suorittaa jonkin tekstin tulostuksen konsoliin. Tämä tulostus siis tapahtuu aina, kun http-pyyntö eli nämä metodit suoritetaan. Tulostus näkyy komentotulkissa, kun http-pyyntö tehdään. Mikäli tilanhallinta toimii oikein, saadaan molemmista metodeista tulostus, kun sovellus on juuri käynnistetty ja Paikat- tai Reitit-ikkuna avataan ensimmäistä kertaa

käynnistyksen jälkeen. Tämän jälkeen tulostusta ei pitäisi näkyä ikkunasta toiseen siirryttäessä, sillä tiedot pitäisi olla haettavissa suoraan tilasta.

Ohjelmakoodi 18 Tulostusmetodit http-pyyntö-metodeissa

```
print('HTTP-pyyntö - PAIKAT');
...
print('HTTP-PYYNTO - REITIT');
```

Kun sovellus käynnistetään ja avataan Paikat-ikkuna ensimmäistä kertaa, komentotulkissa nähdään äsken luodun tulostusmetodin teksti Kuvan 6 mukaisesti, kuten kuuluukin.

Kuva 6 Paikkatietojen tulostusmetodi

```
I/flutter (27122): Response:
I/flutter (27122): 200
I/flutter (27122): HTTP-pyyntö - PAIKAT
```

Kun valitaan jokin paikka kartalta, avautuu Reitit-ikkuna ja komentotulkkiin ilmestyy tulostusmetodin teksti Kuvan 7 mukaisesti.

Kuva 7 Reittitietojen tulostusmetodi

```
I/flutter (27122):
I/flutter (27122): HTTP-PYYNTO - REITIT
```

Tässä kohtaa pitäisi siis paikka- ja reittitiedot olla haettuna tilaan. Kun Reitit-ikkunasta siirrytään takaisin Paikat-ikkunaan, huomataan komentotulkista, että tulostusmetodia ei enää näy. Vaikka valitsisi jonkun uuden paikan, http-pyyntöjä ei enää tehdä, sillä ne kaikki ovat haettu tilaan. Sama ilmenee reittien kanssa, eli tässä vaiheessa voidaan todeta tilanhallinnan toimivan kuten on suunniteltu. Koska haettua tietoa ei ole tarpeen säilyttää käyttökertojen välissä, tila siis tyhjenee aina sovelluksen sammuessa ja tiedot haetaan uudelleen sovelluksen käynnistyessä.

Vaikka loppukäyttäjälle sovellus näyttää käytännössä täysin samalta kuin aiemmin, voi mahdollisesti sovelluksen nopeudessa havaita pieniä eroja. Kun http-pyyntöjä ei enää tehdä joka kerta siirryttäessä Paikat- tai Reitit-ikkunalle, näyttää sovellus paikka- ja reittimerkinnät nopeammin kartalla. Tämä voi olla hyödyksi tilanteissa, joissa nettiyhteys on hidas tai pätkivä, mikäli ensimmäiset http-pyynnöt saadaan tehtyä sovelluksen käynnistyessä.

6 Asiakkaan palaute

Kun opinnäytetyö oli siinä pisteessä, että tilanhallintaratkaisu oli toteutettu ja sen toimivuus oli testattu, järjestettiin asiakkaan kanssa työn katselmointi. Asiakkaan (HAMK Smart -tutkimusyksikkö) edustajana toimi opinnäytetyön ajan sovelluskehittäjä Juuso Saarinen.

Katselmoinnissa käytiin läpi tehdyt muutokset ja saavutetut tulokset, joita verrattiin työn alussa luotuihin suunnitelmiin ja tavoitteisiin. Asiakaspalautteen tuloksena todettiin, että opinnäytetyön aikana pidetyt säännölliset palaverit olivat hyödyllisiä. Tällaiset palaverit eivät ole joka projektissa itsestäänselvyyksiä, ja niiden avulla vältyttiin epäselvyyksiltä ja varmistettiin, että työn tekijällä ja tilaajalla on yhtäläinen ymmärrys esimerkiksi työn tavoitteista sekä suunnitelmasta.

Työn koodiosuus oli hyvin rakennettu ja se oli tarpeeksi yksinkertainen. Tilanhallintaratkaisujen vaihtoehtoja oli vertailtu kattavasti ja valinta tilanhallintaratkaisusta vaikutti asiakkaan näkökulmasta oikealta. Opinnäytetyöstä tulee olemaan hyötyä tutkimusyksikölle jatkossa myös uusien sovellusten kannalta, ja työ auttaa ymmärtämään ja käyttämään tässä työssä käsiteltyä aihetta, vaikka Flutterista ei aikaisempaa kokemusta olisikaan. Työ on auttanut ajattelemaan ja ymmärtämään, että mitä tilanhallinnalta vaaditaan etukäteen ja mitä tilanhallinta loppujen lopuksi mahdollistaa. Lopputuloksena todettiin, että työ on onnistunut ja asiakkaan asettamat vaatimukset ja odotukset saavutettiin.

7 Yhteenveto

Opinnäytetyön alussa tutustuttiin Järvisuunnistus-sovellukseen liittyviin työkaluihin, kehitykseen sekä rajapintoihin. Kaikkiin tutkimuskysymyksiin saatiin vastaus työn aikana: erilaisiin tilanhallintaratkaisuihin tutustuttiin, sopivin tilanhallintaratkaisu etsittiin tätä sovellusta varten ja lopuksi valittu tilanhallintaratkaisu (Riverpod) rakennettiin sovellukseen.

Vielä opinnäytetyötä aloittaessani Flutter-kehys sekä dart-ohjelmointikieli olivat minulle täysin uusia asioita. Työn aikana opin käyttämään Flutteria ja ymmärtämään sen logiikkaa sekä lukemaan ja kirjoittamaan dart-kielen syntaksia. Tilanhallinta käsitteenä oli tuttua aikaisemmista mobiilisovellusprojekteista, mutta Flutter-kehyksessä tilanhallintaratkaisu toimii hieman omintakeisella tavalla, joten se vaati myös jonkun verran opettelua. Opin paljon myös erilaisista tilanhallintaratkaisuista: minkälaisia ratkaisuja on tarjolla, mitä asioita ratkaisuja valittaessa kuuluu huomioida ja mitä rajoitteita niillä on. Työssä suurin vaikeus oli tehdä muutoksia isoon sovellukseen, jota ei ole ollut itse alun alkaen kehittämässä. Tilanhallintaratkaisun lisääminen ei siis aina sujunut ongelmitta, sillä se vaati isoja muutoksia sovelluksen koodiin.

Lopputuloksena syntyi versio sovelluksesta, jossa paikka- ja reittitietojen koordinaatit haetaan REST-rajapinnasta ja kyseiset tiedot tallennetaan Riverpod-tilanhallintaratkaisun avulla sovelluksen välimuistiin. Ratkaisun avulla sovelluksesta saatiin nopeampi. Opinnäytetyötä voidaan hyödyntää kyseisen sovelluksen jatkokehityksessä sekä muissa tutkimusyksikön projekteissa, joihin tilanhallintaa aiotaan käyttää.

Lähteet

Red Hat. 2020. *What is a REST API?.* Haettu 31.1.2023 osoitteesta https://www.redhat.com/en/topics/api/what-is-a-rest-api

Gillis, A. (n.d.). *What is REST API (RESTful API)?.* TechTarget. Haettu 26.1.2023 osoitteesta https://www.techtarget.com/searchapparchitecture/definition/RESTful-API

Google Developers. (23.2.2018). Introducing Flutter [video]. YouTube. https://www.youtube.com/watch?v=fq4N0hgOWzU&ab_channel=GoogleDevelopers

Flutter. (n.d.-a). *Flutter SDK releases.* Haettu 25.1.2023 osoitteesta https://docs.flutter.dev/development/tools/sdk/releases?tab=windows

Google Developers. (17.10.2012). Introducing Dart [video]. YouTube. https://www.youtube.com/watch?v=5KlnlCq2M5Q&ab_channel=GoogleDevelopers

Flutter. (n.d.-b). *Introduction to widgets.* Haettu 31.1.2023 osoitteesta https://docs.flutter.dev/development/ui/widgets-intro

pub.dev. 2022-a. *get | Flutter Package*. Haettu 27.1.2023 osoitteesta https://pub.dev/packages/get

Flutter. (n.d.-c). *Start thinking declaratively.* Haettu 3.2.2023 osoitteesta https://docs.flutter.dev/development/data-and-backend/state-mgmt/declarative

Flutter. (n.d.-d). *Differentiate between ephemeral state and app state.* Haettu 30.1.2023 osoitteesta https://docs.flutter.dev/development/data-and-backend/state-mgmt/ephemeral-vs-app

Flutter. (n.d.-e). *List of state management approaches.* Haettu 2.2.2023 osoitteesta https://docs.flutter.dev/development/data-and-backend/state-mgmt/options

Flutter. (n.d.-f). *Simple app state management.* Haettu 3.2.2023 osoitteesta https://docs.flutter.dev/development/data-and-backend/state-mgmt/simple

pub.dev. 2022-b. *provider | Flutter Package.* Haettu 3.2.2023 osoitteesta https://pub.dev/packages/provider

Flutter. (n.d.-g). *InheritedWidget class.* Haettu 3.2.2023 osoitteesta https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html

Imperva. (n.d.). *Lazy Loading.* Haettu 3.2.2023 osoitteesta https://www.imperva.com/learn/performance/lazy-loading/

Riverpod. (n.d.-a). *Riverpod.* Haettu 5.2.2023 osoitteesta https://riverpod.dev/

Riverpod. (n.d.-b). *Getting started.* Haettu 5.2.2023 osoitteesta https://riverpod.dev/docs/getting_started

Riverpod. (n.d.-c). *Providers.* Haettu 5.2.2023 osoitteesta https://riverpod.dev/docs/concepts/providers

Riverpod. (n.d.-d). *.family.* Haettu 6.2.2023 osoitteesta https://riverpod.dev/docs/concepts/modifiers/family

Riverpod. (n.d.-e). *.autoDispose.* Haettu 6.2.2023 osoitteesta https://riverpod.dev/docs/concepts/modifiers/auto_dispose

pub.dev. (n.d.). *ProviderScope class.* Haettu 14.2.2023 osoitteesta https://pub.dev/documentation/flutter_riverpod/latest/flutter_riverpod/ProviderScope-class.html

Liite 1: Aineistonhallintasuunnitelma

Järvisuunnistus-sovelluksen projektikansio säilytetään C-asemalla, ja koodiin tehdyt muutokset ajetaan GitHub-versionhallintajärjestelmään GitHub Desktop-työpöytäsovelluksen kautta. Opinnäytetyötiedostoa säilytetään OneDrive-pilvipalvelussa, sillä se ei sisällä luottamuksellista tai salassapidettävää tietoa. Aineistoa säilytetään yksi vuosi hyväksymispäivän jälkeen.

Opinnäytetyön aikana pidetään päiväkirjaa, johon kirjataan tehdyt työt jokaiselta päivältä. Päiväkirjaa säilytetään OneDrive-pilvipalvelussa, ja sitä säilytetään yksi vuosi opinnäytetyön valmistumisen jälkeen.

Pidetyistä palavereista kirjoitetut muistiinpanot säilytetään OneDrive-pilvipalvelussa. Muistiinpanoihin kirjataan vain lyhyesti tärkeimpiä kohtia, jotka pitää ottaa huomioon työtä tehdessä.

Asiakaspalautepalaveri pidettiin etänä. Palaverin sisällöstä kirjattiin tärkeimmät asiat, joista koostettiin palauteteksti. Teksti lähetettiin asiakkaalle hyväksyttäväksi ennen sen liittämistä opinnäytetyöhön.