Frisbeegolf-sovelluskokonaisuuden kehitys React Nativella ja Node.js:llä

Jimi Kurko

Haaga-Helia ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Opinnäytetyö
2024

Tiivistelmä

Tekijä(t)
Jimi Kurko

Tutkinto
Tradenomi

Raportin/Opinnäytetyön nimi
Frisbeegolf-sovelluskokonaisuuden kehitys React Nativella ja Node.js:llä

Sivu- ja liitesivumäärä
73 + 0

Tämän opinnäytetyön tehtävänä oli suunnitella, kehittää ja julkaista laaja frisbeegolf-sovelluskokonaisuus, jota hyödyntäen käyttäjä pystyy mobiililaitteellaan muun muassa kirjaamaan ylös frisbeegolf-pelikierroksen pisteitä sekä tarkkailemaan pelaamiaan kierroksia.

Sovelluskokonaisuuden kehittämisessä käytettiin React Native-mobiiliohjelmointikehystä käyttäjälle näkyvän frontendin kehityksessä sekä Node.js-ajoympäristöä frontendille dataa syöttävän backendin kehityksessä. Kaikessa koodissa hyödynnettiin JavaScriptiin pohjautuvaa TypeScriptiä, ja ohjelmoinnissa hyödynnettiin VSCode-tekstinkäsittelyohjelmaa. Backendin tietokantana hyödynnettiin PostgreSQL-tietokantaa.

Sovelluskokonaisuus toteutettiin vesiputousmallia hyödyntäen. Toteutus alkoi sovelluksen frontendin, backendin sekä tietokannan perusteellisesta vaatimusmäärittelystä sekä vaatimusmäärittelyyn pohjautuvasta suunnittelusta. Backend, tietokanta sekä frontend toteutettiin luotujen suunnitelmien mukaan, ja sekä frontend, backend että tietokanta julkaistiin tuotantokäyttöön. Kaikki sovelluksen ohjelmakoodi julkaistiin myös GitHubissa.

Tämä opinnäytetyön raportti selostaa koko ohjelmointiprosessin yksityiskohtaisesti alusta loppuun asti. Raportissa annetaan ensin teoriaosuudessa pohjatietoja muun muassa käytetyistä teknologioista sekä joistain ohjelmointiperiaatteista, joita hyödynnettiin vahvasti itse sovelluskokonaisuuden toteutuksessa. Kaikkien sovelluskokonaisuuden osa-alueiden toteutus kuvataan raportissa perusteellisesti havainnollistavien kuvien kera. Lopussa uppoudutaan pohdintaan projektin onnistumisista ja epäonnistumisista sekä jatkokehityksen mahdollisuuksista.

Asiasanat
ohjelmistokehitys, mobiililaitteet, tietokannat, frisbeegolf

Sisällys

1 Johdanto 1
1.1 Sovelluskokonaisuuden sisältö ja sovelluksen käyttö lyhyesti 1
1.2 Opinnäytetyöraportin kulku 2
2 Taustatietoa toteutettavasta projektista 3
2.1 Frisbeegolf lyhyesti 3
2.2 JavaScript ja TypeScript 3
2.3 Mobiilisovellukset ja React Native 4
2.4 Node.js ja Express 5
2.5 PostgreSQL ja Drizzle 5
2.6 Full-stack ohjelmointi 6
2.7 MVC-arkkitehtuuri 6
2.8 VSCode 6
2.9 Eri suunnittelutyökaluja 7
3 Projektin aloittaminen 7
3.1 Frontendin vaatimusmäärittely sekä suunnittelu 7
3.1.1 Frontendin vaatimusmäärittely 7
3.1.2 Frontendin ulkonäön suunnittelu ja prototyyppi 9
3.2 Backendin vaatimusmäärittely sekä suunnittelu 13
3.2.1 Backendin vaatimusmäärittely 13
3.2.2 Backendin sekä tietokannan suunnittelu 14
4 Backendin ja tietokannan toteutus 16
4.1 Node.js-projektin initialisointi 16
4.2 Projektin rakenteen luominen MVC-arkkitehtuurin mukaisesti 16
4.3 Versionhallinta 17
4.4 Paikallisen tietokannan luominen ja Drizzlen lisääminen projektiin 17
4.5 Model-tiedostot 19
4.6 Controller-tiedostot ja virheidenkäsittely 20
4.7 API-päätepisteet 22
4.8 Syötetyn datan validointi 23
4.9 Käyttäjien hallinta ja autentikaatio 24
4.10 Backendin käyttöönotto lokaalisti 29
5 Frontendin toteutuksen aloitus 29
5.1 Mobiilisovelluksen initialisointi 30

5.2 Sovelluksen rakenne ... 30
5.3 Käyttäjänhallinta JWT:ta hyödyntäen ... 31
5.3.1 Zustand-tilanhallintakirjasto käyttäjänhallinnassa ... 31
5.3.2 Axios-instanssin luominen ja refresh tokenien hallinta ... 32
5.4 Services-hakemisto ... 33
6 Sovelluksen näkymät ja komponentit ... 34
6.1 Layout-näkymä ... 34
6.2 Sovelluksen päänäkymä ... 36
6.3 Kirjautumis- ja rekisteröitymisnäkymät ... 39
6.4 Karttanäkymä ... 42
6.5 Pelaajavalintanäkymä ... 47
6.6 Kierrosnäkymä ... 49
6.7 Profiilinäkymä ... 57
7 Sovelluskokonaisuuden julkaisu ... 65
7.1 Backendin julkaisu ... 65
7.2 Frontendin julkaisu ... 66
8 Pohdinta ... 67
8.1 Onnistumiset sekä epäonnistumiset ... 67
8.2 Sovelluskokonaisuuden jatkokehitys ... 68
8.3 Mitä projektista opin? ... 69

1 Johdanto

Suomalaiset ovat hyvin metsäläheistä väkeä, ja se näkyy suomalaisten elämäntavasta sekä harrastuksista. Yksi varsinkin viime vuosina eniten esillä olevista luonnonläheisistä harrastuksista on frisbeegolf. Jo pelkästään Suomessa lajilla on harrastajia jo yli 100 000, ja määrä on jatkuvasti kasvamaan päin (LDG s.a.) Kyseessä on siis suhteellisen suosittu laji.

Lajissa ominaista on golfin tapaan pisteiden laskeminen jotain työkalua käyttäen. Pistetilanteen kirjaaminen paperille on yksi perinteinen tapa pitää kirjaa pelinkulusta, mutta teknologiakeskisessä nyky-yhteiskunnassa puhelimessa oleva helppokäyttöinen pisteidenlaskusovellus tuntuu monen mielestä houkuttelevammalta ajatukselta.

Tässä toiminnallisessa opinnäytetyössäni pyrin toteuttamaan yksinkertaisen mobiilipohjaisen frisbeegolf-pisteidenlaskusovelluksen. Opinnäytetyön tuotoksena syntyvän mobiilisovelluksen on tarkoitus olla varteenotettava vaihtoehto frisbeegolfia harrastaville ihmisille, jotka haluavat hyvin helppokäyttöisen sekä yksinkertaisen pisteidenlaskusovelluksen. Itse lopputuotoksena syntyvä sovelluskokonaisuus ei ole täysin opinnäytetyön keskiössä, vaan tavoitteenani on opinnäytetyöraportissa kuvata eri ohjelmistokehityksen vaiheita sekä mitä teknologioita kussakin vaiheessa käytettiin.

1.1 Sovelluskokonaisuuden sisältö ja sovelluksen käyttö lyhyesti

Tässä toiminnallisessa opinnäytetyössä toteuttamani frisbeegolf-pisteidenlaskun toteuttava sovelluskokonaisuus tulee pitämään sisällään kolme eri osaa: itse käyttäjän näkemän interaktiivisen frontendin, joka on kännykällä toimiva mobiilisovellus, sekä frontendin kanssa tiiviisti toimiva erillinen backend, joka syöttää dataa frontendille ja ottaa vastaan dataa frontendiltä. Backend tulee olemaan yhteydessä tietokantaan, jolle tallennetaan kaikki sovelluksessa tarvittava data, kuten sovellukseen rekisteröityneet käyttäjät sekä pelatut kierrokset.

Mobiilisovelluksella tulee olemaan käyttäjänhallinta, ja halutessaan käyttäjä voi rekisteröityä sovellukseen omalla tunnuksellaan. Sovellusta käyttämällä käyttäjä pystyy helposti valitsemaan radan kartalta ja aloittamaan uuden kierroksen. Kartalla sijaitsevat radat kertovat tietoa radasta, kuten radan osoitteen sekä radalla sijaitsevat väylät. Vaihtoehtoisesti, jos rataa ei löydy kartalta, käyttäjä pystyy itse määrittelemään radan kierrosta varten ja aloittamaan kierroksen.

Seuraavaksi käyttäjä valitsee kierrokselle osallistuvat pelaajat ja aloittaa kierroksen. Jokaisen väylän jälkeen käyttäjä merkitsee jokaisen pelaajan pistemäärän pelaajien heittämälle väylälle. Kierroksen loputtua käyttäjä näkee jokaisen pelaajan heittomäärän ja lopputuloksen. Tarkoituksena

on myös, että pelaaja pystyy halutessaan tallentamaan kierroksen tietokantaan, ja myöhemmin tarkastelemaan pelattua kierrosta.

Pisteidenlaskun lisäksi sovellukseen on myös tarkoituksenani lisätä käyttäjille oma profiilisivu, jolta käyttäjä näkee muun muassa pelaamansa kierrokset. Eri ratoja voi käydä myös karttanäkymässä merkitsemässä jo pelatuiksi. Pelaajan tulee myös pystyä lisäämään muita sovellukseen rekisteröityneitä käyttäjiä kavereiksi ja tarkkailemaan omaa kaverilistaa profiilistaan.

1.2 Opinnäytetyöraportin kulku

Raportissa aion aluksi kuvailla projektissa käytettäviä teknologioita sekä arkkitehtuureita lyhyesti, antaen taustaa projektin toteutukselle tietoperustan muodossa.

Sovelluskokonaisuuden tulen toteuttamaan hyödyntäen projektinhallinnassa vesiputousmallia. Vesiputousmallissa projekti on jaettuna eri selkeisiin vaiheisiin: vaatimusten määrittely, suunnittelu, toteutus, testaus ja lopuksi ylläpito. Vesiputousmalli on hyvä tilanteissa, joissa projekti on ennustettava ja selkeä ja vaatimukset ovat suhteellisen selvät. (Thinkingportfolio 25.7.2016)

Raportin kulku tulee olemaan hyvin vahvasti sidoksissa vesiputousmalliin. Raportissa tullaan ensin kuvailemaan mobiilisovelluksen frontendin vaatimusmäärittely. Vaatimusmäärittelyllä tarkoitetaan dokumentointia, johon kirjataan ylös se, mitä joltain ohjelmistolta vaaditaan eli mitä sen on tarkoitus tehdä halutun lopputuloksen saamiseksi (Salescommunications.fi 17.03.2022). Vaatimusmäärittely tulee siis sisältämään sovelluksen tärkeimmät toiminnot. Vaatimusmäärittelyn jälkeen sovelluksen frontendin ulkonäkö sekä toiminnallisuudet suunnitellaan vaatimusmäärittelyn tulosten mukaisesti, ja tämä dokumentoidaan raporttiin. Frontendille tullaan toteuttamaan suunnitelman mukaisesti oma prototyyppi käyttäen protoilutyökalua.

Seuraavaksi toteutan backendin vaatimusmäärittelyn, joka tulee olemaan hyvin samankaltainen frontendin vaatimusmäärittelyn kanssa. Moni frontendin vaatimuksista tulee olemaan hyvin paljon tekemisissä backendin sekä siihen yhteydessä olevan tietokannan kanssa, ja tähän tulen kiinnittämään erityisen paljon huomiota.

Lopulta siirryn itse backendin sekä tietokannan luomiseen, ja raportti tulee sisältämään suhteellisen tarkat selitykset kehitysprosessin tärkeimmistä vaiheista sekä käytetyistä teknologioista. Sama dokumentointi tullaan toteuttamaan frontendin eri osa-alueille. Tämä raportti tulee olemaan kokonaan suomeksi, mutta itse koodin luomisessa tulen käyttämään englantia vanhan tottumuksen mukaan.

Raportin lopussa raportoin valmiin frontendin sekä backendin julkaisusta tuotantokäyttöä varten. Lopuksi raportissa tulee olemaan pohdintaa toteutetusta sovelluksesta kohtaamisistani onnistumisista ja epäonnistumisista, sekä ehdotuksia mahdollista jatkokehitystä varten.

Backendin ja frontendin koodin raportoinnissa ei luonnollisesti tulla esittämään kaikkea mahdollista, sillä muuten tästä raportista tulisi aivan liian laaja. Pyrin kuitenkin mahdollisimman hyvin kertomaan kaikki projektin koodaamiseen liittyvät pääkohdat pääpiirteittäin.

2 Taustatietoa toteutettavasta projektista

2.1 Frisbeegolf lyhyesti

Frisbeegolfilla on, kuten nimestä saattaa huomata, hyvin paljon yhteistä golfin kanssa. Lajia pelataan frisbeegolf-radoilla, jotka koostuvat väylistä, jotka ovat pelialueita, jotka alkavat tiialueelta ja päätyvät väylämaaliin eri koriin. Pelaajien tavoitteena on saada pelaajan pallon sijaan käyttämä frisbee koriin mahdollisimman pienellä määrää heittoja, heittäen väylää pitkin. Heitot jatkuvat aina siitä mihin kunkin pelaajan kiekko on viime heitolla laskeutunut. Pelaajat pelaavat väylät järjestyksessä ensimmäisestä viimeiseen, ja vähimmällä määrää heittoja radan läpäissyt pelaaja on pelaajajoukkonsa voittaja. (Suomen Frisbeegolfliitto 1.1.2024)

2.2 JavaScript ja TypeScript

JavaScript on yksi maailman käytetyimmistä ohjelmointikielistä. Kyseessä on suhteellisen kevyt Brendan Eichin vuonna 1995 kehittämä skriptauskieli, joka mahdollistaa HTML:n sekä CSS:n ohella sivustojen interaktiivisuuden luomisen. JavaScriptistä tekee tehokkaan ohjelmointikielen sen yksinkertaisuus sekä nopeus – selaimet suorittavat JavaScriptiä suoraan ilman tarvetta kääntäjälle. Sille on luotu myös lukuisia eri kirjastoja auttamaan JavaScript-kehitystä. (Jordana 2024)

TypeScript on taas hieman uudempi, vuonna 2012 luotu JavaScriptin kanssa yhteensopiva ohjelmointikieli. Kieli eroaa JavaScriptistä pääosin sillä, että siihen on rakennettu sisäinen tyyppitarkistusjärjestelmä, mikä antaa koodia kirjoittaessa jatkuvasti tietoa esimerkiksi eri muuttujien tyypeistä sekä funktioiden parametreista, ja ilmoittaa mahdollisista virheistä. Tämä nopeuttaa ohjelmointikehitystä huomattavasti perinteisen JavaScriptin käyttöön verrattuna, sillä iso osa virheistä huomataan heti koodia kirjoittaessa sen sijaan että virheestä tulee ilmoitus vasta ohjelmaa suorittaessa. (Sysart s.a.)

Oheinen kuva illustroi hieman JavaScriptin ja TypeScriptin käyttöä. TypeScript ilmoittaa virheestä punaisella poikkiviivalla ohjelmoijan antaessa String-tyyppisen muuttujan vastaanottavalle funktiolle number-tyyppisen muuttujan. (Kuva 1)

src > JS esimerkki.js > ...

```
1  let nimi = "Pekka"
2  let luku = 5
3
4  function sanoTerve(nimi) {
5      console.log(`Terve ${nimi}`)
6  }
7
8  sanoTerve(luku)
```

src > TS esimerkki.ts > ...

```
1  let nimi: string = "Pekka"
2  let luku: number = 5
3
4  function sanoTerve(nimi: string) {
5      console.log(`Terve ${nimi}`)
6  }
7
8  sanoTerve(luku)
```

Kuva 1. JavaScriptin ja TypeScriptin käyttöä ohjelmistokehityksessä

2.3 Mobiilisovellukset ja React Native

Mobiilisovelluksesta puhuttaessa tarkoitetaan jotain sovellusta, joka on luotu pienen laitteen kuten älypuhelimen tai tabletin käyttö mielessä. Mobiilisovellukset voidaan jakaa kolmeen eri kategoriaan – natiiveihin, verkkopohjaisiin tai hybridisovelluksiin. Natiivisovellus on nimenomaan mobiililaitteelle asennettava, sen ominaisuuksia hyödyntävä sovellus. Natiivisovellukset kirjoitetaan käyttöjärjestelmäkohtaisella (iOS tai Android) ohjelmointikielellä, ja ne toimivat vain niille suunnatuilla alustoilla. Verkkopohjainen mobiilisovellus on mobiililaitteen selaimessa toimiva web-toteutus, joka näyttää ja tuntuu natiivisovellukselta. Kaikki mobiililaitteen ominaisuudet eivät kuitenkaan ole web-pohjaisilla sovelluksilla käytettävissä, ja sovellus saattaa myös toimia hitaammin. Hybridisovellukset ovat molempien natiivisovellusten sekä web-pohjaisten sovellusten välimaastoa: ne kykenevät hyödyntämään puhelimien ominaisuuksia ja toimivat millä tahansa käyttöjärjestelmällä. (Bartosińska 2024)

React.js-kehys on Facebookin kehittämä JavaScript-kehys ja -kirjasto. Sitä voidaan käyttää interaktiivisten käyttöliittymien ja verkkosovellusten rakentamiseen nopeasti ja tehokkaasti huomattavasti vähemmällä koodilla kuin normaalilla JavaScriptillä. Reactin pääajatus on, että koodin osat ovat jaettuna komponentteihin, ja komponenteilla on mahdollista hallita omaa tilaa. Tämän lähestymistavan avulla sovellusten kehittäminen Reactilla on selkeää sekä modulaarista, helpottaen niiden ylläpitoa sekä laajentamista. (Dillemuth 11.9.2023)

React Native on React.js-kehykseen pohjautuva alusta hybridimobiilisovellusten kehittämiseen. React Nativella koodi kirjoitetaan, kuten Reactilla, käyttäen JavaScriptiä (tai TypeScriptiä) ja sillä pystytään hyödyntämään mobiililaitteiden ominaisuuksia kuten kameraa ja mikrofonia. Kyseessä on siis hyvin tehokas kehys, jolla voi tehdä mobiilisovelluksia kerralla iOS- sekä Android-alustoille ilman, että tarvitsee nähdä vaivaa kahden erillisen sovelluksen luomiseen (Ruokangas s.a.)

React Nativea pystyy toteuttamaan hyödyntäen Expoa, joka on avoimen lähdekoodin kehys React Native-sovellusten kehittämiseen. Se helpottaa ja nopeuttaa mobiilisovelluksen kehittämistä tarjoamalla erilaisia työkaluja, kuten mahdollisuuden avata kehitysvaiheessa oleva sovellus kehittäjän omalla mobiililaitteella hyödyntäen puhelimelle ladattavaa Expo Client-ohjelmaa. Expo helpottaa myös puhelimen natiivien toimintojen, kuten kameran, mikrofonin ja sijainnin, käyttöönottoa huomattavasti. (Ighowese s.a.)

2.4 Node.js ja Express

Node.js on palvelimilla toimiva Googlen V8-Javascript-moottoriin perustuva suoritusympäristö, joka käyttää ohjelmointikielenä JavaScriptiä. Express taas on web-sovellusten ohjelmointia Node.js:llä helpottamaan luotu ohjelmointirajapinnan tarjoava kirjasto. Käyttäen Expressiä käyttäjä kykenee muun muassa toteuttamaan backendin (Fullstackopen s.a.).

NPM tai Node Package Manager on Node-ympäristössä toimiva pakettien (package) hallintatyökalu, jolla voi esimerkiksi asentaa ja hallita JavaScript-kirjastoja ja aloittaa uusia projekteja hyödyntäen eri pakkauksia (packages) (Metwalli 2024).

API (Application Programming Interface) on mekanismi, jonka avulla kaksi ohjelmakomponenttia pystyy viestittelemään keskenään käyttäen eri protokollia. Yleinen tapa välittää dataa frontendin (käyttäjälle näkyvä sovellus) ja backendin välillä on API-päätepisteet, joita backend tarjoaa ja joiden kautta frontend hakee ja saa dataa, hyödyntäen REST:iä eli Representational State Transferia, joka mahdollistaa tavan frontendille hakea dataa esimerkiksi GET-, DELETE-, PUT- ja POST-pyynnöillä. (Amazon s.a.)

2.5 PostgreSQL ja Drizzle

Relaatiotietokannassa tietokanta koostuu useista eri taulukoista, joissa tiedot on esitetty eri riveillä ja sarakkeilla. Taulukot liittyvät myös usein jollain tavalla toisiinsa, ja taulukoihin on tallennettuna tieto taulukkojen keskinäisistä yhteyksistä eli relaatioista (Sarja 2006).

PostgresSQL on yritystason avoimen lähdekoodin relaatiotietokannan hallintajärjestelmä. Kyseessä on yksi suosituimmista tietokantojen hallintajärjestelmistä, jolla on jo yli 35 vuotta historiaa takana. Kyseessä on hyvin stabiili sekä monia eri ohjelmointikieliä tukeva hallintajärjestelmä, jonka käyttö on hyvin helppoa. (Geeksforgeeks 2023) StackOverflow:n vuonna 2022 tekemän tutkimuksen mukaan PostgreSQL oli kaikista käytetyin tietokanta ohjelmoijien keskuudessa. 46 prosenttia tutkimuksen kyselyyn vastanneista ohjelmointialan työntekijöistä mainitsivat käyttäneensä tietokantana PostgreSQL:ää. (Linuxpolska 2023)

Drizzle on erittäin kevyt ja moderni ORM (Object Relational Mapping)-kirjasto, joka mahdollistaa helpon käytön PostgreSQL:ään tallennetun datan kanssa käyttäen alustanaan Node.js:ää. Drizzleä käyttäen sovelluskehittäjä pystyy esimerkiksi pelkkää koodia käyttäen määrittämään tietokantakaaviot ja viemään ne tietokantaan Drizzlen avulla ja toteuttamaan monimutkaisiakin kyselyitä tietokantaan koodin avulla (Drizzle s.a.)

2.6 Full-stack ohjelmointi

Full-stack ohjelmoinnista puhuttaessa puhutaan prosessista, jossa suoritetaan ohjelmistoprojekti, jossa sovellukselle toteutetaan sekä käyttäjälle näkyvä frontend että näkymätön palvelinpuoli backend, joka on usein myös yhteydessä tietokantaan. Frontend pitää sisällään koodin, joka liittyy käyttäjän vuorovaikuttamiseen sovelluksen kanssa, ja tarjoaa käyttöliittymän, jonka kautta käyttäjä pystyy toteuttamaan eri toimintoja. Backend taas toimii tietokantaan yhteydessä olevana palvelimena, joka kommunikoi frontendin kanssa ja prosessoi sen syöttämää ja sille syötettävää dataa. (Amazon s.a.)

2.7 MVC-arkkitehtuuri

MVC-arkkitehtuurillta tarkoitetaan perinteistä ohjelmistokehityksen suunnittelumallia, jossa sovelluksen esitys- ja logiikkakerrokset ovat täysin eristettyinä toisistaan ja eri vastuualueita on delegoituna eri osa-alueille: mallille (model), ohjaimelle (controller) ja näkymälle (view). Ajatuksena on, että malli hakee, muokkaa ja lisää dataa vuorovaikuttaen tietokannan kanssa. Ohjain toimii näkymän ja mallin välissä, välittäen dataa mallilta näkymälle. Näkymällä tarkoitetaan usein käyttäjälle näkyvää käyttöliittymää, mutta kyseessä voi olla myös JSON-muotoinen representaatio datasta. (Hurja 2023)

2.8 VSCode

VSCode on Microsoftin kehittämä ilmainen avoimen lähdekoodin hyvin tehokas ja selkeä tekstinkäsittelyohjelma. Ohjelmoija voi halutessaan käyttää pelkästään sitä kaikkien projektin osa-

alueiden työstämiseen, myös tapauksissa, joissa ohjelmointikieli eroaa projektin osa-alueiden välillä. VSCodelle voi asentaa monia eri lisäosia helpottamaan ohjelmointia ja käyttää eri ominaisuuksia kuten Snippetiä, joka nopeuttaa koodin luomista. (Manninen, Mepham 16.2.2022)

2.9 Eri suunnittelutyökaluja

Projektien suunnittelua varten on olemassa monia eri varteenotettavia web-sovelluksia.

Figma on frontendin ulkonäön sekä interaktiivisuuden suunnitteluohjelma, jolla pystytään suunnittelemaan verkkosivustojen sekä mobiilisovellusten käyttöliittymiä ilman tarvetta koodaamiselle. Figmalla voidaan protoilla erilaisia näkymiä suunnitteilla olevalle sovellukselle, ja lisätä siihen esimerkiksi painikkeita, jotka simuloivat lopputuloksena syntyvää sovellusta. Tarkoituksena on saada aitoa tuntumaa sovelluksen lopputuloksesta (P 14.10.2021).

Lucidchart on ilmainen ja helppokäyttöinen diagrammien luontisovellus, jota voi käyttää esimerkiksi tietokantakaavioiden luomiseen. Sillä voi myös esimerkiksi luoda hyvin karkeita prototyyppejä sivustoista suunnittelun alkuvaiheessa. (Innovationtraining s.a.).

3 Projektin aloittaminen

3.1 Frontendin vaatimusmäärittely sekä suunnittelu

Projektin aloittaminen alkoi vesiputousmallin mukaisesti vaatimusmäärittelyllä sekä suunnittelulla. Päätin aloittaa frontendin vaatimusmäärittelystä, jotta sain luotua itselleni hyvän idean siitä, mitä toiminnallisia sekä ei-toiminnallisia ominaisuuksia sovelluksella tulisi olla. Toiminnallisilla vaatimuksilla tarkoitetaan vaatimuksia ominaisuuksista, jotka käyttäjä havaitsee. Ei-toiminnalliset vaatimukset ovat taas vaatimuksia, jotka määrittelevät miten sovelluksen tulisi toimia sisäisesti (Visuresolutions.com s.a.).

3.1.1 Frontendin vaatimusmäärittely

Kuvassa (Kuva 2) näkyvät vaatimukset ovat kaikki toimintoja, mitkä määrittelevät sovelluksen ytimen ja sen, mitä sen odotetaan käyttäjän näkökulmasta tekevän. Toiminnallisten vaatimusten perusteella loin myös listan käyttötapauksista, jotka pohjautuvat kyseisiin vaatimuksiin (Kuva 3). Käyttötapauksilla tarkoitetaan käyttäjän ja järjestelmän välistä vuorovaikutuksia, jotka käyttäjä

ensin suorittaa toiminnoillaan, jotta sovellus tuottaa jonkin hänelle jonkin hänen tavoittelemansa toiminnon (Saavutettavuusmalli.hel.fi 4.3.2021).

Toiminnalliset vaatimukset
Toiminto
Rekisteröityminen sovellukseen
Kirjautuminen sovellukseen
Kirjautuminen ulos sovelluksesta
Omien kierrosten/pelattujen ratojen tarkastelu profiilissa
Kavereiden/pelaajien lisääminen sekä tarkastelu profiilissa
Ratojen tarkastelu kartalla
Radan valitseminen kartalta/oman radan määrittäminen
Pelaajien valitseminen, uusien pelaajien lisääminen
Kierroksen aloittaminen
Pisteiden lasku kierroksen aikana, pistetilanteen tarkastelu
Kierroksen tallentaminen

Kuva 2. Frontendin toiminnallisten vaatimusten määrittely

Käyttötapaukset	
Tapaus	Näkymä
Käyttäjä painaa nappia rekisteröityäkseen, täyttää lomakkeen ja rekisteröityy sovellukseen	Päänäkymä
Käyttäjä painaa nappia kirjautuakseen sovellukseen, täyttää kirjautumistiedot ja kirjautuu sovellukseen	Päänäkymä
Käyttäjä painaa nappia kirjautuakseen ulos sovelluksesta, käyttäjä uloskirjautuu	Päänäkymä
Käyttäjä menee päänäkymän kautta profiiliin, tarkastelee kavereita ja lisää niitä ensin hakemalla käyttäjää ja sitten lähettämällä kaveripyynnön	Päänäkymä -> Profiili
Käyttäjä näkee profiilissa saadut kaveripyynnöt, kykenee hyväksymään niitä	Profiili
Käyttäjä painaa nappia päänäkymässä josta hän pääsee tarkastelemaan ratoja kartalla, yksityiskohtaisia tietoja jokaisesta radasta	Päänäkymä -> Radat
Käyttäjä painaa nappia josta alkaa kierros, valitsee radan ratanäkymästä, valitsee pelaajat, aloittaa uuden kierroksen	Päänäkymä -> Radat -> Pelaajavalinta -> Kierros
Käyttäjä painaa nappia josta alkaa kierros, painaa custom-nappia josta hän voi itse määritellä radan väylineen, valitsee pelaajat, aloittaa uuden kierroksen	Päänäkymä -> Radat -> Custom -> Pelaajavalinta -> Kierros
Kierroksen alettua käyttäjä merkitsee pisteet jokaiselle pelaajalle per väylä	Kierros
Käyttäjä pystyy tarkastelemaan pistetilannetta kierroksen aikana	Kierros
Kierroksen loputtua käyttäjä näkee pistetilanteen ja pystyy tallentamaan halutessaan kierroksen	Kierros

Kuva 3. Toiminnallisiin vaatimuksiin pohjautuvia yksinkertaisia käyttötapauksia

Määrittelin frontendille myös ei-toiminnallisia vaatimuksia (Kuva 4). Myöhemmin toteutettavassa backendin vaatimusmäärittelyssä tullaan hyödyntämään joitain frontendin ei-toiminnallisessa vaatimusmäärittelyssä esiin tulleita asioita, kuten huomio käyttäjähallinnan tietoturvallisuuden tärkeydestä.

Ei-toiminnalliset vaatimukset
Käyttöliittymän tulee olla helppokäyttöinen
Sovelluksen tulee toimia kaikilla alustoilla
Sovelluksen tulee olla skaalautuva ja kykenevä käsittelemään suurta määrää dataa kerralla
Sovelluksen ja komponenttien ulkonäkö tulee olla silmää miellyttävä
Käyttäjähallinnan tulee olla tietoturvallinen

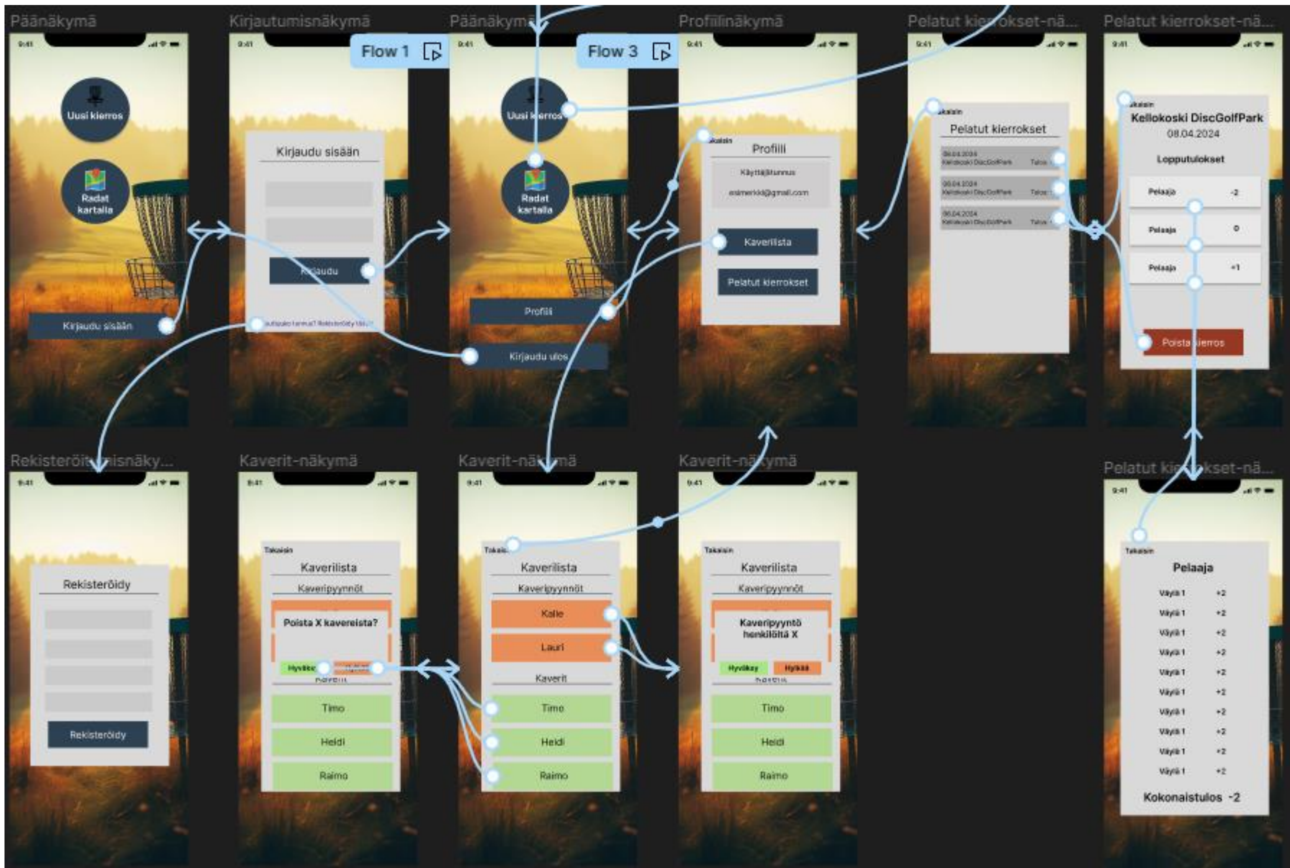
Kuva 4. Ei-toiminnallisia vaatimuksia

3.1.2 Frontendin ulkonäön suunnittelu ja prototyyppi

Vaatimusmäärittelyn jälkeen siirryin toteuttamaan sovelluksen ulkonäön suunnittelun. Työkaluksi suunnitteluun valitsin Figman. Tähän käyttötarkoitukseen kyseinen työkalu on erittäin kätevä, sillä pääsin sillä myös simuloimaan eri näkymistä siirtymistä toiseen melko vaivatta, ja periaatteessa pystyin pelkkää Figmaa käyttäen luomaan täydellisen prototyypin koko sovelluksesta ja se antoi todella hyvän idean siitä minkälaista sovellusta olin lähdössä tekemään.

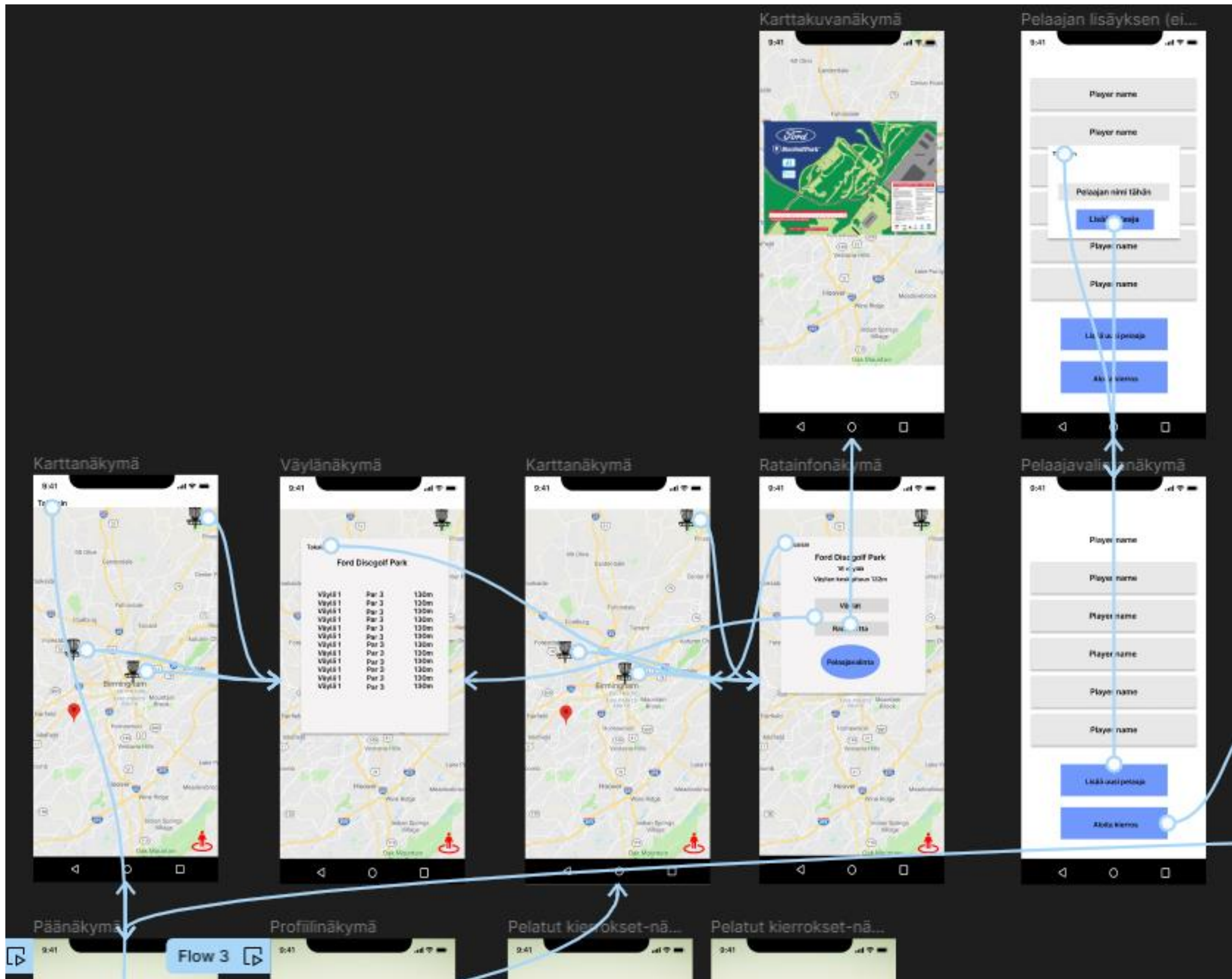
Frontendin vaatimusmäärittelyn ja sen pohjalta tehtyjen käyttötapausten sekä ei-toiminnalliset vaatimukset huomioon ottaen syntyi sovelluksesta lähes kaikki toiminnallisuudet esittävä prototyyppi. Prototyyppi kuvaa koko sovelluksen toimintaa sovelluksen käynnistämisestä yhden pelatun kierroksen lopettamiseen. Se sisältää myös muita sivutoimintoja, kuten profiilin tarkastelua. Kaikki käyttötapauksissa olevat käyttäjän mahdollisesti tekemät toiminnot on otettu prototyyppiä luodessa huomioon. Käyn läpi prototyypin osio kerrallaan.

Kuvassa 5 näkyy päänäkymän sekä profiilin prototyyppi. Nuolet kuvaavat navigointia näkymistä toisille. Prototyyppi kuvaa käyttäjän kirjautumis- sekä rekisteröitymisprosessia sekä oman profiilin tarkkailua.



Kuva 5. Prototyyppi päänäkymästä sekä profiiliosiosta

Kuvassa 6 näkyy prototyypin osa, joka kuvaa karttanäkymän käyttöä sekä hieman kierroksen aloittavan käyttäjän pelaajavalinnan käyttöä.



Kuva 6. Prototyyppi karttanäkymästä sekä pelaajavalinnasta

Viimeinen osa käyttöliittymän prototyyppiä on itse sovelluksen ydin, eli pelaajavalinnan jälkeinen pisteidenlaskentaosuus (Kuva 7). Prototyypin osat kuvaavat väylältä toiseen siirtymistä, pelaajien pisteiden merkkaamista, pistetilanteen tarkastelua sekä kierroksen lopettamista, joka ohjaa käyttäjän takaisin päänäkymään.

Kuva 7. Prototyypin pisteidenlaskentaosuus

Sovelluksen taustakuvan luomisessa hyödynsin Bingin tarjoamaa DALL-E 3-tekoälyä. Kyseessä on kuvia generoiva OpenAI:n luoma tekoälymalli, jonka avulla käyttäjä voi luoda omia kuvia omien avainsanojen avulla (Pykes, 1.11.2023).

Tarkoituksena oli toteuttaa frontend hyödyntäen React Nativea. Sovelluksen on tarkoitus olla käytettävissä kaikilla eri mobiilialustoilla, ja sovellusta ei voinut toteuttaa verkkopohjaisesti, sillä tarkoituksena oli hyödyntää sovelluksessa joitain mobiililaitteiden natiiveja ominaisuuksia.

Puhelimelle asennettava sovellus on myös tämänkaltaiselle pisteidenlaskuohjelmalle paras ja kätevin muoto selaimella toimivan sovelluksen sijaan.

3.2 Backendin vaatimusmäärittely sekä suunnittelu

Frontendin vaatimusmäärittelyn ja suunnittelun pohjalta oli hyvä lähteä tekemään samoja asioita backendille. Backendin on tarkoituksena tarjota kaikki frontendin tarvitsema data käyttäen hyödyksi tietokantaa, jolle kaikki tieto kuten käyttäjien kaverit sekä järjestelmään lisätyt radat ovat tallennettuina.

3.2.1 Backendin vaatimusmäärittely

Kuvissa 8 ja 9 ovat listattuina kaikki backendille asettamani toiminnalliset sekä ei-toiminnalliset vaatimukset.

Backendin toiminnalliset vaatimukset
Toiminto
Käyttäjien rekisteröinti ja talletus
Ratojen lisäys, poisto ja muokkaus
Ratojen haku
Käyttäjäkohtaisten kavereiden lisäys ja poisto
Pelattujen kierrosten lisäys ja poisto

Kuva 8. Vaatimusmäärittely backendin toiminnallisista vaatimuksista

Backendin ei-toiminnalliset vaatimukset
Suojaus HTTPS-yhteydellä
Skaalautuva, pystyy käsittelemään paljon dataa kerralla
Kunnollinen virheenkäsittely virheiden sattuessa
Sovelluksen ja komponenttien ulkonäkö tulee olla silmää miellyttävä
Käyttäjähallinnan tulee olla tietoturvallinen

Kuva 9. Ei-toiminnallinen vaatimusmäärittely backendille

3.2.2 Backendin sekä tietokannan suunnittelu

Backend suunniteltiin toteutettavaksi hyödyntäen Node.js:ää ja Expressiä. Itseltäni löytyy jo valmiiksi kokemusta kyseisten teknologioiden käytöstä, joten valinta tuntui luonnolliselta.

Tarkoituksena oli kytkeä frontend ja backend keskenään API:n avulla, ja backendin tuli toimia REST API:na, joka tarjoaa eri API-päätepisteitä frontendille, joihin frontend tekee erityyppisiä pyyntöjä datan hakua, muokkausta, lisäystä tai poistoa varten. Kyseiset päätepisteet piti ensin suunnitella, jotta backendin toteuttamista varten sain hyvän kuvan siitä mitä olin toteuttamassa (Kuva 10).

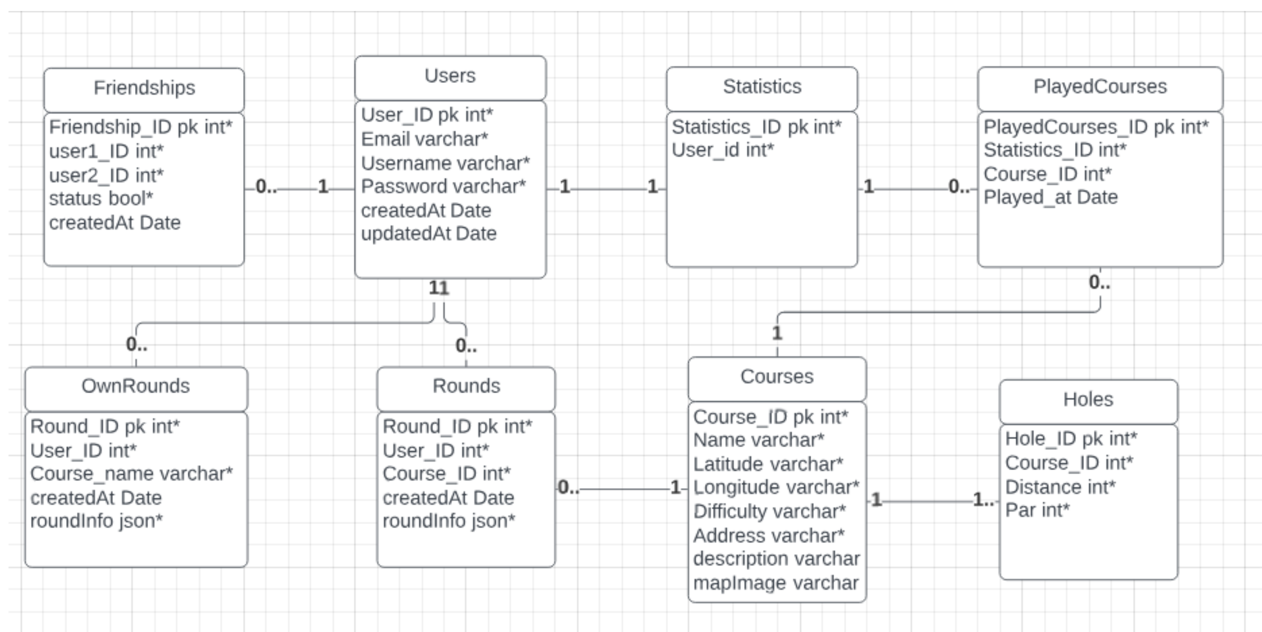
Metodi	Endpoint	Tarkoitus
GET	/api/courses	Kaikkien ratojen hakeminen
POST	/api/courses	Uuden radan lisäys
GET	/api/courses/:id	Tietyn radan hakeminen id:n perusteella
DELETE	/api/courses/:id	Tietyn radan poistaminen id:n perusteella
PUT	/api/courses/:id	Tietyn radan muokkaaminen id:n perusteella
GET	/api/rounds	Kaikkien pelaajien pelaamien kierrosten haku
POST	/api/rounds	Pelatun kierroksen lisääminen
GET	/api/users/:id/rounds	Tietyn pelaajan pelattujen kierrosten haku
DELETE	/api/users/:id/rounds/:roundId	Tietyn radan poisto
GET	/api/users	Kaikkien käyttäjien hakeminen
POST	/api/users	Uuden käyttäjän lisäys
GET	/api/users/:id	Tietyn käyttäjän haku
PUT	/api/users/:id	Tietyn käyttäjän muokkaus
GET	/api/users/:id/friend-requests	Kaikkien kaveripyyntöjen haku käyttäjäid:llä
GET	/api/users/:id/friendships	Kaikkien kaverien haku käyttäjäid:llä
POST	/api/users/:id/friend-requests	Kaveripyynnön lisäys käyttäjäid:llä
PUT	/api/users/:id/friend-requests/:id	Kaveripyynnön muokkaus (hyväksyminen/hylkäys)
GET	/api/users/:id/played-courses	Tietyn pelaajan pelattujen ratojen haku käyttäjäid:llä
POST	/api/users/:id/played-courses	Pelatun radan lisäys tietylle pelaajalle käyttäjäid:llä
DELETE	/api/users/:id/played-courses/:courseId	Pelatun radan poistaminen pelatuista radoista id:llä

Kuva 10. Suunnitelma backendin tarjoamista päätepisteistä

Päätepisteiden perusteella oli hyvä lähteä suunnittelemaan backendille omaa tietokantaa, johon kaikki sovelluksessa tarvittava tieto tallennetaan.

Valitsin PostgreSQL:in käytettäväksi tässä projektissa, sillä minulta löytyi jo valmiiksi jonkin verran kokemusta relaatiotietokantojen hallinnasta käyttäen kyseistä tietokantojen hallintajärjestelmää.

Tietokannan suunnittelussa sekä relaatiokaavion luomisessa hyödynsin Lucidchart-websovellusta. Ensin loin tietokannasta hyvin karkean relaatiokaavion. Hyödyntäen karkeaa suunnitelmaa lisäsin kullekin taulukolle omat attribuutit ja tarkensin taulukoiden keskinäisiä yhteyksiä, normalisoiden tietokantakaavion (Kuva 11). Tarkoituksenani oli tallentaa data kierroksista JSON-tyyppisenä datana Rounds-taulun roundInfo-attribuuttiin, sillä en nähnyt tarvetta luoda erillisiä tauluja koska kyseistä dataa ei tarvitse ikinä muokata, ja datan poisto kohdistuisi muutenkin koko Rounds-tauluun. PlayedCourses- sekä Users-taulujen välille loin ”Statistics”-taulun jotta mahdollisten statistiikkataulujen lisääminen tietokantaan olisi tulevaisuudessa helpompaa. Käyttäjien itse määrittelemillä radoilla pelatuille kierroksille loin oman ”OwnRounds”-taulun, jotta ”Rounds”-taulusta ei tulisi turhaan liian monimutkaista.



Kuva 11. Tietokannan normalisoitu relaatiotietokannan kaavio

Käyttäjien hallinnan suunnittelin toteutettavaksi käyttäen JSON Web Tokeneita tai JWT:ita. JWT:t ovat JSON-tyyppisiä digitaalisesti allekirjotettuja informaatiota sisältäviä objekteja, joiden avulla voi turvallisesti välittää dataa kahden osapuolen välillä (Babladi s.a.). Ajatuksenani oli, että käyttäjä syöttää frontendin kautta backendille salasanan, ja backend vertailee annettua sanaa käyttäjän

taulussa sijaitsevaan kryptattuun versioon salasanasta. Tämän perusteella käyttäjälle lähetetään backendin täyttämä JWT, jota käytetään kirjautuneen käyttäjän todentamiseen seuraavissa pyynnöissä.

4 Backendin ja tietokannan toteutus

Vaikka toteutin ensin vaatimusmäärittelyn sekä suunnittelun frontendille, päätin aloittaa luontiprosessin backendistä sekä tietokannasta. Frontend tulee olemaan niistä hyvin riippuvainen, ja frontendin kehittämisvaiheessa on hyödyllistä lähettää pyyntöjä backendille ja ottaa vastaan pyyntöjä backendiltä, jotta samalla pystyy varmistamaan, että kaikki toimii niin kuin pitäisi. Koodaustyökaluna käytin backendin toteuttamisessa jo aiemmin paljon käyttämääni VSCode-tekstinkäsittelyohjelmaa.

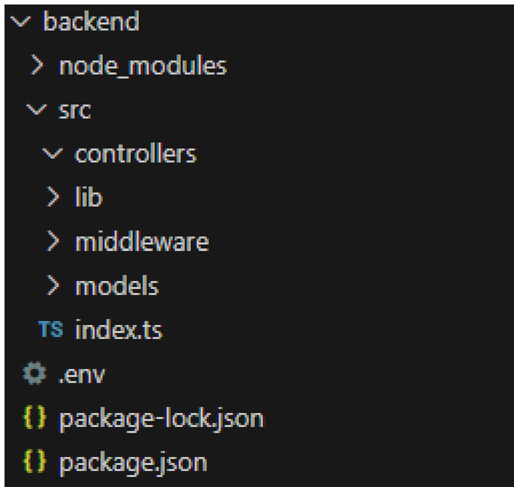
4.1 Node.js-projektin initialisointi

Loin aluksi hakemiston, jolle on tarkoitus lisätä frontend sekä backend. Loin backendille oman hakemiston, jossa käytin NPM:n tarjoamaa “npm init -y” komentoa, joka generoi hakemiston juureen oman ”package.json”-tiedoston, joka toimii ikään kuin ohjelman sydämenä, josta selviää muun muassa kaikki eri projektissa käytetyt riippuvuudet (dependencies) sekä eri ohjelman käynnistämiseen liittyvät skriptit.

Oletuksena Node.js ympäristössä ohjelmointikielenä on käytössä JavaScript. Halusin kuitenkin hyödyntää projektissa TypeScriptiä, joten asensin siis NPM:llä TypeScriptin ja alustin projektiin ”tsconfig.json”-tiedoston, mikä mahdollisti TypeScriptin käytön projektissa.

4.2 Projektin rakenteen luominen MVC-arkkitehtuurin mukaisesti

Seuraavaksi vuorossa oli jonkinlaisen järkevän projektirakenteen luominen ottaen huomioon MVC-arkkitehtuurin. Syntyi alustava projektin rakenne (Kuva 12). Kaikki projektin lähdekoodi oli tarkoitus sijoittaa ”src”-hakemistoon, johon oli erikseen luotuna omat hakemistot ohjaimille sekä malleille. Mallien (models) oli tarkoitus suorittaa toimintoja tietokannan kanssa, ja ohjaimien (controllers) oli tarkoitus käyttää malleja ja prosessoida backendiä käyttävän käyttäjän pyyntö. ”Lib”-hakemistoon tuli kaikenlainen muu data, kuten TypeScript-tyypit tai hyödyllisiä koko projektissa hyödynnettäviä funktioita. ”Middleware”-hakemistoon tuli ohjaimiin liitettäviä funktioita, jotka jollain tapaa käsittelevät käyttäjältä tulevaa pyyntöä. Laittamalla eri osat omiin kansioihin tavoitteenani oli tehdä projektista mahdollisimman selkeä sen hetkistä kehitystä sekä mahdollista jatkokehitystä varten.



Kuva 12. Projektin rakenne

4.3 Versionhallinta

Päätin luoda projektille oman versionhallinnan hyödyntäen GitHubia. GitHub on palvelu, johon käyttäjä pystyy tallentamaan eri repositorioihin omien projektiensa lähdekoodia ilmaiseksi, ja jakamaan sitä muiden käyttäjien kesken (Linux s.a.). Loin projektille oman repositorion, käyttäen nimeä ”kiekotus-backend”. GitHubilla on tarjolla monia eri ominaisuuksia kuten mahdollisuus luoda monia eri haaroja projektille, mutta koska kyseessä on minun yksin toteuttamani projekti päädyin luomaan vain yhden ”development”-haaran, jolle aikomuksenani oli tallentaa muuttamani projekti samalla kun varmasti toimiva versio elää koskemattomana main-haarassa.

4.4 Paikallisen tietokannan luominen ja Drizzlen lisääminen projektiin

Alkaessani itse hommiin backendin luonnissa päätin aloittaa tietokannasta. Backendin ollessa tuotannossa se luonnollisesti käyttää jotain hostattua tietokantaa, mutta kehitystarkoituksessa loin oman paikallisen PostgreSQL-tietokannan hyödyntäen PostgreSQL:n omaa komentoriviä, jonka avulla loin uuden tietokannan “kiekotusdbdevelopment” omalle tietokoneelleni. Kaikki tulevat tietokantakyselyt kehitysvaiheessa olevalta backendiltä tulivat kohdistumaan kyseiselle tietokannalle.

Tietokannan hallinnassa Node.js:llä olen aikaisemminkin käyttänyt ORM-työkaluna Drizzleä ja kokenut sen hyväksi vaihtoehdoksi. Asensin sen projektiin NPM:ää hyödyntäen, ja lisäsin sille oman kansion ”Lib”-hakemistoon. Loin Drizzlelle oman index.ts tiedoston, joka suoritetaan aina

backendin käynnistyessä. Tiedosto luo oman tietokantamuuttujan, jota myöhemmin käytetään muualla koodissa kyselyjen tekemiseen itse tietokantaan.

Seuraavaksi loin Drizzlelle oman ”schema.ts”-tiedoston, jossa loin aikaisemman esitetyn tietokannan relaatiokaavion (Kuva 11) perusteella TypeScriptin avulla tietokantataulut relaatioineen yhdessä tiedostossa. Esimerkissä (Kuva 13) on esitettynä ”courses”-taulun sekä siihen liittyvän ”holes”-taulun sekä molempien taulujen relaatioiden luominen.

```
export const courses = pgTable("courses", {
    id: serial("id").primaryKey(),
    name: text("name").notNull(),
    latitude: text("latitude"),
    longitude: text("longitude"),
    difficulty: text("difficulty"),
    address: text("address"),
    description: text("description"),
    mapImage: text("map_image")
})

export const courseRelations = relations(courses, ({ many }) => ({
    holes: many(holes),
    statistics: many(statistics),
    rounds: many(rounds)
}))

export const holes = pgTable("holes", {
    id: serial("id").primaryKey(),
    distance: integer("distance"),
    par: integer("par").notNull(),
    courseId: integer("course_id")
})

export const holeRelations = relations(holes, ({ one }) => ({
    course: one(courses, {
        fields: [holes.courseId],
        references: [courses.id]
    })
}))
```

Kuva 13. Osa ”schema.ts”-tiedostoa, jossa kuvataan ”courses”-taulu ja ”holes”-taulu relaatioineen

Seuraavaksi loin Drizzlelle oman ”migration.ts”-tiedoston, jossa konfiguroin tavan, miten Drizzle vie ”schema.ts”-tiedoston datan tietokantaan. Määrittelin ”package.json”-tiedostoon skriptin jolla kykenin generoimaan schema.ts:ssa määritellyn koodin perusteella oikeantyyppiset SQL-kyselyt ja viemään ne paikalliseen tietokantaan. Suoritettuani kyseisen skriptin PostgreSQL:llä oli siis kaikki

tarvittavat taulut luotuna omalla tietokoneellani sijaitsevassa paikallisessa tietokannassa, ja tietokanta oli valmiina ottamaan vastaan kyselyitä.

4.5 Model-tiedostot

Päätin jatkaa backendin toteutusta luomalla MVC-arkkitehtuuria mukaillen ”model”-tason, eli osan koodista, joka hakee ja manipuloi tietokannalla olevaa dataa. Loin käyttäjille, kierroksille, radoille, kaverisuhteille sekä statistiikalle omat tiedostot ”models”-hakemistoon. Tiedostoissa määrittelin eri funktioita, joilla käsittelin tietokannalla sijaitsevaa dataa eri tavoin, käyttäen aiemmin Drizzlellä määrittelemääni ”db”-muuttujaa, joka antaa mahdollisuuden tehdä pyyntöjä tietokantaan. Oheisessa kuvassa (Kuva 14) on koodia ”course.model.ts”-tiedostosta. GetSingleCourse hakee annetun id:n perusteella tietokannasta id:tä vastaavan radan. GetAllCourses hakee kaikki mahdolliset radat. AddCourse lisää uuden radan annetun ”CourseType”-tyyppisen JSON-objektin perusteella, ja lisää ”addHole”-funktion avulla uusia reikiä tietokantaan, joilla on yhteys lisättävään rataan. DeleteCourse poistaa id:n mukaisen radan.

```
export const getSingleCourse = async (id: number) => {
    const course = await db.query.courses.findFirst({
        where: eq(courses.id, id), with: { holes: { columns: { distance: true, par: true } } }
    })

    return course ? course : "No course found with that id"
}

export const getAllCourses = async () => {
    return await db.query.courses.findMany()
}

export const addCourse = async (input: CourseType) => {
    const course = await db.insert(courses).values(input).returning()
    if (input.holes) {
        for (let hole of input.holes) {
            await addHole(hole, course[0].id)
        }
    }
    return course[0]
}

export const deleteCourse = async (id: string) => {
    const course = await db.delete(courses).where((eq(courses.id, Number(id)))).returning()
    return course[0]
}
```

Kuva 14. Eri funktioita ”course.model.ts”-tiedostosta

4.6 Controller-tiedostot ja virheidenkäsittely

Seuraavaksi loin aikaisemmassa vaiheessa luodun päätepiste-listaa (Kuva 10) hyödyntäen jokaiselle tulevalle päätepisteelle omat ohjaimet eli controllerit. Käyttäjän lähettämä pyyntö ohjataan omalle käsittelijälle eli handlerille, joka käsittelee pyynnön ja palauttaa vastauksen, mahdollisesti JSON-tyyppisenä oliona.

Kuvassa (Kuva 15) näkyy ratoihin liittyviä handler-funktioita. GetSingleCourseHandler nappaa käyttäjän pyynnössä ilmoittaman radan id:n ja hyödyntää course.model.ts:ssä määriteltyä ”getSingleCourse”-funktiota yhden tietyn radan hakuun tietokannasta edellä mainitun id:n perusteella. Samaa periaatetta hyödynsin muissa käsittelijäfunktioissa.

```
export const getSingleCourseHandler = async (req: Request, res: Response, next: NextFunction) => {
    try {
        const id = Number(req.params.id)
        const course = await getSingleCourse(id)
        return res.status(200).send(course)
    } catch (error) {
        logger.error(error)
        next(error)
    }
}

export const getAllCoursesHandler = async (req: Request, res: Response, next: NextFunction) => {
    try {
        const courses = await getAllCourses()
        return res.status(200).send(courses)
    } catch (error) {
        logger.error(error)
        next(error)
    }
}

export const postCourseHandler = async (req: Request, res: Response, next: NextFunction) => {
    try {
        const newCourse = await addCourse(req.body)
        return res.status(200).send(newCourse)
    } catch (error) {
        logger.error(error)
        next(error)
    }
}
```

Kuva 15. Eri funktioita ”course.controller.ts”-tiedostosta

Käsittelijäfunktioihin lisäsin myös virheidenkäsittelyn, sillä model-funktioissa tai muualla saattaa tiedon käsittelyn aikana ilmeentyä virheitä, ja on tärkeää, että ne käsitellään asianmukaisesti.

Try-catch block on hyvin tärkeä osa Node.js-ohjelmia, sillä sen avulla voidaan välttää ohjelman kaatuminen tilanteessa, jossa ohjelma ei tiedä mitä tehdä virheellä. Expressissä suositellaan

määrittelemään virheenkäsittelijä-middlewaren, joka ottaa ohjelmassa ilmoitetut virheet vastaan ja prosessoi ne asianmukaisesti (Expressjs 2017). Loin siis ”middlewares”-hakemistoon sovellukselle oman ”errorHandler”-funktion (Kuva 16), joka ottaa virheet vastaan ja tarkistaa, onko virhe tyyppiä AppError. Jos virhe ei ole kyseistä tyyppiä, niin se palauttaa geneerisen 500-statuskoodin ja virheen JSON-muotoisena viestinä. AppError (Kuva 17) on erikseen luomani virheluokka, jota pystyin hyödyntämään ”model”-funktioissa ohjelman ilmoittaessa yleisimmistä virheistä, kuten väärästä id:stä tai käyttäjästä.

```
export const errorHandler = async (error: any, res: Response) => {

    if (error instanceof AppError) {
        return res.status(error.statusCode).json({ message: error.message, errorCode: error.errorCode })
    }

    return res.status(500).json(error)
}
```

Kuva 16. ”Errorhandler”-middleware

```
export class AppError extends Error {
    statusCode: number
    errorCode: number
    constructor(message: string, statusCode: number, errorCode: number) {
        super(message)
        this.statusCode = statusCode
        this.errorCode = errorCode
    }
}
```

Kuva 17. ”AppError”-luokka

Aikaisemmassa kuvassa (Kuva 15) näkyy suoritettavat toiminnot virheiden sattuessa (sijoitettuna catch-blockiin). Loin ”utils”-hakemistoon oman ympäri sovellusta hyödynnettävän ”logger”-funktion, joka käyttää ”Pino”-nimistä kirjastoa tapahtumien kunnollisten lokitietojen ilmoittamiseen perinteisen JavaScriptin sisäisen ”console.log”-funktion sijaan. Jos käsittelijäfunktio huomaa minkä tahansa virheen tapahtuneen, siirtää se ”next”-funktion avulla virheen käsiteltäväksi ”errorHandler”-funktiolle.

4.7 API-päätepisteet

Controller-tiedostojen perusteella käyttäen hyödyksi suunnitelmaa päätepisteistä (Kuva 10) loin ”src”-hakemistoon uuden ”routes.ts”-tiedoston, jossa määrittelin jokaiset päätepisteet. Jokaiselle päätepisteelle määrittelin käsittelijäfunktion, jolle päätepisteelle tehty pyyntö ohjautuu (Kuva 18). Esimerkkinä sovelluksen tai käyttäjän tehdessä pyynnön https://backendin-osoite.com/api/courses/5 ohjautuu pyyntö getSingleCourseHandlerille, joka käsittelee pyynnön käyttäen osoitteessa ilmoitettua ID:tä 5 radan hakemiseen.

```
// Courses
app.get("/api/courses/:id", getSingleCourseHandler)
app.get("/api/courses", getAllCoursesHandler)
app.post("/api/courses", postCourseHandler)
app.delete("/api/courses/:id", deleteCourseHandler)

// Holes
app.post("/api/courses/:id", postHoleHandler)
app.put("/api/courses/:id/:holeId", postHoleHandler)

// Rounds
app.get("/api/rounds/", getallRoundsHandler)
app.get("/api/rounds/:userId", getRoundsHandler)
app.delete("/api/rounds/:id", getRoundsHandler)
app.post("/api/rounds", postRoundHandler)

// Users
app.get("/api/users", getAllUsersHandler)
app.get("/api/users/:id", getSingleUserHandler)
app.post("/api/users", postUserHandler)
app.put("/api/users/:id", editUserHandler)
app.delete("/api/users/:id", deleteUserHandler)

// Statistics
app.get("/api/users/:id/statistics", getUserStatisticsHandler)
app.post("/api/users/:id/statistics/played-courses", postNewPlayedCourseHandler)
app.delete("/api/users/:id/statistics/played-courses/:courseId", deletePlayedCourseHandler)

// Friendships
app.post("/api/users/:userId/friend-requests", addFriendHandler)
app.put("/api/users/:userId/friend-requests/:id", handleFriendRequestHandler)
app.get("/api/users/:userId/friend-requests", getAllFriendRequestsHandler)
app.get("/api/users/:userId/friendships", getAllFriendsHandler)
```

Kuva 18. ”routes.ts”-tiedoston API-päätepisteitä

4.8 Syötetyn datan validointi

Missä tahansa sovelluksessa sille syötetyn datan validointi on elintärkeää bugien, tietoturvariskien sekä muiden odottamattomien tapahtumien välttämiseksi. Yksi tehokkaista tavoista validoida syötetty JSON-data on Zod-kirjaston avulla. Zod mahdollistaa schema-tyyppisten tiedostojen luomisen, joissa olevien objektien avulla sovelluskehittäjät pystyvät tarkastamaan onko esimerkiksi lomakkeen avulla syötetty data oikean mallista, ennen kuin data annetaan ohjelman funktioille käsiteltäväksi. (Mehta 2023)

Koin esimerkiksi käyttäjien tai ratojen lisäämisen yhteydessä syötettävän JSON-tyyppisen objektin kunnollisen validoinnin olevan tärkeää, joten loin kaikki tarvittavat schema-tiedostot. Esimerkkinä kierrosten schema-tiedosto (Kuva 19), jossa määritellään, että pyynnössä lähetetyn JSON-objektin tulee sisältää nimi, koordinaatit latitudina ja longitudina, vaikeustaso sekä osoite. Reiät tai kuvaus radasta ovat valinnaisia attribuutteja. Vaikeustason tuli vastata olla difficulties-muuttujaan määritellyn taulukon vaikeustasoa, ja latitudin ja longitudin tuli sopia niille asetettuihin regexeihin tai säännöllisiin lausekkeisiin.

```
const latitudeRegex = /^(-?\d{1,2}(?:\.\d+)?|[-+]?[1-8]?\d{1}(?:\.\d+)?)$/
const longitudeRegex = /^(-?\d{1,3}(?:\.\d+)?|[-+]?(?:1[0-7]|[1-9])?\d{1}(?:\.\d+)?)$/

const difficulties = ["AAA1", "AA2", "AA1", "A3", "BB1", "B2", "C1", "D1", "C3", "B1"]

export const createHoleSchema = object({
    distance: number(),
    par: number({ required_error: "Par is required" }).min(1).max(10),
})

export const createCourseSchema = object({
    name: string({ required_error: "Need to add course name" })
        .min(2, "Too short name for course"),
    latitude: string({ required_error: "Latitude required" })
        .regex(new RegExp(latitudeRegex), "Incorrect latitude format"),
    longitude: string({ required_error: "Longitude required" })
        .regex(new RegExp(longitudeRegex), "Incorrect longitude format"),
    difficulty: string()
        .refine(val => difficulties.includes(val), { message: "Incorrect difficulty" }).optional(),
    address: string({ required_error: "Address required" }).min(5, "Too short address"),
    description: string(),
    holes: array(createHoleSchema)
})
```

Kuva 19. ”Round.schema.ts”-tiedoston validointiobjektit

Loin seuraavaksi uuden middlewaren, joka käsittelee pyynnön, ja käyttää sille syötettyä schema-objektia datan validoimiseen parse-funktiolla. Jos kyseinen middleware huomaa virheen annetun datan muodossa, palauttaa se virheilmoituksen. Lisäsin kyseisen middlewaren sitä tarvitseviin päätepisteisiin, esimerkkinä kaksi päätepistettä, joissa toteutetaan POST-pyyntöjä (Kuva 20). Sovelluksen tai käyttäjän tekemä POST-pyyntö ohjautuu ensin validointia suorittavalle middlewarelle, ja sitten vasta käsittelijäfunktiolle.

```
app.post("/api/courses", validateBody(createCourseSchema), postCourseHandler)
app.post("/api/users", validateBody(createUserSchema), postUserHandler)
```

Kuva 20. Validointi-middleware ennen käsittelijäfunktiota

4.9 Käyttäjien hallinta ja autentikaatio

Tässä vaiheessa backend oli periaatteessa käyttövalmis, mutta siitä puuttui erittäin tärkeä osa – käyttäjien tunnistaminen sekä autentikointi. Jotkin päätepisteet piti suojata sovelluksen ulkopuolisilta käyttäjiltä, ja joissakin päätepisteissä sijaitsevat käsittelijäfunktiot tarvitsivat tietoa kutsun tehneestä käyttäjästä. Esimerkiksi poistaessa jotain käyttäjää halutaan aina varmistaa, että pyynnön tehnyt käyttäjä on kyseinen käyttäjä itse. Alkuvaiheessa tekemässäni suunnitelmassa päätin hyödyntää JSON Web Tokeneita (JWT) käyttäjien autentikoinnissa, ja päätin tässä vaiheessa implementoida niiden hyödyntämisen backendiin.

Tehokas tapa hyödyntää JWT:itä on käyttää JWT-tokeneina saatua pidempiaikaista refresh tokenia sekä lyhytaikaisempaa access tokenia. Ajatuksena on, että access tokeniin on tallennettuna informaatiota käyttäjästä, ja sitä käytetään käyttäjän autentikoinnissa. Ne ovat hyvin lyhytaikaisia erilaisten väärinkäytösten estämiseksi, jos esimerkiksi jokin ulkopuolinen tekijä pääsee käsiksi siihen. Sen sijaan että käyttäjän pitää kirjautua jatkuvasti uudelleen järjestelmään access tokenin mentyä vanhaksi käyttää sovellus hänellä, yleensä evästeissä tallennettuna olevaa refresh tokenia access tokenin päivittämiseen sen vanhennuttua. (Babladi s.a.)

Loin aluksi ”jwt.utils.ts”-tiedoston, johon lisäsin kaikenlaisia JWT-tokeneihin liittyviä funktioita. Kuvassa (Kuva 21) näkyvä signJWT-funktio on itse tokenien luomisesta vastuussa oleva funktio. Se ottaa vastaan objektin, joka sisältää tietoa käyttäjästä, kuten käyttäjätunnuksen, sähköpostin sekä käyttäjän roolin. Kyseinen objekti lisätään luotavaan tokeniin, ja token allekirjoitetaan ja palautetaan. Allekirjoituksessa käytin backendin hakemiston juuressa elävään ”.env”-tiedostoon

lisäämääni ympäristömuuttujaa privateKey, jota käytetään datan suojaukseen RS-256 algoritmin mukaisesti. VerifyJWT-funktio taas ottaa vastaan JWT:n, ja purkaa sen sisällön luettavaan muotoon käyttäen ympäristömuuttujana olevaa julkista avainta publicKey. Decoded-muuttuja on tässä tapauksessa objekti, joka sisältää tietoa käyttäjästä. Jos funktio huomaa jotain virheitä sille syötetyssä JWT:ssä, palauttaa se virheen. ReIssueAccessToken-funktio ottaa vastaan refreshTokenin ja tarkistaa onko se oikeanlainen JWT. Seuraavaksi funktio varmistaa, että refreshTokenissa oleva käyttäjä on olemassa, ja lopuksi allekirjoittaa uuden accessTokenin ja palauttaa sen. Tätä funktiota hyödyntämällä accessTokenia saadaan siis päivitettyä refreshTokenin avulla ilman, että käyttäjän täytyy kirjautua palveluun jatkuvasti uudelleen.

```
const privateKey = process.env.PRIVATE_KEY || ""
const publicKey = process.env.PUBLIC_KEY || ""

export const signJWT = (object: Object, options?: jwt.SignOptions | undefined) => {
    return jwt.sign(object, privateKey, { ...(options && options), algorithm: "RS256" })
}

export const verifyJwt = (token: string) => {
    try {
        const decoded = jwt.verify(token, publicKey)
        return {
            valid: true,
            expired: false,
            decoded: decoded as TokenPayload
        }
    } catch (error: any) {
        return {
            valid: false,
            expired: error.message === "jwt expired",
            decoded: null
        }
    }

}

export const reIssueAccessToken = async (refreshToken: string) => {
    const { decoded } = verifyJwt(refreshToken) as JwtPayload
    const userId = get(decoded?.user, "id")

    if (!decoded || !userId) {
        return false
    }

    let user = await getSingleUser(Number(decoded.user.id))

    if (!user) return false

    const accessTokenTtl = "15m"
    const accessToken = signJWT({ user: { email: user.email, name: user.username, _id: user.id } }, { expiresIn: accessTokenTtl })
    return accessToken
}
```

Kuva 21. ”Jwt.utils.ts”-tiedostossa olevia funktioita

Seuraavaksi lisäsin ”users.model.ts”-tiedostoon käyttäjän sisäänkirjautumisessa käytettävän funktion (Kuva 22). Ensin käyttäjä validoidaan käyttäen funktiota, joka vertailee käyttäjän tietokantaan tallennettua salasanaa kirjautuessa annettuun salasanaan. Jos käyttäjä löytyy, palauttaa se sekä lyhytaikaisemman accessTokenin että pidempiaikaisemman refreshTokenin hyödyntäen edellämainittua signJWT-funktiota (Kuva 21). Molempia palautettuja tokeneita tullaan hyödyntämään käyttäjien autentikoinnissa.

```
export const loginUser = async (input: SessionType) => {
    const user = await validatePassword(input)

    if (!user) {
        throw new AppError("Wrong password!", 401, 401)
    }

    const accessToken = signJWT({ user: { email: user.email, name: user.username, id: user.id } }, { expiresIn: "15m" })
    const refreshToken = signJWT({ user: { email: user.email, name: user.username, id: user.id } }, { expiresIn: "30d" })

    return { accessToken, refreshToken }
}
```

Kuva 22. Käyttäjän kirjautumiseen liittyvä funktio

LoginUser-funktiota hyödyntämään loin erillisen ohjaimen controllers-kansioon, jossa oleva loginHandler-käsittelijäfunktio hyödyntää loginUser-funktiota käyttäjän kirjaamiseen sisään (Kuva 23). Funktio palauttaa refreshTokenin kiinnittäen httpOnly-evästeen res-olioon, joka on käyttäjälle pyynnössä palautettava olio. Tämä tallentaa refresh tokenin backendia käyttävän käyttäjän evästeisiin tulevia kutsuja varten. HttpOnly-eväste on turvallinen eväste, joka estää evästeeseen kaiken muun paitsi backendilta, vähentäen riskiä, että joku varastaisi evästeen hyödyntäen pahantahtoista koodia (CookiePro s.a.). AccessToken palautetaan pyynnössä JSON-tyyppisenä oliona, jonka frontendin on tarkoitus tallentaa paikallisesti ja hyödyntää sitä sen lähettäessä pyyntöjä backendille.

```
export const loginHandler = async (req: Request, res: Response, next: NextFunction) => {
    try {
        const tokens = await loginUser(req.body)
        return res.cookie("refreshToken", tokens.refreshToken, { httpOnly: true })
            .send({ accessToken: tokens.accessToken })
    } catch (error: any) {
        next(error)
    }
}
```

Kuva 23. Loginhandler-käsittelijäfunktio

Loin käyttäjän kirjautumista sisään varten ”routes.ts”-tiedostoon oman päätepisteen, joka hyödyntää loginHandleria käyttäjän kirjaamiseen sisään. Nyt olin implementoinut tavan käyttäjälle saada sekä access- että refreshTokenit backendiltä, mutta seuraavaksi piti keksiä tapa, miten backend vastaanottaa kyseiset tokenit käyttäjältä ja käsittelee ne asianmukaisesti.

Tein middleware-hakemistoon uuden middleware-funktion deserializeUser (Kuva 24). Funktio ottaa käyttäjän pyynnössä authorization-kuljetuskerroksessa eli headerissa olevan accessTokenin vastaan. Funktio ottaa myös pyynnössä tulevissa evästeissä olevan refreshTokenin ja asettaa sen refreshToken-muuttujaan. Koko funktion tarkoitus on purkaa accessTokenissa oleva käyttäjä ja asettaa se ”res.locals.user”-muuttujaan, jota backend pääsee muissa osissa hyödyntämään, esimerkiksi tarkistamalla pyynnön tekevän käyttäjän nimen tai ID:n. Jos accessToken on vanhentunut ja refreshToken löytyy, pyrkii funktio hankkimaan uuden accessTokenin käyttäen refreshTokenia, lähettämään uuden accessTokenin käyttäjälle ”x-access-token”-nimisessä kuljetuskerroksessa ja tallentamaan käyttäjän ”res.locals.user”-muuttujaan. Tämä on siis refreshTokenin perimmäinen tarkoitus – tuottaa uusia accessTokeneita edellisen vanhetessa siihen asti, kun itse refreshToken vanhenee, jolloin käyttäjä joutuu kirjautumaan uudelleen backendiin.

```
export const deserializeUser = async (req: Request, res: Response, next: NextFunction) => {
    const accessToken = get(req, "headers.authorization")?.slice(7)
    const refreshToken = req.cookies["refreshToken"]

    if (!accessToken) {
        return next()
    }
    const { decoded, expired } = verifyJwt(accessToken)
    if (decoded) {
        const { user } = decoded
        res.locals.user = user
        return next()
    }

    if (expired && refreshToken) {
        const newAccessToken = await reIssueAccessToken(refreshToken)
        console.log(newAccessToken)
        if (newAccessToken) {
            res.setHeader("x-access-token", newAccessToken)
            const result = verifyJwt(newAccessToken)
            res.locals.user = result.decoded
            return next()
        }
        return next()
    }

    return next()
}
```

Kuva 24. DeserializeUser-middlewarefunktio

Jotkin käsittelijäfunktiot tarvitsevat tietoja pyynnön tehneestä käyttäjästä, tai muuten vaan ovat suojattuja sisäänkirjautumattomilta käyttäjiltä. Jotta pystytään varmistamaan että pyynnön tekee sisäänkirjautunut käyttäjä hyödyntäen accessTokeniaan tein requireUser-middlewaren (Kuva 25), joka tarkistaa onko ”res.locals.user”-muuttujaa olemassa. Jos se puuttuu, palauttaa se statuskoodin 403 eli forbidden, ja pääsy resurssiin estetään. Hyödynsin kyseistä middlewarea lisäämällä sen kaikkiin aikaisemmin määrittelemiini päätepisteisiin (Kuva 18) niihin missä sitä tarvittiin, kuten esimerkiksi kierrosten lisäämisestä vastuussa olevaan päätepisteeseen.

```
export const requireUser = (req: Request, res: Response, next: NextFunction) => {
    const user = res.locals.user
    if (!user) {
        return res.status(403).end()
    }
    return next()
}
```

Kuva 25. RequireUser-middleware

4.10 Backendin käyttöönotto lokaalisti

Backend oli tässä vaiheessa periaatteessa valmis, ja se piti vain asettaa käyttövalmiiksi omaan kehitysympäristöön frontendin koodausta varten. Frontendiä koodatessa on hyvä samalla varmistaa molempien osien yhteensopivuus, mikä onkin tärkein syy sille miksi aloitin projektin toteuttamisen backendistä.

Index.ts-tiedosto näytti loppuvaiheessa tältä (Kuva 26). Kyseessä on ohjelman ”sydän”, ja suorittamalla kyseinen tiedosto käynnistyy koko backend aikaisemmin määriteltyine koodeineen. Kuvassa näkyy useita middlewareja, joita backend ottaa käyttöön ennen ”listen”-funktiossa tapahtuvaa päätepisteiden käyttöönottoa routes-funktion avulla. CookieParser on kirjasto, joka mahdollistaa evästeiden lukemisen sekä lisäämisen. CORS (Cross-Origin Resource Sharing) on mekanismi, joka mahdollistaa eri verkkotunnuksissa olevia sovelluksia välittämään data keskenään ilman konflikteja (Amazon s.a.). Cors-kirjastoa hyödynsin asettamaan tulevan frontendin IP:n sallituksi, jotta se kykenee tekemään pyyntöjä backendiin. Express.json on middleware, joka mahdollistaa JSON-tyyppisen datan käsittelyn. DeserializeUser on aikaisemmin luomani middleware, joka käsittelee pyynnössä tulevat JWT:t.

```
const PORT = process.env.PORT
const app = express()

app.use(cookieParser())
app.use(cors({
    exposedHeaders: ["x-access-token"],
    origin: ["http://localhost:3000,", "80.220.95.201"],
    methods: ["POST", "PUT", "DELETE"], credentials: true
}))
app.use(express.json())
app.use(deserializeUser)

app.listen(PORT, async () => {
    logger.info(`Listening to port ${PORT}`)
    routes(app)
})
```

Kuva 26. Index.ts-tiedosto

5 Frontendin toteutuksen aloitus

Frontendin toteutettiin React Nativella hyödyntäen Expo-kehystä. Kaikessa frontendin koodauksessa käytin VSCode-tekstinkäsittelyohjelmaa, kuten backendin kehityksessä. Suunnittelin ensin luovani projektin rakenteen sekä yleisesti käytetyt etukäteen suunnitellut toiminnot kuten

käyttäjänhallinnan sekä backendiin hakuja tekevät funktiot, ja sitten siirtyväni itse näkymien ja komponenttien toteuttamiseen.

5.1 Mobiilisovelluksen initialisointi

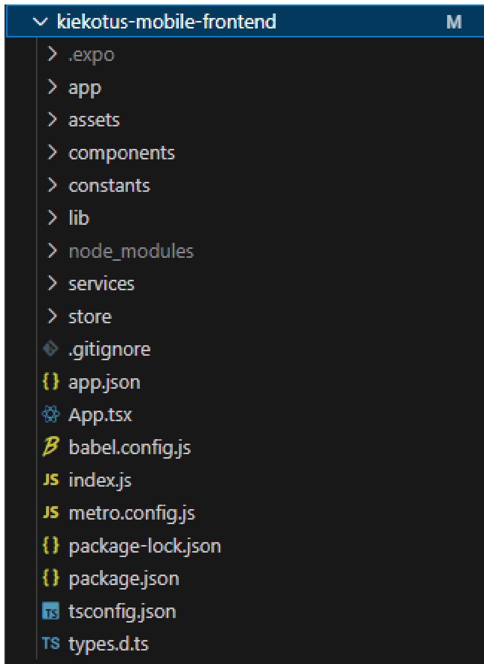
Lähtiessäni luomaan projektia aloitin projektin komennolla “npx create-expo-app -t expo-template-blank-typescript”, ja valitsin projektin nimeksi ”kiekotus-mobile-frontend”. Tämä loi uuden, periaatteessa mobiililaitteella jo toimivan sovelluksen, joka hyödyntää TypeScriptiä. Tulin kuitenkin lisäämään sovellukseen muita riippuvuuksia, kuten sovelluksessa navigointiin liittyvän expo-routerin sekä react-native-screensin, jotka molemmit asensin käyttäen npx install-komentoa.

Loin seuraavaksi mobiilisovellukselle oman GitHub-repositorion versionhallintaa varten, samalla tavalla kuten backendin kanssa.

5.2 Sovelluksen rakenne

Loin projektille rakenteen, hyödyntäen aikaisemmin tekemääni suunnitelmaa frontendistä, erityisesti Figmalla luomaani sovelluksen prototyyppiä.

Alustin heti aluksi projektin tarvittavat hakemistot, ja lopputulos näyttää tältä (Kuva 27). App-hakemistoon kuuluu kaikki sovellukseen liittyvät näkymät. Assets-hakemistoon kuuluu muun muassa sovelluksessa käytetyt kuvat. Components-hakemistoon kuuluu kaikki sovelluksessa hyödynnetyt Reactin kaltaiset komponentit. Constants-hakemistoa hyödynsin helpottamaan projektissa useasti hyödynnettyjen elementtien, kuten esimerkiksi tiettyjen kuvien käyttöä useassa paikassa erilaisten muuttujien muodossa. Lib-hakemistoon lisäsin kaikenlaista ylimääräistä, esimerkiksi hyödyllisiä eri paikoissa sovellusta käytettäviä funktioita. Services-hakemistoon oli tarkoitus lisätä backendin kanssa tekemisissä olevia funktioita, joita frontend hyödyntää. Store-hakemiston loin komponenttien välistä tilanhallintaa varten. Tarkoituksenani on selittää myöhemmin, mikä tarkalleen on jokaisen hakemiston tarkoitus.



Kuva 27. Frontendin hakemistot

5.3 Käyttäjänhallinta JWT:ta hyödyntäen

5.3.1 Zustand-tilanhallintakirjasto käyttäjänhallinnassa

Kehitteillä olevalla sovelluksella on monta eri näkymää, jotka ovat riippuvaisia kirjautuneesta käyttäjästä. Jouduin keksimään jonkin hyvän tavan tämän ominaisuuden lisäämiseen, hyödyntäen backendin syöttämiä refresh tokeneita sekä access tokeneita.

Zustand on hyvin yksinkertainen tilanhallintakirjasto Reactia sekä React Nativea varten. Koko sovelluksen kattavan tilan asetus sillä on hyvin helppoa ja vaivatonta verrattuna esimerkiksi yleisemmin käytettyyn tilanhallintakirjasto Reduxiin. Zustandia käyttäen voidaan määritellä tila ja tilaa muuttavat toiminnot yhteen tiedostoon, ja jakaa kyseinen tila ja sitä muuttavat toiminnot kaikkien sitä tarvitsevien komponenttien kesken. (George 2024)

Päätin käyttää Zustandia tilanhallintaan, saadakseni tehokkaan tavan jakaa tietoa kirjautuneesta käyttäjästä eri komponenteille ympäri sovellusta. Jotkin komponentit tarvitsivat tiedon myös siitä, onko käyttäjä ylipäätään kirjautunut sovellukseen, ja tuottaa näkymiä komponentin saaman tiedon perusteella.

Zustandia käyttäen loin stores-hakemistoon tiedoston authStore.ts, jossa loin tilan autentikaatiolle (Kuva 28). Tilassa on muuttujana käyttäjä, joka on objekti, johon kuuluu käyttäjän nimimerkki, sähköposti sekä id. LoggedIn on muuttuja, joka kertoo true tai false arvolla onko käyttäjä kirjautuneena sisään vai ei. Tila määrittelee myös funktioita, kuten login-funktion, jota käytetään käyttäjän kirjaamiseen sisään. Kyseinen funktio ottaa vastaan backendiin kirjautuessa saadun access tokenin, purkaa sen ja asettaa access tokenin perusteella saadun tiedon käyttäjästä tilan user-muuttujaan. Funktio myös tallentaa access tokenin mobiililaitteen lokaaliin SecureStore-muistiin. Käynnistettäessä sovellusta uudelleen user-muuttuja palautuu takaisin tyhjäksi, mutta relogin-funktiota hyödyntämällä käyttäjä kirjataan uudelleen sisään, hyödyntäen SecureStoressa olevaa access tokenia. Logout kirjaa käyttäjän ulos ja poistaa accessTokenin SecureStoresta.

```
const useAuthStore = create<AuthStore>(
    (set) => ({
        user: null,
        loggedIn: false,
        async login(token: string) {
            const decodedToken: TokenPayload = jwtDecode(token)
            set(state => ({
                ...state, user: { user: decodedToken.user.name, email: decodedToken.user.email, id: decodedToken.user._id }, loggedIn: true
            }))
            await SecureStore.setItemAsync("accessToken", token)
        },
        async relogin() {
            const token = await SecureStore.getItemAsync("accessToken")
            if (!token) return
            const decodedToken: TokenPayload = jwtDecode(token)
            set(state => ({
                ...state, user: { user: decodedToken.user.name, email: decodedToken.user.email, id: decodedToken.user._id }, loggedIn: true
            }))
        },
        async logout() {
            await SecureStore.deleteItemAsync("accessToken")
            set(state => ({
                ...state, user: null, loggedIn: false
            }))
        }
    }))
```

Kuva 28. UseAuthStore-tila Zustandilla

5.3.2 Axios-instanssin luominen ja refresh tokenien hallinta

Sovelluksen backendiin tekemissä pyynnöissä päätin käyttää perinteisen JavaScriptiin sisäänrakennetun Fetch API:n sijaan Axios-kirjastoa, joka tarjoaa selkeämmän sekä tehokkaamman vaihtoehdon fetchille.

Loin uuden tiedoston (Kuva 29), jossa määrittelin axios-instanssin, joka tekee pyynnöt sille syötettyyn baseURL:iin. WithCredentials-ominaisuus mahdollistaa sen, että pyynnön mukana kulkee evästeet. Instanssin luomisen lisäksi tiedostossa määritellään instanssin “interceptors” ominaisuuteen eri toimintoja JWT:hin liittyen. Ensimmäinen hakee SecureStoreen tallennetun

access tokenin, ja lisää sen pyynnössä olevaan headeriin. Näin jokaisessa pyynnössä pystytään todentamaan pyynnön tehnyt käyttäjä access tokenin perusteella. Toinen ottaa vastaan “x-access-token”-headerin, jonka backend lähettää takaisin access tokenin vanhennuttua evästeissä syötettyä validia refresh tokenia vastaan. Jos kyseinen token on eri verrattuna SecureStoreen tallennettuun access tokeniin, vaihtaa se vanhan access tokenin uuteen. Backend konfiguroitiin aikaisemmin käsittelemään näitä kyseisiä toimintoja (Kuva 24).

```
const instance = axios.create({ baseURL: "http://192.168.1.66:1337", withCredentials: true })

export const getToken = async () => {
    const token = await SecureStore.getItemAsync("accessToken")
    if (typeof token === "string") { return token }
    return null
}

export const setAccessToken = async (token: string) => {
    await SecureStore.setItemAsync("accessToken", token)
}

instance.interceptors.request.use(async (req) => {
    const accessToken = await getToken()
    req.headers["Authorization"] = `Bearer ${accessToken}`
    return req
})

instance.interceptors.response.use(async (res) => {
    const xToken = res.headers["x-access-token"]
    const accessToken = await getToken()
    if (xToken !== accessToken) {
        setAccessToken(xToken)
    }
    return res
})
```

Kuva 29. Axios-kirjastolla luotu instanssi ja sen interceptors-ominaisuuteen liitetyt middleware-funktiot

5.4 Services-hakemisto

Frontend tulee tekemään backendiin jatkuvasti erilaisia pyyntöjä käyttäjien tehdessä eri toimintoja, joiden on tarkoitus suorittaa haku- tai lisäystoimintoja tietokannan kanssa. Näitä pyyntöjä suorittavat eri funktiot, jotka päätin laittaa omaan erilliseen services-hakemistoonsa, josta frontendin komponentit hakevat backendin kanssa vuorovaikuttavia funktioita eri pyyntöjä varten. Loin services-hakemistoon tiedostot courseService, friendService, roundService sekä userService. Jokaiseen tiedostoon lisäsin tiedostonimeen liittyviä funktioita.

Esimerkkinä yhdestä services-hakemiston tiedostosta näkyy kuvassa 30 roundService.ts-tiedoston funktioita, jotka hyödyntävät edellisessä luvussa luotua axios-instanssia erityyppisten pyyntöjen tekemiseen backendiin. Näitä funktioita tullaan hyödyntämään myöhemmin eri komponenteissa.

```
export const getUserRounds = async (userId: string) => {
    try {
        const response = await instance.get(`/api/users/${userId}/rounds`)
        const data = await response.data
        return data
    } catch (error) {
        console.error(error)
    }
}

export const addUserRound = async (userId: string, round: Round) => {
    try {
        const response = await instance.post(`/api/users/${userId}/rounds`, round)
        const data = await response.data
        return data
    } catch (error) {
        console.error(error)
    }
}

export const addPlayedRound = async (userId: string, courseId: number) => {
    try {
        const response = await instance.post(`/api/users/${userId}/statistics/played-courses`, { courseId, date: new Date().toISOString() })
        const data = await response.data
        return data
    } catch (error) {
        console.error(error)
    }
}
```

Kuva 30. RoundService.ts-tiedoston funktioita

6 Sovelluksen näkymät ja komponentit

Seuraavaksi siirryin sovelluksen näkymien luomiseen Figmalla luotua prototyyppiä hyödyntäen. Ajatuksena oli, että näkymien luonti järjestyksessä käyttäjän sovelluksessa kulku huomioiden olisi järkevintä. Näkymätiedostoissa renderöidään käyttäjälle mobiililisovellusta käytettäessä näkyvät näkymät, ja tiedostoissa on mukana mm. eri funktioita tuomassa niihin toiminnallisuutta. Näkymissä hyödynnetään paljon myös erillisiä komponentteja, jotta kaikkea koodia ei tarvitse sijoittaa yhteen näkymätiedostoon. Näin koodi on modulaarisempaa ja täten selkeämpää.

6.1 Layout-näkymä

Yksi ohjelmoinnin kultaisista säännöistä on pyrkimys välttää liiallista toistoa. Kaikissa näkymissä on tiettyjä toistuvia elementtejä, joten piti keksiä tapa välittää kyseiset elementit jokaiselle näkymälle ilman turhaa toistoa.

Toteutin tämän luomalla layout-näkymän. Käyttäjän ei ole tarkoitus ”nähdä” kyseistä näkymää, mutta kaikki app-hakemistoon lisäämäni näkymät renderöidään kyseisen _layout.tsx-näkymän läpi (Kuva 31). Projektissa käyttämäni expo-router tunnistaa kyseisen näkymän nimen perusteella. Layout määrittelee SafeAreaViewin, joka varmistaa, että kaikki sisältö sijoitetaan suhteessa käytetyn mobiililaitteen reunuksiin. Tämä varmistaa, että sisältö sijoitetaan oikein riippumatta mobiililaitteiden eroavista reuna-alueista, joita esimerkiksi akun latauksen määrän indikaattori voi luoda. QueryClientProvider mahdollistaa erään useQuery-kirjaston käytön kaikissa näkymissä. ImageBackground määrittelee koko sovelluksessa käytettävän taustakuvan, joka on kaikkien näkymien taustalla. Toast-komponentti kuuluu react-native-toast-message ilmoitusviestikirjastoon, ja sijoittamalla Toast-komponentin layout-tiedostoon näkyy uusi ilmoitusviesti missä tahansa näkymässä. Lopuksi Slot kertoo, mihin kohtaan renderöitävä näkymä, johon koko layout-näkymä tulee vaikuttamaan, renderöidään.

Tyylittely liitetään React Nativen elementteihin niiden style-attribuutin avulla. Loin jokaiselle näkymälle erillisen styles.ts-tiedoston, joka luo tyylittelyn määrittelevän olion hyödyntäen React Nativen StyleSheet-moduulia (Kuva 32). StyleSheetillä luotu olio tuodaan näkymään tai komponenttiin, ja oliossa määritellyt tyylit lisätään style-attribuuteilla sitä vaativiin elementteihin tai komponentteihin. Samaa tyylittelyperiaatetta käytin kaikissa näkymissä sekä komponenteissa.

```
const queryClient = new QueryClient()

const _layout = () => {
    return (
        <SafeAreaView style={styles.safeAreaView}>
            <QueryClientProvider client={queryClient}>
                <ImageBackground source={images.background} style={styles.background}>
                    <Toast />
                    <Slot />
                </ImageBackground>
            </QueryClientProvider>
        </SafeAreaView>
    )
}
```

Kuva 31. Layout-näkymä, joka lisää erilaisia ominaisuuksia kaikkiin sovelluksen näkymiin

```
import { StyleSheet } from "react-native"

export const styles = StyleSheet.create({
    safeAreaView: {
        flex: 1,
        backgroundColor: "grey"
    },
    background: {
        flex: 1,
        justifyContent: "center",
        alignItems: "center"
    },
    header: {
        backgroundColor: "grey"
    },

})
```

Kuva 32. ”Layout.style.ts”-tiedoston sisältö

6.2 Sovelluksen päänäkymä

Aloitin käyttäjille näkyvien näkymien luomisen päänäkymästä, jonka sovelluksen käyttäjä näkee heti käynnistettyään sovelluksen. Loin app-hakemiston juureen tiedoston index.ts. Tiedoston nimen perusteella expo-router tietää kyseessä olevan sovelluksen päänäkymä (Kuva 33).

Index.ts-tiedostoon tuodaan aikaisemmin Zustandilla määritellyn AuthStoren loggedIn-muuttuja sekä logout- ja relogin-metodit. LoggedIn-muuttujaa hyödynnetään tiedostossa määrittelemään käyttäjälle renderöitävää sisältöä. Logout-metodia käytetään käyttäjän kirjaamiseen ulos, ja relogin-metodia taas käyttäjän tietojen asettamiseen authStoreen uudelleen SecureStoressa olevaa access tokenia käyttäen.

Loin tässä vaiheessa myös erillisen Zustand-storen, roundStoren. Tämä store tulee sisältämään tietoa pelaajan aloittaneeseen kierrokseen liittyen. Lisäsin sinne tiedon siitä, onko käyttäjä luomassa uutta kierrosta (creatingRound-boolean), ja metodin setCreatingRound, joka asettaa creatingRound-booleanille arvoksi true. Tämä tulee vaikuttamaan myöhemmin sovelluksessa.

```
export default function IndexPage() {
  const router = useRouter()
  const { loggedIn, logout, relogin } = useAuthStore()
  const { setCreatingRound } = useRoundStore()
  useEffect(() => {
    setCreatingRound(false)
    relogin()
  }, [])

  const createRoundHandler = () => {
    if (!loggedIn) {
      Toast.show({
        type: "error",
        text1: "Kirjaudu sisään aloittaaksesi kierroksen!",
      })
      return
    }
    router.push({ pathname: "/map" })
    setCreatingRound(true)
  }

  const logOut = () => {
    logout()
  }

  return (
    <View style={styles.container}>
      <StatusBar style='light' backgroundColor='black' />
      <View style={styles.upperButtonContainer}>
        <TouchableOpacity style={styles.circleButton} onPress={() => createRoundHandler()}>
          <Image source={images.basket} resizeMode='contain' style={styles.iconStyle} />
          <Text style={styles.textStyle}>Uusi kierros</Text>
        </TouchableOpacity>
        <TouchableOpacity style={styles.circleButton} onPress={() => router.push({ pathname: "/map" })}>
          <Image source={images.mapicon} resizeMode='contain' style={styles.iconStyle} />
          <Text style={styles.textStyle}>Radat kartalla</Text>
        </TouchableOpacity>
      </View>
      <View style={styles.lowerButtonContainer} >
        {loggedIn ?
          <>
            <TouchableOpacity style={styles.rectangleButton}
              onPress={() => router.push({ pathname: "/profile" })}>
              <Text style={styles.textStyle}>Profiili</Text>
            </TouchableOpacity>
            <TouchableOpacity style={styles.rectangleButton} onPress={logOut}>
              <Text style={styles.textStyle}>Kirjaudu ulos</Text>
            </TouchableOpacity>
          </>
          : <>
            <TouchableOpacity style={styles.rectangleButton}
              onPress={() => router.push({ pathname: "/login" })}>
              <Text style={styles.textStyle}>Kirjaudu sisään</Text>
            </TouchableOpacity>
          </>}
      </View>
    </View>
  );
}
```

Kuva 33. Index.ts-tiedoston sisältö

Ensimmäinen toiminnallisuus kuvan 33 näkymässä on siihen asettamani useEffect-koukku. Reactissa useEffect on hyvin usein käytetty koukku, jota käyttämällä voidaan tuottaa sivuvaikutuksia komponenteissa. Koukku on funktio, joka ottaa vastaan kaksi arvoa: suoritettavan

funktion sekä taulukossa määritellyt riippuvuudet, jotka laukaisevat kyseisen sivuvaikutuksen. Asetin useEffect-koukun toteuttamaan authStoren relogin-funktion, ja asetin riippuvuudeksi tyhjän taulukon, mikä tarkoittaa sitä, että koukku suoritetaan vain kerran näkymään tultaessa. Asetin kyseisen koukun, sillä käyttäjän käynnistäessä sovelluksen uudelleen tyhjenee authStoressa oleva tieto kirjautuneesta käyttäjästä ja koukun avulla tieto lisätään authStoreen takaisin.

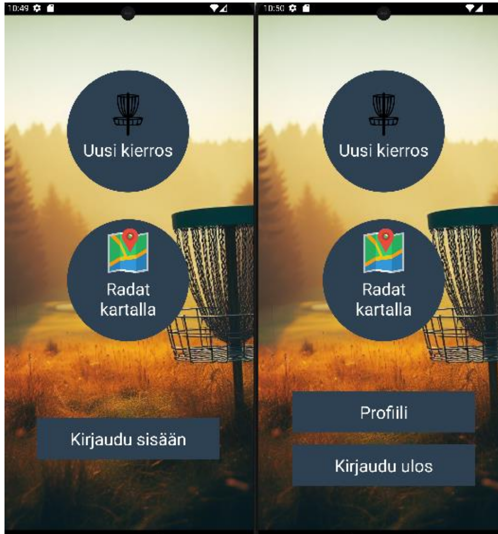
React Nativessa View-komponentti toimii HTML:n div-elementin tavoin. Kaikki sisältö React Nativessa laitetaan View-tyylisten komponenttien sisään, ja tyylittelemällä kyseiset komponentit niiden style-attribuuttia käyttäen pystyin mm. säätämään niiden kokoa sekä määrittelemään miten niiden sisälle asetetut muut komponentit sijoitetaan.

StatusBar on React Nativen tuoma komponentti, joka mahdollistaa puhelimen tilapalkin muokkaamisen.

Navigaatioon liittyvien nappien tai painikkeiden luonnissa hyödynsin React Nativen tarjoamaa TouchableOpacity-komponenttia. Kykenin onPress-attribuuttia hyödyntäen asettamaan jokaiselle painikkeelle niiden painautuessa suoritettavat funktiot.

Useat näkymissä olevista painikkeista ohjaavat käyttäjän toiseen näkymään käyttäen tiedoston alussa määritellyn router-olion push-funktiota. Painikkeet renderöidään näkymään loggedIn-muuttujan totuusarvon perusteella (Kuva 34). Jos käyttäjä ei ole kirjautuneena sisään eli loggedIn on false, on näkymässä Kirjaudu sisään-painike. Jos käyttäjä on kirjautuneena sisään eli loggedIn on true, on näkymässä Kirjaudu ulos- sekä Profiili-painikkeet. Uloskirjautumiseen liittyvä nappi kirjaa käyttäjän ulos käyttäen authStoren tarjoamaa logout-funktiota.

’Uusi kierros’-painike käynnistää tiedostossa määrittelemäni createRoundHandler-funktion, joka tarkistaa onko käyttäjä kirjautunut sisään. Jos käyttäjä on kirjautuneena sisään, asettaa se creatingRound-booleanin arvoksi true, ja käyttäen router-funktiota ohjaa käyttäjän map-näkymään. Jos käyttäjä ei ole kirjautuneena sisään, ilmoittaa se virheestä käyttäen react-native-toast-message kirjaston tarjoamaa funktiota, joka saa virheilmoituksen ilmestymään käyttäjän ruudulle.



Kuva 34. Index.ts-tiedoston kaksi mahdollista tuottamaa näkymää käytännössä

6.3 Kirjautumis- ja rekisteröitymisnäkymät

Käyttäjän halutessa kirjautua tai rekisteröityä sovellukseen navigoi hän Login-näkymään päänäkymästä Kirjaudu sisään-painikkeen avulla. Näkymän koodi (Kuva 35) on suhteellisen yksinkertainen. Se sisältää yhden TouchableOpacity-painikkeen, joka on sijoitettu ylälaitaan ja ohjaa käyttäjän takaisin päänäkymään. Pohdin, miten tyylittelisin painikkeen, ja päädyin käyttämään AntDesigniä. AntDesign on Reactille tarkoitettu UI-komponenttikirjasto, ja se tarjoaa valmiita UI-komponentteja React-sovelluksille, kuten erilaisia kuvakkeita (Uxpin s.a.). Hyödynsin AntDesigniä tuomalla koodiin kirjaston tarjoaman nuolikuvakkeen, ja lisäsin sen TouchableOpacity-komponenttiin visualisoimaan edelliselle sivulle vievää painiketta.

```
const LoginPage = ({ }) => {
    const router = useRouter()

    return (
        <View style={styles.container}>
            <TouchableOpacity style={styles.backButton} onPress={() => router.push("/")}>
                <AntDesign name="back" size={24} color="black" />
            </TouchableOpacity>
            <View style={styles.header}>
                <Text style={styles.mainText}>Kirjaudu sisään</Text>
                <View style={styles.line} />
            </View>
            <LoginForm />
        </View>
    );
};
```

Kuva 35. Kirjautumisnäkymän koodi

Koodin modularisointia Reactin periaatteiden mukaan ajatellen loin kirjautumiseen liittyvän lomakkeen erikseen LoginForm-nimisen komponentin muodossa projektin components-hakemistoon (Kuva 36). Koodissa määritellään lomake, joka sisältää TextInput-komponentteja, jotka toimivat kenttinä HTML:n input-elementtien lailla. Lomakkeessa hyödynnetään react-hook-form kirjastoa, joka mahdollistaa useForm-koukun käytön. UseForm-koukun tarkoitus on tehdä lomakkeiden hallinnasta helpompaa. UseFormin tuomalla register-funktiolla pystyin helposti hallitsemaan kenttiä asettamalle niille tiettyjä käyttäjältä odotettuja vaatimuksia, kuten tarpeen sähköpostin oikealle muodolle. Pystyin myös asettamaan funktion avulla kenttien tuottamat virheilmoitukset. Virheilmoituksia pystyin hyödyntämään useFormin tarjoaman formStaten errors-oliolla. Kyseistä oliota hyödyntämällä pystyin määrittelemään tavan, jolla virheet ilmoitetaan käyttäjälle. Päädyin ilmoittamaan virheistä syötteissä TextInput-komponentin alle ilmaantuvilla teksteillä.

```
const LoginForm = ({ }) => {
    const router = useRouter()
    const { login } = useAuthStore()
    const { register, handleSubmit, setValue, formState: { errors } } = useForm();

    const onSubmit = async (formData: FieldValues) => {
        try {
            const token = await loginToServer({ password: formData.password, email: formData.email })
            if (token) {
                login(token)
                router.push({ pathname: "/" })
            }
        } catch (error) {
            Toast.show({
                type: "error",
                text1: "Virhe kirjautuessa sisään! Väärä käyttäjätunnus ja/tai salasana.",
            })
        }
    }

    const onChangeField = (name: any) => (text: any) => {
        setValue(name, text);
    };

    return (
        <>
            <View style={styles.formContainer}>
                <TextInput style={styles.input} placeholder="Sähköposti" onChangeText={onChangeField('email')}
                    {...register("email", {
                        required: "Syötä sähköposti!", pattern: {
                            value: /^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i,
                            message: 'Sähköpostin tulee olla muotoa XXX@XXX.XXX',
                        }
                    })} />
                {errors.email && <Text style={styles.errorText}>{errors.email.message as string}</Text>}
                <TextInput style={styles.input} secureTextEntry placeholder="Salasana" onChangeText={onChangeField('password')}
                    {...register("password", { required: "Syötä salasana!" })} />
                {errors.password && <Text style={styles.errorText}>{errors.password.message as string}</Text>}
                <TouchableOpacity onPress={handleSubmit(onSubmit)} style={styles.button}>
                    <Text style={styles.buttonText}>
                        Kirjaudu sisään
                    </Text>
                </TouchableOpacity>
            </View>
            <Text style={styles.registerLink} onPress={() => router.push({ pathname: "/registration" })}>Puuttuuko tunnus? Rekisteröidy tästä!</Text>
        </>
    );
};
```

Kuva 36. Kirjautumislomakkeen koodi

Käyttäjän syöttäessä omat tietonsa ja painaessaan “Kirjaudu sisään”-painiketta aktivoituu kuvan 36 koodissa määritelty onSubmit-funktio, joka hyödyntää services-hakemistossa olevaa loginToServer-funktiota käyttäjän tietojen lähettämiseen ja tokenien hakemiseen. Jos kaikki on kunnossa, kirjaa funktio käyttäjän sisään authStoren login-funktiota käyttäen ja vie käyttäjän takaisin päänäkymään. Jos kirjautumisessa tapahtui virhe, ilmoitetaan siitä käyttäjälle ilmoitusviestillä.

Kuten kuvasta 36 näkee, on koodin alaosassa painike rekisteröitymiselle käyttäjätunnuksen puuttuessa. Painamalla sitä ohjaa sovellus käyttäjän rekisteröitymisnäkymään (Kuva 37). Näkymä käyttää täysin samantyyppistä rakennetta kirjautumisnäkymän kanssa, eristäen rekisteröitymislomakkeen omaan komponenttiin.

```
const Registration: FC<RegistrationProps> = ({ }) => {
    const router = useRouter()

    return (
        <View style={styles.container}>
            <TouchableOpacity style={styles.backButton} onPress={() => router.push("/login")}>
                <AntDesign name="back" size={24} color="black" />
            </TouchableOpacity>
            <View style={styles.header} >
                <Text style={styles.mainText}>Rekisteröidy</Text>
                <View style={styles.line} />
            </View>
            <RegisterForm />
        </View>
    )
}
```

Kuva 37. Rekisteröitymisnäkymän koodi

Rekisteröitymisnäkymässä oleva lomake (Kuva 38) on hyvin samankaltainen kirjautumisnäkymän lomakkeen kanssa. Siinä käytetään lomakkeen hallintaan kirjautumislomakkeen tavoin react-hook-formia. OnSubmit-funktiossa tarkistetaan ensin, että käyttäjän syöttämät salasanat täsmäävät, ja seuraavaksi funktiossa käytetään registerToServer-funktiota backendiin rekisteröitymiseen. Jos sovelluksen käyttäjä onnistuu rekisteröimään uuden tunnuksen, ohjataan hänet takaisin kirjautumisnäkymään ilmoitusviestin kera. Kirjautumisnäkymässä hän pystyy syöttämään uuden tunnuksen kirjautumistiedot ja kirjautumaan sisään sovellukseen.

```
const RegisterForm = ({ }) => {
    const router = useRouter()
    const { register, handleSubmit, setValue, formState: { errors } } = useForm();

    const onSubmit = async (formData: FieldValues) => {
        try {
            if (formData.password !== formData.confirmPassword) {
                Toast.show({ type: "error", text1: "Salasanat eivät vastaa toisiaan!" })
            }
            const registered = await registerToServer({ password: formData.password, confirmPassword: formData.confirmPassword,
                email: formData.email, username: formData.username })
            if (registered) {
                Toast.show({ type: "success", text1: "Käyttäjä luotu onnistuneesti, voit nyt kirjautua sisään." })
                router.push({ pathname: "/login" })
            }
        } catch (error) {
            Toast.show({ type: "error", text1: "Virhe luodessa käyttäjää: käyttäjätunnus tai sähköposti on jo olemassa." })
        }
    };

    const onChangeField = useCallback((name: any) => (text: any) => {
        setValue(name, text);
    }, []);

    return (
        <>
            <View style={styles.formContainer}>
                <TextInput style={styles.input} placeholder="Käyttäjätunnus" onChangeText={onChangeField('username')} {...register("username",
                    { minLength: { value: 2, message: "Käyttäjätunnuksen minimipituus 2 merkkiä" }
                    , maxLength: { value: 15, message: "Käyttäjätunnuksen maksimipituus 15 merkkiä" } })} />
                {errors.username && <Text style={styles.errorText}>{errors.username.message as string}</Text>}
                <TextInput style={styles.input} placeholder="Sähköposti" onChangeText={onChangeField('email')}  {...register("email", {
                    required: "Syötä sähköposti!",
                    pattern: { value: /^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i, message: 'Sähköpostin tulee olla muotoa XXX@XXX.XXX', }
                })} />
                {errors.email && <Text style={styles.errorText}>{errors.email.message as string}</Text>}
                <TextInput style={styles.input} secureTextEntry placeholder="Salasana" onChangeText={onChangeField('password')}
                {...register("password", { required: "Syötä salasana!" })} />
                {errors.password && <Text style={styles.errorText}>{errors.password.message as string}</Text>}
                <TextInput style={styles.input} secureTextEntry placeholder="Toista salasana" onChangeText={onChangeField('confirmPassword')}
                {...register("confirmPassword", { required: "Syötä salasanan vahvistus!" })} />
                {errors.confirmPassword && <Text style={styles.errorText}>{errors.confirmPassword.message as string}</Text>}
                <TouchableOpacity onPress={handleSubmit(onSubmit)} style={styles.button}>
                    <Text style={styles.buttonText}>
                        Rekisteröidy
                    </Text>
                </TouchableOpacity>
            </View>
        </>
    );
};
```

Kuva 38. Rekisteröitymislomakkeen koodi

6.4 Karttanäkymä

Päänäkymässä (Kuva 34) käyttäjän painaessa “Uusi kierros”- tai “Radat kartalla”-painiketta ohjautuu sovellus karttanäkymään. Karttanäkymän komponentti on melko yksinkertainen, ja se sisältää kaikki sovelluksen kartan toiminnallisuudet kattavan komponentin CourseMap (Kuva 39).

UseState-koukut ovat Reactissa hyvin usein käytettyjä koukkuja, jotka mahdollistavat tilan hallinnan komponentissa. Ne ottavat parametrina vastaan alkutilan, ja palauttavat taulukon, joista ensimmäinen alkio on tila ja toinen on tilan päivittävä funktio. Tilan päivittävää funktiota kutsuessa

komponentti päivittyy uudelleen ja renderöidään muuttuneella useState-koukulla, jonka tilan päivittävää funktiota käytettiin. (Herrera 2023)

CourseMapissa käytetään useita useState-koukkuja hallitsemassa komponentin tilaa. Kuten kuvasta 39 näkyy, useState-koukkuja on muun muassa ratojen tallentamista varten luotu courses-koukku, käyttäjän sijaintia tallentava userLocation-koukku, tämänhetkiselle sijainnille tarkoitettu location-koukku sekä valitun radan id:n tallentava koukku visibleCourseId.

```
const CourseMap = () => {
    const [courses, setCourses] = useState<Course[]>([])
    const [userLocation, setUserLocation] = useState<Coordinates | null>(null);
    const [location, setLocation] = useState<Coordinates | null>({ latitude: 60.2963679, longitude: 25.0382604 });
    const [visibleCourseId, setVisibleCourseId] = useState<null | number>(null)
    const [creatingCustom, setCreatingCustom] = useState(false)
    const creatingRound = useRoundStore(state => state.creatingRound)

    // Käyttäjän sijainnin pyyntäminen ja hyödyntäminen
    const getUserLocation = async () => {
        let { status } = await Location.requestForegroundPermissionsAsync();
        if (status !== 'granted') {
            return;
        }
        let location = await Location.getCurrentPositionAsync({});
        const { latitude, longitude } = location.coords
        return { latitude, longitude }
    }

    useEffect(() => {
        getUserLocation().then(coordinates => {
            if (coordinates) {
                setLocation({ ...coordinates })
                setUserLocation({ ...coordinates })
            }
        })
        getAllCourses().then(courses => setCourses(courses))
    }, [])

    return (
        <View style={styles.container}>
            <MapView style={styles.map} showsUserLocation onRegionChangeComplete={(event) => setLocation(event)} region={{
                latitude: location ? location.latitude
                    : 0, longitude: location ? location.longitude : 0, latitudeDelta: 2, longitudeDelta: 2
            }} onPress={() => setVisibleCourseId(null)}>
                {courses.map((course, i) => (
                    <Marker coordinate={{ latitude: parseFloat(course.latitude), longitude: parseFloat(course.longitude) }} title={course.name}
                        description={course.description} key={i} image={images.basket} onPress={() => setVisibleCourseId(course.id)} />
                ))}
            </MapView>
            <Toast  />
            {userLocation &&
                <TouchableOpacity style={styles.locateIcon} onPress={async () => { setLocation(userLocation) }}>
                    <Image source={images.locateUser} style={styles.locateIconImage} />
                </TouchableOpacity>
            }
            {visibleCourseId && <CourseInfo visibleCourseId={visibleCourseId} setVisibleCourseId={setVisibleCourseId} />}
            {creatingCustom && <CustomCourseForm setCreatingCustom={setCreatingCustom} />}
            {creatingRound && !creatingCustom && <TouchableOpacity style={styles.createOwnCourseButton} onPress={() => setCreatingCustom(true)}>
                <Text>Custom rata</Text>
            </TouchableOpacity>}
        </View>
    )
}
```

Kuva 39. CourseMap-komponentin koodi

CourseMap-komponentissa on myös useEffect-koukku. Koukku hyödyntää getUserLocation-funktiota, joka käyttää expo-location kirjastoa. Expo-location tarjoaa mahdollisuuden käyttää puhelimen natiiveja sijaintiominaisuuksia. Funktio pyytää ensin lupaa käyttää sijaintia, ja käyttäjän salliessa sovelluksen käyttää sijaintia hakee funktio käyttäjän koordinaatit latitudina ja longitudina. UseEffect-koukku käyttää kyseistä funktiota, ja saadessaan käyttäjän koordinaatit asettaa se kyseiset koordinaatit location- sekä userLocation-muuttujiin. UseEffect hakee lopuksi kaikki radat käyttäen services-hakemistossa sijaitsevaa funktiota getAllCourses, ja asettaa ne courses-tilaan hyödyntäen setCourses-funktiota.

Itse kuvan 39 komponentin tuottamassa kartassa hyödynsin react-native-mapsin tarjoamaa MapView-karttakomponenttia. Kartalle asetin showsUserLocation-ominaisuuden, joka aiheuttaa sen, että käyttäjän sen hetkinen sijainti näkyy aina kartalla. OnRegionChangeComplete-ominaisuus tarjoaa tapahtumankäsittelijän, joka suoritetaan aina kun käyttäjä liikuttaa karttanäkymää. Sille asettamani funktio muuttaa useStatella määrättyä “location”-tilaa. Region-ominaisuus määrittelee kartalla näytetyn sijainnin ominaisuudelle annettujen koordinaattien mukaan. Se on riippuvainen “location”-tilasta, ja aina kun “location”-tila muuttuu, muuttuu kartalla näytetty sijainti.

Kartan sisälle renderöidään radat Marker-komponentteina, jotta ne näkyvät kartalla ja käyttäjä pystyy painamaan niitä. Käytin “courses”-tilaan asetettujen ratojen iterointiin map-funktiota, ja loin jokaiselle radalle oman Marker-komponentin MapViewin sisään. Markereille asetettiin koordinaatit, nimike sekä kuvaus. Muokkasin myös jokaisen kartalla näkyvän Marker-komponentin frisbeegolf-korin näköiseksi käyttäen niiden ”image”-ominaisuutta.

Kuvan 39 CourseMap-komponenttiin renderöidään myös käyttäjän sijainnin paikannus tilanteessa, jos käyttäjä vetää karttaa muualle ja haluaa keskittää kartan näkymän takaisin hänen omaan sijaintiinsa. Loin TouchableOpacity-komponenttia hyödyntävän painikkeen, joka renderöidään näkymään, jos käyttäjä on sallinut sijainnin käytön. Painamalla painiketta asetetaan ”userLocation”-koukkuun tallennetut koordinaatit ”location”-koukkuun, ja MapView päivittyy käyttäjän senhetkiseen sijaintiin, jos käyttäjä on liikkunut kartalla jonnekkin muualle.

Seuraavaksi rakensin komponentin, joka antaa käyttäjälle tiedot painamastaan radasta. Näytettävä rata renderöidään kuvan 39 CourseMapin ”visibleCourseId”-koukun avulla, ja radalle syötetään kyseinen koukku ominaisuuksina. Marker-komponenteissa on onPress-ominaisuus, joka määrittää sen, että käyttäjän painaessa sitä asetetaan Marker-komponentissa olevan radan id ”visibleCourseId”-koukkuun. Jos ”visibleCourseId”-koukku ei ole null, renderöidään karttanäkymään ”CourseInfo”-komponentti (Kuva 40), joka ilmoittaa radan nimen, vaikeustason,

osoitteen, mahdollisen linkin ratakarttaan ja kaikki radan väylät listana haettuaan tiedot ”getCourseData”-servicefunktion avulla.

```
const CourseInfo: FC<CourseInfoProps> = ({ visibleCourseId, setVisibleCourseId }) => {
    const { setRoundInfo, creatingRound } = useRoundStore(state =>
        ({ setRoundInfo: state.setRoundInfo, creatingRound: state.creatingRound }))
    const [course, setCourse] = useState<Course | null>(null)
    const router = useRouter()

    useEffect(() => {
        if (visibleCourseId) {
            getCourseData(visibleCourseId).then(course => setCourse(course))
        }
    }, [visibleCourseId])

    if (!visibleCourseId || !course) {
        return null
    }

    const handlePlayerSelectionPress = () => {
        setRoundInfo({ course, players: [] })
        router.push({ pathname: "/playerselection" })
    }

    return (
        <View style={styles.container}>
            <BackButton onPress={() => setVisibleCourseId(null)} />
            <View style={styles.courseInfoContainer}>
                <Text style={styles.courseTitle}>{course.name} {course.difficulty}</Text>
                <Text style={styles.courseInfo}>{course.address}</Text>
                {course.mapAddress ?
                    <TouchableOpacity style={styles.courseMap}>
                        <MaterialIcons name="map" size={24} color="blue" />
                        <Text>Ratakartta</Text>
                    </TouchableOpacity>
                    :
                    <Text>Ei ratakarttaa</Text>}
                <Text style={styles.fairwayHeader}>Väylät</Text>
                <ScrollView style={styles.fairwayList}>
                    {course.holes.map((hole, i) => (
                        <View style={styles.fairwayInfo} key={i}>
                            <Text style={styles.fairwayNumber}>Väylä {i + 1}</Text>
                            <View style={styles.distanceAndPar}>
                                <Text>{hole.distance} metriä</Text>
                                <Text>Par {hole.par}</Text>
                            </View>
                        </View>

                    ))}
                </ScrollView>
                {creatingRound &&
                    <TouchableOpacity style={styles.playerChooseButton}
                        onPress={() => handlePlayerSelectionPress()} >
                        <Text>Pelaajavalinta</Text>
                    </TouchableOpacity>}
            </View>
        </View>
    )
}
```

Kuva 40. CourseInfo-komponentin koodi

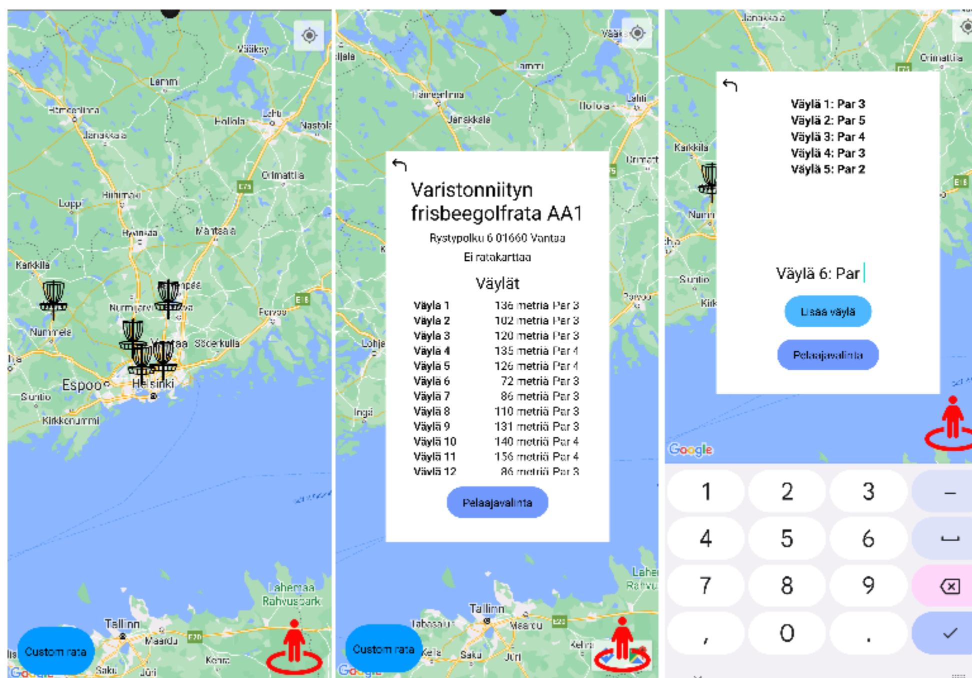
Kuvan 40 CourseInfo-komponentissa käytetään myös roundStoren ”creatingRound”-tilaa josta selviää tieto siitä onko käyttäjä aloittamassa kierrosta, ja sen mukaan komponenttin renderöidään painike, josta käyttäjä pääsee pelaajavalintaan kierroksen luomisen yhteydessä. Samalla aktivoituu komponentissa määritelty ”handlePlayerSelectionPress”-funktio, joka asettaa roundStoren ”roundInfo”-tilaan valitun radan ja tyhjän taulukon pelaajista, jota muokataan seuraavassa tulevissa näkymissä.

Lopuksi rakensin näkymää varten vielä komponentin, jonka avulla käyttäjä pystyy rakentamaan oman radan ja aloittamaan kierroksen määrittelemillään väylillä. Komponentti aktivoidaan painamalla kuvan 35 alalaidassa näkyvää ”Custom rata”-painiketta. CustomCourseForm-komponentissa (kuva 41) määritellään TextInput, johon käyttäjä syöttää väylän par-lukeman, ja painaa sitten ”Lisää väylä”-painiketta väylän lisäämiseen komponentin sisäiseen tilaan komponentissa määritellyn addFairway-funktion avulla, joka päivittää komponentissa olevan FlatList-komponentin näyttämään lisätyt väylät listana. Lopuksi käyttäjä painaa ”Pelaajavalinta”-painiketta, kuten CourseInfo-komponentissakin pelaajien valitsemiseksi kierrosta varten.

```
const CustomCourseForm: FC<CustomCourseFormProps> = ({ setCreatingCustom }) => {
    const { setRoundInfo } = useRoundStore(state =>
        ({ setRoundInfo: state.setRoundInfo, creatingRound: state.creatingRound }))
    const [fairways, setFairways] = useState<number[]>([])
    const [input, setInput] = useState("")
    const router = useRouter()

    const addFairway = () => {
        if (Number(input)) {
            setFairways(prev => [...prev, Number(input)])
            setInput("")
        } else {
            Toast.show({ type: "error", text1: "Väärä muoto väylälle, laitathan vain numeroita." })
        }
    }

    const handlePlayerSelectionPress = () => {
        const course: CustomCourse = { name: "Custom rata", holes: fairways.map(fw => ({ distance: 0, par: fw })) }
        setRoundInfo({ course, players: [] })
        router.push({ pathname: "/playerselection" })
    }

    return (
        <View style={styles.container}>
            <Toast />
            <BackButton onPress={() => setCreatingCustom(false)} />
            <View style={styles.newCourseContainer}>
                <FlatList data={fairways} renderItem={({ item, index }) =>
                    <View><Text style={styles.fairwayText}>Väylä {index + 1}: Par {item}</Text></View>} />
                <View style={styles.inputContainer}>
                    <Text style={styles.inputText}>Väylä {fairways.length + 1}: Par </Text>
                    <TextInput value={input} style={styles.inputText} onChangeText={(txt) => setInput(txt)}
                        keyboardType='numeric' autoFocus />
                </View>
                <TouchableOpacity onPress={addFairway} style={styles.addFairwayButton}>
                    <Text>Lisää väylä</Text>
                </TouchableOpacity>
                <TouchableOpacity style={styles.playerChooseButton}
                    onPress={() => handlePlayerSelectionPress()} >
                    <Text>Pelaajavalinta</Text>
                </TouchableOpacity>
            </View>
        </View>

    )
}
```

Kuva 41. CustomCourseForm-komponentin koodi

Oheisessa kuvassa (Kuva 42) näkyy kaikki tässä kappaleessa luotujen koodien tuottamat näkymät.



Kuva 42. Karttanäkymä, ikkuna josta näkee radan tiedot sekä ikkuna oman radan lisäykselle

6.5 Pelaajavalintanäkymä

Käyttäjän valittua haluamansa radan siirtyy hän näkymään, jossa hän pääsee valitsemaan kierroksen pelaajat (Kuva 43). Näkymässä käytetään useEffect-koukkua services-hakemistossa sijaitsevan getLocalUserFriends- sekä getAllUserFriends-funktioiden suorittamiseen. Kyseisillä funktioilla haetaan kaikki paikallisesti kännykän muistiin SecureStoreen tallennetut kaverit sekä backendin tietokantaan tallennetut kaverit, ja tallettaa kaikki ne komponentin friends-tilaan. Friends-tilaa hyödynnetään FlatList-komponentissa kaikkien kavereiden renderöimiseen listana. FlatList-komponentissa jokaiselle renderöidylle kaverille asetetaan onPress-funktio, joka aktivoi setPlayers-funktion. Kyseinen funktio asettaa selectedFriends-tilaan kyseisen kaverin. Jos kyseinen kaveri löytyy listalta jo valmiiksi, funktio poistaa kyseisen kaverin listalta. Pelaajan valittua kaikki haluamansa kaverit voi hän painaa Aloita kierros-painiketta, josta alkaa kierros, ja sovellus siirtyy kierrosnäkymään.

```
const PlayerSelectionPage = () => {
    const { roundInfo, setRoundInfo } = useRoundStore(state => ({ roundInfo: state.roundInfo, setRoundInfo: state.setRoundInfo }))
    const [friends, setFriends] = useState<Friend[]>([])
    const { user } = useAuthStore(state => ({ user: state.user }))
    const [selectedFriends, setSelectedFriends] = useState<Friend[]>([])
    const [addingFriend, setAddingFriend] = useState(false)
    const router = useRouter()
    useEffect(() => {
        if (user) {
            getLocalUserFriends().then(data => setFriends(prev => [...prev, ...data]))
            getAllUserFriends(user.id).then(data => setFriends(data))
        }
    }, [])

    const setPlayers = (clickedFriend: Friend) => {
        if (selectedFriends.some(selectedFriend => selectedFriend.id === clickedFriend.id)) {
            setSelectedFriends(selectedFriends.filter(selectedFriend => selectedFriend.id !== clickedFriend.id))
        } else {
            setSelectedFriends(prev => ([...prev, clickedFriend]))
        }
    }

    const startRoundHandler = () => {
        const roundPlayers: RoundPlayer[] = selectedFriends.map(friend => ({ player: friend, scores: [] as number[] }))
        if (user) {
            setRoundInfo({ course: roundInfo!.course, players: [...roundPlayers, { player: { id: Number(user.id), name: user.user }, scores: [] }] })
        }
        router.push({ pathname: "/round" })
    }

    return (
        <View style={styles.container}>
            {addingFriend && <NewNonregisteredFriendForm setAddingFriend={setAddingFriend} setFriends={setFriends} />}
            <FlatList
                data={friends}
                renderItem={({ item }) =>
                    <TouchableOpacity style={selectedFriends.some(selectedFriend => selectedFriend.id === item.id) ? styles.playerCardSelected : styles.playerCard}
                        onPress={() => setPlayers(item)}>
                        <Text style={styles.text}>{item.name}</Text>
                    </TouchableOpacity>}
                keyExtractor={item => item.id + ""}
                style={styles.playerList}
            />
            <View style={styles.buttonContainer}>
                <>
                    <TouchableOpacity style={styles.button} onPress={() => setAddingFriend(prev => !prev)}>
                        <Text style={styles.text}>Lisää uusi pelaaja</Text>
                    </TouchableOpacity>
                    <TouchableOpacity style={selectedFriends.length > 0 ? styles.button : styles.notReadyButton}
                        onPress={() => startRoundHandler()}>
                        <Text style={styles.text}>Aloita kierros</Text>
                    </TouchableOpacity>
                </>
            </View>
        </View>
    )
}
```

Kuva 43. PlayerSelectionPage-näkymän koodi

Kuvassa 43 näkyy myös painike pelaajan lisäämiselle. Tarkoituksenani oli, että jos pelaaja haluaa lisätä kierrokseen jonkin uuden pelaajan, joka ei ole kaverina erilaisista mahdollisista syistä, kykenee hän lisäämään lokaalin kaverin SecureStorea hyödyntäen. Loin uuden komponentin NewNonregisteredFriendForm (Kuva 44). Komponentin TextInput-komponenttiin syötetään pelaajan nimi, ja ”Lisää pelaaja”-painikkeesta lisätään kyseinen pelaaja SecureStoreen käyttäen luomaani addLocalFriend-funktiota. SecureStoreen voidaan tallentaa vain merkkijonoja, ja siten sinne on hankalaa suoriltaan tallentaa taulukon tyyppistä dataa. Käyttäen JavaScriptin sisäistä JSON.parse-funktiota, addLocalFriend jäsentää SecureStoreen merkkijonona tallennetun datan taulukkomuotoon. Sen jälkeen käsiteltävänä olevaan taulukkoon lisätään kaveri, joka on muutettu TypeScript-tyypin Friend muotoon olioksi. Friend-tyyppi vaatii oman id:n nimen lisäksi, jotta

pelaajavalinta toimisi oikein, joten pelaajalle lisätään satunnaisesti generoitu ID hyödyntäen uuid-kirjastoa. Sen jälkeen lista tallennetaan taas merkkijonomuotoisena SecureStoreen. Seuraavaksi pelaajavalintanäkymältä saadun setFriends-funktion avulla lisätään kaveri kyseisessä komponentissa sijaitsevaan tilaan, jotta kaverilista päivittyy ja lopuksi suljetaan ikkuna asettamalla setAddingFriend-funktiolla ikkunan näkyvyyden määrittävä boolean false-arvoiseksi.

```
const NewNonregisteredFriendForm: FC<NewNonregisteredFriendFormProps> = ({ setAddingFriend, setFriends }) => {
    const [input, setInput] = useState("")

    const addLocalFriend = async () => {
        if (input.length < 2 && input.length > 10) {
            Toast.show({ type: "error", text1: "Nimen minimipituus 2 merkkiä, maksimipituus 10 merkkiä!" })
            return
        }
        let friends = await SecureStore.getItemAsync("friends")
        if (!friends) {
            friends = "[]"
        }
        const parsedFriends = JSON.parse(friends)
        if (parsedFriends instanceof Array) {
            const friend = { name: input, id: uuidv4() }
            parsedFriends.push(friend)
            SecureStore.setItemAsync("friends", JSON.stringify(parsedFriends))
            setFriends(prev => [...prev, friend])
            setAddingFriend(prev => !prev)
        }
    }

    return (
        <View style={styles.container}>
            <BackButton onPress={() => setAddingFriend(prev => !prev)} />
            <View style={styles.newPlayerFormContainer}>
                <TextInput style={styles.input} placeholder="Kaverin nimi" value={input}
                    onChangeText={txt => setInput(txt)} maxLength={20} />
                <TouchableOpacity onPress={addLocalFriend} style={styles.button}>
                    <Text style={styles.buttonText}>
                        Lisää pelaaja
                    </Text>
                </TouchableOpacity>
            </View>
        </View>
    )
}
```

Kuva 44. Rekisteröimättömän kaverin lisäyskomponentin koodia

6.6 Kierrosnäkymä

Käyttäjän valittua kierroksen pelaajat siirtyy hän sovelluksen oleellisimpaan vaiheeseen eli itse pisteiden laskuun. Tähän vaiheeseen minulla upposi ylivoimaisesti eniten aikaa verrattuna aikaisempiin vaiheisiin, koska pisteidenlaskuun piti keksiä jonkinlainen toimiva logiikka. Kierrosnäkymässä RoundPage (Kuva 45) haetaan RoundStoresta kierrokseen liittyvät tiedot, sekä asetetaan tila väylän numerolle (holeNumber), näkymässä näytettävän pelaajan indeksille (displayedPlayer) sekä tila, joka määrittää näkyykö tulostaulu vai ei (displayScoreboard). Kaikkia tiloja hyödynnetään neljässä eri elintärkeässä sekä laajahkossa komponentissa, jotka määrittävät pelinäkymässä tapahtuvat asiat, kuten painikkeet pisteiden määrittelemiselle kullekin pelaajalle.

RoundPage-sivulla sijaitseva koodi on suhteellisen monimutkaista monine komponentteineen, ja käyn läpi ne järjestyksessä ylhäältä alas.

```
const RoundPage = ({ }) => {
    const { roundInfo, setRoundInfo } = useRoundStore()
    const [holeNumber, setHoleNumber] = useState(0)
    const [displayedPlayer, setDisplayedPlayer] = useState(0)
    const [displayScoreboard, setDisplayScoreboard] = useState(false)

    if (!roundInfo) {
        return null
    }

    const { course, players } = roundInfo
    const { holes } = course

    const playerScore: number = players[displayedPlayer].scores[holeNumber]
    const par = holes[holeNumber].par

    const lastScore = holeNumber + 1 === holes.length && displayedPlayer + 1 === players.length
        && roundInfo.players[roundInfo.players.length - 1].scores[holeNumber] !== undefined

    return (
        <View style={styles.container}>
            {displayScoreboard && <Scoreboard holeNumber={holeNumber} roundInfo={roundInfo} course={course}
                setDisplayScoreboard={setDisplayScoreboard} lastScore={lastScore} />}
            <HoleInfo distance={holes[holeNumber].distance} holeNumber={holeNumber} par={par} />
            <PlayerInfo player={players[displayedPlayer].player.name} playerScore={playerScore}
                displayedPlayer={displayedPlayer} holeNumber={holeNumber} players={players} lastScore={lastScore}
                setDisplayedPlayer={setDisplayedPlayer} setHoleNumber={setHoleNumber} setDisplayScoreboard={setDisplayScoreboard} />
            <ScoreButtons displayedPlayer={displayedPlayer} holeNumber={holeNumber} holes={holes} par={par}
                players={players} roundInfo={roundInfo} setDisplayScoreboard={setDisplayScoreboard}
                setDisplayedPlayer={setDisplayedPlayer} setHoleNumber={setHoleNumber} setRoundInfo={setRoundInfo} lastScore={lastScore} />
            {lastScore ?
                <TouchableOpacity style={styles.finishButton}>
                    <Text style={styles.finishText} onPress={() => setDisplayScoreboard(true)}>Lopputulokset</Text>
                </TouchableOpacity> :
                <TouchableOpacity style={styles.resultButton} onPress={() => setDisplayScoreboard(true)}>
                    <View style={styles.leaderboard}><Text><AntDesign name="profile" size={50} /></Text><Text>Tulostaulu</Text></View>
                </TouchableOpacity >}

        </View >
    )
}
```

Kuva 45. RoundPage-sivun koodia

HoleInfo-komponentti (Kuva 46) on komponenteista kaikista lyhin, mutta se on silti hyvin tärkeässä asemassa. Komponentti ottaa ominaisuuksina RoundPagelta vastaan tietoa väylästä ja renderöi pelinäkymään väylän numeron, väylän par-lukeman sekä väylän pituuden, jos väylän pituusarvoksi on merkattu yli 0.

```
const FairwayInfo: FC<FairwayInfoProps> = ({ holeNumber, par, distance }) => {
    return (
        <View style={styles.fairwayInfo}>
            <Text style={styles.text}>Väylä {holeNumber + 1}</Text>
            <Text style={styles.text}>Par {par}</Text>
            <Text style={styles.text}>{distance > 0 && `${distance}m`}</Text>
        </View>
    )
}
```

Kuva 46. HoleInfo-komponentin koodia

PlayerInfo-komponentti (Kuva 43) kertoo näkymässä tällä hetkellä olevan pelaajan pelitilanteesta nimen sekä väylän tuloksen muodossa. Komponentti tarjoaa mahdollisuuden navigoida pelattuja väyliä edestakaisin nuolinäppäinten avulla, jolloin pystytään siirtymään väylältä toiselle esimerkiksi muokkaamaan virheellistä tulosta. Tämän logiikan hoitavat handleBackPress- sekä handleNextPress-funktiot.

```
const PlayerInfo: FC<PlayerInfoProps> = ({ player, players, playerScore, displayedPlayer,
    holeNumber, setDisplayedPlayer, setHoleNumber, lastScore }) => {

    const handleBackPress = () => {
        if (displayedPlayer === 0) {
            setHoleNumber(prev => prev - 1)
            setDisplayedPlayer(players.length - 1)
            return
        }
        setDisplayedPlayer(prev => prev - 1)
    }

    const handleNextPress = () => {
        if (displayedPlayer === players.length - 1) {
            setHoleNumber(prev => prev + 1)
            setDisplayedPlayer(0)
            return
        }
        setDisplayedPlayer(prev => prev + 1)
    }

    return (
        <>
            <View>
                <Text style={styles.text}>{player}</Text>
                <Text style={styles.text}>{playerScore && (playerScore > 0 ? `+${playerScore}` : playerScore) + ""}</Text>
                <View style={styles.navigateButtons}>
                    {(holeNumber > 0 || displayedPlayer > 0) ?
                        <TouchableOpacity style={styles.backButton} onPress={handleBackPress}>
                            <Text><AntDesign name="leftcircleo" size={30} color="green" /></Text>
                        </TouchableOpacity>
                        : <TouchableOpacity disabled style={styles.backButton} onPress={handleBackPress}>
                            <Text><AntDesign name="leftcircleo" size={30} color="grey" /></Text>
                        </TouchableOpacity>}
                    {lastScore ? <TouchableOpacity style={styles.backButton} disabled>
                        <Text><AntDesign name="rightcircleo" size={30} color="grey" /></Text>
                    </TouchableOpacity> :
                        <TouchableOpacity style={styles.backButton} onPress={handleNextPress}>
                            <Text><AntDesign name="rightcircleo" size={30} color="green" /></Text>
                        </TouchableOpacity>}
                </View>
            </View>
        </>
    )
}
```

Kuva 47. PlayerInfo-komponentin koodia

ScoreButtons-komponentti (Kuva 48) on itse pisteenlaskennan kannalta olennainen komponentti, sillä se sisältää itse pisteidenlaskennan logiikan. Tässä osassa minulla meni ehkä eniten aikaa koko sovelluksen kehittämisessä. Komponentti renderöi viisi erilaista painiketta pisteiden lisäykselle, ja niitä painamalla aktivoituu handleScorePress-funktio, joka asettaa annetun pistetuloksen roundInfo-tilaan tällä hetkellä käsiteltävälle pelaajalle. Annettuaan pisteet kaikille pelaajille funktio asettaa kierrokselle seuraavan väylän setHoleNumber-funktion avulla, ellei väylä ole viimeinen.

```
const ScoreButtons: FC<ScoreButtonsProps> = ({ displayedPlayer, roundInfo, holeNumber, players, holes,
    setRoundInfo, setDisplayedPlayer, setHoleNumber }) => {

    if (!roundInfo) {
        return null
    }

    const handleScorePress = (playerIndex: number, score?: number) => {
        if (score === undefined) {
            console.log(score)
            return
        }
        setRoundInfo({
            ...roundInfo, players: roundInfo.players.map((player, i) => {
                if (i !== playerIndex) return { ...player }
                player.scores.splice(holeNumber, 1, score)
                return ({ ...player })
            })
        })
        if (displayedPlayer + 1 === players.length) {
            if (holeNumber + 1 === holes.length) {
                return
            }
            setHoleNumber(prev => prev + 1)
            setDisplayedPlayer(0)
        } else {
            setDisplayedPlayer(prev => prev + 1)
        }
    }

    return (
        <View style={styles.scoreContainer}>
            <ScoreButton handleScorePress={handleScorePress} playerIndex={displayedPlayer} text='-...' customScoreStyle='minus' />
            <ScoreButton handleScorePress={handleScorePress} score={-1} playerIndex={displayedPlayer} text='-1' />
            <ScoreButton handleScorePress={handleScorePress} score={0} playerIndex={displayedPlayer} text='0' />
            <ScoreButton handleScorePress={handleScorePress} score={1} playerIndex={displayedPlayer} text='+1' />
            <ScoreButton handleScorePress={handleScorePress} playerIndex={displayedPlayer} text='+...' customScoreStyle='plus' />
        </View>
    )
}
```

Kuva 48. ScoreButtons-komponentin koodia

Pistepainikkeet kuvan 48 komponenttiin on toteutettu ScoreButton-komponenteilla (Kuva 49). Kyseisen komponentin toimivaksi saaminen oli melko monimutkainen prosessi. Koodissa näkyvä alempi return aktivoituu, jos komponentille on syötetty score-ominaisuus ja customScoreStyle ei ole null. Silloin komponentti palauttaa yksinkertaisen pistepainikkeen, jota painamalla aktivoituu handleScorePress-funktio kyseisen komponentin saamalla score-ominaisuudella, jolloin kyseinen score-ominaisuutena määritelty pistemäärä syötetään näkymässä olevalle pelaajalle. Pistemäärille

-1, 0 sekä +1 tein omat kyseistä logiikkaa noudattavat painikkeet, mutta kaikille muille pistemäärille päätin tehdä painikkeet, joihin käyttäjä itse syöttää tuloksen. Jos CustomScoreStyle-merkkijono löytyy ja se on joko ”plus” tai ”minus” ja score-ominaisuutta ei ole määritelty, palauttaa komponentti TextInput-komponentin sisältävän painikkeen, jonka avulla käyttäjä pystyy itse määrittelemään pistetuloksen kyseiselle väylälle. HandleEndEditting-funktio aktivoituu käyttäjän hyväksyessä syöttämänsä tuloksen, ja funktio ensin poistaa kaikki mahdolliset ylimääräiset merkit ja sitten syöttää tuloksen pelaajalle käyttämällä komponentille syötettyä handleScorePress-funktiota.

```
export const ScoreButton: FC<ScoreButtonProps> = ({ text, score, playerIndex, handleScorePress, customScoreStyle }) => {
    const [scoreContent, setScoreContent] = useState(customScoreStyle === "plus" ? "+..." : "-...")

    const handleEndEditting = (playerIndex: number, isPlus: boolean) => {
        const score = scoreContent.replace(/[.,\-_]/g, "")
        const adjustedScore = isPlus ? Number(score) : -Math.abs(Number(score))
        handleScorePress(playerIndex, adjustedScore)
        setScoreContent(customScoreStyle === "plus" ? "+..." : "-...")
    }

    if (score === undefined && customScoreStyle) {
        return (
            <TouchableOpacity style={styles.buttonStyle}>
                {customScoreStyle === "plus" ?
                    <TextInput keyboardType="numeric" onEndEditing={() => handleEndEditting(playerIndex, true)}
                        style={styles.text} placeholder={text} onChangeText={(e) => setScoreContent(e)}
                        onPressIn={() => setScoreContent("+")} >{scoreContent}</TextInput> :
                    <TextInput keyboardType="numeric" onEndEditing={() => handleEndEditting(playerIndex, false)}
                        style={styles.text} placeholder={text} onChangeText={(e) => setScoreContent(e)}
                        onPressIn={() => setScoreContent("-")}>{scoreContent}</TextInput>}
            </TouchableOpacity>
        )
    }

    return (
        <TouchableOpacity onPress={() => handleScorePress(playerIndex, score)} style={styles.buttonStyle}>
            <Text style={styles.text}>{text}</Text>
        </TouchableOpacity>
    )
}
```

Kuva 49. ScoreButton-komponentin koodia

Palataan takaisin kuvan 45 toteutettuun kierroksen päänäkymään RoundPage, jossa edellämainitut komponentit elävät. Komponenttien lisäksi lisäsin päänäkymään displayScoreboard-muuttujan avulla esiin tulevan pistetaulun. Lisäsin näkymän alalaitaan painikkeen, jota painamalla käyttäjä pystyy avaamaan pistetaulun. Koodissa näkyy myös määrittelemäni lastScore-boolean, joka kertoo, onko kierroksen viimeinen väylätulos syötetty. Kyseisen booleanin avulla kierrosnäkymään renderöidään ”Lopputulokset”-painike, joka avaa pistetaulu.

Itse pistetaulu on toteutettu komponentilla Scoreboard (Kuva 50). Komponentti renderöi listan pelaajista ja jokaisen pelaajan omasta tuloksesta, käyttäen countScore-funktiota kaikkien pelaajien taulukossa olevien pisteiden summaamiseen. Komponentissa hyödynnetään myös RoundPagessa määriteltyä lastScore-booleania ilmoittamaan onko kyseessä lopputulokset, ja renderöimään ”Lopeta kierros”-painike jos kierros on todellakin ohi.

```
const countScore = (scores: number[]) => {
    const score = scores.reduce((a, b) => a + b, 0)
    return score > 0 ? `+${score}` : score
}

const Scoreboard: FC<ScoreboardProps> = ({ roundInfo, course, setDisplayScoreboard, lastScore }) => {
    const [selectedPlayer, setSelectedPlayer] = useState<RoundPlayer | null>(null)
    const [displaySaveRoundDialog, setDisplaySaveRoundDialog] = useState(false)

    if (selectedPlayer) {
        return <SinglePlayerScores selectedPlayer={selectedPlayer} setSelectedPlayer={setSelectedPlayer} />
    }

    if (displaySaveRoundDialog) {
        return <SaveRoundDialog roundInfo={roundInfo} />
    }

    return (
        <View style={styles.scoreboardContainer}>
            <BackButton onPress={() => setDisplayScoreboard(false)} />
            <Text style={styles.title}>{course.name}</Text>
            <View style={styles.resultContainer}>
                {lastScore && <Text style={styles.text}>Lopputulokset</Text>}
                <FlatList
                    data={roundInfo.players}
                    renderItem={({ item, index }) =>
                        <TouchableOpacity style={styles.playerCard} onPress={() => setSelectedPlayer(item)}>
                            <Text style={styles.text}>{item.player.name}</Text><Text style={styles.text}>
                                {countScore(roundInfo.players[index].scores)}
                            </Text>
                        </TouchableOpacity>}
                    keyExtractor={item => item.player.id + ""}
                    style={styles.playerList}
                />
            </View>
            {lastScore && <TouchableOpacity style={styles.finishButton}
                onPress={() => setDisplaySaveRoundDialog(true)}>
                <Text>Lopeta kierros</Text>
            </TouchableOpacity>}
        </View>
    )
}
```

Kuva 50. Scoreboard-komponentin koodia

Kuten kuvan 50 komponentissa näkyy, jos selectedPlayer-tilaan on asetettu pelaaja painamalla listassa olevaa pelaajaa, palauttaa komponentti pistetaulun sijaan yksinkertaisen, samantyylisen

SinglePlayerScores-komponentin (Kuva 51), joka kertoo pelaajien väyläkohtaiset pisteet per väylä, sekä kokonaistuloksen.

```
const SinglePlayerScores: FC<SinglePlayerScoresProps> = ({ selectedPlayer, setSelectedPlayer }) => {

    const result = selectedPlayer?.scores.reduce((a, b) => a + b, 0)

    return (
        <View style={styles.scoreboardContainer}>
            <BackButton onPress={() => setSelectedPlayer(null)} />
            <Text style={styles.title}>Pelaajan {selectedPlayer?.player.name} kortti</Text>
            <View style={styles.resultContainer}>
                <FlatList
                    data={selectedPlayer?.scores}
                    renderItem={({ item, index }) =>
                        <View style={styles.scoreCard}>
                            <Text style={styles.text}>Väylä {index + 1}</Text>
                            <Text style={styles.text}>{item > 0 && "+"}{item}</Text>
                        </View>}
                    keyExtractor={item => (item + Math.floor(Math.random() * 1000)) + ""}
                    style={styles.playerList}
                />
            </View>
            <Text style={styles.text}>Kokonaistulos {result! > 0 ? `+${result}` : result}</Text>
        </View>
    )
}
```

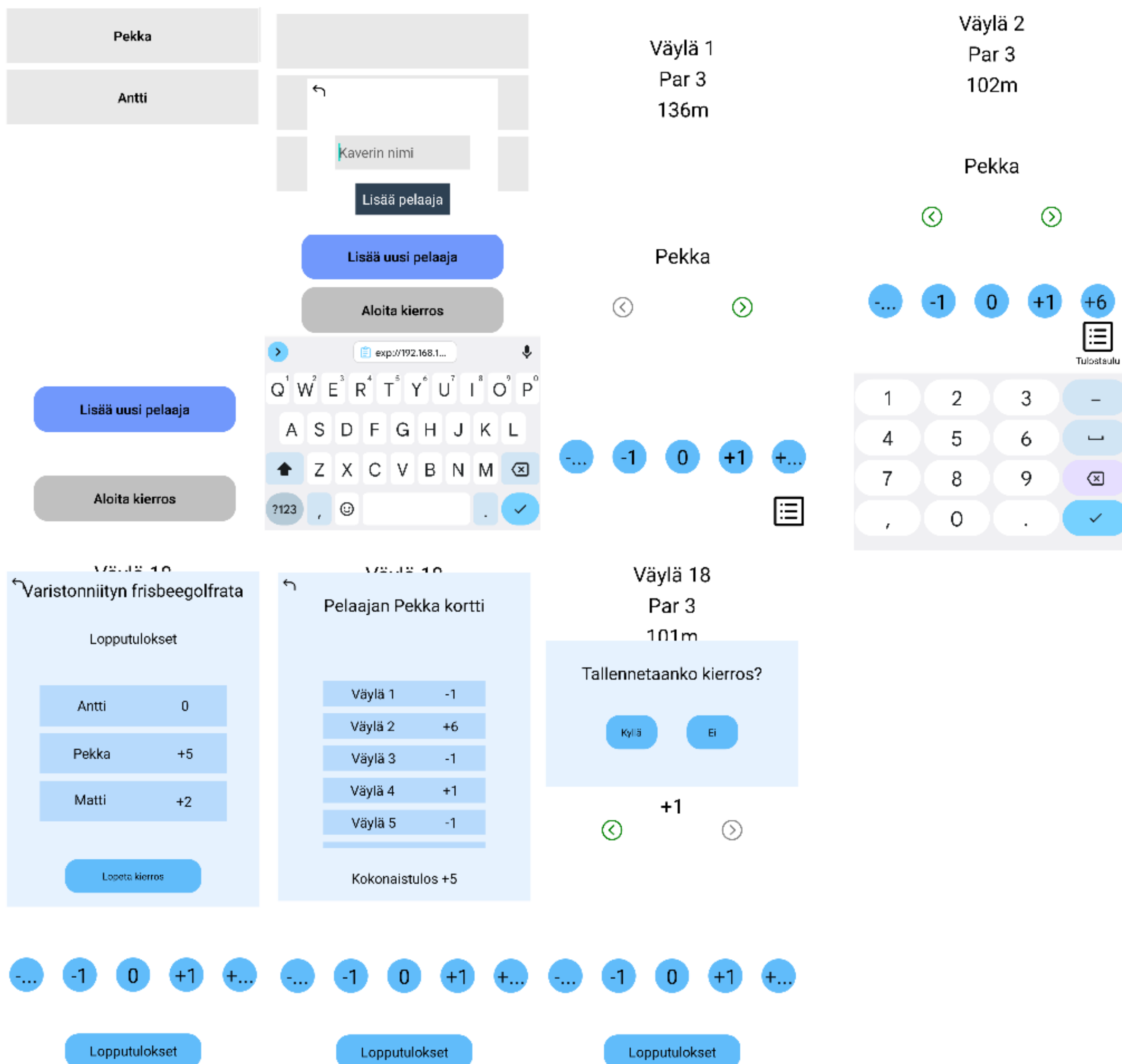
Kuva 51. SinglePlayerScores-komponentin koodia

Kuvan 50 komponentista näkee myös, että kierroksen ollessa ohi, käyttäjän painaessa ”Lopeta kierros”-painiketta pistetaulunäkymässä, palauttaa ScoreBoard-komponentti käyttäen displaySaveRoundDialog-tilaa SaveRoundDialog-komponentin (Kuva 52). Komponentti palauttaa yksinkertaisen ikkunan, joka kysyy käyttäjältä hänen aikomustaan tallentaa kierros tietokantaan vai ei. Käyttäjän painaessa ”Ei”-painiketta, lisätään services-hakemiston addPlayedRound-funktiota käyttäen backendin kautta tietokantaan tieto siitä, että kyseinen käyttäjä on pelannut kyseisen radan, mutta itse pelattua kierrosta ei tallenneta. Jos radan ID on 0, tarkoittaa se sitä, että kyseinen rata on pelaajan itse tekemä ja lisäystä ei tehdä. Jos käyttäjä painaa ”Kyllä”-painiketta, käytetään services-hakemiston addUserRound-funktiota lisäämään tietokantaan pelaajan pelaama kierros samalla kun lisätään tieto pelatusta radasta. Sen jälkeen käyttäjä ohjataan takaisin sovelluksen päänäkymään.

```
const SaveRoundDialog: FC<SaveRoundDialogProps> = ({ roundInfo }) => {
    const user = useAuthStore(state => state.user)
    const router = useRouter()

    const submitData = async () => {
        try {
            if (user) {
                if (roundInfo.course.id !== 0) {
                    await addPlayedRound(user?.id, roundInfo.course.id)
                }
                router.push("/")
            }
        } catch (error) {
            Toast.show({ text1: "Error sending data to server!", type: "error" })
        }
    }

    const submitPlayedRound = async () => {
        if (user) {
            try {
                const roundPlayers = roundInfo.players.map(p =>
                    ({ name: p.player.name, score: p.scores.reduce((a, b) => a + b, 0) }))
                const roundData: Round = { courseId: roundInfo.course.id ?? 0, courseName: roundInfo.course.name, roundPlayers }
                await addUserRound(user?.id, roundData)
                await submitData()
            } catch (error) {
                Toast.show({ text1: "Error sending data to server!", type: "error" })
            }
        }
    }

    return (
        <View style={styles.scoreboardContainer}>
            <Text style={styles.title}>Tallennetaanko kierros?</Text>
            <View style={styles.buttonContainer}>
                <TouchableOpacity style={styles.buttonStyle} onPress={async () => submitPlayedRound()}>
                    <Text>Kyllä</Text>
                </TouchableOpacity>
                <TouchableOpacity style={styles.buttonStyle} onPress={async () => submitData()}>
                    <Text>Ei</Text>
                </TouchableOpacity>
            </View>
        </View>
    )
}
```

Kuva 52. SaveRoundDialog-komponentin koodia

Kuvassa 53 näkyy kierroksen näkymät. Ensimmäinen näkymä on pelaajavalinnasta, jossa painetaan ”Lisää uusi pelaaja”-painiketta. Uusi pelaaja lisätään, pelaajat valikoidaan ja painetaan ”Aloita kierros”-painiketta, joka aloittaa kierroksen valitulla radalla. Kolmannessa näkymässä näkyvillä vaaleansinisillä painikkeilla asetetaan pisteet ruudussa näkyvälle pelaajalle ruudulla näkyvällä väylällä. Nuolinäppäimillä käyttäjä pystyy navigoimaan taaksepäin, esimerkiksi tilanteessa, jossa aikaisempi pisteidenlaskenta on todettu virheelliseksi. Neljännessä näkymässä näkyy käyttäjä asettamassa suurempaa pistemäärää pelaajalle. Viides näkymä demonstroi pistetaulua, joka on avattu kierroksen päätyttyä. Kuudennessa näkymässä näkyy pelaajakohtainen pistetaulu, joka ilmoittaa tulokset per väylä. Viimeisessä näkymässä käyttäjältä kysytään, haluaako hän tallentaa kierroksen järjestelmään.



Kuva 53. Kierroksen näkymiä

6.7 Profiilinäkymä

Päätin toteuttaa frontendin kehityksessä viimeiseksi profiilinäkymän. Jätin sen viimeiseksi, koska käyttäjän profiili ei ole kovin olennainen osa sovelluksen pääasiallista toimintaa ajatellen. Se on kuitenkin tarpeellinen esimerkiksi kaverien hallintaa sekä pelattujen pelikierrosten tarkastelua varten.

Ensimmäiseksi loin profiilinäkymän ProfilePage (Kuva 54). Näkymän komponentti on melko yksinkertainen: se renderöi käyttäjän näytölle käyttäjän nimimerkin, sähköpostin sekä painikkeet,

joiden avulla käyttäjä pystyy navigoimaan joko kaverilistaan tai pelattuihin kierroksiin. Näkymäkomponentti käyttää jo aiemmin esiin tullutta konditionaalilogiikkaa: jos showFriends-tilan booleanin arvo on true, näkyy FriendList-komponentti. Samaa logiikkaa käytetään PlayedRounds-komponentin kanssa.

```
const ProfilePage = () => {
    const user = useAuthStore(state => state.user)
    const router = useRouter()
    const [showFriends, setShowFriends] = useState(false)
    const [showRounds, setShowRounds] = useState(false)

    if (showFriends) {
        return <FriendList user={user} setShowFriends={setShowFriends} />
    }

    if (showRounds) {
        return <PlayedRounds user={user} setShowRounds={setShowRounds} />
    }

    return (
        <View style={styles.container}>
            <StatusBar style='light' backgroundColor='black' />
            <BackButton onPress={() => router.push("/")} />
            <View style={styles.header}>
                <Text style={styles.title}>Profiili</Text>
                <View style={styles.line} />
            </View>
            <View style={styles.userInfoContainer}>
                <Text style={styles.text}>{user?.user}</Text>
                <Text style={styles.text}>{user?.email}</Text>
            </View>
            <TouchableOpacity style={styles.button} onPress={() => setShowFriends(true)}>
                <Text style={styles.buttonText}>Kaverilista</Text>
            </TouchableOpacity>
            <TouchableOpacity style={styles.button} onPress={() => setShowRounds(true)}>
                <Text style={styles.buttonText}>Pelatut kierrokset</Text>
            </TouchableOpacity>

        </View>
    )
}

export default ProfilePage
```

Kuva 54. ProfilePage-näkymän komponentin koodia

Ensin käyn läpi FriendList-komponentin (Kuva 55). Komponentti luo monta tilaa, jota useat eri alikomponentit hyödyntävät. UseEffect-koukulla haetaan services-hakemiston funktioita hyödyntäen backendiltä käyttäjän id:n perusteella käyttäjään kohdistuneet kaveripyynnöt sekä kaikki käyttäjän kaverit, ja ne asetetaan friendships- sekä friendRequests-tiloihin. Eri dialogeille on omat tilat showDelete, showApprove sekä showAdding, jotka määrittelevät näkyykö kyseiset dialogit näkymässä.

```
const FriendList: FC<FriendListProps> = ({ user, setShowFriends }) => {
    const [friendships, setFriendships] = useState<Friendship[]>([])
    const [friendRequests, setFriendRequests] = useState<Friendship[]>([])
    const [selectedFriendship, setSelectedFriendship] = useState<Friendship | null>(null)
    const [showDelete, setShowDelete] = useState(false)
    const [showApprove, setShowApprove] = useState(false)
    const [showAdding, setShowAdding] = useState(false)

    useEffect(() => {
        if (user?.id) {
            getAllFriendRequests(user.id).then(data => setFriendRequests(data)).catch(error => {
                Toast.show({ type: "error", text1: "Virhe näytettäessä kavereita!" })
            })
            getUserFriends(user.id).then(data => setFriendships(data)).catch(error => {
                Toast.show({ type: "error", text1: "Virhe näytettäessä kavereita!" })
            })
        }
    }, [])

    return (
        <View style={styles.friendInfoContainer}>
            {showAdding && <AddFriendDialog setShowAdding={setShowAdding} user={user} />}
            <StatusBar style='light' backgroundColor='black' />
            <BackButton onPress={() => setShowFriends(false)} />
            {showDelete && <DeleteDialog friendship={selectedFriendship} setShowDelete={setShowDelete} user={user} />}
            {showApprove && <ApproveDialog friendship={selectedFriendship} user={user} setShowApprove={setShowApprove} />}
            <View style={styles.header}>
                <Text style={styles.title}>Kaverilista</Text>
                <View style={styles.line}></View>
                <TouchableOpacity style={styles.addingButton} onPress={() => setShowAdding(true)}>
                    <Text style={styles.addingText}>Lähetä kaveripyyntö</Text>
                </TouchableOpacity>
            </View>
            <FriendRequests friendRequests={friendRequests} setSelectedFriendship={setSelectedFriendship}
                setShowApprove={setShowApprove} user={user} />
            <Friendships friendships={friendships} setSelectedFriendship={setSelectedFriendship}
                setShowDelete={setShowDelete} user={user} />
        </View>
    )
}
```

Kuva 55. FriendList-komponentin koodia

Kuvan 55 FriendList-komponentissa näkyy koodattuna painike kaveripyynnön lähettämiselle. AddFriendDialog-dialogikomponentti (Kuva 56) ottaa käyttäjältä vastaan lisättävän kaverin nimen, ja hyödyntää services-hakemiston sendFriendRequest-funktiota pyynnön lähettämiseen backendille. Backend tarkistaa löytyykö nimen mukaista henkilöä tietokannasta tai onko ystävyyssuhde solmittu jo käyttäjän ja kyseisen henkilön välillä, ja toimii sen mukaisesti joko lisäämällä uuden rivin friendships-tauluun tai lähettämällä virheilmoituksen.

```
const AddFriendDialog: FC<AddFriendDialogProps> = ({ setShowAdding, user }) => {
    const [input, setInput] = useState("")
        const addFriend = async () => {
        try {
            if (input.length < 2 && input.length > 10) {
                Toast.show({ type: "error", text1: "Nimen minimipituus 2 merkkiä, maksimipituus 10 merkkiä!",
                 topOffset: 20, position: "top" })
                return
            }
            if (user) {
                await sendFriendRequest(user.id, input)
                setShowAdding(false)
            }

        } catch (error: any) {
            Toast.show({ type: "error", text1: error.message })
        }
    }

    return (
        <View style={styles.container}>
            <BackButton onPress={() => setShowAdding(prev => !prev)} />
            <View style={styles.newPlayerFormContainer}>
                <Toast />
                <TextInput style={styles.input} placeholder="Kaverin nimi" value={input}
                    onChangeText={txt => setInput(txt)} maxLength={20} />
                <TouchableOpacity onPress={addFriend} style={styles.button}>
                    <Text style={styles.buttonText}>
                        Lisää kaveri
                    </Text>
                </TouchableOpacity>
            </View>
        </View>
    )
}
```

Kuva 56. AddFriendDialog-dialogikomponentin koodia

Kuvassa 55 näkyy myös FriendRequests-komponentti. Kyseisen komponentin tehtävänä on renderöidä lista kaikista käyttäjään kohdistuneista kaveripyynnöistä. Komponentin (Kuva 57) sisältö on melko yksinkertainen: se ottaa vastaan päänäkymän friendRequests-tilan, ja renderöi FlatList-komponentin avulla listan kaveripyynnön tehneistä kavereista. Jokainen listassa oleva kaverielementti on painike, jota painamalla aktivoituu handleApproveClick, joka taas asettaa kyseisen kaverin selectedFriend-tilaan ja asettaa showApprove-tilan arvoksi true. Tämä aiheuttaa kuvan 55 FriendList-komponentin renderöimään approveDialog-dialogikomponentin (Kuva 58) renderöinnin. Kyseinen komponentti tarjoaa painikkeet kaveripyynnön hyväksymiselle tai hylkäämiselle, ja molemmat painikkeet hyödyntävät services-hakemiston tarjoamia handleAccept- sekä handleDeny-funktioita, jotka joko hyväksyvät tai hylkäävät kaveripyynnön.

```
const FriendRequests: FC<FriendRequestsProps> = ({ friendRequests, user, setSelectedFriendship, setShowApprove }) => {

    const handleApproveClick = (friend: Friendship) => {
        setSelectedFriendship(friend)
        setShowApprove(true)
    }

    return (
        <View>
            <View style={styles.subheader}>
                <Text style={styles.subtitle}>Kaveripyynnöt</Text>
                <View style={styles.line}></View>
            </View>
            <View>
                <FlatList data={friendRequests} renderItem={({ item }) =>
                    <TouchableOpacity style={styles.friendRequestButton} onPress={() => handleApproveClick(item)}>
                        <Text style={styles.text}>{item.firstUser.username}</Text>
                    </TouchableOpacity>}
                    keyExtractor={(item) => item.id + ""} />
            </View>
        </View>
    )
}
```

Kuva 57. FriendRequests-komponentin koodia

```
const ApproveDialog: FC<ApproveDialogProps> = ({ friendship, user, setShowApprove }) => {

    if (!friendship) return null

    const handleAccept = async () => {
        try {
            if (user && friendship) {
                await handleFriendRequest(user.id, { id: friendship?.id, accept: true })
                setShowApprove(false)
            }
        } catch (error) {
            Toast.show({ type: "error", text1: "Virhe hyväksyttäessä kaveria" })
        }
    }

    const handleDeny = async () => {
        try {
            if (user && friendship) {
                await handleFriendRequest(user.id, { id: friendship?.id, accept: false })
                setShowApprove(false)
            }
        } catch (error) {
            Toast.show({ type: "error", text1: "Virhe hylättäessä kaveria" })
        }
    }

    return (
        <View style={styles.container}>
            <Toast />
            <Text style={styles.title}>Kaveripyyntö henkilöltä {friendship.firstUser.username}.</Text>
            <View style={styles.buttonContainer}>
                <TouchableOpacity style={styles.buttonStyle} onPress={() => handleAccept()}><Text>Hyväksy</Text></TouchableOpacity>
                <TouchableOpacity style={styles.buttonStyle} onPress={() => handleDeny()}><Text>Hylkää</Text></TouchableOpacity>
            </View>
        </View>
    )
}
```

Kuva 58. ApproveDialog-dialogikomponentin koodia

Kuvan 55 koodin alaosassa näkyvä Friendships-komponentti listaa kaikki käyttäjän kaverit. Kyseisen listan koodi on melko samankaltainen kuvan 57 FriendRequests-koodin kanssa: se listaa päänäkymän syöttämän friendships-tilan avulla listan kavereista, ja jokainen kaveri on oma painikkeensa. Painamalla kaveria asetetaan kaveri selectedFriend-tilaan, ja showDelete-tilan arvoksi tulee true. Tällöin näyttäytyy dialogikomponentti DeleteDialog (Kuva 59), joka tarjoaa käyttäjälle mahdollisuuden poistaa käyttäjä kavereista, käyttäen services-hakemiston handleDeleteFriend-funktiota.

```
const DeleteDialog: FC<DeleteDialogProps> = ({ friendship, user, setShowDelete }) => {

    if (!friendship) return null

    const handleDelete = async () => {
        try {
            if (user && friendship) {
                await handleDeleteFriend(user.id, friendship.id)
                setShowDelete(false)
            }
        } catch (error) {
            Toast.show({ type: "error", text1: "Virhe poistaessa kaveria" })
        }
    }

    return (
        <View style={styles.container}>
            <Toast />
            <Text style={styles.title}>Poistetaanko henkilö {friendship.firstUser.username} kavereista?</Text>
            <View style={styles.buttonContainer}>
                <TouchableOpacity style={styles.buttonStyle} onPress={() => handleDelete()}><Text>Kyllä</Text></TouchableOpacity>
                <TouchableOpacity style={styles.buttonStyle} onPress={() => setShowDelete(false)}><Text>Ei</Text></TouchableOpacity>
            </View>
        </View>
    )
}
```

Kuva 59. DeleteDialog-dialogikomponentin koodia

FriendList-komponentin lisäksi kuvan 54 ProfilePage-näkymän komponentti renderöi showRounds-tilan ollessa true PlayedRounds-komponentin (Kuva 60). Komponentti hakee useEffect-koukkua hyödyntäen services-hakemiston getUserRounds-funktiolla kaikki käyttäjän tallentamat kierrokset käyttäjän id:n perusteella, ja listaa ne FlatList-komponentin avulla. Kuten pelaajan kavereiden kanssa, jokainen FlatList-komponentin renderöimä kierros on myös painike, jota painamalla asetetaan displayedRound-muuttujaan kyseinen pelaajan painama kierros.

```
const PlayedRounds: FC<PlayedRoundsProps> = ({ user, setShowRounds }) => {
    const [playedRounds, setPlayedRounds] = useState<ImportedCourseData[]>([])
    const [displayedRound, setDisplayedRound] = useState<ImportedCourseData | null>()

    useEffect(() => {
        if (user?.id) {
            getUserRounds(user.id).then(data => setPlayedRounds(data))
                .catch(err => {
                    Toast.show({ type: "error", text1: "Virhe näytettäessä kierroksia!" })
                })
        }
    }, [])

    if (displayedRound) {
        return <RoundData setDisplayedRound={setDisplayedRound} displayedRound={displayedRound} />
    }

    return (
        <View style={styles.playedRoundsContainer}>
            <StatusBar style='light' backgroundColor='black' />
            <BackButton onPress={() => setShowRounds(false)} />
            <View style={styles.header}>
                <Text style={styles.title}>Pelatut kierrokset</Text>
                <View style={styles.line}></View>
            </View>
            <View style={styles.roundListContainer}>
                <FlatList data={playedRounds} renderItem={({ item }) => (
                    <TouchableOpacity style={styles.roundButton} onPress={() => setDisplayedRound(item)}>
                        <Text style={styles.text}>{item.createdAt}</Text>
                        <Text style={styles.text}>{item.course ? item.course.name : "Custom rata"}</Text>
                    </TouchableOpacity>
                )} style={styles.list} />
            </View>
        </View>
    )
}
```

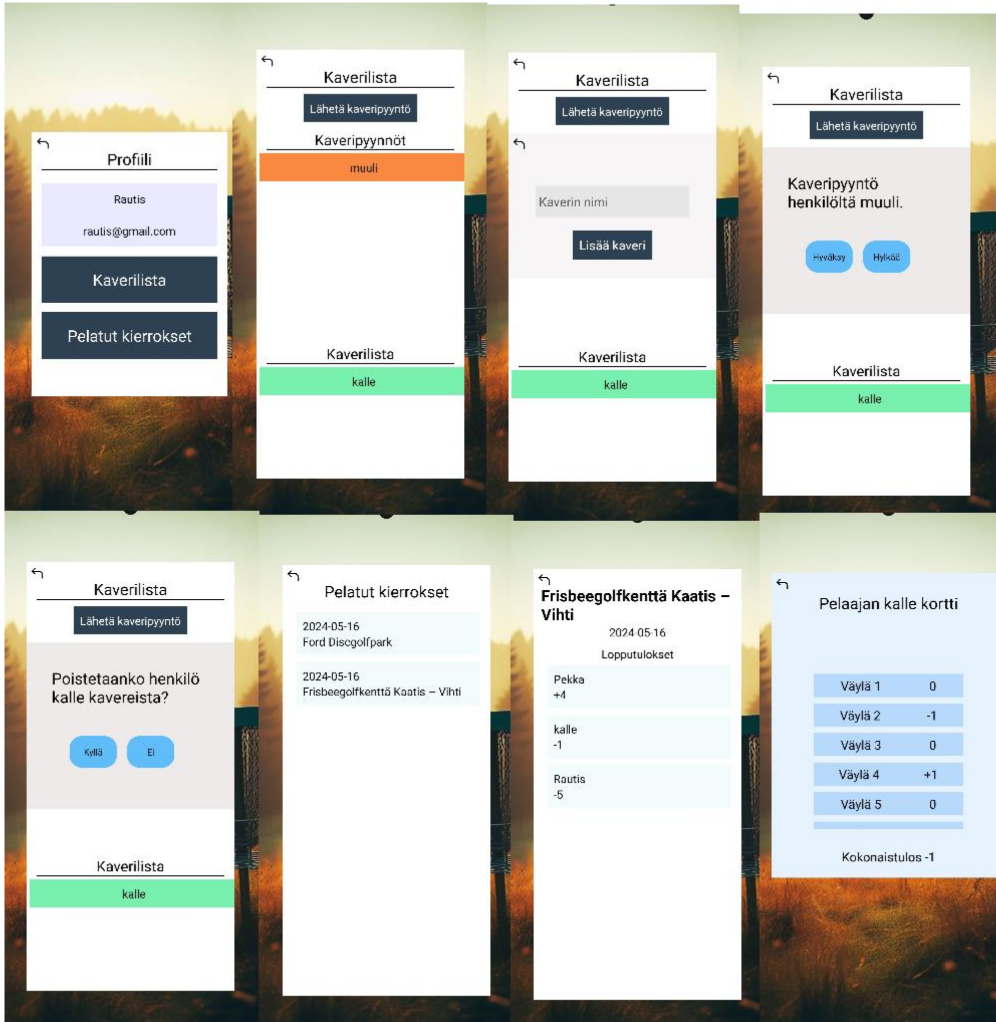
Kuva 60. PlayedRounds-komponentin koodia

Jos kuvan 60 koodissa näkyvä displayedRound ei ole null, palauttaa kyseinen komponentti RoundData-komponentin (Kuva 61), jossa listataan kaikki kierroksella olevat pelaajat. Komponentti kertoo tarkempaa tietoa kierroksesta, kuten kaikki kierrokseen osallistuneet pelaajat lopputuloksineen. Painamalla FlatListin synnyttämää painiketta palauttaa RoundData-komponentti aikaisemmin mainitun, pelikierroksella jo hyödynnetyn SinglePlayerScores-komponentin (Kuva 51), jonka avulla käyttäjä pääsee tarkastelemaan väyläkohtaista tietoa painetusta pelaajasta.

```
const RoundData: FC<RoundDataProps> = ({ displayedRound, setDisplayedRound }) => {
    const [selectedPlayer, setSelectedPlayer] = useState<RoundPlayer | null>(null)
    const parsedPlayers: RoundPlayer[] = JSON.parse(displayedRound.roundPlayers)

    if (selectedPlayer) {
        return <SinglePlayerScores selectedPlayer={selectedPlayer} setSelectedPlayer={setSelectedPlayer} />
    }

    return (
        <View style={styles.playedRoundsContainer}>
            <StatusBar style='light' backgroundColor='black' />
            <BackButton onPress={() => setDisplayedRound(null)} />
            <View style={styles.header}>
                <Text style={styles.title}>{displayedRound.course.name ? displayedRound.course.name : "Custom rata"}</Text>
                <Text style={styles.text}>{displayedRound.createdAt}</Text>
            </View>
            <View style={styles.roundListContainer}>
                <Text style={styles.text}>Lopputulokset</Text>
                <FlatList data={parsedPlayers} renderItem={({ item }) => {
                    const finalScore = item.scores.reduce((a, b) => a + b, 0)
                    return (
                        <TouchableOpacity style={styles.playerButton} onPress={() => setSelectedPlayer(item)}>
                            <Text style={styles.text}>{item.player.name}</Text>
                            <Text style={styles.text}>{finalScore > 0 && "+"}{finalScore}</Text>
                        </TouchableOpacity>
                    )
                }} style={styles.list} keyExtractor={(item) => item.player.name + item.scores} />
            </View>
        </View>
    )
}
```

Kuva 61. RoundData-komponentin koodia

Kuvassa 62 näkyy profiilinäkymän kaverilista sekä pelattuja kierroksia. Kaverilistaan liittyvissä kuvissa demonstroidaan kaverilistaan menemistä, kaverin lisäysdialogia, käyttäjän kaveripyynnön hyväksymisdialogia sekä kaverin poistodialogia. Pelattuihin kierroksiin mentäessä tarkkaillaan pelattuja kierroksia, tarkkaillaan yhtä kierrosta tarkemmin sekä tarkkaillaan tarkemmin yhden tietyn pelaajan pelikorttia kyseiseltä kierrokselta.



Kuva 62. Profiili-näkymän eri komponenttien ulkonäköjä

7 Sovelluskokonaisuuden julkaisu

Tässä vaiheessa sovelluksen frontend sekä backend olivat tuotantovalmiita. Sovelluksella ei ollut mitään rahoitussuunnitelmaa eikä sille odotettu kovin suurta käyttäjävyöryä, joten julkaisualustan suhteen olin hyvin maltillinen. Se ei vaatinut kovin paljon suoritustehoa eikä talletustilaa tietokannalle tässä vaiheessa.

7.1 Backendin julkaisu

Puntaroin eri vaihtoehtoja backendin julkaisualustojen välillä. Backendin saa joihinkin palveluihin pystyyn ilmaiseksi, mutta tällöin se ei välttämättä ole aina saatavilla. PostgreSQL-hostausta tarjottiin myös melko rajatusti omaan pieneen budjettiin nähden.

Päätin toteuttaa backendin pystytyksen hyödyntäen omaa yhden piirilevyn minitietokonettani Raspberry PI:tä. Kyseisellä tietokoneella on erittäin rajattu suorituskyky sekä talletustila, mutta tälle sovellukselle, jolla ei tule ainakaan aluksi olemaan kovin paljon käyttäjiä se on täysin riittävä. Omasin myös valmiiksi kokemusta serverin pystytyksestä Raspberry PI:llä, joten se tuntui luonnolliselta. Myös tietokannan pystytys Raspberry PI:llä on mahdollista, jokseenkin melko rajatulla talletustilalla.

Docker on konttiteknologiaa hyödyntävä alusta, jonka avulla sovellusten jakaminen eri tietokoneiden välillä on helppoa. Dockerilla voidaan pakata sovelluksia eri imageihin, joita pystytään suorittamaan eri ympäristöissä eristetyssä ajoympäristössä hyödyntäen Dockerin tarjoamia Docker Containereja tai Docker-kontteja. (Wallenius 2022)

Hyödynsin backendin tuotantokäyttöön saamisessa Dockeria. Docker on aiheena laaja, ja sen käytöstä tässä kontekstissa voisi kirjoittaa lähes uuden opinnäytetyön. Käyn tekemäni askeleet kuitenkin nopeasti läpi. Ensin pakkasin omalla päätietokoneellani tuotantoon tarkoitetun backend-koodin omaan Docker-imageen, ja laitoin sen jakoon DockerHub-sivustolle, josta hain Raspberry PI:n avulla kyseisen imagen. Sen jälkeen loin Raspberry PI:llä kyseiselle backendille sekä PostgreSQL-instanssille omat kontit, ja käynnistin backendin sekä tietokannan kontit.

Tässä vaiheessa Raspberry PI:llä oli tuotannossa pyörivä backend yhteydessä PostgreSQL tietokantaan, valmiina ottamaan vastaan pyyntöjä. Tein pienen muutoksen frontendin koodiin muuttamalla services-hakemiston axiosInstance.ts-tiedostossa määritellyn, tuotantokäynnössä käyttämäni paikallisen IP:n omaan julkiseen IP:n ja muutin oman reitittimeni asetuksia, jotta pyynnöt IP:lle ohjautuu reitittimelle, jotta frontend pystyy tekemään pyyntöjä backendiin. Tässä vaiheessa hyvin varhainen sekä alkukantainen, mutta toimiva tuotantovaihe backendistä oli käytössä.

Backendin ohjelmakoodin löytää julkisesta GitHub-repositoriostani https://github.com/Rauraurautis/kiekotus-backend.

7.2 Frontendin julkaisu

Frontendin julkaisu oli backendiin verrattuna suoraviivaisempi prosessi. Tarkoituksenani oli alun perin julkaista sovellus Google Play Storeen, mutta päätinkin julkaista kyseisen, hyvin varhaisen version sovelluksesta APK (Android Application Package)-tiedostona. Kyseisen tiedoston pystyy jakamani linkin kautta lataamaan Android-puhelimeen ja Expo go-sovelluksen avulla käyttää sovellusta normaalisti.

Sovelluksen julkaisu ei tässä vaiheessa siis toteutettu iOS-käyttöjärjestelmää hyödyntäville mobiililaitteille, vaikka sovelluksen pitäisi React Native-pohjaisena hybridisovelluksena toimia kaikilla alustoilla.

Kuten backendin ohjelmakoodin, myös frontendin koodin voi löytää julkisesta GitHub-repositoriostani https://github.com/Raurautis/kiekotus.

Frontendin APK-tiedoston pystyy lataamaan puhelimeen osoitteesta https://expo.dev/artifacts/eas/MQ4oNLXx5imHgqVNGUdSU.apk.

8 Pohdinta

Tarkoituksenani toteuttaessani opinnäytetyötä oli luoda laaja sovelluskokonaisuus, jossa on sekä React Nativella toteutettu frontend sekä Node.js:llä toteutettu Backend, joka on yhteydessä PostgreSQL-tietokantaan. Opinnäytetyöraportin tarkoitus oli antaa taustatietoa projektista sekä siinä käytettävistä teknologioista ja raportoida projektin luomisesta elinkaaren alun suunnittelusta lopun julkaisuun asti.

Lopputuotoksena syntyi sovelluskokonaisuus, joka sisältää mobiilisovelluksen, joka on yhteydessä tietokannan kanssa elävään backendiin.

8.1 Onnistumiset sekä epäonnistumiset

Valitsin mielestäni hyvän aiheen, sillä siinä yhdistyy kaksi itseäni kiinnostavaa aihetta – koodaus sekä frisbeegolf. Mielestäni tämä edesauttoi hyvinkin paljon projektin onnistumista, sillä mielenkiinto koodiin pysyi alusta loppuun asti korkealla. Olin myös tietoinen siitä, mitä odottaisin tämän tyyppiseltä sovellukselta, joten pystyin samalla hyödyntämään omaa “asiantuntijuutta” aiheeseen liittyen.

Itse toteuttamaani sovellukseen olen tyytyväinen. Olen hyödyntänyt sovellusta oikeassa elämässä pelatessani frisbeegolfia, ja se toimii moitteetta. Olen erityisesti tyytyväinen pisteidenlaskunäkymään. Mielestäni sovelluksen sisältö vastaa kaikkia suunnitteluvaiheessa sovellukselle asettamiani vaatimuksia.

Sovellusta kehittäessä kysyin välillä itseltäni joidenkin ominaisuuksien tarpeellisuudesta. Esimerkiksi käyttäjänhallinta ei ollut täysin tarpeellinen sovelluksen toiminnan kannalta – olisin hyvin voinut hyödyntää jotain natiivisti käytettävää paikallista tietokantaa tietojen tallentamiseen

sen sijaan että olisin tehnyt sovelluskokonaisuudesta yhtään sen monimutkaisempaa kokonaisuutta. Halusin alun perin kuitenkin tehdä kunnollisen, käyttäjänhallinnan omaavan full stack-kokonaisuuden, ja toteutin suunnittelemani ominaisuudet.

Projektia toteuttaessani kohtasin kuitenkin myös useita puutteita, joita en ollut ottanut suunnitteluvaiheessa huomioon. Esimerkiksi ratojen lisäykselle en ollut keksinyt mitään järkevää tapaa, ja tällä hetkellä radat täytyy lisätä järjestelmään manuaalisesti. Suomessa on lukuisia eri ratoja, ja kaikkien niiden lisääminen yksittäin on hyvin vaivalloinen prosessi. Myös kaverien lisäys tuntuu jokseenkin epäkäytännölliseltä. Sovellusta käyttäessä oikeassa elämässä tuli myös esiin erilaisia puutteita. Esimerkiksi pelaajaa ei pysty kierroksen aikana poistamaan, tai jos pelaaja painaa vahingossa puhelimen natiivia nuolinäppäintä, poistuu koko kierros.

Raportin alussa loin prototyypin projektista, ja pyrin noudattamaan sitä suhteellisen tarkasti. Itse sovellusta käyttäessä kohtasin kuitenkin erilaisia asioita, joita oli jäänyt puuttumaan prototyypistä, kuten mahdollisuus kavereiden lisäykselle. Päädyin myös joissain näkymissä prototyypistä eroaviin design-päätöksiin. Mielestäni protoilu onnistui hyvin, mutta olisin voinut upottaa siihen enemmän aikaa, jotta olisin jo aiemmin tiennyt tasan tarkkaan mitä sovelluksen tulee pitää sisällään.

Raportin kirjoittaminen kävi mielestäni melko vaivattomasti. Oli kuitenkin haastavaa pyrkiä kirjoittamaan niin, että jokin asiasta vähemmän tunteva henkilö ymmärtäisi tekstin. Asetin raporttiin useita kuvia havainnollistamaan koodia, ja yritin saada koodia pilkottua järkeviin osiin, jotta kuvat eivät olisi liian isoja ja tekstistä saisi selvää. Välillä saattoi pilkkomisen takia tuntua, että kuvia on liikaa.

Työtä sekä tätä raporttia toteuttaessani välillä myös tuntui siltä, että valitsin itselleni liian ison palan purtavaksi. Jos olisin valinnut suppeamman aiheen, kuten pelkän backendin toteutuksen jo olemassa olevalle tai kuvitteelliselle frontendille, olisin voinut opinnäytetyössä pureutua syvemmälle aihepiirin saloihin. Mielestäni raportoin frontendin, backendin sekä tietokannan toteutukset melko hyvin, mutta uskon, että olisin voinut selittää aiheista paljon syvällisemminkin. Tästä raportista olisi kuitenkin tullut aivan liian laaja, koska se kattaa kaikki kyseiset osa-alueet.

8.2 Sovelluskokonaisuuden jatkokehitys

Viime aliluvussa mainitsin eri havaitsemistani puutteista, ja niiden perusteella sovellusta voisi sovelluskokonaisuutta jatkokehittää paremmaksi. Esimerkiksi ratojen lisäämiselle tietokantaan voisi suunnitella ja kehitellä jonkin järkevän tavan, ja käyttäjän painaessa vahingossa nuolta kierroksen aikana ei keskeneräisen kierroksen tulisi poistua kokonaan.

Vaikka sovelluksen on tarkoitus olla suhteellisen yksinkertainen sovellus pisteidenlaskua varten, voisi sovellusta mahdollisesti jatkokehittää lisäämällä siihen erilaisia ominaisuuksia, kuten esimerkiksi käyttäjän kavereiden kierrosten tarkastelun. Sovelluskokonaisuuteen voisi myös lisätä erilaista käyttäjäkohtaista statistiikkaa, kuten käyttäjän voittamat sekä parhaiten heittämät ratakohtaiset kierrokset.

Sovelluskokonaisuudesta puuttui kokonaan myös oleellinen osa ohjelmistokehitystä: koodin testaus. Tehokkaan testausjärjestelmän implementointi varmistaisi, että sovellus toimii kehityksessä tehtyjen muutosten jälkeen oikein. Pelkästään backendin sekä frontendin testauksesta voisi kirjoittaa uuden pitkän opinnäytetyön. Sovelluksessa ei ole myöskään otettu yhtään huomioon sovelluksen rahoitusta esimerkiksi lisäämällä mainoksia, tai mahdollisesti jonkin premium-tyylisen tilauksen implementoinnilla.

8.3 Mitä projektista opin?

Kuten aiemmin on selvinnyt, omasin jo valmiiksi melko paljon omakohtaista kokemusta kyseisestä aihepiiristä. Opin projektin aikana kuitenkin paljon vesiputousmalliin liittyvästä vaatimusmäärittelystä sekä suunnittelusta. En ollut aikaisemmin hyödyntänyt niitä omissa projekteissani, ja käytettyäni niitä tässä projektissa olen ymmärtänyt enemmän niiden tärkeydestä. Sovelluskehitys tuntui paljon luontevammalta, kun olin asettanut projektille selkeän päämäärän ja pystyin keskittymään suunnittelun jälkeen suunniteltujen asioiden toteuttamiseen.

Tein tämän opinnäytetyön myös suhteellisen tiukalla aikataululla, ja mielestäni tulin paremmaksi ohjelmistoprojektin aikataulun hahmottamisessa. Uskon, että projekti teki minusta vahvemman ohjelmoijan ja vahvisti valmiuksiani työelämän pyörteille.

Lähdeluettelo

Amazon s.a. What is an API (Application Programming Interface)? Luettavissa: https://aws.amazon.com/what-is/api/. Luettu: 08.04.2024.

Amazon s.a. What is CORS? Luettavissa: https://aws.amazon.com/what-is/cross-origin-resource-sharing/. Luettu: 18.04.2024.

Amazon s.a. What is full-stack development? Luettavissa: https://aws.amazon.com/what-is/full-stack-development/. Luettu: 11.04.2024.

Babladi S. s.a. Refresh Tokens: When to Use Them and How They Interact with JWTs. Luettavissa: https://www.loginradius.com/blog/identity/refresh-tokens-jwt-interaction/. Luettu: 10.4.2024.

Bartosińska I. What Is a Mobile App – All You Should Know as a Future Product Owner. Luettavissa: https://www.thedroidsonroids.com/blog/what-is-a-mobile-app. Luettu: 11.04.2024.

CookiePro s.a. What is an HttpOnly Cookie? Luettavissa: https://www.cookiepro.com/knowledge/httponly-cookie/. Luettu: 16.04.2024.

Dillemuth J. 11.9.2023. 5 syytä valita ReactJS. Luettavissa: https://www.tieturi.fi/blogi/5-syyta-valita-reactjs/. Luettu: 11.04.2024.

Drizzle s.a. Drizzle ORM. Luettavissa: https://orm.drizzle.team/docs/overview. Luettu: 12.4.2024.

Expressjs 2017. Error handling. Luettavissa: https://expressjs.com/en/guide/error-handling.html. Luettu: 13.04.2024.

Fullstackopen s.a. Node.js ja Express. Luettavissa: https://fullstackopen.com/osa3/node_js_ja_express. Luettu: 08.04.2024.

Geeksforgeeks 28.2.2023. What is PostgreSQL – Introduction. Luettavissa: https://www.geeksforgeeks.org/what-is-postgresql-introduction/. Luettu: 09.04.2024.

George C. 15.2.2024. React State Management – using Zustand. Luettavissa: https://medium.com/globant/react-state-management-b0c81e0cbbf3. Luettu: 21.04.2024.

Herrera E. 7.2.2023. UseState in React: A complete guide. Luettavissa: https://blog.logrocket.com/guide-usestate-react/. Luettu: 20.04.2024.

Hurja, 2.2.2023. MVC for dummies: malli, näkymä ja ohjain -arkkitehtuuri web-sovelluksissa. Luettavissa: https://www.hurja.fi/blogi/mvc-for-dummies-malli-nakyma-ja-ohjain-arkkitehtuuri-web-sovelluksissa/. Luettu: 10.04.2024.

Ighosewe E. Should I Use Expo For React-Native. Luettavissa: https://upstackhq.com/blog/software-development/should-i-use-expo-for-react-native. Luettu: 11.04.2024.

Innovationtraining s.a. What is Lucidchart and how to use it for visual collaboration. Luettavussa: https://www.innovationtraining.org/what-is-lucidchart-and-how-to-use-it-for-visual-collaboration/. Luettu: 12.4.2024.

Jordana A. 26.2.2024. What Is JavaScript: A Beginner's Guide to the Basics of JS. Luettavissa: https://www.hostinger.com/tutorials/what-is-javascript. Luettu: 11.04.2024.

LDG s.a. Mitä on frisbeegolf? Luettavissa: https://ldg.fi/?page_id=130. Luettu: 06.04.2024.

Linux.fi s.a. GitHub. Luettavissa: https://www.linux.fi/wiki/GitHub. Luettu: 10.04.2024.

Linuxpolska 14.03.2023. PostgreSQL – the database most frequently chosen by developers and the future of DBMS. Luettavissa: https://linuxpolska.com/en/knowledge-base/blog/postgresql-the-database-most-frequently-chosen-by-developers-and-the-future-of-dbms/. Luettu: 14.05.2024.

Manninen N, Mepham L. 16.2.2022. VSCoden monipuolisuus ilahduttaa opiskelijoita. Luettavissa: https://blogi.oamk.fi/2022/02/16/vscoden-monipuolisuus-ilahduttaa-opiskelijoita. Luettu: 19.10.2023.

Mehta, R. 11.12.2023. Data validation in JavaScript with Zod. Luettavissa: https://www.cloudthat.com/resources/blog/data-validation-in-javascript-with-zod. Luettu: 14.04.2024.

Metwalli S. What is NPM? Luettavissa: https://builtin.com/software-engineering-perspectives/npm. Luettu: 10.04.2024.

P, T. 14.10.2021. Figma – Paras työkalu käyttöliittymien suunnitteluun. Luettavissa: https://webguru.fi/figma. Luettu: 07.04.2024.

Pykes, K. 1.11.2023. An Introduction to Using DALL-E 3: Tips, Examples, and Features. Luettavissa: https://www.datacamp.com/tutorial/an-introduction-to-dalle3. Luettu: 08.04.2024.

Ruokangas J. s.a. React Native: kaikki mitä olet aina halunnut kysyä. Luettavissa: https://www.fraktio.fi/blogi/react-native-kaikki-mita-olet-aina-halunnut-kysya. Luettu: 11.04.2024.

Saavutettavuusmalli.hel.fi 4.3.2021. Moduuli 2, Käyttötapaukset. Luettavissa: https://saavutettavuusmalli.hel.fi/saavutettavuus-palvelukehityksessa/kayttotapaukset/. Luettu: 07.04.2024.

Salescommunications.fi 17.03.2022. Millainen on hyvä vaatimusmäärittely? Luettavissa: https://www.salescommunications.fi/blog/millainen-on-hyva-vaatimusmaarittely. Luettu: 06.04.2024.

Sarja, J. Relaatiotietokanta. Luettavissa: https://verkkopedagogi.net/vanhat/fi/sisalto/materiaalit/access2003/luku0375c6.html. Luettu: 09.04.2024.

Suomen Frisbeegolfliitto s.a. Frisbeegolfin virallinen sääntökirja ja kilpailuopas. Luettavissa: https://frisbeegolfliitto.fi/frisbeegolfin-virallinen-saantokirja-ja-kilpailuopas/. Luettu: 06.04.2024.

Sysart s.a. Vaihtamalla paranee – 3 hyvää syytä valita TypeScript. Luettavissa: https://www.sysart.fi/blog/2017/12/04/vaihtamalla-paranee-3-hyvaa-syyta-valita-typescript. Luettu: 11.04.2024

Tarvainen J. 30.7.2016. Mikä on TypeScript? Luettavissa: https://www.symfony.fi/artikkeli/mika-on-typescript. Luettu: 10.04.2024.

Thinkingportfolio s.a. Kuinka valita sopiva menetelmä projektiin? Luettavissa: https://thinkingportfolio.com/kuinka-valita-sopiva-menetelma-projektiin/ . Luettu: 06.04.2024.

Uxpin s.a. Ant Design 101 – Introduction to a Design System for Enterprises. Luettavissa: https://www.uxpin.com/studio/blog/ant-design-introduction/. Luettu: 26.04.2024.

Visuresolutions.com s.a. Mitä ovat toiminnalliset vaatimukset: esimerkit, määritelmä, täydellinen opas. Luettavissa: https://visuresolutions.com/fi/blogi/toiminnalliset-vaatimukset/. Luettu: 07.04.2024.

Wallenius N. 23.2.2022. Mikä on Docker ja mitä hyötyä siitä on? Luettavissa: https://niklaswallenius.fi/mika-on-docker/. Luettu: 13.05.2024.