Haaga-Helia
ammattikorkeakoulu Oy

Rutiiniseurantasovelluksen suunnittelu ja kehittäminen Android-alustalle

Santtu Hyvärinen

Opinnäytetyö
Tietojenkäsittelyn
koulutusohjelma
31.3.2021

Haaga-Helia
ammattikorkeakoulu Oy

Opinnäytetyö

Tekijä(t)
Santtu Hyvärinen

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Raportin/Opinnäytetyön nimi
Rutiiniseurantasovelluksen suunnittelu ja kehittäminen Android-alustalle

Sivu- ja liitesivumäärä
77 + 0

Opinnäytetyön tehtävänä oli suunnitella, kehittää ja julkaista Android-alustalle sovellus, jonka avulla käyttäjä voi kehittää ja seurata rutiinejansa päivittäisten tehtävien avulla. Sovellus kehitettiin käyttäen Android Studio -kehitysympäristöä, ja sovelluksen ohjelmointi toteutettiin kokonaan Kotlin-ohjelmointikielellä. Projektin lopuksi sovellus julkaistiin Google Play -sovelluskaupassa. Opinnäytetyö keskittyy melkein täysin itse sovelluksen kehittämiseen Android Studio -kehitysympäristössä, eikä opinnäytetyö uppoudu sen syvemmin rutiinien kehittämisen teoriaan.

Kehityksen tavoitteena oli luoda käyttöliittymä, jonka kautta käyttäjä voi luoda, hallita ja seurata rutiineja. Rutiinien pohjalta sovellus luo automaattisesti päivittäisiä tehtäviä, joita käyttäjä voi merkitä suoritetuksi. Sovelluksesta löytyy tilastonäkymä, josta käyttäjä voi seurata rutiiniensa edistymistä erilaisten kaavioiden ja tilastojen avulla.

Android-kehityksessä oli tarkoitus noudattaa hyviä kehityskäytäntöjä Android-dokumentaation suositteleman arkkitehtuurin avulla, jossa käyttöliittymäohjainkomponenttien käyttämä data hallitaan Jetpack-kirjaston arkkitehtuurikomponenttien kautta. Sovelluksen navigaatio toteutettiin käyttämällä Jetpack-kirjaston navigointikomponentteja. Sovelluksen tavoitteena oli toimia täysin paikallisesti Android-laitteessa, joten sovelluksen tietokantaratkaisu toteutettiin Room Persistence -kirjaston ja Kotlin-ohjelmointikielen Coroutine-toimintojen avulla. Sovelluksen suunnittelussa oli tavoitteena ottaa huomioon sovelluksen käyttökokemuksen esteettömyys käyttäen apuna eri työkaluja ja esteettömyydestä kirjoitettuja ohjeita. Sovelluksen esteettömyyttä analysoitiin Accessibility Service -työkalulla.

Asiasanat
Android, Android Studio, Kotlin, Jetpack-kirjastot, Rutiinit, Esteettömyys

Sisällysluettelo

Kuvien ja kaavioiden sisällysluettelo

Sanasto

Activity: Androidin käyttöliittymäohjainkomponentti, joka toimii sovelluksen ikkunana.
Android: Käyttöjärjestelmä mobiililaitteille.
Android Studio: Ohjelmointiympäristö Android-kehitystä varten.
BroadcastReceiver: Android-komponentti, jolla voi vastaanottaa järjestelmälähetyksiä.
Coroutine: Asynkronisen koodin suorittaminen Kotlin-ohjelmointikielessä.
DAO-rajapinta: Tietokannan käsittelyä varten oleva rajapinta, joka määrittää tietokannan käsittelyn metodit.
Entiteetti: Room-kirjaston dataluokka, joka kuvastaa taulua tietokannasta.
Foreground Service: Service-komponentti, joka toimii etualalla.
Fragment: Modulaarinen käyttöliittymäohjainkomponentti Android-kehityksessä.
Git: Versionhallintajärjestelmä.
Google Play: Googlen sovelluskauppa Android-alustalle.
Java: Yksi mahdollisista ohjelmointikielistä Android-kehitykseen.
Jetpack: Kokoelma kirjastoja Android-kehitykseen.
Kotlin: Staattisesti kirjoitettu ohjelmointikieli.
LiveData: Datan pitoluokka, jonka muutoksia muut komponentit pystyvät havainnoimaan.
MVVM-arkkitehtuuri: Model-View-ViewModel-arkkitehtuuri, jossa sovelluksen toiminta on jaettu malliin, näkymään ja näkymämalliin.
NavController: Jetpack-navigointikomponentin osa, joka hoitaa näkymien välillä siirtymisen.
NotificationService: Service-komponentti sovelluksessa, joka hoitaa muistutusten lähettämisen.
Preference: AndroidX Preference -kirjaston komponentti, joka esittää yhtä asetusta.
RecyclerView: Jetpack-kirjaston komponentti, jolla voidaan näyttää dataa lista- tai ruudukkomuodossa.
Room Persistence -kirjasto: Jetpack-kirjasto, jolla voidaan toteuttaa paikallinen tietokantaratkaisu.
Safe Args: Jetpack-navigointikomponentin laajennus, jolla voi välittää dataa navigoinnin yhteydessä.
Service: Android-komponentti, joka toimii taustalla, vaikka käyttäjä ei aktiivisesti käytä sovellusta.
ViewModel: Jetpack-kirjaston käyttöliittymäohjaimen datanhallintakomponentti.

1 Johdanto

Tämä on opinnäytetyöni raportti tietojenkäsittelyn koulutusohjelman tutkintoa varten Haaga-Helian ammattikorkeakoulussa. Opinnäytetyön tyyppi on toiminnallinen ja sen tavoitteena on suunnitella ja kehittää rutiiniseurantasovellus Android-alustalle. Sovelluksen kehitys on tarkoitus toteuttaa käyttäen Android Studio -kehitysympäristöä ja Kotlin-ohjelmointikieltä.

Vaikka toteutettavan sovelluksen aiheena on rutiiniseuranta, opinnäytetyö kuitenkin keskittyy melkein täysin itse sovelluksen kehittämiseen Android Studio -kehitysympäristössä. Opinnäytetyöllä ei ollut toimeksiantajaa, vaan sovellus kehitettiin opinnäytetyötä varten. Rutiiniseuranta valittiin sovelluksen aiheeksi, koska aihe antoi hyvän mahdollisuuden joustavasti suunnitella sovelluksen toimintoja ja ominaisuuksia käytettävien teknologioiden ja kirjastojen mukaan, joihin haluttiin keskittyä sovelluksen kehityksessä.

Tämän opinnäytetyön päätavoitteena on toteuttaa julkaisukelpoinen rutiiniseurantasovellus Androidille. Julkaisukelpoisella tarkoitetaan tässä tapauksessa sovellusta, josta löytyy sovelluksen kuvauksessa kuvatut ominaisuudet ja joka täyttää sovelluksen suunnittelun tavoitteissa kuvatut laatuvaatimukset. Tavoitteena on myös julkaista valmis sovellus Google Play -sovelluskaupassa, jossa se on ilmaiseksi ladattavissa Android-laitteille. Projektin päätteeksi aion myös julkaista Android Studio -projektin lähdekoodin GitHub-palvelussa, jossa se on vapaasti luettavissa.

Oppimistavoitteenani oli perehtyä syvemmin Android-sovellusten suunnitteluun ja hyvien käytäntöjen noudattamiseen kehityksessä. Olen kehittänyt Android-sovelluksia työkseni jo useamman vuoden, ja olen huomannut, että hyvät kehityskäytännöt saattavat unohtua, kun keskittyy pelkästään ominaisuuksien ja toimintojen toteuttamiseen. Ohjelmistokehityksessä voi toteuttaa yhden ominaisuuden niiden monella eri tapaa, mutta nopeat ja karkeat ratkaisut saattavat haitata jatkokehittämistä ja sovelluksen laatua. Vaikka itse toteutettavan sovelluksen laajuus ominaisuuksien kannalta ei ole kovin laaja, mutta se antaa mahdollisuuden keskittyä syvemmin itse sovelluksen suunnitteluun ja Androidin hyviin kehityskäytäntöihin.

Hyviä Android-kehityskäytäntöjä on tarkoitus tarkastella virallisen Android-dokumentaation suositteleman sovellusarkkitehtuurin ja Androidin omien Jetpack-kirjastojen näkökulmasta. Projektissa on tarkoitus noudattaa Android-dokumentaation suosittelemaa arkkitehtuurin rakennetta ja käyttää Jetpack-kirjastojen komponentteja sovelluksen eri toimintojen toteut-

tamisessa. Suunnittelun pitäisi noudattaa hyviä kehityskäytäntöjä, jotta sovelluksen ylläpitäminen ja jatkokehittäminen olisi mahdollisimman helppoa ja modulaarista. Raportin johtopäätöksissä tarkastellaan lähemmin, miten projektin toteutus onnistui Android-kehityksestä kirjoitettujen tutkimuksien ja aikaisempien kokemuksieni Android-kehittäjänä pohjalta.

Toivon, että tämä raportti voi auttaa myös muita kehittäjiä Android-sovellusten suunnittelussa ja kehityksessä hyvien käytäntöjen kautta. Tämän raportin tarkoituksena ei ole kuitenkaan toimia yksityiskohtaisina ohjeina alkaville Android-kehittäjille. Android-sovellusten kehityksen alkeista on kirjoitettu jo paljon materiaalia, joten aion keskittyä raportoimaan sovelluksen toiminnan kannalta oleellisimmat kohdat.

Rutiineista aikaisemmin kirjoitettua kirjallisuutta käytetään lähtökohtana sovelluksen suunnittelussa, mutta opinnäytetyö kuitenkin keskittyy melkein täysin itse kehitysprosessiin. Sovelluksen suunnittelussa otetaan myös esteettömyys huomioon, jotta sovellus olisi selkeä ja helppokäyttöinen mahdollisimman usealle eri käyttäjälle. Sovelluksen esteettömyyttä analysoidaan Accessibility Scanner -työkalun avulla.

Valmiin sovelluksen pitää skaalautua eri kokoisille Android-laitteille. Sovelluksen pitäisi olla helppokäyttöinen ja visuaalisesti miellyttävä sekä älypuhelimessa että tabletissa. Sovelluksen pitäisi myös toimia sekä pystysuunnassa että vaakasuunnassa, ja sen pitäisi pystyä vaihtamaan niiden välillä ongelmitta. Sovelluksen käyttöliittymän kieli pitäisi olla saatavilla sekä suomeksi että englanniksi, ja sovelluksen pitäisi myös tukea uusien käännösten lisäämistä tulevaisuudessa helposti.

Tämän opinnäytetyön tavoitteena ei ole kehittää palvelinpuolta tälle sovellukselle, vaan sovelluksen tarkoituksena on toimia täysin paikallisesti Android-laitteessa. Raportissa ei käsitellä sovelluksen testauksen automatisointia, koska yksikkö- tai integraatiotestauksen toteuttaminen ei kuulunut projektin laajuuteen. Testaus toteutettiin manuaalisesti käyttäen Samsung Galaxy S8 -matkapuhelinta ja Lenovo TB-X605L -tablettia. Molemmissa laitteissa oli Android 9 Pie -käyttöjärjestelmäversio.

1.1 Sovelluksen kuvaus

Sovelluksen ideana on auttaa käyttäjää rutiiniensa seurannassa ja tehdä niiden ylläpitämisestä palkitsevaa ja motivoivaa. Sovelluksen avulla käyttäjä voi pitää kirjaa ja seurata rutiinejansa. Rutiineja voivat olla esimerkiksi kuntosalilla käynti kerran viikossa tai vitamiinien ottaminen kerran päivässä.

Sovelluksessa käyttäjällä on mahdollisuus luoda ja hallita rutiineja. Käyttäjä voi asettaa luomillensa rutiineille muun muassa nimen ja toistuvuuden. Rutiinin voi asettaa toistumaan viikonpäivien tarkkuudella. Käyttäjä voi asettaa rutiinin toistumaan esimerkiksi pelkästään maanantaisin ja torstaisin. Käyttäjä voi muokata ja poistaa luomiaan rutiineja.

Sovelluksen päänäkymässä käyttäjä näkee listan päivittäisiä tehtäviä, jotka luodaan käyttäjän luomien rutiinien pohjalta. Listalta käyttäjä voi merkata tehtävän suoritetuksi, epäonnistuneeksi tai ohitetuksi. Sovellus pitää kirjaa käyttäjän merkkaamista tehtävistä. Sovellus pisteyttää rutiinit niiden tehtävien suoritusten tiheyden perusteella. Kun käyttäjä suorittaa rutiinin tehtävän monta kertaa putkeen, rutiinin pisteytys nousee ja sovellus huomauttaa visuaalisesti käyttäjälle siitä. Jos käyttäjä jättää tehtävän suorittamatta tai merkkaa tehtävän epäonnistuneeksi, tehtävän pisteytys nollataan.

Käyttäjä voi seurata omaa edistymistään tilastonäkymästä. Tilastonäkymässä käyttäjälle näytetään rutiinien ja niiden tehtävien suoritusten tiedot visuaalisesti käyttäen apuna erilaisia kaavioita ja tilastoja. Sovellus myös lähettää käyttäjälle muistutuksia avoinna olevista tehtävistä päänäkymästä. Muistutukset voidaan kytkeä pois päältä sovelluksen asetuksista, jonka kautta käyttäjä voi myös hallita sovelluksen muita toimintoja.

1.2 Käytettävät teknologiat

Sovellus kehitetään Android Studio -ohjelmistokehitysympäristössä ja sovelluksen ohjelmointikielenä käytetään Kotlin-ohjelmointikieltä. Työssäni Android-kehittäjänä vaihdoin Java-ohjelmointikielestä Kotlin-ohjelmointikieleen, koska Kotlin-ohjelmointikieltä oli yleisesti kehuttu paremmaksi Android-kehitykseen kuin Java-ohjelmointikieltä. Aluksi vierastin Kotlin-ohjelmointikielen käyttöönottoa, koska olin tottunut Java-ohjelmointikielen toimintoihin Android-kehityksessä. Kun olin opetellut ja käyttänyt Kotlin-ohjelmointikieltä enemmän, Java-ohjelmointikielellä ohjelmointi alkoi tuntumaan kömpelöltä verrattuna Kotlin-ohjelmointikieleen. Itse suosittelen Kotlin-ohjelmointikieltä Android-kehitykseen Javan yli, koska Kotlin-ohjelmointikielessä minun kokemukseni mukaan syntyy paljon vähemmän turhaa vakiokoodia.

Sovelluksen kehittämissä käytetään Androidin Jetpack-kirjaston arkkitehtuurikomponentteja ja kirjastoja, kuten navigointikomponenttia ja elinkaaritietoisia arkkitehtuurikomponentteja. Sovelluksen paikallinen tietokanta toteutetaan käyttäen Jetpack-kirjastoihin kuuluvaa Room Persistence -kirjastoa. Oppimistavoitteenani on myös perehtyä hieman syvemmin Kotlin-ohjelmistokieleen ja miten se toimii Android-ohjelmistokehityksessä. Esimerkiksi tietokantahaut ja muut asynkroniset funktiot ovat tarkoitus toteuttaa Kotlinin Coroutine-toiminnoilla.

1.3 Raportin kulku

Raportti alkaa varsinaisesti projektin taustasta, jossa käydään läpi projektissa käytettyjä Android-komponentteja ja -kirjastoja. Taustassa kerrotaan sovelluksen kehityksessä käytetystä arkkitehtuurista ja esteettömyyden suunnittelusta Android-alustalla.

Projektin varsinainen toteutus alkaa Android Studio -projektin pystyttämisestä ja sovelluksen navigoinnin toteuttamisesta Jetpack-kirjaston navigointikomponentilla. Navigoinnin jälkeen kuvaillaan, miten sovelluksen skaalautuvuus ja lokalisaatio toteutettiin projektissa. Tietokantaosiossa käydään läpi sovelluksen tietokannan toteutuksessa käytetyn Room Persistence -kirjaston luokkia, ominaisuuksia ja toimintoja sovelluksen tietokantaratkaisun toteutuksen yhteydessä. Sovelluksen näkymät ja toiminnot -osiossa kerrotaan ensin Fragment-komponenttien ja ViewModel-arkkitehtuurikomponenttien toteutuksesta yleisesti sovelluksen eri näkymissä, jonka jälkeen käydään läpi jokaisen sovelluksen näkymän toimintojen toteutus yksitellen. Näkymäkohtaisissa osioissa kerrotaan tarkemmin näkymäkohtaisten toimintojen toteuttamisessa käytetyistä kirjastoista, kuten RecyclerView- ja AndroidX Preference -kirjastoista.

Raportin lopussa ennen johtopäätöksiä kerrotaan vielä sovelluksen esteettömyyden suunnittelusta käyttäen apuna esteettömyydestä kirjoitettuja tutkimuksia ja Accessibility Scanner -sovellusta. Johtopäätöksissä verrataan projektin tuloksia projektille asetettuihin tavoitteisiin Android-kehityksestä kirjoitettujen tutkimusten näkökulmasta.

2 Projektin tausta

2.1 Rutiinit

James Clear määrittelee kirjassaan ”Atomic Habits” (2018), että tottumus tai rutiini voidaan kuvata käyttäytymisenä, jota on toistettu niin useasti, että siitä tulee automaattista. Hän jakaa rutiinin rakentamisen neljään eri askeleeseen: merkkiin, tarpeeseen, reaktioon ja palkintoon. Huomattu merkki ympäristössä laukaisee tarpeen palkinnon saamiseksi, joka taas laukaisee reaktion eli rutiininomaisen käyttäytymisen, jolla palkinto saavutetaan. Tämä toimii palautekierteenä, jonka kautta rutiinit muodostuvat. (luku 3.)

Charles Duhigg määrittelee The Power of Habit -kirjassaan rutiinin samankaltaisesti kuin Clear. Hänen määritelmässänsä on kolme askelta: merkki, rutiini ja palkinto. Hän kirjoittaa, että rutiinit auttavat meitä helpottamaan aivojen työtaakkaa, jolloin aivot voivat keskittyä muihin asioihin. (Duhigg 2013, 17–19.)

Rutiinien seuranta auttaa muistamaan suorittaa rutiinit, tekee niiden suorittamisesta palkitsevampaa ja auttaa näkemään niiden edistyksen, joka myös motivoi rutiinien ylläpitämisessä (Clear 2018, luku 16).

2.2 Kotlin

Kotlin on staattisesti kirjoitettu ohjelmointikieli, jonka on kehittänyt JetBrains. Kotlin ohjelmointikielenä keskittyy koodin selkeyteen, yhteen toimivuuteen ja turvallisuuteen. Kotlin kääntyy samanlaiseen tavukoodiin kuin Java, joten se on yhteen toimiva Javan kanssa ja yhdessä ohjelmassa voi käyttää sekä Kotlinia että Javaa. (Heller 23.3.2020.) Vuoden 2019 toukokuussa Google ilmoitti, että Kotlin-ohjelmointikieli on ensisijainen ohjelmointikieli Android-kehityksessä ja että uudet Jetpack rajapinnan ominaisuudet tulevat ensimmäiseksi Kotlin-ohjelmointikielelle (Lardinois 7.5.2019).

Oliveiran, Teixeiran ja Ebertin tutkimuksessa, jossa selvitettiin Kotlin-ohjelmointikielen käyttöönottoa Android-kehittäjien keskuudessa, selvisi, että kehittäjät pitivät Kotlin-ohjelmointikieltä nykyaikaisena ohjelmointikielenä, joka vähentää koodin määrää ja tehostaa tuottavuutta. Tutkimuksen mukaan kehittäjät pitivät myös Kotlin-ohjelmointikielen etuna sitä, että sen käyttämiseen ei tarvitse siirtyä kokonaan heti, vaan sitä voi käyttää Javan ohella. (Oliveira, Teixeira & Ebert 2020, 214.)

Banerjeen, Bosen, Kundun ja Mukherjeen suorittamassa Java- ja Kotlin-ohjelmointikielien välisessä vertailussa Kotlin-ohjelmointikieltä pidettiin vähemmän alttiimpana ohjelmointivirheille kuin Java-ohjelmointikieltä, ja Kotlinilla pystyi ohjelmoimaan saman asian kuin Javalla vähemmällä koodilla. Vertailussa kuitenkin suositeltiin Java-ohjelmointikieltä aloittaville kehittäjille, koska Javalle oli kirjoitettu enemmän ohjeita ja materiaalia avuksi Android-kehitykseen kuin Kotlin-ohjelmointikielelle. (Banerjee, Bose, Kundu & Mukherjee 2018, 44.)

Arditon, Coppolan, Malnatin ja Torchianon tutkimus (2020, luvut 6.2 ja 7), joka vertaili Kotlin- ja Java-ohjelmointikielien tehokkuutta Android-kehityksessä, myös löysi, että Kotlin-ohjelmointikielellä tuotti heidän vertailussansa tiiviimpää koodia kuin Java, vaikka pienien projektien koodin ylläpitämisen kannalta tutkimus ei löytänyt merkittäviä eroja ohjelmointikielten välillä.

Kotlin-ohjelmointikieli on suunniteltu vähentämään tyhjien viittausten käsittelemisestä syntyviä virheitä. Kotlin-ohjelmointikielessä voidaan pitää määritellä erikseen, että voiko olioon viittaus olla myös tyhjä. Oletuksena viittaukset eivät voi olla tyhjiä. Olion muuttujia ja metodeja voidaan kutsua turvallisesti, vaikka viitattava olio olisikin tyhjä. (Kotlin Programming Language 2020.) Arditon, Coppolan, Malnatin ja Torchianon tutkimuksen (2020, luku 6.3) vertaisussa selvisi, että Kotlin-ohjelmointikielellä kehittäessä tuli vähemmän NullPointerException-virheitä kehityksen aikana kuin Java-ohjelmointikielellä.

2.3 Android-kehitys

Android-kehitykseen liittyvä teoriataustani pohjautuu suurimmaksi osin Android Developers -sivuston Android-sovelluskehityksen dokumentaatioon. Suuri osa Android-kehityksestä kirjoitetuista artikkeleista ja oppaista näyttivät minun selvitykseni perusteella pohjautuvan tähän dokumentaation joko suoraan tai epäsuoraan. Minun kokemukseni mukaan tämä dokumentaatio selittää selkeästi ja kattavasti eri komponenttien toiminnan ja käyttöönoton suhteellisen neutraalisti. Dokumentaatio kuitenkin luonnollisesti suosittelee omien kirjastojensa ratkaisuja ja käytäntöjä kolmansien osapuolien ratkaisujen yli.

2.3.1 Android Studio

Android Studio on virallinen ohjelmistokehitysympäristö Android-sovellusten kehitystä varten. Android Studio perustuu IntelliJ IDEA -ohjelmointiympäristöön. Android Studion mukana tulee muun muassa Android-emulaattori ja sekä virheenjäljitys- että suorituskyvyn profilointityökalut. (Android Developers 2020a.)

Google julkisti Android Studion ensimmäisen kerran I/O konferenssissa vuonna 2013. Android Studion tarkoituksena oli toimia virallisena kehitysympäristönä Androidille ja korvata aikaisempi Eclipse-ohjelmistokehitysympäristö. Android Studion ensimmäinen vakaa versio julkaistiin vuonna 2014. (Protalinski 8.12.2014.)

2.3.2 Android Studio -projekti

Android Studio käyttää Gradle-työkalua Android Studio -projektin kääntämiseen sovellukseksi. Gradle-työkalu kääntää projektin resurssitiedostot ja ohjelmistokoodin APK-tiedostoon, jolla sovellus voidaan asentaa Android-laitteeseen. Android Studio luo automaattisesti tarvittavat Gradle-käännöskonfiguraatiotiedostot projektille, kun projekti luodaan. Projektin tarvitsemat riippuvaisuudet pitää olla määritettynä käännöskonfiguraatiotiedostoissa. Android-käännösautomaatiojärjestelmä tukee useiden eri käännösversiokonfiguraatioita käyttämistä. Sovelluksesta voi kääntää esimerkiksi erikseen versiot testausta ja julkaisua varten. (Android Developers 2020b.)

Activity-komponentti on tärkeä osa sovelluksien toiminnan kannalta Android-järjestelmässä. Activity-komponentti toimii ikkunana sovelluksen käyttöliittymälle, ja sen kautta käyttäjä käyttää sovellusta. (Android Developers 2020c.)

Fragment-komponentti kuvastaa sovelluksen käyttöliittymän osaa. Fragment-komponentit ovat modulaarisia ja uudelleenkäytettäviä, joten sovelluksen käyttöliittymässä on mahdollista esittää useita eri Fragment-komponentteja samaan aikaan. Fragment-komponentilla on oma elinkaarensa, ja se hoitaa oman käyttöliittymänsä sisällön asettelun ja kosketustapahtumat. (Android Developers 2020d.)

Androidin Service-komponentti eli palvelu suorittaa toimintoja taustalla, vaikka käyttäjä ei käyttäisikään sovellusta aktiivisesti. Service-luokka toimii pohjana kaikille palveluille. Service-komponentti suorittaa ohjelmakoodia oletuksena sovelluksen pääsäikeellä, joten komponentti saattaa hidastaa sovelluksen käyttöliittymäkomponenttien toimintaa. Tämän takia suositellaan, että raskaat ohjelmatoiminnot suoritetaan toisella säikeellä Service-komponentissa. (Android Developers 2021a.)

Android-järjestelmä ja Android-sovellukset lähettävät eri tapahtumista lähetyksiä, joita muut sovellukset voivat vastaanottaa. BroadcastReceiver-luokka toimii pohjana lähetyksiä vastaanottavalle koodille. Sovellus voi rekistöröidä vastaanottamaan tiettyjä järjestelmälähetyksiä, kuten laitteen käynnistyksestä kertovan lähetyksen. (Googler 2020.)

Jokaisella projektilla Android Studiossa pitää olla mukana AndroidManifest.xml-tiedosto, joka sisältää tietoa sovelluksesta Android-koontityökaluja, -käyttöjärjestelmää ja Google Play -kauppaa varten. AndroidManifest.xml-tiedostossa pitää olla lueteltuna muun muassa sovelluksen käyttämät Activity-, Service- ja Receiver-komponentit. Sovelluksen tarvitsemat käyttöoikeudet ovat myös määritelty tässä tiedostossa. (Android Developers 2020e.)

Android-projektilla on omat resurssikansiot, jotka säilyttävät sovelluksen käyttämät staattiset resurssit ja tiedostot. Resurssit on kategorisoitu eri resurssityyppeihin ja niitä säilytetään niiden omissa alikansioissa. Resurssityyppejä ovat muun muassa näkymän asettelutiedostot, värit, tekstit, animaatiot ja tyylit. Sovellukselle voi luoda myös konfiguraatioriippuvaisia resurssikansiota, jotka ladataan sovelluksen ajoaikana laitteen konfiguraation perusteella. (Android Developers 2020f.)

2.3.3 Jetpack-kirjastot

Android Jetpack on kokoelma kirjastoja, joiden tavoitteena on tukea hyviä kehityskäytäntöjä ja vähentää toistuvan koodin määrää Android-kehityksessä (Android Developers 2020g). Jetpack-kirjastot ovat rakennettu yhteensopivaksi myös vanhempien Android-versioiden kanssa (Protalinski 8.5.2018).

Verdecchian, Malavoltan ja Lagon suorittamassa tutkimuksessa selvitettiin osakysymyksenä, että mihin kirjastoihin viitataan eniten kirjallisuudessa, missä mietitään Android-sovelluksen arkkitehtuuria. Tutkimuksen mukaan Jetpack-kirjasto oli kolmanneksi viitatuin kirjasto RxJava- ja Dagger-kirjastojen jälkeen, vaikka Jetpack-kirjasto oli vasta julkaistu tutkimuksen julkaisuaikaan. (Verdecchia, Malavolta & Lago 2019, 144.) Vaikka tutkimuksen datasetissä Jetpack-kirjastoon oli viitattu suhteellisen vähän, mutta se osoittaa kuitenkin, että Jetpack-kirjasto esiintyy jo Android-arkkitehtuurista kirjoitetusta kirjallisuudesta sen uutuudesta huolimatta. Google julkaisi Android Jetpack -kirjaston toukokuussa 2018 (Protalinski 8.5.2018).

Jetpackin Room Persistence -kirjasto mahdollistaa datan tallentamisen paikalliseen tietokantaan. Kirjasto toimii rajapintana SQLite-tietokannan päällä, joka mahdollistaa tietokannan sujuvan käytön ilman turhaan toistuvaa vakiokoodia. Kirjasto tarjoaa myös SQL-lauseiden varmistuksen ohjelmakoodin käännöksen aikana. (Android Developers 2020h.)

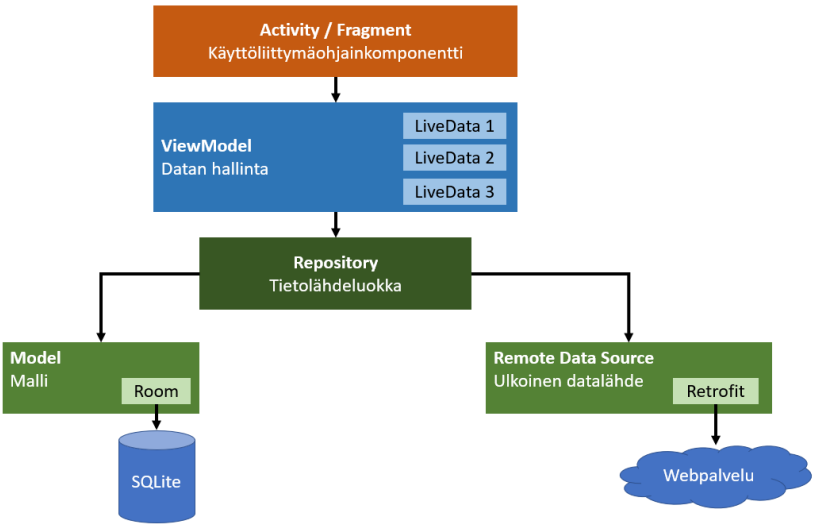
Jetpack-kirjaston navigointikomponentti auttaa Android-sovelluksen navigoinnin toteuttamisessa. Navigointikomponentti hoitaa muun muassa Fragment-komponenttien eli näkymien vaihtamisen ja linkityksen, ja sen voi integroida helposti navigointikäyttöliittymäkomponentteihin, kuten alanavigointipalkkiin. (Android Developers 2020i.)

Jetpack-kirjastosta löytyy RecyclerView-komponentti, jota voidaan käyttää ison datasetin esittämiseen esimerkiksi lista- tai ruudukkomuodossa. RecyclerView-komponentti pystyy esittämään suuria datasettejä hyvällä suorituskyvyllä. RecyclerView-komponenttiin liitettävä Adapter-komponentti ja ViewHolder-komponentti määrittävät yhdessä, miten datasetti näytetään RecyclerView-komponentissa. ViewHolder-komponentti sisältää näkymän asetelman yksittäiselle datasetin osalle. Adapter-komponentti luo ja hallitsee näitä ViewHolder-komponentteja ja sitoo datasetin datan niihin. (Android Developers 2020j.)

2.3.4 Arkkitehtuuri ja arkkitehtuurikomponentit

Android Developers -sivuston dokumentaatio suosittelee välttämään käyttöliittymäohjainkomponenttien eli Activity- ja Fragment-komponenttien käyttöä sovelluksen käyttämän datan lähteenä. Käyttöliittymäohjainkomponenttien suositellaan käyttävän Jetpack-kirjastojen arkkitehtuurikomponentteja, kuten ViewModel- ja LiveData-komponentteja, datan hallintaan. Dokumentaatio suosittelee myös jakamaan sovelluksen toiminnallisuudet mahdollisimman itsenäisiin moduuleihin, joille on määritelty selkeät vastuut. (Android Developers 2021b.)

Alla olevassa kaaviossa (Kaavio 1) näkyy Android-dokumentaation suositteleman arkkitehtuurin rakenne, missä Activity- tai Fragment-komponentti toimii käyttöliittymänohjaimena ja ViewModel-komponentti paljastaa tietolähdeluokan kautta haetun datan käyttöliittymäohjaimelle LiveData-komponenttien avulla. Tietolähdeluokka ei ole osa Jetpack-kirjaston arkkitehtuurikomponentteja, mutta sitä suositellaan koodin erottamisen ja arkkitehtuurin vuoksi (Muntenescu 2021).



Kaavio 1. Android-dokumentaation suosittelema sovellusarkkitehtuuri (Mukaillen Android Developers 2021b)

Android-dokumentaation suosittelemaa arkkitehtuurin muotoa voisi kuvailla Model-View-ViewModel-arkkitehtuuriksi eli MVVM-arkkitehtuuriksi, koska miten siinä käytetään ViewModel-komponenttia, kuten nimestä voi päätellä. Verdecchian, Malavoltan ja Lagon tutkimuksessa (2019, 144) odotettiin, että MVVM-arkkitehtuurin käyttö Android-kehityksessä saattaa kasvaa tulevaisuudessa nopeasti ViewModel-komponentin julkaisun seurauksena.

John Gossmann esitteli MVVM-arkkitehtuurin blogissaan, jossa hän kuvaa kyseisen arkkitehtuurin toimintaa. Hän määrittelee, että MVVM-arkkitehtuurissa näkymäkomponentti eli ”View” sisältää ohjelman käyttöliittymän ja mallikomponentti eli ”Model” sisältää ohjelman käyttämän datan. Näkymämallikomponentti eli ”ViewModel” hallitsee dataa näkymäkomponentin ja mallikomponentin välillä. Hän sanoo kanssa, että MVVM-arkkitehtuurissa käytetään datasidontaa sitomaan näkymäkomponentin attribuutteja näkymämallin hallitsemaan dataan. (Gossmann 8.10.2005.)

Loun pro gradu -tutkielmassa, jossa verrattiin eri arkkitehtuureja Android-kehityksessä, Model-View-ViewModel- ja Model-View-Presenter-arkkitehtuurit olivat kehitettävyydeltään, testattavuudeltaan ja suorituskyvyltään tehokkaampia kuin Model-View-Controller-arkkitehtuuri. Tutkielma ei löytänyt merkittäviä eroja MVVM- ja MVP-arkkitehtuurien välillä paitsi, että MVVM-arkkitehtuurilla oli parempi testattavuus, mutta huonompi muokattavuus. (Lou 2016, 39–40.) Vaikka tutkielma suoritettiin noin kaksi vuotta ennen Jetpack-kirjaston julkaisua, MVVM-arkkitehtuurin käyttöönoton hyödyt kuitenkin pitäisi päteä Jetpack-kirjaston arkkitehtuurikomponenttien kanssa.

Verdecchian, Malavoltan ja Lagon tutkimuksessa koottiin ohjeita Android-arkkitehtuuriin eri arkkitehtuurityylien näkökulmasta, kuten MVVM-arkkitehtuurin. Tutkimuksen ohjeet koottiin Android-arkkitehtuurista kirjoitetun kirjallisuuden pohjalta. MVVM-arkkitehtuurikohtaisia neuvoja olivat muun muassa, että näkymämallikomponentit kannattaa pitää mahdollisimman yksinkertaisena ja että näkymämallikomponentit eivät saa viitata näkymäkomponentteihin. (Verdecchia, Malavolta & Lago 2019, 145—147.)

Android-kehityksessä ViewModel-komponenttien tarkoitus on valmistella ja hallita dataa käyttöliittymäohjainkomponentteja varten. Android-järjestelmä luo käyttöliittymäohjainkomponentit uudelleen konfiguraatiomuutoksien, kuten esimerkiksi näytön kääntämisen, aikana, jonka vuoksi kaikki ohjainkomponenttien sisällä tallennettu data katoaa konfiguraatiomuutoksen jälkeen. Toisin kuin käyttöliittymäohjainkomponentteja, ViewModel-kompo-

nenttia ei tuhota ja ladata uudelleen konfiguraatiomuutoksien takia. Kun käyttöliittymäohjainkomponentti uudelleen luodaan konfiguraatiomuutoksen seurauksena, ViewModel-komponentti kiinnittyy siihen automaattisesti. ViewModel-komponentin elinkaari jatkuu, kunnes sitä vastaava käyttöliittymäohjainkomponentti suljetaan. (Android Developers 2020k.)

LiveData on elinkaaritietoinen datan pitämiseen tarkoitettu luokka, jonka sisältämän datan muutoksia käyttöliittymäohjainkomponentit ja muut komponentit voivat seurata. Koska LiveData-komponentti on elinkaaritietoinen, se ilmoittaa sen pitämän datan muutoksista vain sitä seuraaville käyttöliittymäohjainkomponenteille, jotka ovat aktiivisia. Käyttöliittymäkomponentit lopettavat automaattisesti LiveData-komponentin seuraamisen, kun ne tuhotaan. LiveData-komponentin avulla voidaan varmistaa, että käyttöliittymä näyttää aina ajantasaista dataa. (Android Developers 2021c.)

2.4 Esteettömyys

Sovelluksen käytettävyydessä voi olla esteitä, jotka tekevät sovellusten käyttämisen hankalaksi henkilöille, jotka käyttävät sovellusta aputeknologian, kuten näytönlukijan, avulla. Esteitä sovelluksen käytölle voivat olla muun muassa puuttuvat sisällönkuvaustekstit kuvapainikkeissa, joita näytönlukijat tarvitsevat toimiakseen, tai liian pienet kosketusalueet, joiden painaminen vaatii tarkkuutta. Myös useat identtiset tai epäselvät sisällönkuvaustekstit saattavat hämmentää käyttäjiä, jotka käyttävät näytönlukijaa sovellusta käyttäessään. (Ross, Zhang, Fogarty & Wobbrock 2020, 11—14.)

Vendomen, Solanon, Linánin ja Linares-Vásquezin tutkimuksessa (2019, 44), jossa analysoitiin Android-sovellusten esteettömyyttä, selvisi, että melkein puolelta tutkimuksessa analysoidulta Android-sovellukselta puuttui sisällönkuvaustekstit ainakin puolelta sovelluksen elementeistä. Myös Rossin, Zhangin, Fogartyn ja Wobbrockin tutkimus (2020, 2) löysi, että heidän testaamiltaan sovelluksista 46 prosentilta puuttui sisällönkuvaustekstit ainakin 90 prosentilta kuvakepainikkeista. Molemmissa tutkimuksessa analysoitiin useita tuhansia eri Android-sovelluksia, joten sisällönkuvaustekstien puuttuminen sovelluksista ei vaikuta olevan harvinainen ongelma.

Android-sovelluksen esteettömyyden suunnittelussa voi hyödyntää erilaisia työkaluja ja esteettömyydestä kirjoitettuja ohjeistuksia. Material Design -ohjeistuksessa (s.a.) on oma osionsa mobiilikäyttöliittymien esteettömyydelle, jossa neuvotaan muun muassa sovelluksen väreistä, fonteista ja kosketusalueista esteettömyyden parantamista varten.

Google on kehittänyt Accessibility Scanner -sovelluksen Android-käyttöjärjestelmälle, jonka avulla Android-kehittäjä voi tunnistaa ongelmia sovelluksensa esteettömyydessä. Kun Accessibility Scanner -sovellus skannaa kuvankaappauksen sovelluksesta, se ehdottaa parannuksia sovelluksen esteettömyyden parantamiseksi. (Google Help 2020.) Vendome, Solano, Linán ja Linares-Vásquez (2019, 49) sanoivat heidän Android-sovellusten esteettömyyttä analysoivassa tutkimuksessa, että vaikka Accessibility Scanner -sovellus tarjoaa eri esteettömyyden testausmahdollisuuksia, se ei ole kaiken kattava ratkaisu esteettömyyden testaamiseen. Elerin, Rojasen, Gen ja Fraserin tutkimuksessa (2018, 117), jossa selvitettiin esteettömyyden testauksen automatisointia, todettiin, että Accessibility Scanner -sovellus ei kuitenkaan ehkä skaalaudu isoimmille sovelluksille, koska kehittäjän pitää manuaalisesti käydä läpi kaikki sovelluksen näkymät läpi.

3 Projektin aloitus

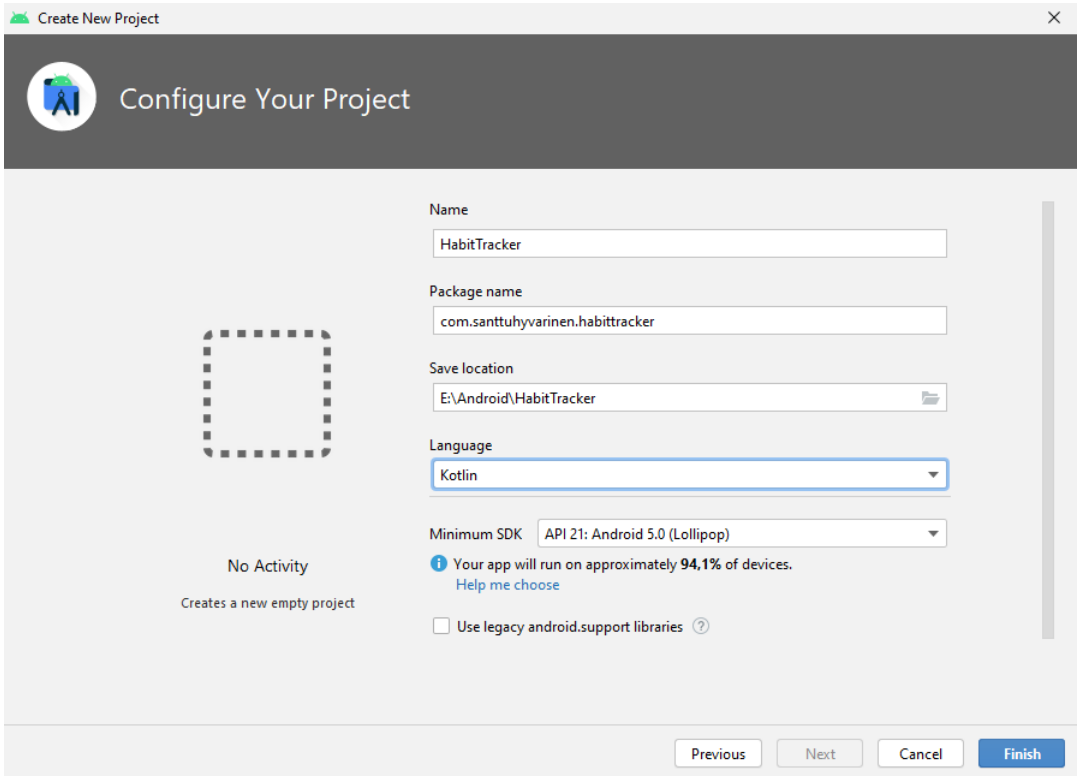
3.1 Versionhallinta

Projektin alussa heti ensimmäiseksi pystytin Git-tietovaraston GitHub-palveluun omalla henkilökohtaisella GitHub-tunnuksellani. Annoin Git-tietovarastolle nimeksi HabitTrackingApp. Tarkoituksena oli käyttää tätä Git-tietovarastoa Android Studio -projektini versionhallintaan.

Suunnitelmani oli soveltaa projektin versionhallinnassa Gitflow Workflow -työnkulkua (Atlassian s.a.), jossa projektin historia tallennetaan eri oksiin versionhallinnassa, joilla on eri käyttötarkoitus. Tämän projektin versiohallinnassa master-oksaan tallennetaan julkaisukelpoinen ja testattu versio sovelluksesta. Development-oksassa eli kehitysoksassa tehdään varsinainen kehitystyö. Koska työskentelin tässä projektissa yksin, en nähnyt tarvetta luoda jokaiselle uudelle ominaisuudelle omaa oksaa, joten ison osa kehitystyöstä aioin tehdä kehitysoksassa. Kuitenkin aion tehdä tarvittaessa joidenkin ominaisuuksien kehitykselle omat oksat versionhallintaan, jos ominaisuus oli tarpeeksi laaja ja itsenäinen kokonaisuus.

3.2 Android Studio projektin pohja

Käytin projektissa Android Studion versiota 4.1.1, joka oli projektin aloittamisaikana Android Studion uusin versio. Aivan ensimmäiseksi Android Studiossa loin uuden projektin kehitettävälle rutiiniseurantasovellukselleni. Android Studio tarjoaa projektimalleja, joiden avulla voi luoda projektin, jossa on muun muassa alanavigaatiopalkin runko valmiina. Projektimallit ovat käteviä, jos haluaa nopeasti päästä aloittamaan projektin. Halusin kuitenkin aloittaa projektin mahdollisimman tyhjästä, joten valitsin No Activity -mallin. Projektin alkukonfiguraatiossa (Kuva 1) annoin projektille nimeksi HabitTracker ja asetin sovelluksen paketin nimeksi com.santtuhyvarinen.habittracker. Alkukonfiguraatiossa valitsin kanssa ohjelmointikieleksi Kotlinin Javan sijasta. Android Studion luomassa tyhjässä projektissa ei ollut montaa tiedostoa, mutta siitä löytyi kaikki tarvittava sovelluksen kehittämisen aloittamista varten.



Kuva 1. Android Studio -projektin alkukonfiguraatio

3.3 Navigoinnin suunnittelu ja toteutus

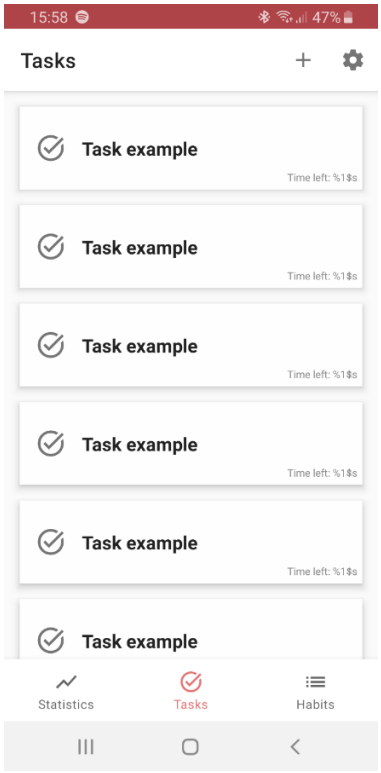
3.3.1 Jetpack-kirjaston navigointikomponentti

Päätin aloittaa kehittämisen luomalla sovellukselle navigointistruktuurin ja sen tarvitsemat Fragment-komponentit eli näkymät käyttäen apuna Jetpack-kirjaston navigointikomponentteja. Suunnitelmani oli käyttää sovelluksessa vain yhtä Activity-luokkaa, joka pitää ja hallitsee kaikkia sovelluksen näkymiä. Näin pystyn toteuttamaan koko sovelluksen navigaation Jetpackin navigointikomponentin kautta, joka tekee sovelluksen navigaation suunnittelusta helpompaa ja selkeämpää.

Jetpackin navigointikomponentti koostuu kolmesta osasta, jotka ovat navigointikaavio, NavHost eli navigointi-isäntä ja NavController eli navigointikontrolleri. Navigointikaavio määrittää sovelluksen navigoinnin käyttämät näkymät ja reitit näkymien välillä. NavHost-osa toimii näkymän kehyksenä ja näyttää sen sisällä navigointikaaviossa määritetyt näkymät. NavController-osa hoitaa navigoinnin ja vaihtamisen näkymien välillä. (Android Developers 2020i.)

Ensimmäiseksi loin uuden MainActivity-komponentin, joka toimii sovellukseni ainoana Activity-komponenttina. Kun sovellus avataan, tämä Activity-komponentti avataan laitteen näytölle. Se toimii sovelluksen muiden näkymien kehyksenä. MainActivity-komponentin

käyttämässä asetelmatiedostossa on määritetty FragmentContainerView-komponentti eli NavHost, jonka tehtävänä on näyttää sovelluksen eri näkymät sen sisässä. Sen yläpuolella on työkalupalkki, jossa on kuvakepainikkeet eri sovelluksen toimintoja varten. Alalaidassa taas on alanavigointipalkki, jonka kautta voi navigoida sovelluksen ylätason näkymien välillä. Sovelluksen ensimmäisestä versiosta otetusta kuvankaappauksesta (Kuva 2) voi nähdä yllä kuvatun rakenteen, jossa lisäksi FragmentContainerView-komponentti näyttää tehtävälistanäkymän malliversion sen tilalla.



Kuva 2. Sovelluksen ensimmäinen versio

Sekä työkalupalkki että alanavigointipalkki käyttävät Jetpack-kirjaston navigointikontrolleria niiden navigointitoimintojen toteuttamiseen. MainActivity-komponentissa työkalupalkki ja alanavigointipalkki -komponentit yhdistetään NavController-komponentin kanssa (Kuva 3), jotta työkalupalkin otsikko näyttäisi nykyisen näkymän otsikon ja alanavigointipalkki avaisi oikean näkymän, kun käyttäjä painaa sen painikkeista.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        //Navigation
        val navHostFragment = supportFragmentManager.findFragmentById(R.id.fragmentContainerView) as NavHostFragment
        val navController = navHostFragment.navController

        //Set up ToolBar and BottomNavigationBar with NavController
        val appBarConfiguration = AppBarConfiguration(setOf(
            R.id.tasksFragment,
            R.id.habitsFragment,
            R.id.statisticsFragment)
        )
        toolbar.setupWithNavController(navController, appBarConfiguration)
        bottomNavigation.setupWithNavController(navController)
```

Kuva 3. NavController-komponentin käyttöönotto MainActivity-komponentissa

Navigointikaavio on XML-tiedosto, jossa määritetään sovelluksen määränpäät ja niiden välillä navigoinnin. Se kuvastaa koko sovelluksen navigoinnin rakenteen ja reitit. Android Studiossa on navigointieditori, jossa voi graafisen käyttöliittymän kautta muokata navigointikaaviota. Vaihtoehtoisesti kehittäjä voi muokata navigointikaavion XML-tiedostoa suoraan. (Android Developers 2020l.) Projektini navigointikaaviona toimii navigation_graph.xml-tiedosto, joka sijaitsee navigation-resurssikansiossa.

3.3.2 Navigaation näkymät

Sovelluksen ensimmäiseen versioon loin tyhjän Fragment-komponentin melkein jokaiselle sovelluksen tulevalle näkymälle. Näkymien ei tarvinnut sisältää ensimmäisessä versiossa vielä mitään toiminnallisuuksia, vaan ne toimivat apuna navigaation suunnittelussa. Näihin Fragment-komponentteihin viitataan sovelluksen navigointikaavion tiedostossa (Kuva 4), jonka avulla sovelluksen navigoinnin rakenne ja reitit luodaan. Sovelluksen navigoinnin rakenteen voi nähdä avaamalla navigointikaavion Android Studion navigointieditorissa (Kuva 5). Navigoinnin rakenne on kuvattu myös tekemässäni kaaviossa (Kaavio 2), jonka avulla hahmottelin sovelluksen navigaatiota.

Tehtävälista toimii myös sovelluksen kotinäkymänä, jonka määrittelin navigointikaaviossa. Tehtävälista on ensimmäinen näkymä, joka näytetään käyttäjälle, kun hän avaa sovelluksen. Tehtävälistanäkymä valittiin kotinäkymäksi, koska tehtävien merkitseminen suoritetuksi on tarkoituksena olla yksi tärkeimmistä syistä, minkä takia käyttäjä avaa sovelluksen.

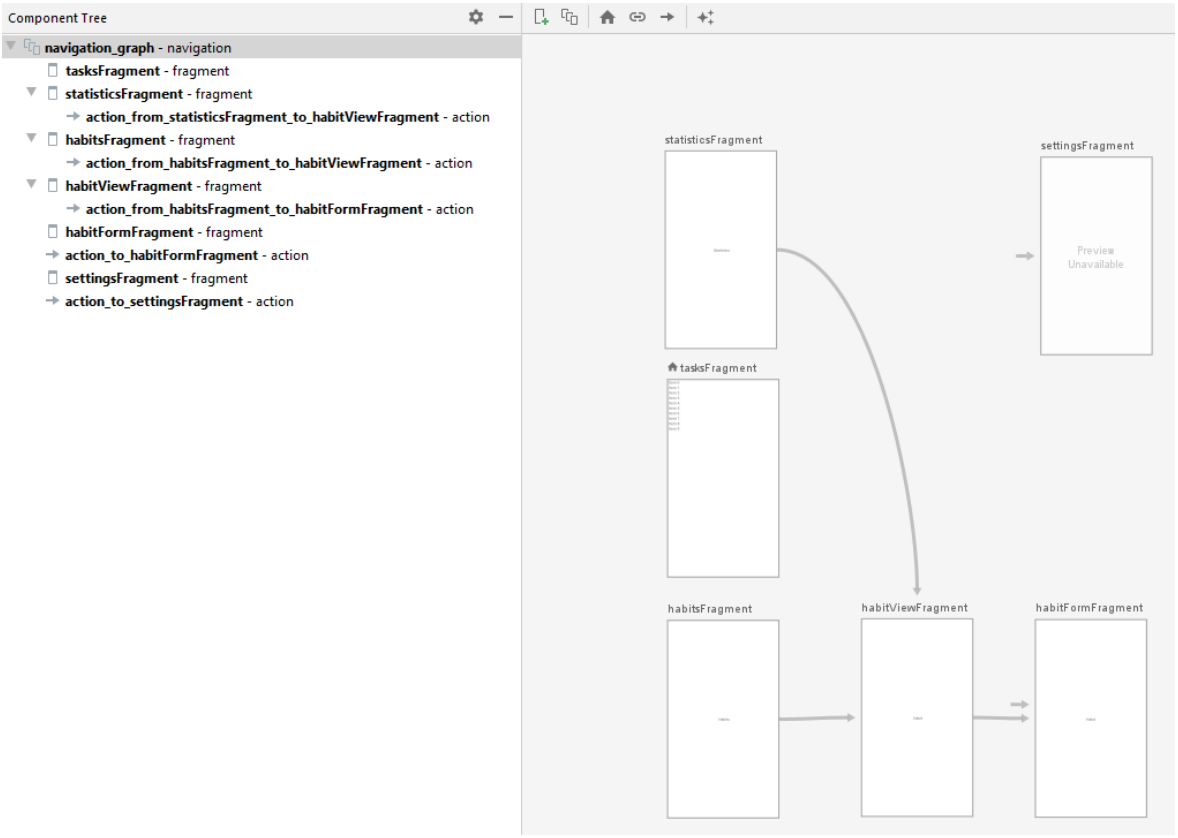
Rutiinilistanäkymän tarkoituksena on näyttää käyttäjän luomat rutiinit listana, josta käyttäjä voi avata rutiinin oman sivun tarkempaa katselua ja hallintaa varten. Rutiinilistanäkymä

toimii kanssa sovelluksessa ylätason näkymänä, joten sillä on oma painike alanavigointipalkissa (Kuva 2), josta sen voi avata nopeasti. Alanavigointipalkissa on myös painike tilastonäkymälle. Tilastonäkymästä käyttäjä voi seurata rutiiniensa edistymistään.

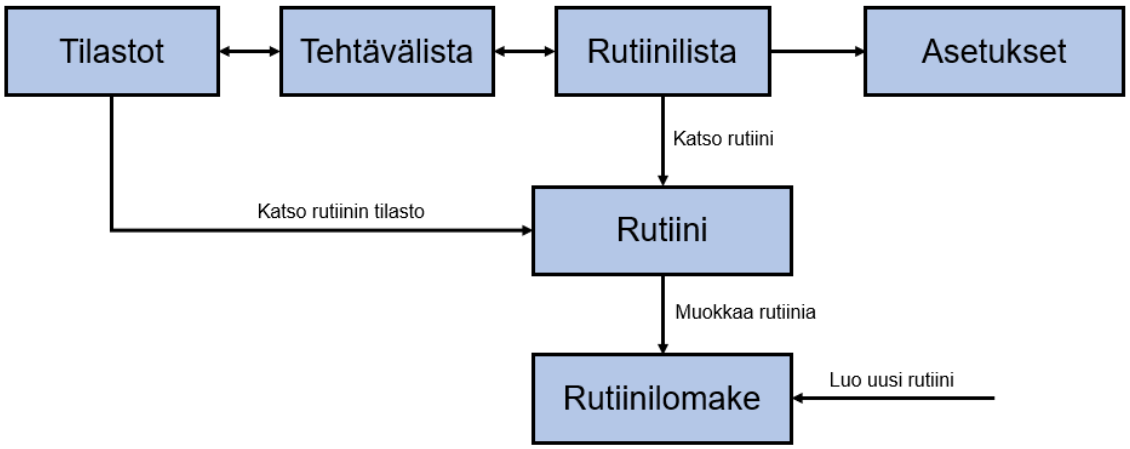
Sovelluksen työkalupalkista löytyvät painikkeet rutiinilomakkeen ja sovelluksen asetuksien avaamiseen (Kuva 2). Rutiinilomakenäkymän kautta käyttäjä voi luoda uusia rutiineja. Rutiinilomakkeen on myös tarkoitus toimia näkymänä jo luodun rutiinin muokkaamiselle. Sovelluksen asetuksista käyttäjä voi hallita muun muassa sovelluksen lähettämiä muistutuksia.

```xml
<?xml version="1.0" encoding="utf-8"?>
<navigation
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/navigation_graph"
    app:startDestination="@id/tasksFragment">

    <!--Home and task list screen-->
    <fragment
        android:id="@+id/tasksFragment"
        android:label="Tasks"
        android:name="com.santtuhyvarinen.habittracker.fragments.TasksFragment"
        tools:layout="@layout/fragment_tasks">
    </fragment>

    <!--Statistics screen-->
    <fragment
        android:id="@+id/statisticsFragment"
        android:label="@string/statistics"
        android:name="com.santtuhyvarinen.habittracker.fragments.StatisticsFragment"
        tools:layout="@layout/fragment_statistics">

        <action
            android:id="@+id/action_from_statisticsFragment_to_habitViewFragment"
            app:enterAnim="@anim/anim_slide_in"
            app:exitAnim="@anim/anim_slide_out"
            app:destination="@id/habitViewFragment"/>
    </fragment>

    <!--Habit list screen-->
    <fragment
        android:id="@+id/habitsFragment"
        android:label="Habits"
        android:name="com.santtuhyvarinen.habittracker.fragments.HabitsFragment"
        tools:layout="@layout/fragment_habits">

        <action
            android:id="@+id/action_from_habitsFragment_to_habitViewFragment"
            app:enterAnim="@anim/anim_slide_in"
            app:exitAnim="@anim/anim_slide_out"
            app:destination="@id/habitViewFragment"/>
    </fragment>

    <!--Habit view screen-->
    <fragment
        android:id="@+id/habitViewFragment"
        android:label="@string/habit"
```

Kuva 4. Navigaatiokaavion XML-tiedosto



Kuva 5. Navigaatiokaavion XML-tiedosto avattuna navigointieditorissa



Kaavio 2. Sovelluksen navigaation rakenne

3.3.3 Näkymien välillä siirtyminen

Kun sovelluksessa siirrytään näkymästä eli Fragment-komponentista toiseen, useimmissa tapauksissa käytetään yksinkertaisesti NavController-komponentin navigate-metodia, jolle annetaan parametrinä projektin navigaatiokaaviossa määritetyn reitin ID-viite, jonka perusteella NavController-komponentti navigoi reittiä vastaavaan näkymään. Sovelluksessa on kuitenkin tilanteita, joissa pitää siirtää tietoa näkymien välillä, kuten esimerkiksi tilanteissa, missä avattavan näkymän tarvitsee esittää tietoa jostain yksittäisestä rutiinista.

Jetpack-kirjaston navigointikomponentilla on Safe Args -laajennus Gradlelle, jonka avulla voi navigoida ja siirtää dataa näkymien välillä tyyppiturvallisesti. Kun Safe Args -laajennus on otettu käyttöön projektin juuren build.gradle-tiedostossa, laajennus luo automaattisesti navigointikaavion näkymien ja reittien pohjalta näkymille Directions- ja Args-luokat ja niiden metodit navigointia ja datan siirtämistä varten. Directions-luokka luodaan reitin lähtöpaikalle, joka lähettää datan. Args-luokat luodaan reitin määränpäälle, joka vastaanottaa lähetetyn datan. (Android Developers 2021d.)

Aikaisemmissa projekteissani olin siirtänyt dataa Fragment-komponenttien välillä Bundle-olioiden avulla. Aluksi tässäkin projektissa käytettiin Bundle-olioita datan siirtämiseen, mutta koska käytin tässä projektissa Jetpack-kirjaston navigointikomponenttia navigoinnin toteuttamiseen, päätin kokeilla Safe Args -laajennusta tiedon välittämiseen näkymien välillä. Projektissani Safe Args -laajennus loi automaattisesti kaikille navigointikaaviossa määritetyille reiteille Directions- ja Args-luokat, joiden kautta määränpäähän voi välittää dataa, kun siihen siirrytään toisesta näkymästä.

Esimerkiksi sovelluksessa tarvitsee siirtää rutiinin ID-numero navigoinnin yhteydessä, kun rutiinilistanäkymästä eli HabitsFragment-komponentista avataan yksittäisen rutiinin näkymä eli HabitViewFragment-komponentti. Komponenttien välinen reitti on määritelty navigointikaaviossa (Kuva 6).

```
<!--Habit list screen-->
<fragment
    android:id="@+id/habitsFragment"
    android:label="Habits"
    android:name="com.santtuhyvarinen.habittracker.fragments.HabitsFragment"
    tools:layout="@layout/fragment_habits">

    <action
        android:id="@+id/action_from_habitsFragment_to_habitViewFragment"
        app:enterAnim="@anim/anim_slide_in"
        app:exitAnim="@anim/anim_slide_out"
        app:destination="@id/habitViewFragment">
        <argument
            android:name="habit_id"
            app:argType="long"
            android:defaultValue="-1L" />
    </action>
</fragment>
```

Kuva 6. HabitsFragment-näkymä ja reitti HabitsFragment-komponentista HabitViewFragment-komponenttiin navigointikaaviossa

Safe Args -laajennus loi automaattisesti HabitsFragmentDirections-luokan navigointikaavion pohjalta. HabitsFragmentDirections-luokalla on actionFromHabitsFragmentToHabitViewFragment-metodi, joka automaattisesti luotiin navigointikaaviossa määritetyn reitin pohjalta (Kuva 6). Metodille annetaan rutiinin ID-numero parametrinä. Metodi palauttaa

NavDirections-olion, joka voidaan antaa parametrinä NavController-komponentin navigate-metodille, mikä navigoi HabitViewFragment-komponentin näkymään eli rutiininäkymään. (Kuva 7.)

```
private fun openHabitView(habit: Habit) {
    val action = HabitsFragmentDirections.actionFromHabitsFragmentToHabitViewFragment(habit.id)
    findNavController().navigate(action)
}
```

Kuva 7. Rutiininäkymään navigoiminen Safe Args -laajennuksen avulla

Reitin määränpäälle eli HabitViewFragment-komponentille luotiin myös automaattisesti HabitViewFragmentArgs-luokka, jonka kautta HabitViewFragment-komponentti pystyy hakemaan navigoinnin mukana annetun rutiinin ID-numeron (Kuva 8).

HabitViewFragment.kt

```
39      private val args: HabitViewFragmentArgs by navArgs()
40
41      override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
42          super.onViewCreated(view, savedInstanceState)
43
44          //If habitId is negative, should navigate up and exit fragment
45          if(args.habitId < 0) {
46              findNavController().navigateUp()
47              return
48          }
```

Kuva 8. HabitViewFragmentArgs-luokan käyttäminen HabitViewFragment-komponentissa

Safe Args -laajennuksen käyttöönotto vaatii hieman enemmän taustatyötä kuin Bundle-olioiden käyttäminen, mutta mielestäni se toi johdonmukaisuutta ja selkeyttä sovelluksen navigaatioon. Laajennuksen automaattisesti luomista luokista näkee helposti suoraan, että minkälaista dataa komponentille voi välittää navigoinnin yhteydessä. Laajennus myös auttaa varmistamaan, että komponentille ei voi vahingossa välittää väärään tyyppistä dataa. Sen käyttö on kuitenkin riippuvainen Jetpack-kirjaston navigointikomponenteista eli Safe Args -laajennuksesta ei ole hyötyä, jos sovelluksen navigoinnissa käyttää jotain muuta ratkaisua kuin Jetpack-kirjaston navigointikomponentteja.

3.4 Skaalautuvuus

Sovelluksen suunnittelun yhtenä osatavoitteena oli sovelluksen käyttöliittymän skaalautuvuus eri kokoisille laitteille. Sovelluksen käyttöliittymän pitäisi olla yhtä helppokäyttöinen ja selkeä sekä pienen kännykän näytöllä että iso kokoisen tabletin näytöllä.

Skaalautuvuuden parantamiseksi tein kaksi erillistä mitta-arvojen tiedostoa (Kuva 9) resurssikansioihin. Molempien mittatiedostojen nimi on dimens.xml. Mittatiedostossa on määritelty sovelluksen asettelutiedostoissa käytetyt marginaalit, tekstien koot ja eri komponenttien mitat. Ensimmäinen mittatiedosto toimii oletusmittatiedostona sovellukselle. Toinen mittatiedosto on taas suunnattu tableteille. Oletusmittatiedostoa säilytetään values-kansiossa muiden oletusarvojen resurssitiedostojen kanssa. Tableteille tarkoitettua mittatiedostoa säilytetään values-sw600dp nimisessä resurssikansiossa.

Sovellus valitsee oikean mittatiedoston laitteen näytön leveyden perusteella. Tableteille suunnatun mitta-arvojen tiedoston tabletin leveyden raja-arvona toimii 600 dp, mikä on määritetty resurssikansion suffiksissa (sw600dp). Jos laitteen leveys on alle 600 dp, sovellus käyttää oletuksena olevaa mittatiedostoa. Jos taas laitteen leveys on yli 600 dp, se käyttää tableteille tarkoitettua mittatiedostoa.

values\dimens.xml

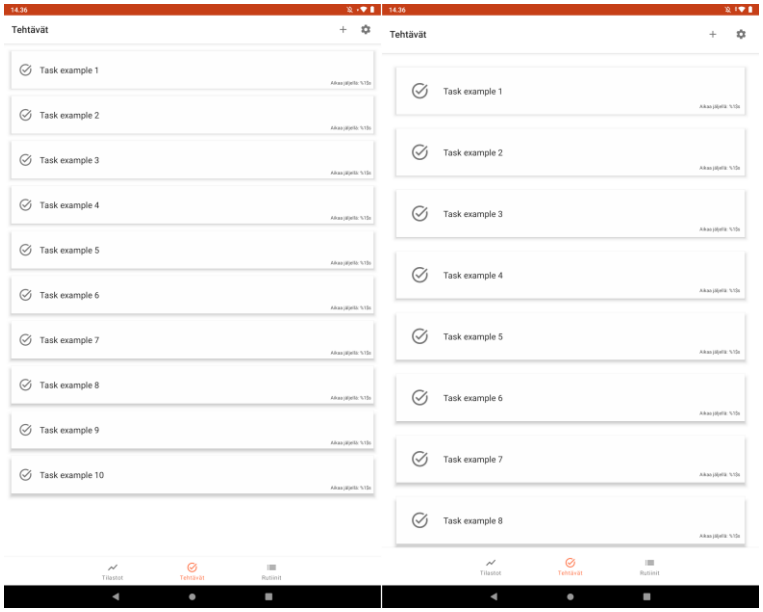
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!--Generic-->
    <dimen name="smallMargin">5dp</dimen>
    <dimen name="mediumMargin">10dp</dimen>
    <dimen name="largeMargin">15dp</dimen>

    <dimen name="smallPadding">5dp</dimen>
    <dimen name="mediumPadding">10dp</dimen>
    <dimen name="largePadding">15dp</dimen>

    <dimen name="extraSmallTextSize">10sp</dimen>
    <dimen name="smallTextSize">18sp</dimen>
    <dimen name="mediumTextSize">20sp</dimen>
    <dimen name="titleTextSize">26dp</dimen>

    <dimen name="elevation">5dp</dimen>

    <!--Navigation-->
    <dimen name="bottomNavigationHeight">60dp</dimen>
    <dimen name="toolbarHeight">60dp</dimen>
    <dimen name="toolbarButtonSize">48dp</dimen>

    <dimen name="buttonHeight">60dp</dimen>
    <dimen name="roundButtonSize">48dp</dimen>

    <dimen name="floatingActionButtonSize">50dp</dimen>

    <!--Tasks-->
    <dimen name="taskCardHeight">80dp</dimen>
    <dimen name="taskIconSize">30dp</dimen>
```

sw600dp\dimens.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!--Generic-->
    <dimen name="smallMargin">10dp</dimen>
    <dimen name="mediumMargin">20dp</dimen>
    <dimen name="largeMargin">30dp</dimen>

    <dimen name="smallPadding">10dp</dimen>
    <dimen name="mediumPadding">20dp</dimen>
    <dimen name="largePadding">30dp</dimen>

    <!--Navigation-->
    <dimen name="bottomNavigationHeight">80dp</dimen>
    <dimen name="toolbarHeight">80dp</dimen>
    <dimen name="toolbarButtonSize">60dp</dimen>

    <dimen name="buttonHeight">80dp</dimen>
    <dimen name="roundButtonSize">60dp</dimen>

    <dimen name="floatingActionButtonSize">60dp</dimen>

    <!--Tasks-->
    <dimen name="taskCardHeight">100dp</dimen>
    <dimen name="taskIconSize">40dp</dimen>

    <!--Habits-->
    <dimen name="itemHeight">100dp</dimen>
    <dimen name="itemIconSize">40dp</dimen>
</resources>
```

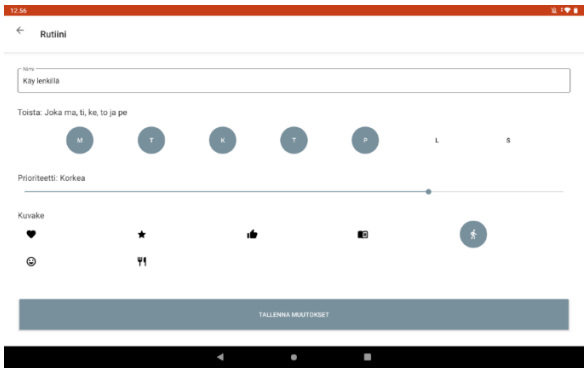
Kuva 9. Mitta-arvojen tiedostot. Oletusmitat (Vasen) ja tableteille tarkoitetut mitat (Oikea)

Kuten mittatiedostoja vertaamalla näkyy (Kuva 9), tableteille tarkoitetussa mittatiedostossa on asetettu hieman korkeammat arvot kuin oletusmittatiedostossa. Alla on kaksi kuvankaappausta sovelluksen päänäkymästä (Kuva 10), jotka otettiin Android-tabletista. Vasemmassa kuvankaappauksessa näkyy, että sovellus käyttää vielä samoja arvoja mitoille kaikissa mahdollisissa laitekonfiguraatioissa. Oikeassa kuvankaappauksessa taas käytetään tableteille tarkoitettua mittatiedostoa, johon on asetettu suuremmat arvot. Tämän pitäisi tehdä sovelluksen käyttöliittymästä vähemmän ahtaan tableteilla. Myös isommat painikkeet pitäisi tehdä niiden koskettamisesta helpompaa tableteilla.



Kuva 10. Sovelluksen päänäkymä (ensimmäinen versio) tabletissa ilman erillistä mittatiedostoa (Vasen) ja mittatiedoston kanssa (Oikea).

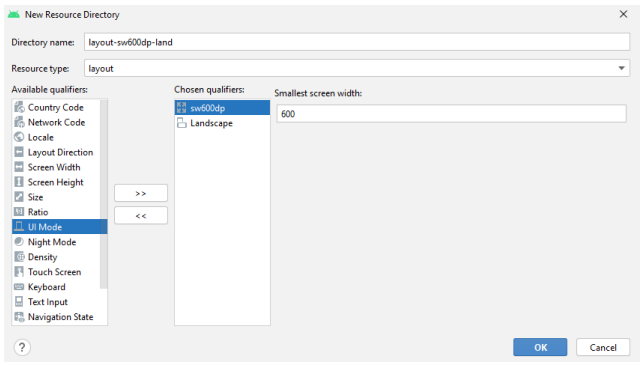
Eri mittatiedostot konfiguraatioille auttavat käyttöliittymän skaalaamisessa eri kokoisille näytöille, mutta jotkut näkymät käyttöliittymässä saattavat tarvita myös erilliset asetelmatiedostot. Useimmat näkymät sovelluksessa skaalautuvat hyvin tabletissa ilman erillistä asetelmatiedostoa, mutta parin näkymän käytettävyys kärsi ilman erillistä asetelmatiedostoa. Esimerkiksi rutiinilomakenäkymä osoittautui huonosti skaalautuvaksi tablettilaitteissa, kun laitteen näyttö oli vaakatasossa (Kuva 11). Kuten viitatussa kuvassa näkyy, lomake venyy koko näytön halki, joka tekee näkymästä sekavan ja hankalakäyttöisen. Sovelluksen pitäisi olla helppokäyttöinen myös silloin, kun laitteen näyttö on vaakatasossa.



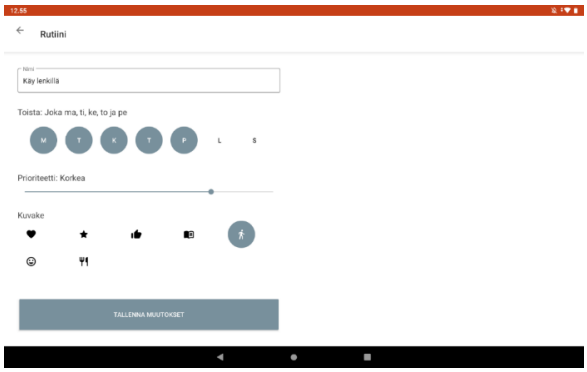
Kuva 11. Rutiinilomake vaakatasossa tabletissa

Rutiinilomakkeen skaalautuvuuden parantamiseksi tein uuden asetelmatiedoston rutiinilomakkeelle. Samaan resurssikansioon voi määrittää useita eri konfiguraatiokriteerejä (Kuva 12), joten tein tämän asetelmatiedoston layout-sw600dp-land nimisen resurssikansion sisään. Tämän resurssikansion resurssit ladataan vain, jos laitteen näytön leveys on yli 600 dp ja laitteen näyttö on vaakatasossa. Jos laitteen näyttö on pystysuorassa tai laitteen näytön leveys on alle 600 dp, käytetään vielä oletusasetelmatiedostoa.

Uudessa rutiinilomakkeen asetelmatiedostossa lomake ei enää veny koko näytön halki, vaan se venyy vain puolet näkymän leveydestä (Kuva 13). Tämän pitäisi tehdä rutiinilomakkeen käytöstä helpompaa tableteilla, kun niiden orientaatio on vaakatasossa.



Kuva 12. Resurssikansion luonti-ikkuna

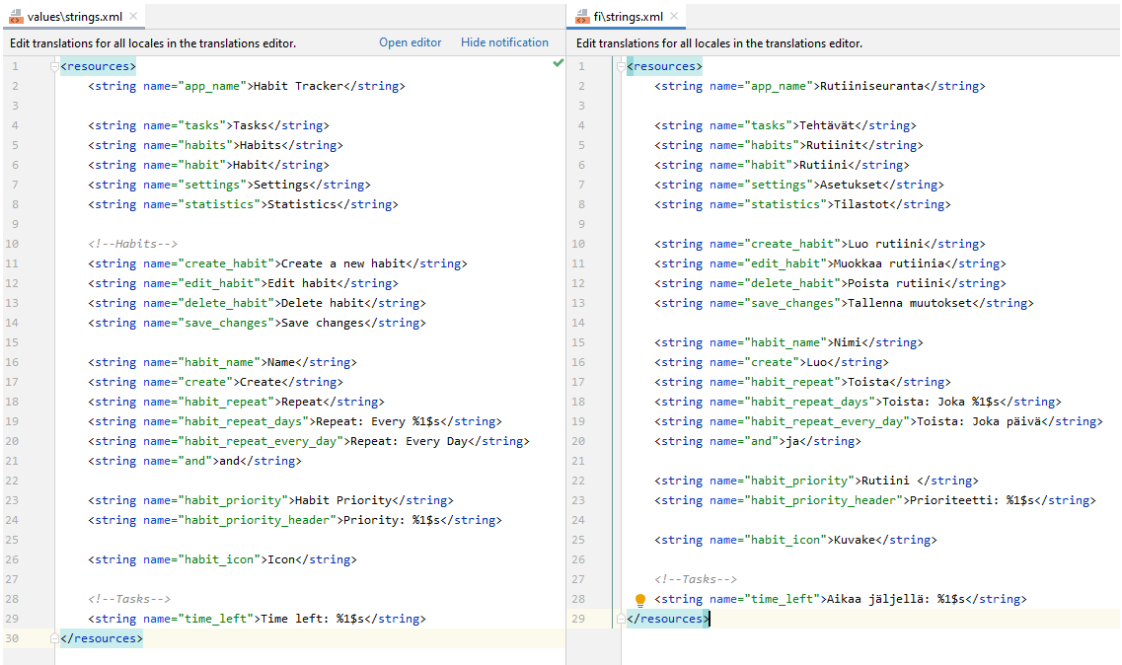


Kuva 13. Rutiinilomakenäkymä vaakatasossa tabletissa (Uusi asetelmatiedosto)

3.5 Lokalisaatio

Sovelluksen kaikki tekstisisältö on säilytetty string.xml- tai arrays.xml-tiedostoissa values-resurssikansioissa. Vain lokeja tai testausta varten oleva teksti on kovakoodattu, koska ne eivät tule näkymään koskaan loppukäyttäjälle. Tällä tavalla kaikki sovelluksen teksti löytyy yhdestä paikasta, josta tekstiä on helppo käydä päivittämässä ilman, että tarvitsee käydä päivittämässä kovakoodattua tekstiä erikseen eri tiedostoista. Tämä tapa myös tekee eri kielten lisäämisen jatkokehityksen aikana erittäin helpoksi.

Sovellus on käännetty kahdeksi eri kieleksi: suomeksi ja englanniksi. Sovelluksessa ei ole erikseen kielenvalinta-asetusta, koska sovellus valitsee kielen laitteen kielen perusteella. Sovelluksella on kaksi resurssikansiota, jotka säilyttävät kielikohtaiset strings.xml- ja arrays.xml-tiedostot (Kuva 14). Resurssikansioiden nimet ovat values ja values-fi. Kansion nimen suffiksi, kuten tässä tapauksessa fi-suffiksi, määrittää kielen. Jos resurssikansiossa ei ole suffiksia kieltä varten, se toimii oletusresurssikansiona, kun laitteen valitulle kielelle ei löydy sitä vastaavaa resurssikansiota. Sovelluksen oletuskieli on englanti. Jos laitteen kieli on joku muu kuin suomi tai englanti, sovellus käyttää englantia käyttöliittymän kielenä.



Kuva 14. Sovelluksen strings.xml-tiedostot (englannin ja suomen kielille)

4 Tietokanta

Sovelluksen yhdet tärkeimmät toiminnallisuudet olivat rutiinien luonti ja hallinta. Käyttäjän pitäisi olla mahdollista luoda rutiineja, joiden pohjalta sovellus luo päivittäisiä tehtäviä. Jotta käyttäjän rutiinit eivät katoaisi, kun sovellus suljetaan, sovellus tarvitsee tietokannan, johon käyttäjän rutiinit tallennetaan.

Päätin käyttää sovelluksen tietokantaratkaisuna Jetpackin Room Persistence -kirjastoa. Room Persistence -kirjaston avulla voi tallentaa dataa sovelluksen paikalliseen SQLite-tietokantaan ilman SQLite-rajapinnan käyttämistä suoraan (Android Developers 2020h). Projektin tavoitteisiin ei kuulunut ulkoisen tietokannan pystyttämistä palvelimelle, joten Room-kirjasto tuntui olevan hyvä ratkaisu sovelluksen tietokannan toteuttamiseen paikallisesti.

Room-kirjaston käyttöönottoa varten projektissa tarvitsee lisätä Room-kirjaston riippuvaisuudet projektin app/build.gradle-tiedostoon (Android Developers 2020h). Tietokannan toteutuksen aikana viimeisin versio Room-kirjastosta oli 2.2.6.

Room-kirjastossa on pääasiassa kolme komponenttia, joiden avulla tietokantaratkaisu toteutetaan. Tietokantaluokan kautta saa yhteyden sovelluksen tietokantaan ja sen sisältämään dataan. Entiteetit määrittävät tietokannan taulujen rakenteen. DAO-rajapinnat tarjoavat metodit, joilla sovelluksen muut osat voivat hakea, päivittää ja poistaa tietokannan dataa. (Android Developers 2020h.)

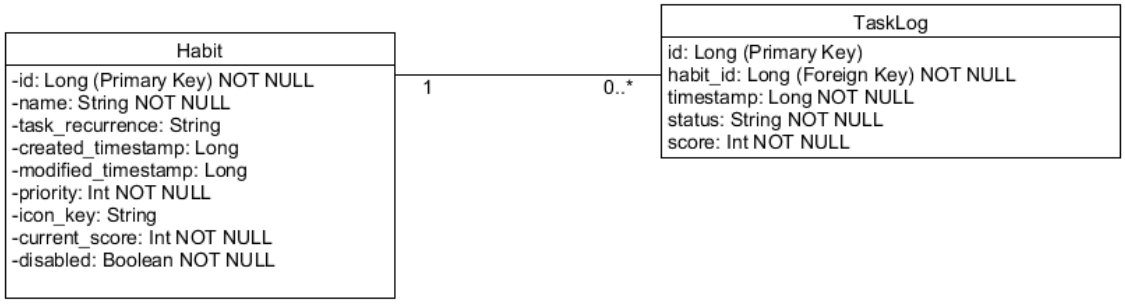
4.1 Tietokantataulujen rakenne

Ensimmäiseksi suunnittelin yksinkertaiset relaatiokaaviot sovelluksen tietokannan tauluja varten (Kaavio 3). Tietokannan taulujen suunnittelussa oli tärkeää ottaa huomioon, että suunnittelemani toiminnallisuudet sovellukseen olisi mahdollista toteuttaa ilman taulujen rakenteen päivittämistä myöhemmin. Room-tietokannan taulujen rakenteita voi päivittää myöhemmin, mutta se vaatii Migration-reittien luomista eri tietokanta versioiden välille (Android Developers 2020m). Jos tietokannan taulujen rakenteita muuttaa ilman, että luo tarvittavia Migration-reittejä, sovellus kaatuu, kun sovelluksen päivittää vanhasta versiosta uuteen (Kuva 15). Tämän takia tietokantarakenteen kannattaa olla hyvin suunniteltu, jotta välttyy turhasta tietokantaversiohallinnasta kehityksen aikana.

```
2021-01-04 11:16:19.662 7975-8079/com.santtuhyvarinen.habittracker E/AndroidRuntime: FATAL EXCEPTION: arch_disk_io_0
    Process: com.santtuhyvarinen.habittracker, PID: 7975
    java.lang.RuntimeException: Exception while computing database live data.
        at androidx.room.RoomTrackingLiveData$1.run(RoomTrackingLiveData.java:92)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1167)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:641)
        at java.lang.Thread.run(Thread.java:764)
     Caused by: java.lang.IllegalStateException: A migration from 1 to 2 was required but not found. Please provide the necessary Migration path via RoomDatabase.Builder
        at androidx.room.RoomOpenHelper.onUpgrade(RoomOpenHelper.java:117)
        at androidx.sqlite.db.framework.FrameworkSQLiteOpenHelper$OpenHelper.onUpgrade(FrameworkSQLiteOpenHelper.java:124)
        at android.database.sqlite.SQLiteOpenHelper.getDatabaseLocked(SQLiteOpenHelper.java:400)
        at android.database.sqlite.SQLiteOpenHelper.getWritableDatabase(SQLiteOpenHelper.java:298)
        at androidx.sqlite.db.framework.FrameworkSQLiteOpenHelper$OpenHelper.getWritableSupportDatabase(FrameworkSQLiteOpenHelper.java:92)
        at androidx.sqlite.db.framework.FrameworkSQLiteOpenHelper.getWritableDatabase(FrameworkSQLiteOpenHelper.java:53)
        at androidx.room.RoomDatabase.inTransaction(RoomDatabase.java:476)
        at androidx.room.RoomDatabase.assertNotSuspendingTransaction(RoomDatabase.java:281)
        at androidx.room.RoomDatabase.query(RoomDatabase.java:324)
        at androidx.room.util.DBUtil.query(DBUtil.java:83)
        at com.santtuhyvarinen.habittracker.database.dao.HabitDao_Impl$7.call(HabitDao_Impl.java:180)
        at com.santtuhyvarinen.habittracker.database.dao.HabitDao_Impl$7.call(HabitDao_Impl.java:177)
        at androidx.room.RoomTrackingLiveData$1.run(RoomTrackingLiveData.java:90) <3 more...>
```

Kuva 15. Room-tietokannan puuttuvan Migration-reitin virheviesti

Suunnittelemiini ominaisuuksiin tarvitsin vain kaksi tietokantataulua (Kaavio 3). Yksi taulu rutiineille, joka säilyttää käyttäjien luomien rutiinien tiedot. Toinen taulu on taas rutiinien tehtävien lokeille. Rutiinin pohjalta luodut tehtävät eivät tarvitse erillistä taulua tietokantaan, koska ne luodaan sovelluksen ajoaikana tilapäisesti rutiinien ja tehtävälokien perusteella. Kun käyttäjä merkitsee tehtävän suoritetuksi, epäonnistuneeksi tai ohitetuksi, tapahtumasta luodaan loki tietokantaan, jonka perusteella sovellus tietää, että mikä on tehtävän tila.



Kaavio 3. Tietokannan relaatiokaaviot (Ensimmäinen versio)

4.1.1 Rutiinin toiminta ja tietokantataulun rakenne

Rutiinin tietokantataulu käyttää ID-kenttää tietokantarivin pääavaimena, jolla tietokantarivit identifioidaan toisistaan. Kun rutiinista luodaan rivi tietokantaan, rivin ID-numero generoidaan automaattisesti, jotta kaikilla tietokantarivillä olisi yksilöivä ID.

Rutiiniin tallennetaan aikaleimat, kun rutiini luodaan ensimmäisen kerran tai rutiinia on muokattu käyttäjän toimesta rutiinilomakkeen kautta. Rutiinilla on myös tärkeysaste (priority), joka tallennetaan numerona. Käyttäjä voi asettaa rutiinin tärkeysasteen rutiinilomakkeessa liukusäätimen kautta. Rutiinien tehtävät järjestetään oletuksena rutiinin tärkeysasteen perusteella tehtävälistanäkymässä.

Rutiinille voi asettaa valinnaisesti kuvakkeen, joka näkyy rutiinin tehtävissä ja rutiinin sivulla. Sovelluksessa on ennalta määritetty kokoelma kuvakkeita, joista käyttäjä voi valita rutiinilomakkeessa. Jokaisella kuvakkeella on oma tekstiavain, joka tallennetaan rutiiniin tietokantaan, jos käyttäjä valitsee kyseisen kuvakkeen. Kenttä jätetään tyhjäksi, jos käyttäjä ei valinnut mitään kuvaketta rutiinille lomakkeessa. Projektissa tekemäni IconManager-luokka hallitsee kuvakkeita, ja sen kautta kuvakkeet voidaan hakea tekstiavaimen perusteella.

4.1.2 Tehtävälokin toiminta ja tietokantataulun rakenne

Tehtävälokeilla sovellus pitää kirjaa käyttäjän suorittamista tehtävistä. Sovelluksen tilastot pohjautuvat näihin tehtävälokeihin. Jokaisella tehtävälokilla on yhden suhde moneen -yhteys johonkin rutiiniin tietokannassa. Yhdellä tehtävälokilla voi olla vain yhteys yhteen rutiiniin. Tehtävälokissa on kenttä rutiinin ID-numerolle, joka toimii viiteavaimena, jolla viitataan tehtävälokin omistavaan rutiiniin. Jos tehtävälokin viittaama rutiini poistetaan tietokannasta, tehtäväloki poistetaan kanssa.

Jokaiseen tehtävälokiin kanssa merkataan aikaleima, milloin tehtäväloki luotiin tietokantaan. Tehtävälokien aikaleimojen perusteella sovellus voi selvittää, että onko rutiinin päivittäinen tehtävä jo suoritettu, kun sovellus luo päivittäisiä tehtäviä.

Tehtävälokin statuskenttään merkataan tekstimuodossa lokin status, minkä perusteella nähdään, että onko tehtävä suoritettu vai ei. Tehtävä voidaan merkitä joko suoritetuksi, epäonnistuneeksi tai ohitetuksi. Kun tehtävä merkitään suoritetuksi, tehtävän rutiinin pisteytystä nostetaan yhdellä. Jos taas tehtävä merkitään epäonnistuneeksi, rutiinin pisteytys nollataan. Jokaiseen tehtävälokiin merkitään myös sen omistavan rutiinin pisteytys tehtävälokin luontihetkellä tilastoja varten.

4.2 Entiteetit

Tein jokaiselle tietokantaan tulevalle taululle sitä vastaavan dataluokan. Kotlin-ohjelmointikielessä dataluokkaa voidaan käyttää, kun luokan pääasiallisena tehtävänä on tiedon pitäminen (Kotlin Programming Language 2019). Tein kaksi luokkaa tietokannan tauluja varten: Habit- ja TaskLog-luokat. Habit-luokka kuvastaa yhtä rutiinia, ja TaskLog-luokka kuvastaa yhtä tehtävälokia.

Room-kirjastossa jokaista tietokannan taulua kohden määritetään luokka entiteetiksi @Entity-notaatiolla. Room-kirjasto luo jokaiselle entiteetille oman taulun tietokantaan, kunhan entiteetteihin on viitattu Database-luokassa. (Android Developers 2020n.)

Lisäsin @Entity-notaation Habit- ja TaskLog-luokille, jotta Room-kirjasto loisi niille taulun tietokantaan. Tekemieni relaatiokaavioiden pohjalta (Kaavio 3) lisäsin Habit- ja TaskLog-entiteetteihin kentät, jotka vastasivat relaatiokaavioideni kenttiä (Kuva 16 ja Kuva 17). Oletuksena Room-kirjasto luo sarakkeen tietokantaan vastaamaan jokaista entiteetin kenttää, ellei kenttään ole määritetty erikseen @Ignore-notaatiota (Android Developers 2020n).

Jokaiselle entiteetille pitää olla kenttä pääavainta varten, johon viitataan @PrimaryKey-notaatiolla. Room-kirjasto määrittää automaattisesti ID-arvon entiteetille, jos @PrimaryKey-notaation autoGenerate-kentän arvo on asetettu positiiviseksi. (Android Developers 2020n.)

Sovelluksen toiminnan kannalta oli tärkeää, että tehtävälokin entiteetillä on suhde johonkin rutiiniin, jotta sovellus tietää, että mille rutiinille tehtävä on merkattu suoritetuksi. Tätä varten määritin TaskLog-entiteetin @Entity-notaatiossa viiteavaimeksi rutiinin ID-numeron (Kuva 17). Viiteavaimen avulla sovellus varmistaa, että entiteettien välinen suhde on oikein (Android Developers 2020o).

Entiteetin viiteavaimen onDelete-kentän kohtaan voi määrittää, että mitä tehdään, kun entiteetin omistama rivi poistetaan tietokannasta. Jos onDelete-kenttään määrittää ForeignKey.CASCADE-arvon, rivi poistetaan myös tietokannasta, jos entiteetin viittaama rivi poistetaan. (Android Developers 2020o.). Sovelluksen toiminnan kannalta ei ollut tärkeää säilyttää tietokannassa niitä tehtävälokeja, joiden viittaama rutiini on poistettu tietokannasta. Määritin TaskLog-entiteetin viiteavaimeen, että tehtäväloki poistetaan tietokannasta, kun käyttäjä poistaa sen viittaaman rutiinin tietokannasta (Kuva 17).

Habit.kt

```kotlin
package com.santtuhyvarinen.habittracker.models

import ...

@Entity
data class Habit(@PrimaryKey(autoGenerate = true) val id : Long = 0) {

    var name : String = ""

    var taskRecurrence : String = ""

    var priority : Int = 0

    var iconKey : String? = null

    var creationDate : Long = 0L

    var modificationDate : Long = 0L

    var score : Int = 0

    var disabled : Boolean = false

    override fun toString(): String {
        return "${name}, taskRecurrence = ${taskRecurrence}, priority = ${priority}, iconKey = ${iconKey}"
    }
}
```

Kuva 16. Rutiinin entiteettiluokka

TaskLog.kt

```kotlin
package com.santtuhyvarinen.habittracker.models

import ...

@Entity(foreignKeys = [
    ForeignKey (
        entity = Habit::class,
        parentColumns = arrayOf("id"),
        childColumns = arrayOf("habitId"),
        onDelete = ForeignKey.CASCADE
)])
data class TaskLog(@PrimaryKey(autoGenerate = true) val id : Long = 0) {

    var habitId : Long = 0

    var score : Int = 0

    var status : String = ""

    var timestamp : Long = 0L
}
```

Kuva 17. Tehtävälokin entiteettiluokka

4.3 DAO-rajapinnat

Tein molempia luomiani entiteettejä vastaavat DAO-rajapinnat. Room-kirjaston DAO-rajapinnat määrittelevät metodit, joilla sovellus voi hakea, päivittää, luoda ja poistaa dataa sovelluksen tietokannasta (Android Developers 2020p).

DAO-lyhenne tulee sanoista Data Access Object. Room-kirjastolla rajapinnan voi määritellä DAO-rajapinnaksi @Dao-notaatiolla. DAO-rajapinnan metodeilla määritetään, miten

sovellus voi käyttää tietokannan dataa. DAO-rajapintaan voi määritellä helposti metodit ilman SQL-lauseita, joilla sovellus voi lisätä, päivittää ja poistaa rivejä tietokannasta, käyttämällä @Insert-, @Update- ja @Delete-notaatioita vastaavasti. DAO-rajapintaan voi myös tehdä metodin, joka suorittaa SQL-kyselyn tietokantaan, käyttämällä @Query-notaatiota. (Android Developers 2020p.)

Luomassani HabitDao-rajapinnassa näkyy, miten edellä mainittuja notaatiota on käytetty (Kuva 18). HabitDao-rajapinnan kautta sovellus voi luoda, päivittää ja poistaa rutiineja tietokannasta. Rajapinnan kautta voi myös hakea yksittäisen rutiinin sen ID-numeron perusteella. Tehtävälokeille on vastaava DAO-rajapinta, jonka nimi on TaskLogDao. TaskLogDao-rajapinta tarjoaa vastaavanlaiset toiminnot tehtävälokien käsittelemiseen kuin HabitDao-rajapinta.

HabitsViewModel.kt | HabitsFragment.kt | HabitDao.kt

```kotlin
package com.santtuhyvarinen.habittracker.database.dao

import ...

@Dao
interface HabitDao {
    @Query( value: "SELECT * FROM habit")
    fun getAll(): LiveData<List<Habit>>

    @Transaction
    @Query( value: "SELECT * FROM habit")
    fun getHabitsWithTaskLogs(): LiveData<List<HabitWithTaskLogs>>

    @Query( value: "SELECT * FROM habit WHERE id IN (:id)")
    suspend fun getById(id : Long) : Habit?

    @Query( value: "SELECT * FROM habit WHERE id IN (:id)")
    suspend fun getHabitWithTaskLogsById(id : Long) : HabitWithTaskLogs?

    @Insert
    suspend fun create(habit: Habit) : Long

    @Update
    suspend fun update(habit: Habit) : Int

    @Delete
    suspend fun delete(habit: Habit) : Int
}
```

Kuva 18. HabitDao-rajapinta

4.4 Sulautettu olio

Tietokantataulujen relaatioita voi myös käyttää sulautettujen olioiden kautta. Sulautetun olion luokkaan voi @Embedded-notaatiolla merkata viitattavan entiteetin kentän. @Relation-notaatiolla voi kuvastaa yhden suhde yhteen tai yhden suhde moneen -yhteyksiä entiteettien välillä. DAO-rajapintaan voi lisätä metodin, joka palauttaa kaikki rivit sulautetusta

luokasta, jossa entiteetti on sulautettu yhteen siihen viittaaman entiteetin tai entiteettien kanssa. (Android Developers 2020q.)

Sovelluksessa ei ole paljon toimintoja, jotka vaatisivat pelkästään tehtävälokien hakemisen tietokannasta. Melkein jokaisessa käyttötapauksessa tehtävälokit pitää ensin yhdistää niiden viittaamiin rutiineihin ennen kuin niitä voi käyttää sovelluksessa. Sovelluksen aikaisemmassa versiossa ensin haettiin rutiinit tietokannasta, jonka jälkeen haettiin tehtävälokit rutiinin pääavaimen perusteella. Tämä lähestymistapa vaati entiteettien manuaalisen yhdistämisen ja uuden tietokantahaun suorittamista jokaista rutiinia kohden.

Sovelluksen päivitetyssä versiossa sulautetun olion avulla tietokannasta pystyy hakemaan kaikki rutiinit, joihin kaikkiin on yhdistetty niiden omat tehtävälokit, yhden tietokantahaun kautta. Sulautetun dataluokan nimi on HabitWithTaskLogs, joka sisältää rutiinin ja listan rutiinin omistamista tehtävälokeista (Kuva 19).

Lisäsin HabitDao-rajapintaan getHabitsWithTaskLogs-metodin, joka palauttaa kaikki tietokannasta löytyvät rutiinit yhdistettynä niiden tehtävälokeihin (Kuva 18). Metodiin on lisätty @Transaction-notaatio, joka varmistaa, että koko toimenpide suoritetaan atomisesti (Android Developers 2020q), koska metodi tekee kaksi tietokantahakua.

HabitsViewModel.kt | HabitsFragment.kt | HabitDao.kt | HabitWithTaskLogs.kt

```
1  package com.santtuhyvarinen.habittracker.models
2
3  import ...
5
6  data class HabitWithTaskLogs (
7      @Embedded
8      val habit: Habit,
9      @Relation(
10         parentColumn = "id",
11         entityColumn = "habitId"
12     )
13     val taskLogs: List<TaskLog>
14 ) {
15     init {
16         val shouldResetHabitScore = ScoreUtil.shouldResetHabitScore( habitWithTaskLogs: this)
17         if(shouldResetHabitScore) habit.score = 0
18     }
19 }
```

Kuva 19. HabitWithTaskLogs-luokka

Sovelluksen päivitetty versio käyttää sekä viiteavaimia että sulautettuja olioita entiteettien suhteiden kuvaamiseen. Entiteettien väliset relaatiot voisi teoriassa toteuttaa pelkästään joko viiteavaimien tai sulautettujen olioiden kautta, mutta päätin käyttää niitä yhdessä. Viiteavaimen avulla voin varmistaa, että tehtävälokit poistetaan, kun niiden viittaama rutiini

poistetaan. Sulautettu olio taas yksinkertaistaa taulujen tietojen yhdistelemistä ja useiden entiteettien hakemista tietokannasta kerralla.

4.5 Tietokantaluokka

Tietokantaluokka pitää sisällään varsinaisen yhteyden sovelluksen tietokantaan. Luokka määritetään tietokantaluokaksi @Database-notaatiolla, jossa on määritetty tietokannan käyttämät entiteetit. Luokan tarvitsee myös periytyä Room-kirjaston RoomDatabase-luokasta. (Android Developers 2020h.)

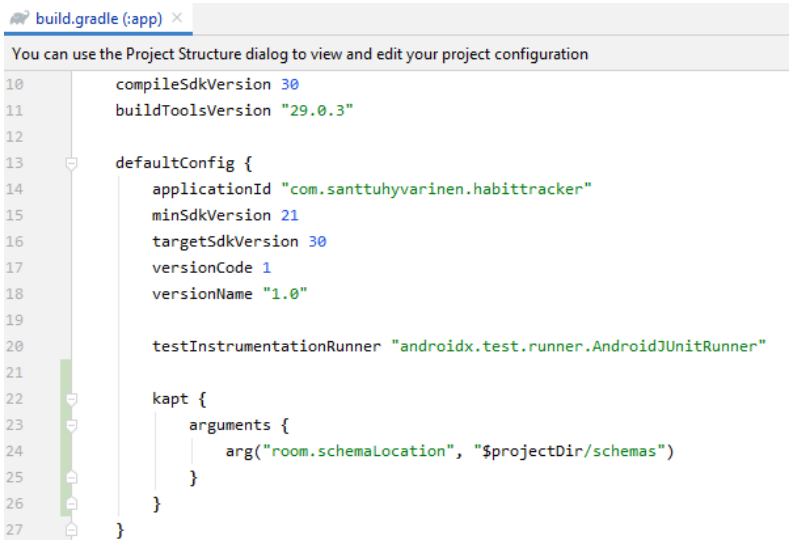
Tein luokan nimeltä AppDatabase, joka toimii projektini tietokantaluokkana. Luokassa on määritetty aikaisemmin luomani DAO-rajapinnat, joiden kautta tietokannan tietoja voidaan käsitellä (Kuva 20). AppDatabase-luokasta on olemassa sovelluksen ajoaikana vain yksi instanssi, joka luodaan vain, kun tietokantaluokkaa tarvitaan ensimmäisen kerran sovelluksen ajoaikana. Näin vältytään luomasta useita eri yhteyksiä tietokantaan samaan aikaan (Muntenescu 2021).

Room-kirjasto voi vaihtoehtoisesti kirjoittaa tietokannan kaavion JSON-tiedostoon (Android Developers 2020m), jos tietokantaluokan @Database-notaatioon on määritetty exportSchema-arvo todeksi (Kuva 20) ja app/build.gradle-tiedostoon on lisätty tiedostopolku room.schemaLocation-arvolle (Kuva 21).

HabitViewModel.kt | HabitViewFragment.kt | AppDatabase.kt | build.gradle (:app)

```
import ...

@Database(entities = [Habit::class, TaskLog::class], version = 1, exportSchema = true)
abstract class AppDatabase : RoomDatabase() {
    abstract fun habitDao() : HabitDao
    abstract fun taskLogDao() : TaskLogDao

    companion object {
        private const val DATABASE_NAME = "database_app"
        const val DATABASE_LOG_TAG = "database_log"

        @Volatile
        private var DATABASE_INSTANCE : AppDatabase? = null

        fun getDatabase(context: Context) : AppDatabase {
            return DATABASE_INSTANCE?: synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                        context.applicationContext,
                        AppDatabase::class.java, DATABASE_NAME)
                        .build()

                DATABASE_INSTANCE = instance

                return@synchronized instance
            }
        }
    }
```

Kuva 20. AppDatabase-luokka



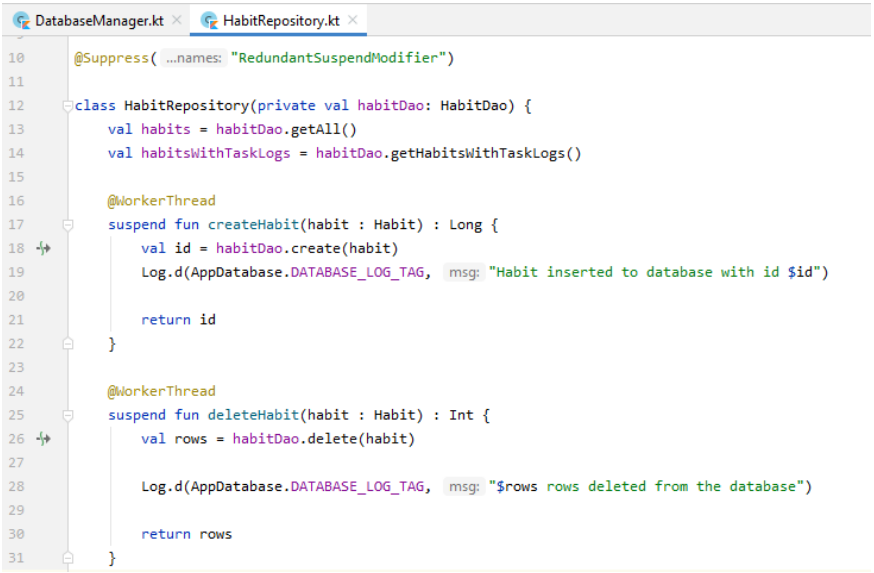
Kuva 21. app/build.gradle-tiedosto

4.6 Tietolähdeluokat

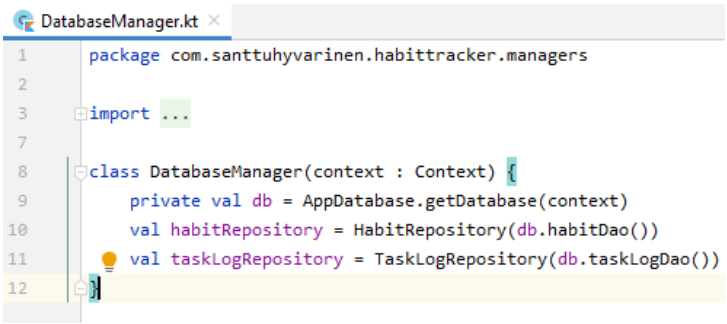
Tietolähdeluokka eli Repository-luokka toimii rajapintana, joka pelkistää tietokannan käsittelyn muulle sovellukselle ja mahdollistaa tiedon hakemisen useasta eri lähteestä. Tietolähdeluokat eivät ole osa Jetpack-arkkitehtuuria, mutta niitä suositellaan käytettäväksi sovelluksen arkkitehtuurissa koodin erottelun parantamiseksi. Tavallisesti tietolähdeluokka voi mahdollistaa tiedon hakemisen joko verkosta tai paikallisesta välimuistista. (Muntenescu 2021.)

Vaikka projektin laajuuteen ei kuulunut ulkoisen tietokannan käsittelemistä verkon kautta, tein rutiineille (Kuva 22) ja tehtävälokeille omat tietolähdeluokat. Tietolähdeluokat helpottavat ulkoisen tietokannan integroimista sovellukseen mahdollisen jatkokehityksen aikana, ja tietolähdeluokan tietokantakäsittely metodeihin on mahdollista lisätä lokien kirjaamista virheenetsintää varten. Kun tietolähdeluokka luodaan, sille annetaan AppDatabase-luokan viittaama DAO-rajapinta parametrinä, jonka kautta tietolähdeluokka käsittelee tietokannan dataa.

Tein DatabaseManager-luokan, joka pitää viittaukset sovelluksen tietokantaan ja tietolähdeluokkiin (Kuva 23). Tarkoitukseni oli, että DatabaseManager-luokka toimii reittinä, jonka kautta muut sovelluksen komponentit pääsevät käsiksi tietolähdeluokkiin ja sitä kautta itse tietokantaan. Näin muiden sovelluskomponenttien ei tarvitse pitää viittausta tietokantaluokkaan tai tietolähdeluokkiin suoraan itse, vaan komponenttien tarvitsee vain luoda DatabaseManager-olio, jonka kautta ne voivat käyttää tietolähdeluokkia.



Kuva 22. Rutiinien tietolähdeluokka



Kuva 23. DatabaseManager-luokka

4.7 Kotlin-ohjelmointikielen Coroutine-toiminnot ja tietokannan käyttäminen

Kotlinin Coroutine-toiminnot mahdollistavat asynkronisen koodin suorittamisen taustalla, joka ei tuki sovelluksen pääsäiettä tai käyttöliittymän säiettä. Coroutine-toimintojen avulla voidaan suorittaa tehtäviä, jotka saattavat kestää kauan ilman, että se tukkii käyttöliittymän käyttöä. (Google Developers Training team 2020a.) Android-dokumentaatio suosittelee Coroutine-toimintojen käyttämistä asynkronisessa ohjelmoinnissa Android-kehityksessä (Android Developers 2020r).

Room-kirjaston tietokantaa ei voi käsitellä sovelluksen pääsäikeestä suoraan. Sovellus kaatuu, jos tietokantaa yrittää käyttää suoraan pääsäikeestä (Kuva 24). Jotta sovellus voisi käyttää tietokantaa, tietokannan käsittely pitää suorittaa taustasäikeeltä esimerkiksi Kotlinin Coroutine-toimintojen avulla.

```
Process: com.santtuhyvarinen.habittracker, PID: 11993
java.lang.IllegalStateException: Cannot access database on the main thread since it may potentially lock the UI for a long period of time.
    at androidx.room.RoomDatabase.assertNotMainThread(RoomDatabase.java:267)
    at androidx.room.RoomDatabase.query(RoomDatabase.java:323)
    at androidx.room.util.DBUtil.query(DBUtil.java:83)
    at com.santtuhyvarinen.habittracker.database.dao.HabitDao_Impl.getByIdTest(HabitDao_Impl.java:405)
    at com.santtuhyvarinen.habittracker.database.repositories.HabitRepository.getHabitByIdTest(HabitRepository.kt:48)
```

Kuva 24. Virheviesti, kun tietokantaa yrittää käyttää pääsäikeestä

Coroutine-toiminnon laajuus määrittää, että missä yhteydessä Coroutine-toiminto suoritetaan. Jetpack-arkkitehtuurin komponentit tukevat Coroutine-toimintojen käyttöä, ja komponenteilla on omat Coroutine-laajuudet. Esimerkiksi ViewModel-luokat voivat käyttää ViewModelScope-laajuutta, kun niiden tarvitsee suorittaa Coroutine-toimintoja. Coroutine-toiminto, joka laukaistaan ViewModelScope-laajuudessa, automaattisesti perutaan, kun sen laukaissut ViewModel-instanssi tuhotaan. (Google Developers Training team 2020a.)

Sovelluksessa käytetään pääasiassa Coroutine-toimintoja ViewModelScope-laajuudessa, kun sovelluksen tarvitsee käsitellä tietokantaa. Esimerkiksi kun tehtävälistan näkymässä merkataan tehtävä suoritetuksi, siitä luodaan tehtäväloki tietokantaan. Kun tehtävälistan näkymän TasksViewModel-komponentti aloittaa tehtävälokin luomisen TaskManager-komponentin avulla, toimenpide suoritetaan Coroutine-toiminnon kautta ViewModelScope-laajuudessa (Kuva 25). Tämä Coroutine-toiminto suoritetaan tiedonsiirrolle varatulla säikeellä, jonka Dispatchers.IO määrittää (Android Developers 2020r).

```
class TasksViewModel(application: Application) : AndroidViewModel(application) {

    private val databaseManager = DatabaseManager(application)
    private val taskManager = TaskManager(databaseManager)
    val iconManager = IconManager(application)

    fun createTaskLog(taskModel: TaskModel, status : String) {
        viewModelScope.launch(Dispatchers.IO) {  this: CoroutineScope
            taskManager.insertTaskLog(taskModel, status)
        }
    }
```

Kuva 25. Coroutine-toiminnon käyttäminen TasksViewModel-luokassa

5 Sovelluksen näkymät ja toiminnot

5.1 Fragment- ja ViewModel-luokat

Jokaisella sovelluksen näkymällä on oma Fragment-komponenttinsa, jotka toimivat näkymän käyttöliittymäohjainkomponenttina, joka hallitsee näkymän käyttöliittymäkomponentteja. Jokaista Fragment-komponenttia kohden on asettelutiedosto, joka sijaitsee layout-kansiossa sovelluksen resursseissa. Asettelutiedostoissa on määritelty näkymien asettelut ja Fragment-komponentin hallitsemat käyttöliittymäkomponentit, kuten listat, painikkeet ja tekstikentät.

Projektin on tarkoitus noudattaa Android-dokumentaation suosittelemaa arkkitehtuuria (Android Developers 2021b), jossa käyttöliittymäohjainkomponenttien eli Activity- ja Fragment-komponenttien käyttämä data hallinnoidaan Jetpack-kirjaston ViewModel- ja LiveData-komponenttien kautta. Fragment-komponenttien näkymän tarvitsemaa dataa voisi säilyttää suoraan Fragment-komponentissa itsessään, mutta koska Android-järjestelmä tuhoaa ja lataa käyttöliittymäohjainkomponentit uudestaan konfiguraatiomuutosten aikana, niiden sisältämä data katoaa ja pitää ladata uudelleen (Android Developers 2020k). Tämä on käyttäjän kannalta huono asia, jos hän menettää tekemänsä muutokset esimerkiksi näytön kääntämisen seurauksena.

Projektissani Fragment-komponenttien tehtävä on vain ohjata ja hallita sen asettelutiedostossa määriteltyjä käyttöliittymäkomponentteja. Fragment-komponentin näyttämä data valmistellaan ja haetaan sen oman ViewModel-luokan kautta, joten Fragment-komponentin ei tarvitse itse säilyttää mitään dataa tai sisältää mitään datakäsittelyn logiikkaa. Fragment-komponentti vain ohjaa, miten ja missä ViewModel-komponentin kautta haettua tai LiveData-komponentin kautta havainnoitua dataa näytetään. Fragment-komponentti myös ohjaa, että mitä tehdään, kun käyttäjä painaa esimerkiksi painiketta käyttöliittymässä.

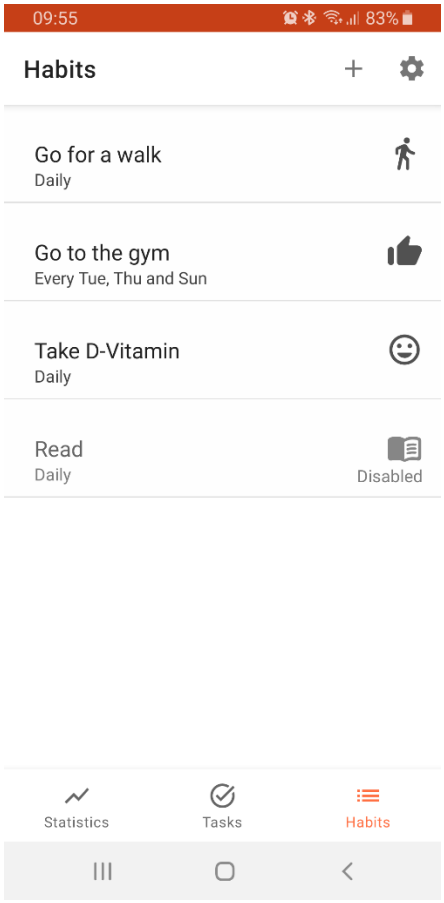
Fragment-komponenttia vastaavan ViewModel-luokan vastuu on taas hallita Fragment-komponentin toimintojen tarvitsemaa dataa ja toimia välikätenä Fragment-komponentin ja muiden toimintalogiikkaa suorittavien komponenttien välillä. ViewModel-luokka ei ole koskaan suoraan tekemisessä sen vastinparina toimivan Fragment-komponentin asetelman käyttöliittymäkomponenttien kanssa, vaan Fragment-komponentti on ainoastaan itse vastuussa sen asetelman käyttöliittymäkomponenttien hallitsemisesta.

Aikaisemmissa projekteissani olen tavallisesti säilyttänyt Fragment-komponenttien käyttämän dataa ja toimintalogiikkaa niissä itsessään. Tämä toimii ihan hyvin käyttötarkoituksissa, joissa esimerkiksi näyttöä ei pysty kääntämään tai näkymän datan säilyttäminen ei

ole tärkeää. Kuitenkin tämä lähestymistapa tekee kuitenkin koodista sekavan, koska muuttujat ja toimintalogiikan metodit ovat sekaisin Fragment-komponentissa. Tämän projektin aikana huomasin, että projektin kokonaisuuden hahmottaminen oli helpompaa, kun käyttöliittymän ohjaus hoidetaan Fragment-komponentissa ja datan hallinta hoidetaan sitä vastaavassa ViewModel-komponentissa.

5.2 Rutiinilistanäkymä

Sovelluksen rutiinilistanäkymässä näytetään lista, johon on listattu kaikki tietokannasta löytyvät rutiinit (Kuva 26). Listalta käyttäjä voi valita minkä tahansa rutiinin. Rutiinin valinnasta sovellus siirtyy rutiininäkymään, josta käyttäjä voi katsoa yksittäisen rutiinin tietoja tarkemmin ja siirtyä muokkaamaan sen tietoja rutiinilomakenäkymän kautta.



Kuva 26. Rutiinilistanäkymä (Sovelluksen toinen versio)

HabitsFragment-komponentti toimii rutiinilistanäkymän käyttöliittymäohjainkomponenttina, joka hallitsee kaikkia näkymän sisältämiä käyttöliittymäkomponentteja. Jetpack-kirjaston RecyclerView-komponentti toimii listana, missä rutiinit näytetään. RecyclerView-komponentissa näytettävien rutiinien esittämistä varten tein HabitsListAdapter-luokan, joka periytyy RecyclerView.Adapter-luokasta. HabitsListAdapter-luokka määrittää, miten rutiinit

näytetään listassa. HabitsListAdapter-luokka luo yksittäisen rutiinin kohdan näkymästä item_habit.xml-asettelutiedoston pohjalta.

Listassa näytettävät rutiinit haetaan HabitsViewModel-luokan kautta, joka hallinnoi rutiinilistanäkymän käyttämää dataa. HabitsViewModel-luokka muodostaa yhteyden tietokantaan DatabaseManager-luokan kautta, jonka avulla luokka hakee kaikki rutiinit tietokannasta listaa varten. Tietokannasta haetut rutiinit palautetaan käärittynä LiveData-komponenttiin, jonka HabitsFragment-komponentti voi hakea avoimen getHabits-metodin kautta. (Kuva 27.)

HabitsViewModel.kt

```
1   package com.santtuhyvarinen.habittracker.viewmodels
2
3   import ...
9
10  class HabitsViewModel(application: Application) : AndroidViewModel(application) {
11
12      val iconManager = IconManager(application)
13      private val databaseManager : DatabaseManager = DatabaseManager(getApplication())
14
15      fun getHabits() : LiveData<List<Habit>> {
16          return databaseManager.habitRepository.habits
17      }
18  }
```

Kuva 27. HabitsViewModel-luokka

HabitsFragment-komponentti antaa HabitsListAdapter-luokalle sen käyttämän rutiinidatan listamuodossa. HabitsFragment-komponentti havainnoi HabitsViewModel-luokalta saamansa LiveData-komponenttia, johon tietokannasta haettu rutiinidata on kääritty. Kun LiveData-komponentin käärimä data päivittyy, HabitsFragment-komponentti antaa rutiinidatan listamuodossa HabitsListAdapter-luokalle, joka valmistelee rutiinidatan näytettäväksi RecyclerView-komponentissa. (Kuva 28.)

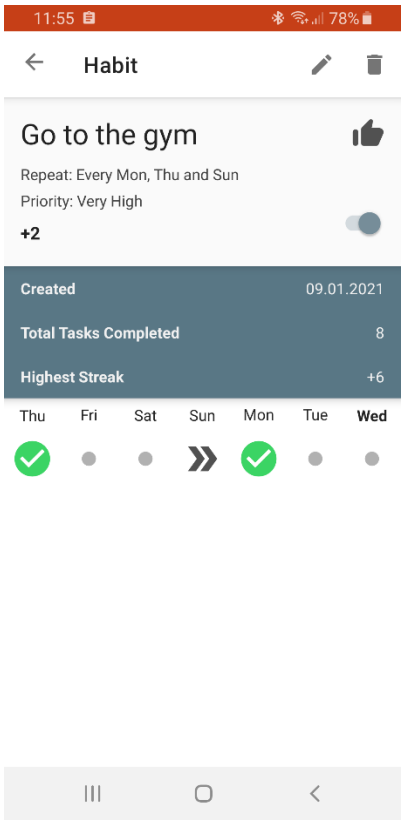
```
//Observer habits from database
val habitsObserver = Observer<List<Habit>> { list ->
    habitsAdapter.data = list
    habitsAdapter.sortData()
    habitsAdapter.notifyDataSetChanged()

    binding.progress.visibility = View.GONE
}
habitsViewModel.getHabits().observe(viewLifecycleOwner, habitsObserver)
```

Kuva 28. LiveData-komponentin havainnoiminen HabitsFragment-luokassa

5.3 Rutiininäkymä

Rutiininäkymässä käyttäjä voi katsoa tarkemmin yksittäisen rutiinin tietoja. Ensimmäisenä rutiininäkymässä näkyvät rutiinilomakkeesta asetettavat tiedot, kuten rutiinin nimi, toistuvuus ja kuvake. Rutiinin nykyinen pisteytys löytyy myös näiden tietojen alta. Näkymästä löytyy myös vipu, jonka avulla käyttäjä voi asettaa rutiinin pois päältä. Kun rutiini on asetettu pois päältä, rutiinille ei luoda päivittäisiä tehtäviä tehtävänäkymässä. (Kuva 29.)

Rutiinin yleisien tietojen alta löytyy rutiinin tilastotietoja, kuten kuinka monta rutiinin tehtävää on suoritettu yhteensä. Rutiinin tilastotiedoista näkyy myös rutiinin tehtävien suoritukset päiväkohtaisesti viimeiseltä seitsemältä päivältä. (Kuva 29.)



Kuva 29. Rutiininäkymä (Sovelluksen toinen versio)

HabitViewFragment-komponentti toimii rutiininäkymän käyttöliittymäohjainkomponenttina, ja HabitViewModel-luokka toimii HabitViewFragment-komponentin datahallintaluokkana. Kun rutiininäkymä avataan rutiinilistanäkymän kautta, rutiininäkymässä näytettävän rutiinin ID-numero annetaan parametrinä navigoinnin yhteydessä. Kun rutiininäkymä alustetaan, HabitViewModel-luokka käy etsimässä parametrinä annettua ID-numeroa vastaavan rutiinin tietokannasta (Kuva 30).

HabitViewModel.kt

```
19
20  class HabitViewModel(application: Application) : AndroidViewModel(application) {
21      private var initialized = false
22
23      private val databaseManager = DatabaseManager(getApplication())
24      val iconManager = IconManager(application)
25
26      private val habitWithTaskLogs : MutableLiveData<HabitWithTaskLogs> = MutableLiveData<HabitWithTaskLogs>()
27
28      //Set true to exit the fragment
29      private val shouldExitView : MutableLiveData<Boolean> = MutableLiveData<Boolean>()
30
31      fun initialize(id : Long) {
32          if (initialized) return
33
34          viewModelScope.launch { this: CoroutineScope
35              habitWithTaskLogs.value = fetchHabit(id)
36          }
37
38          initialized = true
39      }
```

Kuva 30. HabitViewModel-luokan alustaminen

Tietokantahaun palauttama rutiinidata kääritään LiveData-komponenttiin, jonka kautta HabitViewFragment-komponentti päivittää rutiinin tiedot sen hallinnoimiin käyttöliittymäkomponentteihin. Jos rutiinidatan käärimä LiveData-komponentti ei palauta mitään rutiinia tilanteessa, jossa tietokantahaku ei löytänyt ID-numeroa vastaavaa rutiinia tietokannasta, rutiininäkymä suljetaan heti. (Kuva 31.)

HabitViewFragment.kt

```
41      override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
42          super.onViewCreated(view, savedInstanceState)
43
44          //If there is no id set, navigate up
45          if(args.habitId < 0) {
46              findNavController().navigateUp()
47              return
48          }
49
50          updateProgress( showLayout: false)
51
52          habitViewModel = ViewModelProvider( owner: this).get(HabitViewModel::class.java)
53          habitViewModel.initialize(args.habitId)
54
55          //Observe Habit
56          val habitObserver = Observer<HabitWithTaskLogs> { habit ->
57              if(habit != null) {
58                  updateHabitValues(habit)
59              } else {
60                  //Could not load habit
61                  Toast.makeText(requireContext(), "Error: Can't show the habit", Toast.LENGTH_LONG).show()
62                  findNavController().navigateUp()
63              }
64          }
65          habitViewModel.getHabitWithTaskLogs().observe(viewLifecycleOwner, habitObserver)
```

Kuva 31. HabitViewFragment-luokan alustaminen

Kun sovelluksen navigointi on rutiininäkymässä, MainActivity-luokka vaihtaa sen hallinnoiman työkalupalkin painikkeet. Muissa näkymässä työkalupalkki näyttää painikkeet asetuksille ja rutiinin luomiselle, mutta rutiininäkymässä näytetään painikkeet rutiinin muokkaamiselle ja poistamiselle (Kuva 29). Kun käyttäjä painaa muokkauspainiketta, sovellus siirtyy rutiinilomakenäkymään, jossa käyttäjä voi muokata rutiinin tietoja. Poistamispainiketta painamalla käyttäjä voi poistaa rutiinin tietokannasta pysyvästi. HabitsViewFragment-komponentin tarvitsee hakea kyseiset painikkeet sen omistaman MainActivity-luokan kautta, jotta luokka voi asettaa painikkeisiin kuuntelijat, joita kutsutaan, kun käyttäjä painaa painikkeita (Kuva 32). Tämä ei ole kovin modulaarinen ratkaisu, koska HabitViewFragment-komponentti tarvitsee MainActivity-luokassa olevat painikkeet rutiinin muokkaus- ja poistamistoimintojen toteuttamiseen. Kuitenkin tämä ei ole sovelluksen toiminnan kannalta ongelma, koska HabitViewFragment-komponenttia ei koskaan käytetä muussa Activity-luokassa.

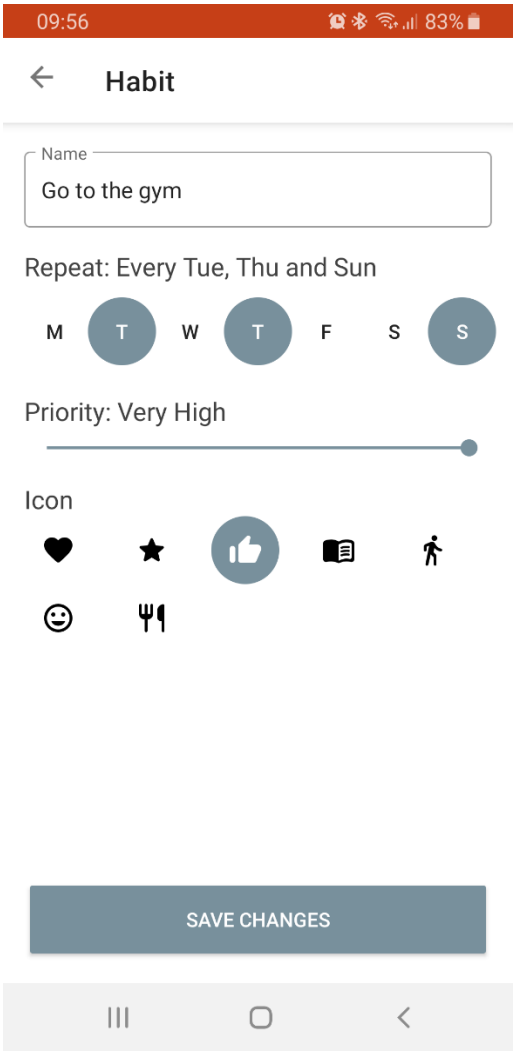
HabitViewFragment.kt

```
73        //Edit button on Activity ToolBar
74        val activity = (activity as MainActivity)
75        activity.getEditButton().setOnClickListener {  it: View!
76            if(context == null) return@setOnClickListener
77
78            val action = HabitViewFragmentDirections.actionFromHabitViewFragmentToHabitFormFragment(args.habitId)
79            findNavController().navigate(action)
80        }
81
82        //Delete button on Activity ToolBar
83        activity.getDeleteButton().setOnClickListener {  it: View!
84            if(context == null) return@setOnClickListener
85
86            showDeleteConfirmationDialog()
87        }
```

Kuva 32. Työkalupalkin painikkeiden alustaminen HabitViewFragment-luokassa

5.4 Rutiinilomake

Sovelluksen rutiinilomake toimittaa sovelluksessa kahta käyttötarkoitusta. MainActivity-luokan työkalupalkissa on painike, jota painamalla käyttäjä voi avata rutiinilomakkeen, kun hän haluaa luoda uuden rutiinin tietokantaan. Rutiinilomaketta käytetään myös, kun rutiininäkymästä siirrytään muokkaamaan jo olemassa olevaa rutiinia. Rutiinilomakkeessa voidaan asettaa rutiinin nimi, toistuvuus, tärkeysaste ja kuvake. (Kuva 33.)



Kuva 33. Rutiinilomakenäkymä (Sovelluksen toinen versio)

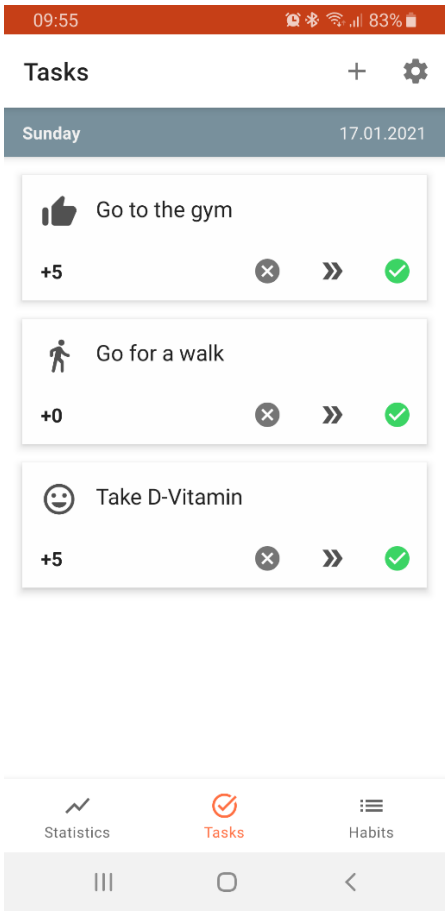
HabitFormFragment-komponentti toimii lomakenäkymän käyttöliittymäohjainkomponenttina, ja sen vastinparina toimii HabitFormViewModel-luokka, joka hallitsee lomakkeen käyttämää dataa ja joka tallentaa täytetyn lomakkeen tiedot tietokantaan, kun käyttäjä haluaa tallentaa rutiinin tiedot.

Kun rutiininäkymästä siirrytään lomakenäkymään, siirtymisen yhteydessä annetaan muokattavan rutiinin ID-numero parametrinä. Kun lomake avataan uuden rutiinin luomista varten, ID-numeron parametriä ei anneta. Kun HabitFormViewModel-luokka alustetaan, luokka tietää ID-numeron perusteella, että muokataanko jo olemassa olevaa rutiinia vai luodaanko uutta. Jos parametrinä on annettu ID-numero, ViewModel-luokka käy hakemassa ID-numeroa vastaavan rutiinin tietokannasta ja antaa sen vastinpari Fragment-komponentille rutiinidatan LiveData-komponentin kautta. Kun HabitFormFragment-komponentti saa tietokannasta haetun rutiinin tiedot LiveData-komponentin kautta, se täyttää haetun rutiinin tiedot lomakkeeseen.

Rutiinin toistuvuus voidaan säätää WeekDayPickerView-käyttöliittymäkomponentin kautta. WeekDayPickerView-komponentti on sovellusta varten luomani käyttöliittymäkomponentti, jossa on painikkeet jokaista viikonpäivää kohden. Painikkeilla painamalla käyttäjä voi säätä, että minä viikonpäivinä tehtävä toistuu.

5.5 Tehtävälistanäkymä

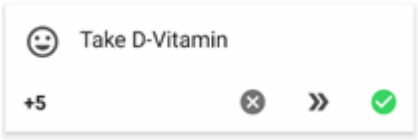
Sovelluksen tehtävälistanäkymässä käyttäjä näkee listan rutiiniensa pohjalta luoduista tehtävistä nykyiselle päivälle (Kuva 34). Listalta käyttäjä voi merkata tehtäviä joko suoritetuksi, epäonnistuneeksi tai ohitetuksi. Kun käyttäjä merkitsee tehtävän, tehtävä poistetaan listalta ja tapahtumasta luodaan uusi tehtäväloki tietokantaan.



Kuva 34. Tehtävälistanäkymä (Sovelluksen toinen versio)

TasksFragment-komponentti toimii näkymän käyttöliittymäohjainkomponenttina. Kuten HabitsFragment-komponentti, TasksFragment-komponentti käyttää RecyclerView-komponenttia ja RecyclerView.Adapter-luokasta periytyvää TasksAdapter-luokkaa päivittäisten tehtävien esittämiseen listamuodossa näkymässä. TasksViewModel-komponentti valmistelee ja hallinnoi TasksFragment-komponentin käyttämää dataa.

Sovelluksessa TaskModel-luokka esittää yksittäistä tehtävää. TaskModel-luokka ei sisällä varsinaisesti mitään muuta kuin viittauksen yhteen rutiiniin, jonka pohjalta tehtävän tiedot täytetään tehtävälistaan. TasksAdapter-luokka määrittää, että miten tehtävät näytetään RecyclerView-komponentissa. Tehtävälistalla tehtävässä näytetään rutiinin kuvake, nimi ja sen hetkinen pisteytys (Kuva 35). Tehtävässä on myös painikkeet, joilla käyttäjä voi merkata tehtävän suoritetuksi, ohitetuksi tai epäonnistuneeksi. Kun käyttäjä painaa jotain painikkeista, TasksAdapter-luokka poistaa tehtävän listalta.



Kuva 35. Esimerkki yksittäisestä tehtävästä

Päivittäisten tehtävien tuottamisen toteuttaa luomani TaskManager-luokka. TaskManager-luokalla on generateDailyTasks-metodi, joka tuottaa listan tehtävistä sille parametrinä annettujen rutiinien ja niiden omistamien tehtävälokien pohjalta. Metodi käy läpi annetun listan rutiineista, joihin on yhdistetty niiden omistamat tehtävälokit. Metodissa luodaan tehtävä rutiinille, jos rutiinin toistuvuus on asetettu sille päivälle ja jos käyttäjä ei ole asettanut rutiinia pois päältä rutiininäkymästä. Lisäksi tehtävää ei luoda, jos joku rutiinin tehtävälokien aikaleimoista on jo asetettu nykyiselle päivälle, mikä tarkoittaa, että käyttäjä on jo merkannut ja poistanut tehtävän listalta. TaskManager-luokka pitää tuotetun tehtävälistan käärittynä LiveData-komponentissa, jonka muutoksia havainnoimalla TasksFragment-komponentti päivittää tehtävälistanäkymän. (Kuva 36.)

```
class TaskManager(private val databaseManager: DatabaseManager) {

    val tasks : MutableLiveData<ArrayList<TaskModel>> by lazy {
        MutableLiveData<ArrayList<TaskModel>>()
    }

    fun generateDailyTasks(habits : List<HabitWithTaskLogs>) {
        val taskList = ArrayList<TaskModel>()

        for(habitWithTaskLogs in habits) {
            if(habitWithTaskLogs.habit.disabled) continue

            if(CalendarUtil.isHabitScheduledForToday(habitWithTaskLogs.habit)) {

                //Check if already added a task log for habit today. If already has a task log for today, don't add the task
                if (!TaskUtil.hasTaskLogForToday(habitWithTaskLogs)) {
                    taskList.add(TaskModel(habitWithTaskLogs.habit))
                }
            }
        }

        taskList.sortWith (compareByDescending<TaskModel> { it.habit.priority }.thenBy { it.habit.name } )

        tasks.value = taskList
    }
```

Kuva 36. TaskManager-luokka

TaskManager-luokka tarjoaa myös metodin tehtävälokin lisäämiseen tietokantaan. Kun käyttäjä merkitsee tehtävän tehtävälistanäkymässä, tehtävä poistetaan listalta ja merkinnästä luodaan uusi tehtäväloki tietokantaan. Tehtävälokiin lisätään sen omistavan rutiinin ID-numero, josta tiedetään, että mille rutiinille tehtäväloki kuuluu. Tehtävälokiin lisätään myös merkinnän status, joka voi olla joko suoritettu, ohitettu tai epäonnistunut. Tehtävälokin luonnin yhteydessä myös päivitetään rutiinin pisteytys. Jos tehtävä merkittiin suoritetuksi, rutiinin pisteytystä nostetaan yhdellä. Epäonnistunut tehtävä nollaa rutiinin pisteytyksen.

Sovellus ei automaattisesti luo tehtävälokia, jos käyttäjä jättää tehtävän merkitsemättä joltain päivältä. Kun sovellus hakee rutiinit yhdistettynä niiden omistamiin tehtävälokeihin tietokannasta, haettujen entiteettien alustuksen aikana sovellus tarkistaa rutiinin tehtävälokien perusteella, että jättikö käyttäjä merkkaamatta rutiinin tehtävän viime tehtävän toistokerralta (Kuva 19). Jos tehtävältä puuttuu loki viime tehtävän toistokerralta, rutiinin pisteytys nollataan.

Kehityksen aikana ongelmaksi muodostui tehtävänäkymän RecyclerView-komponentin käyttämän TaskAdapter-luokan ja LiveData-komponentin toiminen yhdessä. Tavoitteenani oli, että tehtävälistaan tehtävien muutoksiin käytettäisiin animaatioita, jotta listan sommittelun muutokset eivät olisi äkillisiä käyttäjän näkökulmasta. Esimerkiksi listan pitäisi automaattisesti siirtää muiden tehtävien paikkaa animaatiolla, kun käyttäjä merkitsee tehtävän suoritetuksi ja poistaa sen tehtävälistalta. RecyclerView.Adapter-komponentti, josta TaskAdapter-luokka periytyy, tukee animaatioiden käyttämistä, kun dataa lisätään tai poistetaan datasetiltä esimerkiksi notifyItemRemoved- ja notifyItemRangeChanged-metodien avulla (Kuva 37).

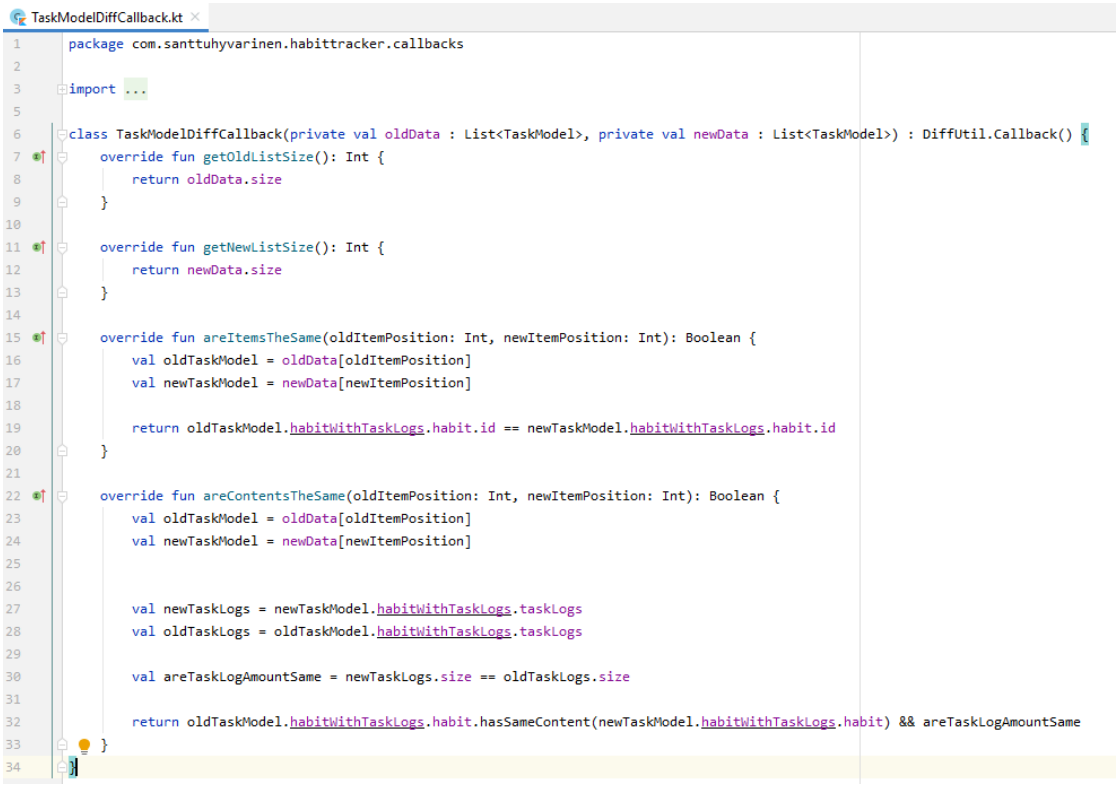
```
animationSet.setAnimationListener(object : Animation.AnimationListener
    override fun onAnimationStart(p0: Animation?) {
    }

    override fun onAnimationEnd(p0: Animation?) {
        layout.visibility = View.GONE
        data.remove(taskModel)
        notifyItemRemoved(position)
        notifyItemRangeChanged(position, itemCount)

        if(data.isEmpty()) tasksAdapterListener?.allTasksDone()
    }

    override fun onAnimationRepeat(p0: Animation?) {
    }
})
```

Kuva 37. Alkuperäinen ratkaisu tehtävän poistamisesta animaation kanssa TaskAdapter-luokassa

Mutta tehtävälistan toinen tärkeä tavoite oli, että tehtävälista näyttäisi LiveData-komponenttia havainnoimalla aina ajan tasalla olevia tehtäviä, jotta käyttäjälle ei näytettäisi tehtäviä, jotka hän on jo merkannut nykyiseltä päivältä. Myös tehtävänäkymässä pitäisi näkyä tehtävät käyttäjän juuri luomille rutiineille ilman, että käyttäjä joutuu manuaalisesti päivittämään tehtävänäkymän. Tehtävien ajan tasalla pitäminen muodostui ongelmaksi animaatioiden kannalta, koska kun TaskManager-olion tehtäviä pitävä LiveData-komponentti päivittyy, TaskAdapter-olion näyttämät tehtävät pitää vaihtaa uusiin. TaskAdapter-olion näyttämien tehtävien päivittäminen vaati sen notifyDatasetChanged-metodin kutsumista, joka päivittää datan, mutta peruu kaikki animaatiot. Tämä näkyi käyttäjälle tehtävien äkillisenä ilmestymisenä listalle tai katoamisena listalta. Käyttäjäkokemuksen kannalta oli tärkeää, että tehtävät eivät katoaisi äkillisesti listalta, kun käyttäjä merkitsee niitä suoritetuksi. Animaatioiden ja tehtävien ajan tasalla säilyttämisen yhteentoimivuuden ongelman ratkaisuksi osoittautui RecyclerView-komponentin DiffUtil-luokka.

Tucker kirjoittaa blogissaan samasta ongelmasta ja ehdottaa ratkaisuksi DiffUtil-luokkaa. DiffUtil on luokka, jolla voidaan laskea, että mitkä datasetin osat pitää päivittää, kun RecyclerView.Adapter-luokkaa pitää päivittää. DiffUtil-luokalla on calculateDiff-metodi, jolle voidaan antaa parametrinä DiffUtil.Callback-olio, jonka perusteella metodi palauttaa tiedon päivitettävistä datasetin osista. (Tucker 2.3.2018.)

Tämän ongelman ratkaisuksi kirjoitetun blogikirjoituksen pohjalta tein DiffUtil.Callback-luokasta periytyvän TaskModelDiffCallback-luokan, jonka avulla voidaan verrata TaskAdapter-komponentin vanhoja tehtäviä päivitettäviin tehtäviin. Kun TaskModelDiffCallBack-olio luodaan, sille annetaan parametrinä listat vanhoista tehtävistä ja uusista tehtävistä. TaskModelDiffCallback-luokka vertaa parametrinä annettuja listoja toisiinsa areItemsTheSame- ja areContentsTheSame-metodien avulla. (Kuva 38.)



Kuva 38. TaskModelDiffCallback-luokka

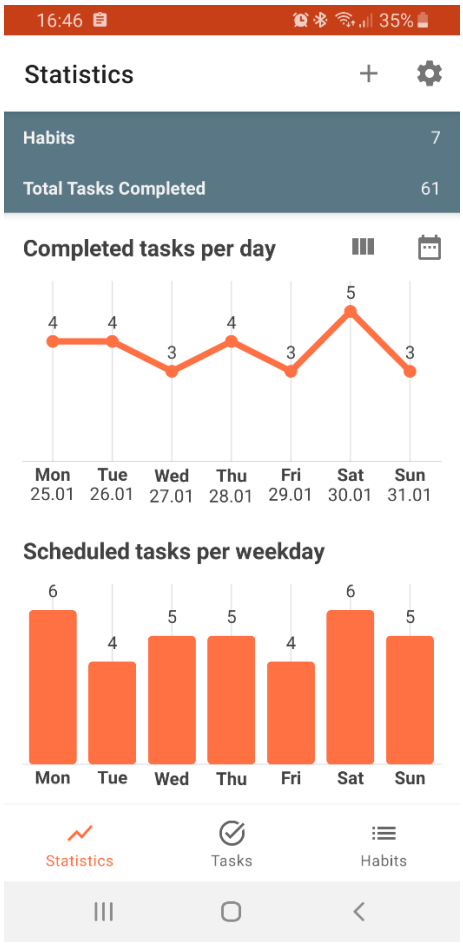
Tein TaskAdapter-luokkaan uuden updateData-metodin, joka päivittää tehtävälistan animaatioiden kanssa DiffUtil- ja TaskModelDiffCallback-luokkien avulla. DiffUtil-luokan calculateDiff-metodille annetaan TaskModelDiffCallback-olio, jonka rakentajalle annetaan parametrinä TaskAdapter-luokan vanha ja päivitetty data tehtävistä. Metodi palauttaa DiffUtil.DiffResult-olion, joka sisältää tiedon päivitettävistä tehtävistä. DiffUtil.DiffResult-olion dispatchUpdatesTo-metodia kutsumalla voidaan päivittää TaskAdapter-luokan data. (Kuva 39.) Nyt tehtävälistanäkymä pystyy pitämään tehtävät listalla ajan tasalla animaatioiden kanssa. Esimerkiksi kun tehtävä poistetaan listalta, muut tehtävät siirtyvät automaattisesti täyttämään poistetun tehtävän jättämän tyhjän tilan.

```
fun updateData(newData : ArrayList<TaskModel>) {
    val diffResult = DiffUtil.calculateDiff(TaskModelDiffCallback(data, newData))
    data = newData
    diffResult.dispatchUpdatesTo( adapter: this)
}
```

Kuva 39. TaskAdapter-luokan uusi metodi tehtävien päivittämistä varten

5.6 Tilastot

Sovelluksen tilastonäkymässä käyttäjä voi tarkemmin tutkia rutiiniensa edistymistä erilaisten tilastojen ja kaavioiden kautta. Tilastot luodaan tietokantaan tallennettujen rutiinien ja tehtävälokien perusteella. Tilastonäkymästä löytyy esimerkiksi viivakaavio, joka näyttää, että kuinka monta tehtävää käyttäjä on suorittanut päiväkohtaisesti. Käyttäjä voi viivakaavion yläpuolella olevista painikkeista vaihtaa kaavion näyttämää ajanjaksoa ja ajanjakson pituutta. Ajanjakson voi asettaa joko viikon, kahden viikon tai kuukauden pituiseksi. (Kuva 40.)



Kuva 40. Tilastonäkymä (Sovelluksen toinen versio)

Android-kehityksessä View-komponentti toimii muiden käyttöliittymäkomponenttien perustana, josta muut käyttöliittymäkomponentit, kuten tekstikentät ja painikkeet, periytyvät. View-komponenttia laajentamalla voi kehittää omia käyttöliittymäkomponentteja tyhjästä, jolloin View-komponenttia laajentava luokka hoitaa komponentin toiminnan ja piirtämisen näytölle. (Google Developers Training team 2020b.) View-komponentin pohjalta voi kehittää omaan kehitystarpeeseensa sopivia käyttöliittymäkomponentteja, kuten esimerkiksi käyttöliittymäkomponentin, joka piirtää viivakaavion sille annetun datan pohjalta.

Projektissa sekä viivakaavio että pylväskaavio käyttävät luomaani ChartView-luokkaa, joka on käyttöliittymäkomponentti, joka periytyy Androidin View-komponentista. ChartView-luokka piirtää sille antaman datan joko viiva- tai pylväskaaviona. Projektissa kehittäjä pystyy säätämään ChartView-komponentin attribuutteja, kuten tekstin kokoa ja värejä, joko asettelutiedostosta (Kuva 41) tai ohjelmakoodista. Resurssitiedostoista löytyy attrs.xml-tiedosto (Kuva 42), johon on määritelty ChartView-komponentin käyttämät attribuutit, joita voi säätää asettelutiedostosta käsin. Pelkästään attribuuttien määrittely attrs.xml-tiedostossa ei tuo asettelutiedostosta asetettujen attribuuttien arvoja sen View-luokkaan. View-luokan alustuksen aikana pitää koodissa manuaalisesti hakea asettelutiedostosta asetetut arvot attributeSet-oliosta ja yhdistää ne haluttuihin muuttujiin luokassa (Kuva 43). Androidin android-ktx-kirjasto tarjoaa withStyledAttributes-laajennuksen, jonka avulla voi yhdistää attribuuttien arvot muuttujiin vähemmällä koodilla (Google Developers Training team 2020b).

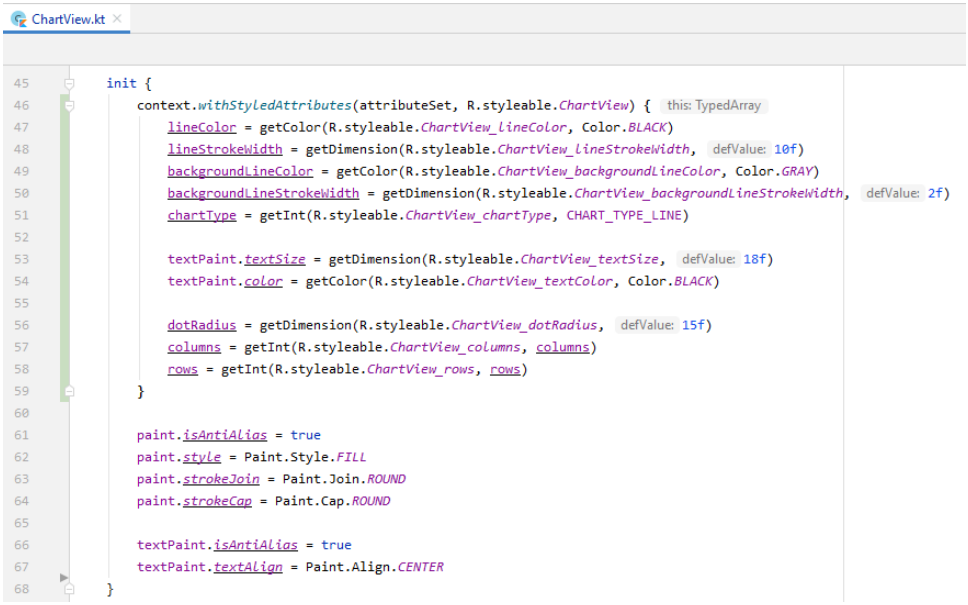
```
<com.santtuhyvarinen.habittracker.views.ChartView
    android:id="@+id/completedTasksChartView"
    android:layout_width="match_parent"
    android:layout_height="200dp"
    android:padding="15dp"

    app:graphType="line"
    app:columnCornerRadius="5dp"
    app:dotRadius="5dp"
    app:backgroundLineColor="@color/colorLineGraphBackground"
    app:textColor="@color/colorTextPrimaryLight"
    app:textSize="14sp"
    app:lineColor="@color/colorPrimary"
    app:lineStrokeWidth="5dp"

    app:layout_constraintTop_toBottomOf="@id/completedTasksHeader"/>
```

Kuva 41. ChartView-komponentti asettelutiedostossa

attrs.xml | layout_toolbar.xml | ChartView.kt | HabitTimelineView.kt

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <attr name="textSize" format="dimension"/>
    <attr name="textColor" format="color"/>

    <declare-styleable name="HabitTimelineView">
        <attr name="days" format="integer"/>
        <attr name="textSize"/>
        <attr name="textColor"/>
    </declare-styleable>

    <declare-styleable name="ChartView">

        <attr name="graphType" format="enum">
            <enum name="line" value="0"/>
            <enum name="column" value="1"/>
        </attr>

        <attr name="columnCornerRadius" format="dimension"/>
        <attr name="columns" format="integer"/>
        <attr name="lineColor" format="color"/>
        <attr name="lineStrokeWidth" format="dimension"/>
        <attr name="backgroundLineColor" format="color"/>
        <attr name="backgroundLineStrokeWidth" format="dimension"/>
        <attr name="dotRadius" format="dimension"/>
        <attr name="textSize"/>
        <attr name="textColor"/>
    </declare-styleable>
</resources>
```

Kuva 42. attrs.xml-tiedosto



Kuva 43. attrs.xml-tiedoston attribuuttien alustus ChartView-luokassa

Kun kehitetään uutta käyttöliittymä komponenttia View-komponentin pohjalta, pitää käyttää käyttöliittymän piirtämiseen View-komponentin onDraw-metodia, jossa piirtäminen toteutetaan Canvas-oliolle. Paint-oliolla voi piirtää Canvas-oliolle esimerkiksi kuvioita, viivoja, bittikarttoja tai tekstiä. (Google Developers Training team 2020b.)

ChartView-komponentti piirtää kaavion tyhjästä komponentin luokan onDraw-metodissa (Kuva 44). ChartView-komponentti laskee onDraw-metodissa kaavion sarakkeiden ja rivien pituudet näkymän leveyden ja pituuden perusteella. Ensimmäiseksi metodissa piirretään kaavion sarakkeiden taustaviivat ja nimikkeet silmukassa.

ChartView-komponenttiin voi määritellä joko asettelutiedostossa tai koodissa, että piirretäänkö kaaviossa näytettävä data viiva- vai pylväskaaviona. Kaavion muoto ei vaikuta ChartView-komponentissa muuten kuin, miten data piirretään kaavioon onDraw-metodissa (Kuva 45). Viivakaaviossa piirretään datan perusteella viivat sarakkeisiin ja ympyrät viivojen kohtauskohtiin. Pylväskaaviossa piirretään pyöreäkulmaiset suorakulmiot viivojen sijasta. Kuitenkin molemmat kaaviot toimivat muuten identtisesti ja saavat kaaviossa näytettävän datan samassa muodossa. Olisin voinut tehdä erilliset käyttöliittymäkomponentit molemmille kaaviolle, mutta yhdistämällä kaaviot yhteen komponenttiin säästin aikaa ja vältyin turhalta koodin toistamiselta projektissa.

```
override fun onDraw(canvas: Canvas) {
    super.onDraw(canvas)

    val textSize = textPaint.textSize
    val columnWidth = (width - paddingLeft - paddingRight) / columns
    val rowHeight = (height - paddingTop - paddingBottom - textSize*2) / rows

    val bottomHeight = (height - paddingBottom - textSize*2)

    //Background lines
    paint.color = backgroundLineColor
    paint.strokeWidth = backgroundLineStrokeWidth

    canvas.drawLine(paddingLeft.toFloat(), bottomHeight, stopX: width - paddingRight.toFloat(), bottomHeight, paint)

    //Use minimized labels if the column labels won't fit the view
    val useMinimizedLabels = shouldUseMinimizedLabels()

    //Draw columns
    for(column in 0 until columns) {
        //Column lines
        val columnX = paddingLeft + (column * columnWidth) + columnWidth / 2f
        canvas.drawLine(columnX, bottomHeight, columnX, paddingTop.toFloat(), paint)
```

Kuva 44. ChartView-luokan onDraw-metodi

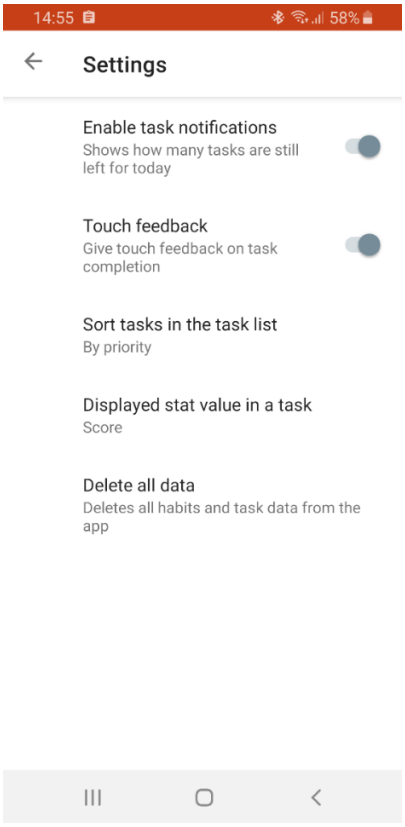
```
when(chartType) {
    CHART_TYPE_LINE -> {
        //Draw line chart data
        var previousX = 0f
        var previousY = bottomHeight
        for (i in 0 until columns) {
            val value = if(i < chartData.size) chartData[i].value else 0

            val x = paddingLeft + (i * columnWidth) + columnWidth / 2f
            val y = bottomHeight - (rowHeight * value)

            if(i > 0) canvas.drawLine(previousX, previousY, x, y, paint)

            canvas.drawCircle(x, y, dotRadius, paint)

            previousX = x
            previousY = y
        }
    }

    CHART_TYPE_COLUMN -> {
        //Draw columns chart data
        val margin = columnWidth / 10

        for (i in 0 until columns) {
            val value = if(i < chartData.size) chartData[i].value else 0

            val left = paddingLeft + (i * columnWidth).toFloat() + margin
            val top = bottomHeight - (rowHeight * value)
            val right = left + columnWidth - (margin*2)
            val bottom = bottomHeight

            canvas.drawRoundRect(left, top, right, bottom, rx: 10f, ry: 10f, paint)
        }
    }
}
```

Kuva 45. Viiva- ja pylväskaavion piirtäminen ChartView-luokan onDraw-metodissa

Kun käyttöliittymäkomponentin piirtämistä ohjelmoi onDraw-metodissa, on tärkeää olla tarkkana, kun laskee piirrettävien kuvioiden koordinaatteja. Väärin lasketut arvot tai muut pienet erehdykset voivat johtaa visuaalisiin virheisiin, joita ei välttämättä helposti huomaa ensisilmäyksellä. Esimerkiksi sovelluksen toisen version tilastonäkymästä (Kuva 40) voi nähdä, että viivakaavion nimikkeiden sijainnin kohdistus pystysuunnassa ei ole täydellinen. Virhe johtui siitä, että onDraw-metodissa nimikkeiden sijaintia pystysuunnassa siirrettiin haluttuun paikaan kaaviossa siirtämällä tekstin y-koordinaattia piirrettävän tekstin korkeuden verran. Nimikkeet eivät kohdistuneet tasaisesti toisiinsa verraten, koska piirrettävän tekstin korkeus vaihtelee eri nimikkeiden välillä. Virheen huomattuani korjasin sen poistamalla piirrettävän tekstin korkeuden onDraw-metodin koordinaattien laskukaavoista ja korvasin sen laskukaavassa tekstin koolla, joka on yhteinen kaikille nimikkeiden teksteille.

5.7 Asetukset

Sovelluksen asetuksista käyttäjä voi vaikuttaa eri sovelluksen toimintoihin, kuten muistutuksiin ja tehtävälistan järjestykseen. Suurin osa sovelluksen asetuksista, joita käyttäjä voi säätää, on esitetty näkymässä listamuodossa. (Kuva 46.)



Kuva 46. Asetukset-näkymä (Sovelluksen toinen versio)

Käyttäjä pystyy asetuksista valitsemaan, miten tehtävät järjestetään tehtävälistanäkymän listalla. Oletuksena tehtävät järjestetään tehtävän rutiinin asetetun prioriteetin perusteella. Tehtävät voidaan myös järjestää aakkosjärjestyksessä tai niiden pisteytyksen mukaan.

Tehtävän alalaidassa näytetään oletuksena tehtävän pisteytys (Kuva 35), mutta asetukset-näkymästä käyttäjä voi valita näytettäväksi sen paikalla rutiinin tehtävien suoritusten määrän. Vaihtoehtona on myös olla näyttämättä mitään sen paikalla. Asetuksen ideana on antaa käyttäjälle mahdollisuus valita, että haluaako hän keskittyä tehtävien suorittamiseen säännöllisesti pisteytyksen avulla vai tehtävien suoritusten määrään ylipäänsä.

Asetukset-näkymästä voidaan myös poistaa kaikki rutiinit ja tehtävälokit tietokannasta tarvittaessa painiketta painamalla. Kun painiketta painetaan, käyttäjältä ensin vahvistetaan, että haluaako hän varmasti poistaa kaikki tiedot tietokannasta.

Android-dokumentaatio suosittelee sovelluksen käyttäjän hallinnoimien asetuksien toteuttamaiseen AndroidX Preference -kirjastoa. AndroidX Preference -kirjastossa asetukset-näyttö luodaan käyttäen Preference-komponentteja, jotka voidaan määritellä joko XML-tiedostossa tai koodissa. (Android Developers 2019.) Päätin käyttää AndroidX Preference -kirjastoa sovelluksen asetuksien toteuttamiseen, koska olin käyttänyt sitä aikaisemmissa projekteissani ja koska mielestäni se tekee sovelluksen asetuksista johdonmukaisen ja nopeasti pystytettävän.

Asetukset-näkymän käyttöliittymäohjainkomponenttina toimii SettingsFragment-komponentti. Toisin kuin muut projektin käyttöliittymäkomponentit, SettingsFragment-komponentti periytyy AndroidX Preference -kirjaston PreferenceFragmentCompat-komponentista. PreferenceFragmentCompat-komponentti on Fragment-komponentti, joka näyttää käyttäjälle listan Preference-komponenteista, jotka voidaan luoda XML-tiedostosta (Android Developers 2020s). SettingsFragment-komponentilla ei ole omaa asettelutiedostoa, vaan komponentti luo sen näkymään listan asetuksista XML-tiedoston perusteella setPreferencesFromResource-metodin avulla (Kuva 47).

```
class SettingsFragment : PreferenceFragmentCompat() {

    private lateinit var settingsViewModel : SettingsViewModel
    override fun onCreatePreferences(savedInstanceState: Bundle?, rootKey: String?) {
        setPreferencesFromResource(R.xml.preferences, rootKey)

        settingsViewModel = ViewModelProvider( owner: this).get(SettingsViewModel::class.java)
```

Kuva 47. SettingsFragment-komponentin alustus

Sovelluksen asetuksia varten lisäsin preferences.xml-tiedoston projektin resurssitiedostoihin. Tiedostossa on määritelty kaikki sovelluksen asetukset-näkymästä löytyvät asetukset, joita käyttäjä voi käyttää sovelluksen toiminnan ja käyttökokemuksen muuttamiseen. Preference-komponentti esittää yksittäistä asetusta asetukset-näkymässä. Jokaiselle Preference-komponentille on määritetty avain, jolla sovellus pystyy hakemaan käyttäjän asettaman arvon SharedPreferences-komponentin kautta. Kaikille Preference-komponenteille on myös määritetty oletusasetukset defaultValue-attribuutin kautta. (Kuva 48.)

preferences.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">

    <SwitchPreferenceCompat
        android:key="enable_notifications_key"
        android:title="Enable task notifications"
        android:summary="Shows how many tasks are still left for today"
        android:defaultValue="true"/>

    <SwitchPreferenceCompat
        android:key="touch_feedback_key"
        android:title="Touch feedback"
        android:summary="Give touch feedback on task completion"
        android:defaultValue="true"/>

    <ListPreference
        android:key="task_sort_key"
        android:title="Sort tasks in the task list"
        android:entries="@array/SortTasksEntries"
        android:entryValues="@array/SortTasksEntryValues"
        android:summary="%s"
        android:defaultValue="priority"/>

    <ListPreference
        android:key="task_display_stat_key"
        android:title="Displayed stat value in a task"
        android:entries="@array/TaskValueEntries"
        android:entryValues="@array/TaskValueEntryValues"
        android:summary="%s"
        android:defaultValue="score"/>

    <Preference
        android:key="delete_habits_key"
        android:title="Delete all data"
        android:summary="Deletes all habits and task data from the app"/>

</PreferenceScreen>
```

Kuva 48. preferences.xml-tiedosto

Esimerkiksi ListPreference-komponentti näyttää käyttäjälle ponnahdusikkunan, josta käyttäjä voi valita listalta arvon, joka tallennetaan tekstinä. Listan arvot ovat määritelty ListPreference-komponentin entries- ja entryValues-attribuuteissa (Kuva 48). Entries-attribuuttiin määritetään taulukko, johon on listattu käyttäjälle näkyvät tekstit listalta. EntryValues-attribuuttiin taas määritellään taulukko, joka sisältää varsinaiset tekstit, jotka tallennetaan asetustiedostoon. ListPreference-komponenttia käytettiin asetuksen Preference-komponenttina, kun käyttäjän piti olla mahdollista valita useista eri vaihtoehdoista.

Yksinkertaisimpiin asetuksiin käytin SwitchPreferenceCompat-komponenttia, joka näyttää käyttäjälle vivun, jonka käyttäjä voi kytkeä päälle ja pois.

Koska asetukset vaikuttavat eri tavalla eri sovelluksen osissa, tein SettingsUtil-luokan, jonka kautta muut sovelluksen osat pystyvät hakemaan asetuksien arvoja. PreferenceFragmentCompat-komponentti käyttää oletuksena Preference-komponenttien asetusten tallentamiseen SharedPreferences-oliota, jonka voi hakea PreferenceManager-luokan getDefaultSharedPreferences-funktion avulla (Android Developers 2020s). Tekemässäni SettingsUtil-luokassa on joukko funktioita, joita kutsumalle voi hakea asetukset-näkymän asetusten arvoja. SettingsUtil-luokka hakee arvot SharedPreferences-olion kautta samojen tekstiavainten perusteella, mitä eri asetukset käyttävät asetukset-näkymässä. Esimerkiksi SettingsUtil-luokassa on isNotificationServiceEnabled-funktio, joka hakee SharedPreferences-olion avulla tiedon siitä, että onko muistutukset kytketty päälle asetuksista (Kuva 49).

```
fun isNotificationServiceEnabled(context: Context) : Boolean {
    val prefs = PreferenceManager.getDefaultSharedPreferences(context)
    return prefs.getBoolean(context.getString(R.string.setting_notification_enable_key),  defValue: true)
}
```

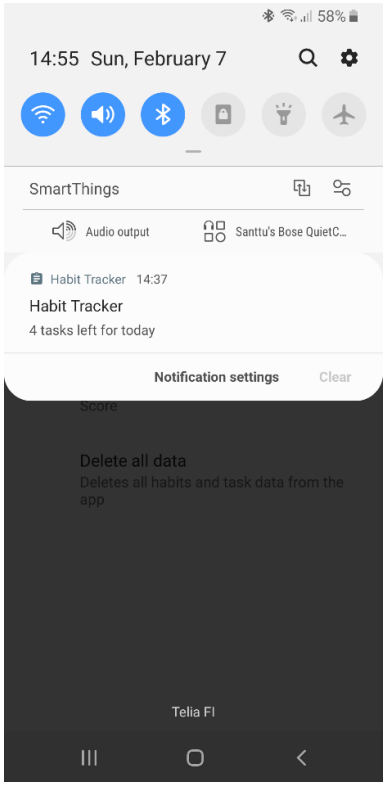
Kuva 49. SettingsUtil-luokan isNotificationServiceEnabled-funktio

5.8 Muistutukset

5.8.1 NotificationService-palvelu

Projektin aikana testasin sovellusta omien rutiinieni seuraamiseen säännöllisesti. Huomasin testauksen aikana, että vaikka rutiinin suorittaisikin päivän aikana, sen tehtävän saattoi kuitenkin unohtaa merkitä suoritetuksi sovelluksessa, mikä nollaa sovelluksen pisteytyksen. Tämän ongelman ratkaisuksi lisäsin sovellukseen NotificationService-palvelun, jonka tehtävänä on näyttää tietoa nykyisen päivän tehtävien tilasta laitteen ilmoituksissa. Oletuksena muistutukset ovat kytketty päälle, mutta käyttäjä voi halutessaan kytkeä muistutukset pois päältä asetukset-näkymästä.

Kun muistutukset ovat päällä, laitteen ilmoituksissa näytetään jatkuvasti muistutus, josta näkyy tieto jäljellä olevien tehtävien määrästä nykyiselle päivälle (Kuva 50). Muistutus toimii myös oikoreittinä sovellukseen. Kun muistutusta koskettaa, se avaa sovelluksen.



Kuva 50. Muistutus päivän tehtäville

Projektissa muistutuksista vastaa tekemäni NotificationService-komponentti, joka pohjautuu Androidin Service-komponenttiin. NotificationService-palvelu toimii taustalla, vaikka sovellus ei olisi aktiivisesti käytössä. Palvelun tehtävänä on näyttää ajan tasainen tieto päivittäisten tehtävien tilasta laitteen ilmoituksissa. NotificationService-palvelu toimii Foreground Service -palveluna.

Foreground Service on palvelu, joka toimii etualalla ja tekee toimintoja, jotka näkyvät käyttäjälle. Etualalla toimivia palveluja voivat käyttää esimerkiksi musiikkisoittimet. Jokaisen etualalla toimivan sovelluksen täytyy näyttää ilmoitus laitteen statuspalkissa. Sovelluksen, joka tähtää Android 9 -versioon tai korkeampaan, tarvitsee pyytää erillistä lupaa etualalla toimivalle palvelulle AndroidManifest.xml-tiedostossa. (Android Developers 2021e.)

Androidin API-tasosta 26 alkaen kaikkien muistutuksien pitää kuulua jollekin muistutuskanavalle. Muistutuskanava kuvastaa, että minkä tyyppisiä muistutuksia kanava lähettää. Muistutuskanavien avulla käyttäjät voivat halutessaan säätää muistutusten asetuksia muistutuskanavakohtaisesti ja valita myös, että mitkä muistutuskanavat ovat aktiivisia. (Yener 2020.) Koska sovellus tähtää myös Android-laitteisiin, joiden API-taso on 26 tai korkeampi, muistutusten lähettämistä varten lisäsin NotificationService-palveluun metodin, joka lisää sovellukselle muistutuskanavan, jos laitteen API-taso on 26 tai korkeampi (Kuva 51).

```
private fun createNotificationChannel() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val serviceChannel = NotificationChannel(
                CHANNEL_ID,
                CHANNEL_NAME,
                NotificationManager.IMPORTANCE_LOW
        )

        val manager = getSystemService(NotificationManager::class.java)
        manager.createNotificationChannel(serviceChannel)
    }
}
```

Kuva 51. Muistutuskanavan luominen

NotificationService-palvelu pitää tehtävien tilan tiedon ajan tasalla käyttämällä TaskManager-luokkaa ja yhteyttä tietokantaan. Palvelu havainnoi tietokannan rutiineja ja tehtävälokeja LiveData-komponentin kautta, joka päivittyy aina, kun tietokannan rutiinien ja tehtävälokien tauluihin tehdään muutoksia. Samalla tavalla kuin tehtävälistanäkymässä, palvelu luo tehtävät tietokannan rutiinien ja tehtävälokien perusteella käyttäen TaskManager-luokan generateDailyTasks-metodia, joka luo TaskModel-oliot esittämään päivittäisiä tehtäviä.

Muistutuksen sisältö päivitetään, kun TaskManager-luokan TaskModel-oliot eli tehtävät pitävä LiveData-komponentti päivittyy. Muistutuksen päivittämisessä LiveData-komponenttia havainnoimalla on se hyvä puoli, että muistutus näyttää aina ajan tasaista tietoa avoinna olevista tehtävistä, koska muistutus päivittyy heti, kun käyttäjä merkitsee tehtävän suoritetuksi sovelluksessa.

Muistutuksen päivittämistä varten lisäsin updateNotification-metodin NotificationService-luokkaan, jossa muistutuksessa näytettävä teksti lisätään jäljellä olevien tehtävien määrän perusteella (Kuva 52). Notification-olion eli muistutuksen luomiseen käytetään NotificationBuilder.Builder-komponenttia. Muistutuksen näyttämistä varten NotificatonCompat.Builder-komponentti tarvitsee vähintään otsikon, tekstisisällön ja kuvakkeen muistutukselle (Yener 2020).

Muistutuksessa näytettävä teksti ja statuspalkin kuvake määräytyy avoinna olevien tehtävien määrästä. Oletuksena muistutuksessa lukee, että kuinka monta tehtävää on vielä jäljellä. Jos avoinna olevia tehtäviä ei enää ole nykyiselle päivälle, muistutuksessa lukee, että kaikki tehtävät on suoritettu. Muistutuksen kuvake myös vaihdetaan jäljellä olevien tehtävien määrän perusteella, jotta pelkästään statuspalkkia vilkaisemalla tietää, että onko tehtäviä vielä jäljellä.

Kun Service-komponentti eli palvelu haluaa toimia etualalla, sen pitää kutsua sen startForeground-metodia, jolle annetaan parametrinä muistutuksen ID-numero ja muistutus eli Notification-olio (Android Developers 2021e). Kun muistutus päivitetään, NotificationService-palvelun updateNotification-metodissa annetaan päivitetty Notification-olio parametrinä startForeground-metodille (Kuva 52), joka myös päivittää muistutuksen.

NotificationService.kt

```
81      private fun updateNotification(tasksLeft : Int) {
82          val allTasksDone = tasksLeft == 0
83
84          val smallIcon = if(allTasksDone) R.drawable.ic_notification_tasks_done else R.drawable.ic_notification_tasks
85          val contentText = when(tasksLeft) {
86              0 -> "All tasks done for today"
87              1 -> "1 task left for today"
88              else -> "{tasksLeft} tasks left for today"
89          }
90
91          val notificationIntent = Intent( packageContext: this, MainActivity::class.java)
92          val pendingIntent = PendingIntent.getActivity( context: this,  requestCode: 0, notificationIntent,  flags: 0)
93
94          val notification = NotificationCompat.Builder( context: this, CHANNEL_ID)
95                  .setContentTitle("Habit Tracker")
96                  .setSmallIcon(smallIcon)
97                  .setContentText(contentText)
98                  .setContentIntent(pendingIntent)
99                  .setPriority(NotificationCompat.PRIORITY_LOW)
100                 .build()
101
102         startForeground(ONGOING_NOTIFICATION_ID, notification)
103     }
```

Kuva 52. Muistutuksen päivittäminen

5.8.2 Receiver-komponentit

Muistutuksien toiminnan kannalta oli tärkeää, että NotificationService-palvelu on aina toiminnassa taustalla, vaikka sovellus ei olisi aktiivisessa käytössä. Android-laitteen uudelleenkäynnistäminen aiheutti ongelmia muistutuksen toiminnalle. Kun laite käynnistettiin uudelleen, sovelluksen muistutusta ei näkynyt enää ilmoituksissa, koska NotificationService-palvelu oli alhaalla. Jos laite käynnistetään uudelleen, Service-komponentit eivät käynnisty itsestään, vaan ne pitää käynnistää erikseen. NotificationService-palvelu käynnistetään, kun käyttäjä avaa sovelluksen, jos muistutukset ovat kytketty päälle. Kuitenkin laitteen uudelleenkäynnistymisen järjestelmälähetyksiä kuuntelemalla on mahdollista käynnistää NotificationService-palvelu ilman, että käyttäjän tarvitsee avata sovellus uudelleenkäynnistymisen jälkeen.

Laitteen uudelleenkäynnistymisen kuuntelemista varten tein DeviceBootUpReceiver-luokan, joka periytyi Androidin BroadcastReceiver-luokasta. Kun laite käynnistyy uudelleen, DeviceBootUpReceiver-luokka käynnistää NotificationService-palvelun. (Kuva 53.)

DeviceBootUpReceiver.kt

```kotlin
package com.santtuhyvarinen.habittracker.receivers

import ...

class DeviceBootUpReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent?) {
        if(intent == null) return

        if(intent.action == Intent.ACTION_BOOT_COMPLETED) SettingsUtil.startNotificationService(context)
    }
}
```

Kuva 53. DeviceBootUpReceiver-komponentti

Ennen kuin BroadcastReceiver-komponentti voi vastaanottaa lähetyksiä, se pitää rekistöröidä vastaanottamaan niitä. Kun BroadcastReceiver-komponentti rekisteröidään AndroidManifest.xml-tiedostossa, se rekisteröidään staattisesti. Kun BroadcastReceiver-komponentti rekisteröidään dynaamisesti sovelluksen tai sovelluskomponentin kontekstin kanssa, BroadcastReceiver-komponentti vastaanottaa lähetyksiä niin kauan, kunnes kontekstin sovellus tai sovelluskomponentti suljetaan. (Googler 2020.)

Koska uudelleenkäynnistyksen järjestelmälähetystä on turha kuunnella, kun sovellus on jo päällä, rekisteröin DeviceBootUpReceiver-komponentin staattisesti projektin AndroidManifest.xml-tiedostossa (Kuva 54). Sovelluksen tarvitsee myös ilmoittaa tarvitsevansa lupaa uudelleenkäynnistyksen lähetyksen vastaanottamiseen AndroidManifest.xml-tiedostossa (Android Developers 2020t).

```xml
<receiver android:name=".receivers.DeviceBootUpReceiver">
    <intent-filter>
        <category android:name="android.intent.category.DEFAULT"/>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
        <action android:name="android.intent.action.QUICKBOOT_POWERON"/>
    </intent-filter>
</receiver>

<service android:name=".services.NotificationService"/>
```

Kuva 54. DeviceBootUpReceiver- ja NotificationService-komponenttien rekistöröinti AndroidManifest.xml-tiedostossa

Android 8 Oreo -käyttöjärjestelmäversio toi rajoitteita palvelujen käynnistämiseen, kun sovellus ei ole aktiivisesti etualalla. Palvelua ei voida käynnistää taustalta, jos sovellus ei ole aktiivisesti etualalla, ellei palvelu ole Foreground Service -tyyppinen. Palveluiden rajoittamisen tarkoituksena oli parantaa sovellusten suorituskykyä, koska taustalla suorittavat prosessit voivat vaikuttaa myös etualalla olevan sovelluksen suorituskykyyn. (Birch

28.8.2017.) Näiden rajoitteiden takia NotificationService-palvelun tarvitsee toimia etualalla, koska palvelua ei voi muuten käynnistää staattisen BroadcastReceiver-komponentin kautta Android 8 tai korkeammissa versioissa.

TaskManager-luokan generateDailyTasks-metodi luo tehtävät nykyisen päivän ja sille annettujen rutiinien toistuvuuden perusteella. Tämä tarkoittaa, että muistutuksen tieto nykyisen päivän tehtävien tilasta pitää päivittää keskiyöllä, kun päivä vaihtuu. Muistutuksen päivittämistä keskiyöllä varten rekisteröin dynaamisesti NotificationService-palveluun BroadcastReceiver-komponentin, joka kuuntelee jokaisen minuutin vaihtumista (Kuva 55). Jos BroadcastReceiver-komponentti rekisteröidään dynaamisesti, sen rekisteröinti pitää myös poistaa, kun sitä ei enää tarvita (Googler 2020). Kun NotificationService-palvelu ajetaan alas, minuuttien vaihtumista kuunteleva BroadcastReceiver-komponentin rekisteröinti poistetaan kanssa. Jokaisen minuutin vaihtumisen jälkeen BroadcastReceiver-komponentti tarkistaa, että ovatko tehtävät päivitetty nykyisen päivän aikana (Kuva 55). Kun päivä vaihtuu, tehtävät luodaan uudestaan ja muistutuksen sisältö päivitetään.

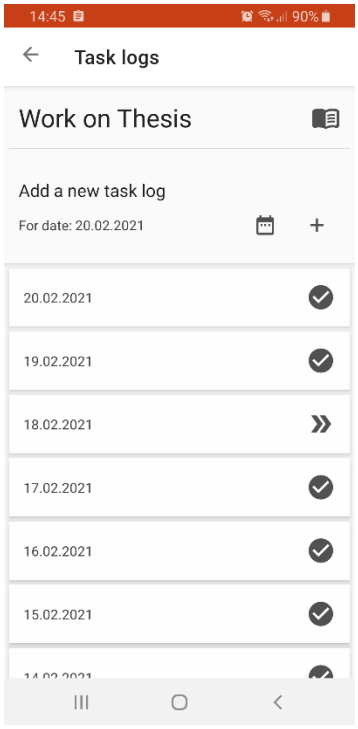
```
//Broadcast Receiver. For updating the notification at midnight
minuteTickReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        val intentAction = intent.action?: return
        if (intentAction.compareTo(Intent.ACTION_TIME_TICK) == 0) {
            //Check if midnight has passed from last tasks update
            val currentDate = DateTime.now()
            if(!CalendarUtil.areSameDate(currentDate, tasksUpdatedDatetime)) {
                Log.d(SERVICE_LOG_TAG,  msg: "Date changed. Updating notification")
                updateTasks()
            }
        }
    }
}

registerReceiver(minuteTickReceiver, IntentFilter(Intent.ACTION_TIME_TICK))
```

Kuva 55. Dynaamisesti rekisteröity BroadcastReceiver-komponentti NotificationService-palvelussa

5.9 Tehtävälokien hallintanäkymä

Sovelluksen kehityksen ja testauksen aikana heräsi tarve tehtävälokien lisäämiseen ja hallitsemiseen tehtävälistanäkymää laajemmin. Tehtävälistanäkymä tekee tehtävien merkitsemisen helpoksi, mutta näkymän kautta ei pystynyt merkitsemään tehtäviä menneille tai tuleville päiville. Myöskään rutiinin omien tilastojen ja yhteisien tilastojen ulkopuolelta käyttäjä ei pystynyt seuraamaan tehtävien lokeja. Näiden ominaisuuksien lisäämistä varten lisäsin tehtävälokien hallintanäkymän, jonka kautta käyttäjä pystyy katsomaan lisäämänsä yksittäisen rutiinin tehtävälokit ja lisäämään uusia tehtävälokeja (Kuva 56).

Tein tehtävälokien hallintanäkymälle, kuten muille näkymille, sen omat Fragment- ja ViewModel-komponentit näkymän käyttöliittymän ohjausta ja datahallintaa varten. Kuten tehtävä- ja rutiinilistanäkymät, tehtävälokien hallintanäkymä näyttää datan eli tehtävälokit RecyclerView-komponentissa listamuodossa käyttäen apuna RecyclerView.Adapter-luokan pohjalta luotua TaskManagementAdapter-luokkaa. TaskManagementAdapter-luokka yksinkertaisesti hallitsee, miten tietokannasta haetut tehtävälokit näytetään listalla eli RecyclerView-komponentissa.



Kuva 56. Tehtävälokien hallintanäkymä (Sovelluksen kolmas versio)

6 Käytettävyys ja julkaisu

Koko kehityksen aikana olen yrittänyt ottaa sovelluksen esteettömyyden huomioon sovelluksen suunnittelussa ja kehittämisessä. Sovelluksen esteettömyys on hyvä ottaa huomioon sovelluksen kehityksessä, koska se loogisesti mahdollistaa laajemman joukon käyttäjiä käyttää sovellustasi sujuvasti. Mielestäni kaikkien sovelluksen käyttäjien käyttökokemus parantuu, kun esteettömyys otetaan huomioon sovelluksen suunnittelussa. Liian pienet painikkeet ja huonosti luettavat tekstit tekevät sovelluksen käyttämisestä ei-optimaalista kaikille käyttäjille.

Vasta kehityksen loppupuolella on mielestäni hieman myöhäistä alkaa ottamaan esteettömyys huomioon sovelluksen kehityksessä. Sovelluksen esteettömyyden parantaminen kehityksen loppuvaiheessa saattaa vaatia sovelluksen toimintalogiikan muuttamista tai sovelluksen eri käyttöliittymäkomponenttien läpikäymistä yksitellen.

Material Design -ohjeistuksen esteettömyysosiossa (s.a.) ohjeistetaan käyttämään vähintään 48 x 48 dp kokoisia kosketusalueita sovelluksen painikkeille, jotta käyttäjien, joilla on vaikeuksia koskettaa pieniä kosketusalueita, olisi helppo koskettaa niitä. Rossin, Zhangin, Fogartyn ja Wobbrockin tutkimuksessa (2020, 21), jossa selvitettiin Android-sovellusten esteettömyyden esteitä, selvisi, että melkein 40 prosenttia tutkimuksessa testattujen sovellusten painike-elementeistä olivat joko liian kapeita tai liian lyhyitä. Kaikki sovelluksestani löytyvät painikkeet ovat joko 48 x 48 dp kokoisia tai suurempia. Koska kaikkien painikkeiden koot ovat määriteltynä values-resurssikansioiden dimens.xml-tiedostoissa, on yksinkertaista varmistaa, että painikkeiden koot noudattavat esteettömyysohjeistusta.

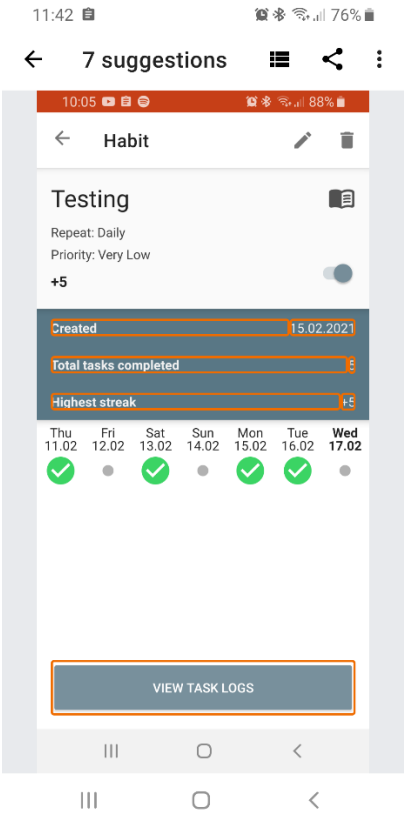
6.1 Accessibility Scanner -sovellus

Sovelluksen käyttökokemuksen ja esteettömyyden testaamisessa olisi hyvä testata sovellus oikeilla loppukäyttäjällä. Kehittäjällä on erilainen näkökulma sovelluksen käyttökokemukseen kuin loppukäyttäjillä, joilla ei ole aikaisempaa kokemusta sovelluksen toiminnasta. Sovelluksen toiminnot saattavat olla epäselkeitä loppukäyttäjille, vaikka ne olisivatkin itsestään selviä kehittäjälle. Käyttäjätestaus oli kuitenkin tämän projektin suunnitellun laajuuden ulkopuolella. On olemassa kuitenkin erilaisia työkaluja, joilla voi löytää parannuskohtia tai ongelmia sovelluksen esteettömyydestä.

Sovelluksen esteettömyyden kehittämisessä käytin apuna Googlen kehittämää Accessibility Scanner -sovellusta. Accessibility Scanner -sovelluksella voi ottaa kuvankaappauksia sovelluksen näkymistä, joiden perusteella Accessibility Scanner -sovellus antaa parannusehdotuksia sovelluksen esteettömyyden parantamiseksi (Google Help 2020).

Latasin Android-testilaitteeseeni Accessibility Scanner -sovelluksen Google Play -sovelluskaupasta, ja asetin sovelluksen päälle laitteen esteettömyysasetuksista. Accessibility Scanner -sovelluksen ohjaimet leijuvat muun näytön sisällön päällä ja sisältävät painikkeet, joilla voi ottaa näytön sisällöstä kuvankaappauksen tai sarjan kuvankaappauksia.

Otin Accessibility Scanner -sovelluksen avulla kuvankaappauksia rutiiniseurantasovelluksen eri näkymistä, joiden pohjalta Accessibility Scanner -sovellus ehdotti parannusehdotuksia asioista, jotka saattoivat haitata sovelluksen esteettömyyttä. Esimerkiksi rutiininäkymän parannusehdotukseksi ehdotettiin tiettyjen tekstien ja niiden taustojen välisten kontrastien parantamista (Kuva 57). Joissakin sovelluksen tekstikentissä käytin tummaa taustaa vaalean tekstin kanssa, mutta asetin tumman taustan turhan vaaleaksi tekstiin verrattuna. Annettujen parannusehdotuksien pohjalta muutin tekstikenttien taustat hieman tummemmaksi, jotta niiden teksti ei sekoittuisi taustan kanssa.



Kuva 57. Kuvankaappaus Accessibility Service -sovelluksen parannusehdotuksista rutiininäkymään

6.2 Sisällönkuvaustekstit

Näytönlukijat lukevat ääneen käyttäjälle painikkeissa ja muissa sovelluksen elementissä olevat sisällönkuvaustekstit. Puuttuvat sisällönkuvaustekstit voivat tehdä kuvapainikkeiden

sisältämät toiminnot hankalaksi tai mahdottomaksi käyttää henkilöille, jotka käyttävät näytönlukijoita (Ross, Zhang, Fogarty, & Wobbrock 2020, 2). Sovelluksen kehittämisessä pyrin siihen, että mahdollisimman monella sovelluksen painikkeisiin ja muihin elementteihin on liitetty niiden toimintoja kuvaavat sisällönkuvaustekstit. Sisällönkuvaustekstin voi liittää View-komponentin contentDescription-attribuuttiin, miten esimerkiksi olin lisännyt ne sovelluksen työkalupalkin kuvapainikkeisiin työkalupalkin asettelutiedostossa (Kuva 58).

```
    <androidx.appcompat.widget.AppCompatImageButton
        android:id="@+id/settingsButton"
        android:layout_width="48dp"
        android:layout_height="48dp"
        android:contentDescription="Settings"

        android:background="?selectableItemBackgroundBorderless"
        android:src="@drawable/ic_settings"
        android:layout_gravity="end"/>

    <androidx.appcompat.widget.AppCompatImageButton
        android:id="@+id/addHabitButton"
        android:layout_width="48dp"
        android:layout_height="48dp"
        android:contentDescription="Create a new habit"

        android:background="?selectableItemBackgroundBorderless"
        android:src="@drawable/ic_add"
        android:layout_gravity="end"/>
</androidx.appcompat.widget.Toolbar>
```

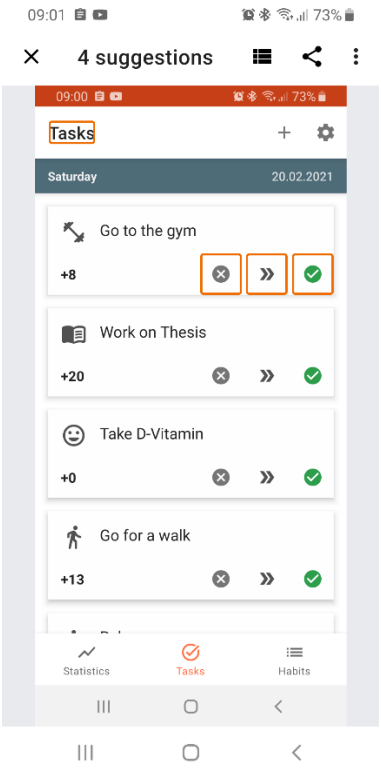
Kuva 58. Kuvapainikkeet, joihin on asetettu sisällönkuvaustekstit, layout_toolbar.xml-tiedostossa

Accessibility Service -sovellus antaa myös palautetta, jos skannatun näkymän kuvapainikkeista puuttuvat sisällönkuvaustekstit tai jos näkymässä toistuu useasti sama sisällönkuvausteksti. Joskus Accessibility Service -sovellus antaa palautetta toistuvista sisällönkuvausteksteistä, vaikka toistuva sisällönkuvausteksti ei olisi tietääkseni ongelma sovelluksen toiminnan kannalta. Esimerkiksi rutiiniseurantasovelluksen skannauksesta huomautettiin, että näkymän otsikko ja alanavigointipalkin painike jakavat sisällönkuvaustekstit, koska alanavigointipalkin painikkeen nimi vastaa loogisesti sen avaaman näkymän otsikkoa (Kuva 59).

Accessibility Service -sovelluksen skannaus kuitenkin huomautti tehtävälistanäkymän tehtävien kuvapainikkeiden toistuvista sisällönkuvausteksteistä (Kuva 59). Koska tehtävät luodaan saman asettelutiedoston pohjalta, kuvapainikkeiden sisällönkuvaustekstit ovat samat. Identtisten sisällönkuvaustekstien takia näytönlukijoilla oli hankala erottaa, että mikä kuvapainike kuului millekin tehtävälle.

Yllä kuvatun ongelman ratkaisemiseksi piti laittaa tehtävien kuvapainikkeet käyttämään sisällönkuvaustekstejä, jotka luodaan dynaamisesti tehtävän mukaan. Laitoin tehtävälistan

tehtävien esittämistä hallitsevan TaskAdapter-luokan liittämään tehtävän rutiinin nimen jokaisen tehtävän kuvapainikkeen sisällönkuvaukseen, kun TaskAdapter-luokka yhdistää tehtävien dataa ViewHolder-komponentteihin (Kuva 60). Dynaamisten sisällönkuvausten ansiosta näytönlukijat lukevat tehtävän kuvapainikkeen toiminnon nimen lisäksi rutiinin nimen, kun käyttäjä painaa tehtävän painiketta.



Kuva 59. Kuvankaappaus Accessibility Service -sovelluksen parannusehdotuksista tehtävälistanäkymään

```
//Set content descriptions for task buttons
holder.successButton.contentDescription = "Mark task as done: {habitName}"
holder.skipButton.contentDescription = "Skip task: {habitName}"
holder.failButton.contentDescription = "Mark task as failed: {habitName}"
```

Kuva 60. Dynaamisten sisällönkuvausten asettaminen TaskAdapter-luokassa

6.3 Julkaisu

Projektin lopuksi sovellus julkaistiin Google Play -sovelluskaupassa, josta sen voi ladata ilmaiseksi Android-laitteille. Google Play -kauppaan julkaisu tapahtui Google Play -konsolin kautta, jota olin käyttänyt aikaisempien harrastusprojektieni julkaisemiseen.

Julkaisuprosessi oli melko suoraviivainen, koska sovelluksessa ei ollut mitään mainoksia tai mitään, mitä käyttäjät voisivat ostaa sovelluksen sisällä. Ennen sovelluksen julkaisemista piti täyttää pari lomaketta koskien sovelluksen kohdeyleisöä ja sisältöä sovelluksen ikärajaa varten.

Android Studiossa tein allekirjoitetun Android App Bundle -tiedoston, jonka latasin Google Play -konsoliin sovelluksen julkaisemista varten. Sovelluksen voi ladata Google Play -konsoliin myös APK-tiedostona. Sovelluksen voi julkaista ensin rajoitetusti ennen kuin sovelluksen julkaisee kaikkien saataville Google Play -sovelluskauppaan. Ennen kuin julkaisin sovelluksen Google Play -sovelluskaupassa, testasin julkaisun ensin sisäisesti, mikä tarkoittaa, että vain henkilöt, joiden sähköpostiosoitteet olin lisännyt testaajien joukkoon, pystyivät lataamaan sovelluksen. Sisäisen julkaisun tarkoituksena oli varmistaa, että varsinaisen julkaisu sujuisi ongelmitta.

Sovellus löytyy Google Play -sovelluskaupasta alla olevan linkin kautta.
https://play.google.com/store/apps/details?id=com.santtuhyvarinen.habittracker&hl=fi-FI

Projektin lopuksi ohjelmakoodi julkaistiin GitHub-palvelussa, josta se on vapaasti katsottavissa ja ladattavissa.
https://github.com/sandels97/HabitTrackingApp

7 Johtopäätökset

Pääosin sanoisin, että johdannossa kuvatut tavoitteet projektille ovat saavutettu. Julkaistussa sovelluksesta löytyvät tavoitteissa kuvatut ominaisuudet. Sovelluksessa käyttäjä voi luoda, hallita ja poistaa rutiineja. Käyttäjä voi asettaa rutiinille nimen, kuvakkeen, toistuvuuden ja tärkeysasteen. Käyttäjän luomien rutiinien pohjalta sovellus luo automaattisesti päivittäisiä tehtäviä. Käyttäjä voi merkitä tehtäviä suoritetuksi, epäonnistuneeksi tai ohitetuksi. Sovellus pitää kirjaa käyttäjän merkitsemistä tehtävistä tehtävälokien avulla. Käyttäjä voi seurata rutiiniensa kehittymistä tehtävälokien pohjalta luotujen kaavioiden ja tilastojen avulla yleisellä ja rutiinikohtaisella tasolla. Sovellus julkaistiin Google Play -sovelluskaupassa, mistä se on ilmaiseksi ladattavissa.

Jos aloittaisin projektin uudestaan, valitsisin todennäköisesti jonkun toisen aiheen kuin rutiiniseurannan. Vaikka sovelluksen aihe toimi pelkästään lähtökohtana itse kehittämiselle, olen huomannut tämän ja muiden projektien aikana, että sovelluksen kehityksen pitäisi lähteä oikeasta kehitystarpeesta. Valitsin rutiiniseurannan aiheeksi, koska ajattelin, että projektin laajuutta voisi sopivasti laajentaa tarvittaessa projektin aikana. Projektin loppupuolella olisin halunnut vielä laajentaa projektin laajuutta, mutta oikeasta tarpeesta lähtevän kehitystarpeen puuttuessa oli vaikeaa lisätä enää mitään merkittäviä ominaisuuksia. Projektin lopuksi sovellus jäi ehkä turhan yksinkertaiseksi mielestäni, vaikka varsinaista ohjelmakoodia kertyi ihan hyvin.

Kuitenkin projektissa tuli tutustuttua laajasti eri Jetpack-kirjastojen komponentteihin. Toteutin sovelluksen navigaation Jetpack-kirjaston navigointikomponentilla. Yhden Activity-luokan rakenne sopi hyvin Jetpack-kirjaston navigointikomponentin kanssa. Fragment-komponenttien käyttäminen sovelluksen näkyminä Activity-komponenttien sijasta mahdollisti pysyvien elementtien, kuten työkalu- ja alanavigointipalkin, jakamisen eri näkymien välillä. Jetpack-kirjaston navigointikomponentti sopi hyvin tähän käyttötapaukseen, jossa oli vain yksi kehys ja jossa navigoitiin eri Fragment-komponenttien välillä. Androidin käyttöliittymässä voi kuitenkin käyttää useita Fragment-komponentteja vierekkäin (Android Developers 2020d), joten olisi mielenkiintoista selvittää toisessa projektissa, miten Jetpack-kirjaston navigointikomponentti sopii hallitsemaan käyttöliittymää, jossa on useita eri Fragment-komponentteja käytössä vierekkäin samaan aikaan.

Kaikki projektin ohjelmointi toteutettiin Kotlin-ohjelmointikielellä, eikä missään välissä projektia tarvinnut käyttää Javaa. Raportissa viitatut Android-dokumentaatiot ja -ohjeistukset perustuivat melkein kokonaan Kotlin-ohjelmointikieleen, joten apua Kotlin-ohjelmointikie-

len käyttämiseen Android-kehityksessä ei ollut hankala löytää. Kotlin-ohjelmointikielen Coroutine-toimintojen hyödyntäminen jäi turhan suppeaksi projektissa. Sovelluksessa ei ollut tarvetta asynkronisille toiminnoille muuten kuin tietokannan käsittelyn yhteydessä, joten projektista ei löytynyt käyttökohteita, missä olisi voinut hyödyntää Coroutine-toimintoja kattavammin ilman projektin laajuuden kasvattamista esimerkiksi ulkoisen tietokannan käyttämisen verkon kautta.

Projektissa noudatettiin Androidin virallisen dokumentaation suosittelemaa sovellusarkkitehtuuria, jossa käyttöliittymäohjainkomponenttien käyttämä data hallitaan Jetpack-kirjaston ViewModel- ja LiveData-komponenttien kautta. Jokaisella sovelluksen näkymällä on oma Fragment-komponentti, joka hallitsee sen asettelutiedostossa määritettyjä käyttöliittymäkomponentteja, ja oma ViewModel-komponenttinsa, joka hallitsee näkymän käyttämää dataa. ViewModel-komponentit hakevat tietokannan datan käyttöliittymäohjainkomponenteille tietolähdeluokkien kautta. ViewModel-komponentit paljastavat haetun datan käyttöliittymäohjainkomponenteille LiveData-komponenttien avulla, joiden käärimän datan muutoksia käyttöliittymäohjainkomponentit voivat havainnoida.

Projektin arkkitehtuurin onnistumista voi reflektoida Android-kehityksestä kerätyistä ohjeista. Projektissa käytettyä arkkitehtuuria voidaan parhaiten kuvailla Model-View-ViewModel-arkkitehtuuriksi, koska miten se käyttää ViewModel-komponentteja. Projektin arkkitehtuuri noudattaa suhteellisen hyvin Verdecchian, Malavoltan ja Lagon tutkimuksessa koottuja ohjeita MVVM-arkkitehtuurissa Android-kehityksestä. Verdecchian, Malavoltan ja Lagon (2019, 147) MVVM-arkkitehtuurikohtaisissa ohjeissa neuvotaan muun muassa käyttämään yhtä tiettyä komponenttia lähteenä sovelluksen käyttämälle datalle ja näkymäkomponenttien viittauksien pitämistä erossa näkymämallikomponenteista. Sovelluksessani DatabaseManager-luokka toimii niin sanottuna porttina sovelluksen käyttämälle datalle, jonka kautta muut sovelluksen osat hakevat ja käsittelevät dataa. Millään sovelluksessa käytetyllä ViewModel-komponentilla ei ole mitään viittausta käyttöliittymäohjain- tai käyttöliittymäkomponentteihin.

Kuitenkin Verdecchian, Malavoltan ja Lagon (2019, 147) MVVM-arkkitehtuurikohtaisissa ohjeissa neuvotaan myös, että kaikki toimintalogiikka näkymäkomponenteista pitäisi siirtää näkymämallikomponenttiin riippumatta operaation koosta. Projektissani sovelluksen käyttöliittymäkomponenteissa tehdään jotain datakäsittelyoperaatioita, kuten päivän hakeminen tai tekstin muodostaminen datan perusteella, tekemieni apuluokkien kautta, joiden staattisia funktioita voi kutsua mistä tahansa komponentista. Vaikka kaiken toimintalogiikan säilyttäminen perusteellisesti ViewModel-komponenteissa tekisi rakenteesta johdonmukaisemman, siinä menettää mielestäni joustavuutta tai muokattavuutta. Apuluokkien

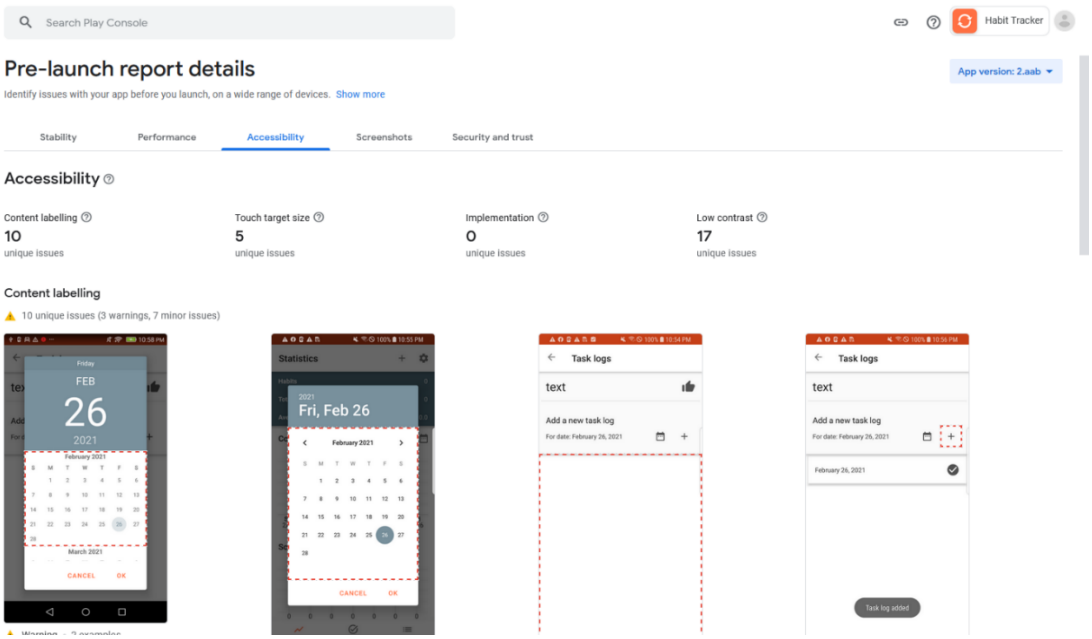
käyttäminen kuitenkin pitää toimintalogiikan koodin itsenäisessä ja loogisessa paikassa, josta sitä voi käyttää mistä tahansa sovelluksen komponenteista.

Sovelluksessa ei käytetä varsinaista datasidontaa, mikä on osa MVVM-arkkitehtuuria. Datasidonnalla voidaan sitoa näkymäkomponentin arvoja suoraan näkymämallikomponentin arvoihin, jolloin näkymä päivittyy automaattisesti, kun siihen sidottu datakin päivittyy (Lou 2016, 19). Koska projektin aikana noudatin pitkälti Android-dokumentaation suosittelemaa arkkitehtuuria, en käyttänyt datasidontaa projektissa, vaikka Jetpack-kirjastoista löytyy oma kirjasto datasidonnallekin. Projektissa data yhdistetään käyttöliittymäkomponentteihin Fragment-komponentin ohjelmakoodissa ViewModel-komponentin LiveData-komponentteja havainnoimalla. LiveData-komponentteja havainnoimalla kuitenkin voi varmistaa, että käyttöliittymän näyttämä data vastaa varsinaisen datan tilaa (Android Developers 2021c).

Android-dokumentaation suositteleman arkkitehtuurin hyviin käytäntöihin kuuluu muun muassa sovelluksien osien modulaarisuus ja niiden vastuiden selkeä määrittäminen (Android Developers 2021b). Myös Verdecchian, Malavoltan ja Lagon tutkimuksen (2019, 146) arkkitehtuurista kerätystä ohjeistuksessa mainitaan sovellusten komponenttien vastuiden selkeä määrittäminen. Sovelluksen näkymät ja eri hallintaluokat, kuten DatabaseManager-, IconManager- ja TaskManager-luokat, ovat modulaarisia, ja niitä oli yksinkertaista hyödyntää eri sovelluksen osissa. Sovelluksen osiin tehdyt muutokset erittäin harvoin aiheuttivat mitään odottamattomia ketjureaktioita, jotka olisivat aiheuttaneet ongelmia sovelluksen toisissa osissa.

Sovelluksen esteettömyyden parantamisessa käytettiin apuna Androidin esteettömyydestä kirjoitettuja tutkimuksia ja Googlen kehittämää Accessibility Scanner -sovellusta. Käyttöliittymäkomponenteissa on yritetty ottaa huomioon, että niillä kaikilla on niiden tarkoitusta kuvaava sisällönkuvausteksti, jonka näytönlukijat voivat lukea. Kehityksessä yritettiin ottaa huomioon myös, että käyttöliittymän painikkeet ovat tarpeeksi suuria, jotta niitä olisi helppo painaa.

Julkaisun yhteydessä osoittautui, että sovelluksessa oli vielä olla epäkohtia sen esteettömyydessä, joita voidaan parantaa mahdollisessa jatkokehityksessä. Google Play -konsolin esijulkaisun raportissa on oma osionsa sovelluksen esteettömyydelle. Iso osa raportissa ilmoitetuista epäkohdista liittyivät sisällönkuvausteksteihin tai sisällön kontrastiin. Jotkut raportin epäkohdista liittyivät Androidin omaan päivävalintaikkunaan, jota projekti käyttää, kun käyttäjän pitää valita päivä. (Kuva 61.) Jatkokehityksessä voisi olla mielenkiintoista selvittää, että paljonko esteettömyyden epäkohtia on Androidin omissa komponenteissa, miten merkittäviä ne ovat ja miten kehittäjät voivat itse vaikuttaa niihin.



Kuva 61. Sovelluksen esteettömyys Google Play -konsolin esijulkaisuraportissa

Sovelluksen käytettävyyttä ja esteettömyyttä voi parantaa erilaisten työkalujen, kuten Accessibility Scanner -sovelluksen, avulla, mutta sovelluksen käytettävyyden ja esteettömyyden toimivuutta on hankala arvioida ilman sovelluksen testaamista oikeilla loppukäyttäjillä. Projektin laajuuteen ei kuulunut käyttäjätestauksen toteuttaminen, mutta seuraavissa projekteissa tai mahdollisessa jatkokehityksessä olisi hyvä ottaa mukaan käyttäjätestaus työkaluksi esteettömyyttä suunnitellessa. Accessibility Scanner -sovellus ei myöskään antanut palautetta itse luoduista käyttöliittymäkomponenteista, kuten viivakaaviosta. Jos itse luotu käyttöliittymäkomponentti piirtää tekstin itse, Accessibility Scanner -sovellus ei vaikuta antavan siitä palautetta.

Projektin aikana opin mielestäni kehittämään Android-projektia johdonmukaisemmin Android-dokumentaation suositteleman arkkitehtuurityylin avulla. Ennen projektia en ollut johdonmukaisesti käyttänyt Android-projekteissani mitään tiettyä arkkitehtuurityyliä, mikä välillä pakotti projektin rakenteen muuttamista myöhemmin. Tämän projektin aikana ei tullut missään vaiheessa tarvetta muuttaa projektin rakennetta laajasti, koska projektin eri osille oli määritelty selkeät vastuut ja riippuvaisuudet. Huomasin myös projektin aikana, että kiinnitin enemmän huomiota jatkokehitettävyyteen ja esteettömyyteen, kun ohjelmoin sovellusta. Yritin tietoisesti välttää niiden arvojen, kuten tekstien ja mittojen, kovakoodaamista, jotka olisivat järkevämpää määrittää resurssitiedostoissa eri laitekonfiguraatioiden kannalta. Yritin myös ottaa huomioon kuvakepainikkeiden ja muiden käyttöliittymäele-

menttien koot ja sisällönkuvaustekstit esteettömyyden takia heti alussa, jotta projektin lopussa ei olisi tarvetta käydä korjailemassa eri käyttöliittymäelementtien esteettömyyttä eri puolilla sovellusta.

Aion jatkossakin hyödyntää Jetpack-kirjaston ViewModel- ja LiveData-komponentteja Android-kehittämisessä. Niiden käyttäminen vaati hieman enemmän työtä projektin alussa, mutta niiden käyttämisestä oli näkyviä hyötyjä verrattuna datakäsittelyyn suoraan Fragment-komponentissa. Helposti näkyvin hyöty oli datan säilyminen laitteen konfiguraation muutosten välillä ilman erillistä manuaalista ratkaisua. Arkkitehtuurikomponenttien käyttäminen teki myös projektin rakenteesta selkeämmän, kun tarvitsi muokata jotain yksittäistä sovelluksen näkymää tai osaa.

Mielestäni tämän projektin aikana osaamiseni Kotlin-ohjelmointikielestä syventyi. Kokemukseni on, että kun opettelee uutta ohjelmointikieltä, sen käyttäminen on aluksi hitaampaa ja vaivalloista verrattuna jo opeteltuihin ohjelmointikieliin. Uuden ohjelmointikielen sisäistämisessä pitää oppia ohjelmointikielikohtaiset tavat toteuttaa asioita. Tämän takia vierastin aluksi Kotlinin käyttöönottoa Android-kehityksessä. Kun ohjelmoin ensimmäisen kerran Kotlin-ohjelmointikielellä, tavallisesti yritin toteuttaa asiat samalla tavalla, miten olisin toteuttaneet ne Javalla. Tavallisesti myöhemmin löytyi siistimpi tapa toteuttaa ne Kotlinilla käyttäen Kotlinin omia toimintoja. Ohjelmointikielen opettelun alussa piti kanssa turvautua dokumentaatioon ja ohjeisiin myös yksinkertaisten toimintojen toteuttamisessa. Tämän projektin aikana ohjelmointi Kotlinilla meni mielestäni erittäin sujuvasti. Hyödynsin paljon enemmän Kotlin-kohtaisia tapoja toteuttaa asioita, ja ohjelmointi ei takkuillut siihen, että piti jatkuvasti selvittää, miten jonkun yksinkertaisen toiminnon toteuttaminen Kotlinilla erosi Javasta. Jatkossa aion käyttää Kotlin-ohjelmointikieltä Android-kehityksessä aina, kun aloitan uuden projektin, koska koen Kotlin-ohjelmointikielen sujuvammaksi käyttää kuin Javan. Kotlinilla syntyy selvästi myös vähemmän vakiokoodia, mikä tekee mielestäni koodista helppolukuisempaa.

Jos olisin tehnyt jotain projektissa eri tavalla, olisin tehnyt enemmän taustatutkimustyötä jo projektin alussa koskien Androidin eri arkkitehtuurityylejä ja käytäntöjä. Aloitin itse sovelluksen kehittämisen heti projektin alussa, koska itseäni kiinnosti eniten itse sovelluksen toteuttaminen käytännössä. Olin jo projektin alussa valinnut Android-dokumentaation suositteleman arkkitehtuurin käytettäväksi, mutta olisi ollut mielenkiintoista verrata sitä enemmän muihin mahdollisiin arkkitehtuurityyleihin.

Sanoisin, että olen suhteellisen tyytyväinen projektin lopputulokseen ja valmistuneeseen sovellukseen. Projekti saatiin toteutettua aikataulussa ilman merkittäviä ongelmia. Projektin palautus meni itse asettamastani määräajasta vähän yli, koska halusin hioa raporttia vielä. Projektin tavoitteet olisivat voineet olla hieman konkreettisempia alussa, mutta ne selkeytyivät projektin aikana. Opin projektin aikana paljon Android-kehityksestä, josta on varmasti hyötyä tulevaisuuteni kannalta Android-kehittäjänä. Uskon, että projektissani oppimistani tavoista tulee olemaan hyötyä myös Android-kehityksen ulkopuolella johdonmukaisemman kehityksen muodossa.

8 Lähdeluettelo

Android Developers 2019. Settings. Luettavissa: https://developer.android.com/guide/topics/ui/settings. Luettu 8.2.2021.

Android Developers 2020a. Meet Android Studio. Luettavissa: https://developer.android.com/studio/intro. Luettu 10.12.2020.

Android Developers 2020b. Configure your build. Luettavissa: https://developer.android.com/studio/build. Luettu 20.1.2021.

Android Developers 2020c. Introduction to Activities. Luettavissa: https://developer.android.com/guide/components/activities/intro-activities. Luettu 16.12.2020.

Android Developers 2020d. Fragments. Luettavissa: https://developer.android.com/guide/fragments. Luettu 8.3.2021.

Android Developers 2020e. App Manifest Overview. Luettavissa: https://developer.android.com/guide/topics/manifest/manifest-intro. Luettu 16.12.2020.

Android Developers 2020f. App resources overview. Luettavissa: https://developer.android.com/guide/topics/resources/providing-resources. Luettu 28.12.2020.

Android Developers 2020g. Android Jetpack. Luettavissa: https://developer.android.com/jetpack. Luettu 11.12.2020.

Android Developers 2020h. Save data in a local database using Room. Luettavissa: https://developer.android.com/training/data-storage/room. Luettu 4.1.2021.

Android Developers 2020i. Navigation. Luettavissa: https://developer.android.com/guide/navigation. Luettu 13.12.2020.

Android Developers 2020j. Create dynamic lists with RecyclerView. Luettavissa: https://developer.android.com/guide/topics/ui/layout/recyclerview. Luettu 31.12.2020.

Android Developers 2020k. ViewModel Overview. Luettavissa: https://developer.android.com/topic/libraries/architecture/viewmodel. Luettu 28.12.2020.

Android Developers 2020l. Get started with the Navigation component. Luettavissa: https://developer.android.com/guide/navigation/navigation-getting-started. Luettu 13.12.2020.

Android Developers 2020m. Migrating Room databases. Luettavissa: https://developer.android.com/training/data-storage/room/migrating-db-versions. Luettu 4.1.2021.

Android Developers 2020n. Defining data using Room entities. Luettavissa: https://developer.android.com/training/data-storage/room/defining-data. Luettu 9.1.2021.

Android Developers 2020o. Foreign Key. Luettavissa: https://developer.android.com/reference/androidx/room/ForeignKey. Luettu 13.1.2021.

Android Developers 2020p. Accessing data using Room DAOs. Luettavissa: https://developer.android.com/training/data-storage/room/accessing-data. Luettu 4.1.2021.

Android Developers 2020q. Define relationships between objects. Luettavissa: https://developer.android.com/training/data-storage/room/relationships. Luettu 13.1.2021.

Android Developers 2020r. Kotlin coroutines on Android. Luettavissa: https://developer.android.com/kotlin/coroutines. Luettu 12.1.2021.

Android Developers 2020s. PreferenceFragmentCompat. Luettavissa: https://developer.android.com/reference/androidx/preference/PreferenceFragmentCompat. Luettu 8.2.2021.

Android Developers 2020t. Manifest.permission. Luettavissa: https://developer.android.com/reference/android/Manifest.permission. Luettu 14.2.2021.

Android Developers 2021a. Service overview. Luettavissa: https://developer.android.com/guide/components/services. Luettu 14.2.2021.

Android Developers 2021b. Guide to app architecture. Luettavissa: https://developer.android.com/jetpack/guide. Luettu 2.3.2021.

Android Developers 2021c. LiveData Overview. Luettavissa: https://developer.android.com/topic/libraries/architecture/livedata. Luettu 8.3.2021.

Android Developers 2021d. Pass data between destinations. Luettavissa: https://developer.android.com/guide/navigation/navigation-pass-data. Luettu 3.2.2021.

Android Developers 2021e. Foreground services. Luettavissa: https://developer.android.com/guide/components/foreground-services. Luettu 12.2.2021.

Ardito, L., Coppola, R., Malnati, G. & Torchiano, M. 2020. Effectiveness of Kotlin vs. Java in android app development tasks. Information and software technology, 127.

Atlassian s.a. Gitflow Workflow. Luettavissa: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow. Luettu 11.12.2020.

Banerjee, M., Bose, S., Kundu, A. & Mukherjee, M. 2018. A COMPARATIVE STUDY: JAVA VS KOTLIN PROGRAMMING IN ANDROID APPLICATION DEVELOPMENT. International Journal of Advanced Research in Computer Science, 9, 3, s. 41-45.

Birch, J. 28.8.2017. Exploring Background Execution Limits on Android Oreo. Medium. Luettavissa: https://medium.com/exploring-android/exploring-background-execution-limits-on-android-oreo-ab384762a66c. Luettu 14.2.2021.

Clear, J. 2018. Atomic Habits: The life-changing million copy bestseller. Kindle Edition. Cornerstone Digital.

Duhigg, C. 2013. The Power of Habit. Random House Books. Lontoo.

Eler, M., Rojas, J., Ge, Y. & Fraser, G. 2018. Automated Accessibility Testing of Mobile Apps. 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), s. 116-126.

Google Developers Training team. 2020a. Android Kotlin Fundamentals: 6.2 Coroutines and Room. Codelabs. Luettavissa: https://developer.android.com/codelabs/kotlin-android-training-coroutines-and-room#0. Luettu 12.1.2021.

Google Developers Training team. 2020b. Advanced Android in Kotlin 02.1: Creating Custom Views. Luettavissa: https://developer.android.com/codelabs/advanced-andoid-kotlin-training-custom-views#0. Luettu 31.1.2021.

Google Help 2020. Get Started with Accessibility Scanner. Luettavissa: https://support.google.com/accessibility/android/faq/6376582?hl=en&visit_id=637440478231941426-1652158357&rd=1. Luettu 20.12.2020.

Googler. 2020. Android fundamentals 07.3: Broadcast receivers. Codelabs. Luettavissa: https://developer.android.com/codelabs/android-training-broadcast-receivers#0. Luettu 14.2.2021.

Gossmann, J. 8.10.2005. Introduction to Model/View/ViewModel pattern for building WPF apps. Microsoft. Luettavissa: https://docs.microsoft.com/en-us/archive/blogs/john-gossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps. Luettu 14.3.2021.

Heller, M. 23.3.2020. What is Kotlin? The Java alternative explained. InfoWorld. Luettavissa: https://www.infoworld.com/article/3224868/what-is-kotlin-the-java-alternative-explained.html. Luettu 11.12.2020.

Kotlin Programming Language. 2019. Data Classes. Luettavissa: https://kotlinlang.org/docs/reference/data-classes.html. Luettu 9.1.2021.

Kotlin Programming Language. 2020. Null Safety. Luettavissa: https://kotlinlang.org/docs/reference/null-safety.html. Luettu 20.1.2021.

Lardinois, F. 7.5.2019. Kotlin is now Google’s preferred language for Android app development. TechCrunch. Luettavissa: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/?ncid=txtlnkusaolp00000616. Luettu 10.12.2020.

Lou, T. 2016. A Comparison of Android Native App Architecture – MVC, MVP and MVVM. Pro gradu -tutkielma. Aalto yliopisto. Luettavissa: https://pure.tue.nl/ws/portalfiles/portal/48628529/Lou_2016.pdf. Luettu 3.1.2021.

Material Design s.a. Accessibility. Luettavissa: https://material.io/design/usability/accessibility.html. Luettu 10.12.2020.

Muntenescu, F. 2021. Android Room with a View – Kotlin. Codelabs. Luettavissa: https://developer.android.com/codelabs/android-room-with-a-view-kotlin#0. Luettu 12.3.2021.

Oliveira, V., Teixeira, L. & Ebert, F. 2020. On the Adoption of Kotlin on Android Development: A Triangulation Study. 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). s. 206-216.

Protalinski, E. 8.12.2014. Google releases Android Studio 1.0, the first stable version of its IDE. VentureBeat. Luettavissa: https://venturebeat.com/2014/12/08/google-releases-android-studio-1-0-the-first-stable-version-of-its-ide/. Luettu 10.12.2020.

Protalinski, E. 8.5.2018. Google launches Android Jetpack, a set of components to speed up app development. VentureBeat. Luettavissa: https://venturebeat.com/2018/05/08/google-launches-android-jetpack-a-set-of-components-to-speed-up-app-development/. Luettu 3.1.2021.

Ross, A., Zhang, X., Fogarty, J. & Wobbrock, J. 2020. An Epidemiology-inspired Large-scale Analysis of Android App Accessibility. ACM transactions on accessible computing, 13(1), s. 1-36.

Tucker, B. 2.3.2018. No More notifyDataSetChanged(). Abnormally Driven. Luettavissa: http://blog.abnormallydriven.com/2018/03/03/diffutil-and-recyclerview/. Luettu 7.2.2021.

Vendome, C., Solano, D., Linán, S. & Linares-Vásquez, M. 2019. Can everyone use my app? An Empirical Study on Accessibility in Android Apps. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), s. 41-52.

Verdecchia, R., Malavolta, I. & Lago, P. 2019. Guidelines for Architecting Android Apps: A Mixed-Method Empirical Study. 2019 IEEE International Conference on Software Architecture (ICSA), s. 141-150.

Yener, M. 2020. Advanced Android in Kotlin 01.1: Using Android Notifications. Luettavissa: https://developer.android.com/codelabs/advanced-android-kotlin-training-notifications#0. Luettu 13.2.2021.