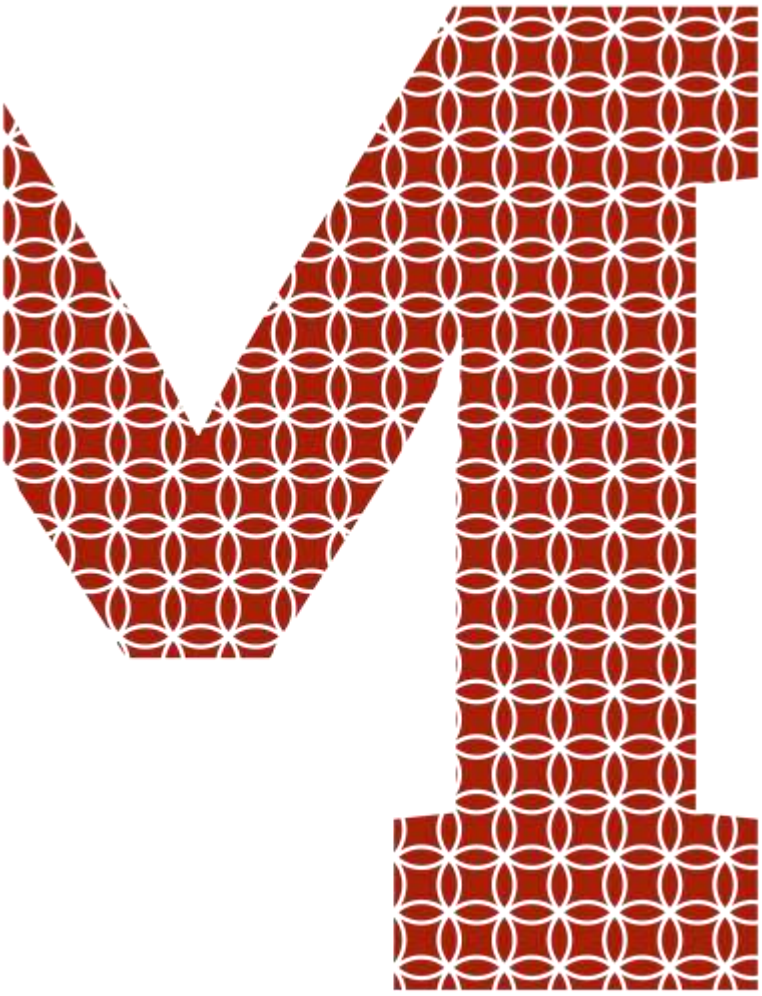


Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Elisa Jalava

Android-sovelluksen koodin ja arkkitehtuurin uudistaminen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

25.5.2019

Tekijä Otsikko Sivumäärä Aika	Elisa Jalava Android-sovelluksen koodin ja arkkitehtuurin uudistaminen 52 sivua 25.5.2019
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Yliopettaja, Erja Nikunen Lehtori, Jussi Alhorinne

Insinöörityössä Android-sovellus uudistettiin koodipohjalta. Tarkoituksena oli päivittää sen tietokanta ja käyttöliittymä käyttäen Androidin arkkitehtuurikomponentteja. Sovelluksen arkkitehtuurista oli tarkoitus saada selkeää ja koodista luettavaa, jotta uusia ominaisuuksia pystyttäisiin toteuttamaan.

Sovelluksessa käytettiin aluksi kovakoodattua SQLite-tietokantaa. Perinteisen SQLiten ollessa työläs käsitellä, ei tietokannan rakennetta pystytty päivittämään. Uusia ominaisuuksia varten tietokanta piti päivittää uuteen versioon ja vanha koodi uudistettava.

Uudistukseen käytettiin Googlen tarjoamaa Room-arkkitehtuurikomponenttia. Room on ORM-tyylinen abstrakti kehys SQLiten ylle. Se mahdollistaa tehokkaan yhteyden Java-koodin ja SQLite-tietokannan välillä muuttamalla tietokantahaun tuloksia Java-objekteiksi ja Java-objekteja tietokantaobjekteiksi.

Migraatio aloitettiin luomalla jokaiselle tietokantataululle omat Entity- ja DAO-luokkansa. Entity-luokat toimivat tietokantataulujen malleina, jotta Room voi annotaatioiden avulla luoda niistä Java-olioita. DAO-luokat tarjoavat abstraktit metodit tietokannan kanssa työskentelemiseen.

DAOt yhdistetään RoomDatabase-olioon, joka rakennetaan sovelluksen ajon alussa ja suorittaa migraatiot. Tämän sovelluksen migraatio oli kaksivaiheinen.

Tietokanta liitetään näkymään joko kontrollerin tai ViewModelin kautta. Kontrolleri sisältää metodit yleisille tietokantatoiminnoille.

ViewModel tarjoaa näkymälle tarkkailijan, joka tarkkailee tietokantakyselyä tai -oliota. Muutoksen tapahduttua näkymää huomautetaan ja UI-päivitys tehdään. Tarkkailtavat ovat LiveData-objekteja, jotka sisältävät kyselyn tuloksen. Jokaiselle näkymäfragmentille tehtiin oma malliluokkansa. Mallit ovat elinkaaririippuvaisia: ne ovat aktiivisina silloin, kun näkymä on.

Tuloksena sovelluksen jatkokehittäminen on sujuvampaa.

Avainsanat	Room, ViewModel, LiveData, arkkitehtuurikomponentit

Author Title Number of Pages Date	Elisa Jalava Updating the Code and Architecture of an Android App 52 pages 25 May 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software engineering
Instructors	Erja Nikunen, Principal Lecturer Jussi Alhorinne, Lecturer

In this thesis a mobile app for Android platforms was upgraded. Updating the app meant refactoring outdated and deprecated code to ease programming new features and introducing new developers to the app. Changes were made to improve the code's architecture, modularity and readability and replace deprecated libraries with the newest, most updated ones.

The app's original database was coded using the SQLite framework. The database's structure was hard-coded and working with the database required a lot of work. A simple database insertion requires multiple lines of boilerplate code, and the hard-coded structure made updating and migrating the database structure nearly impossible. Since the future features that have been designed for the app require migrating the database to a new version, it was decided to use Room to update it. Room is one of Google's architecture components for Android and it provides an abstraction layer over SQLite. As an ORM, it maps the database entries into Java objects and vice versa.

After establishing the structure of the upcoming database, each table in the database was made an Entity class to work as a model for the table. Each entity was written a DAO class that provides the methods to communicate with the database. The methods are abstract and use annotations to identify their task, thus minimizing the amount of code required to communicate with the database. The DAOs are accessed through a RoomDatabase class, which upon building automatically runs the migrations written for it and validates the database. The migration was made in two stages and unit tests were built for the app's purposes.

The database is accessed through a repository class, which contains methods for all the database queries. The queries are run on a separate thread.

ViewModel is an architecture component that introduces a new model for coding. ViewModel accesses the database model and gets a LiveData object from one of Room's queries. LiveData, an architecture component, is provided to the view as an observable that notifies the UI every time the object is updated. ViewModel is aware of the view's lifecycle and the observers are active if the view is active. Through the creative use of different LiveData object types, the UI can be dynamically updated when the user makes changes to the database.

The outcome is an app that is easier to update and introduce new developers to.

Keywords	Android, LiveData, Viewmodel, Room, Architecture

Sisällys

Lyhenteet

ORM - Object Relational Mapping. Olioiden kartoitus relatiivisesti, esim. SQL-kyselystä Java-olioksi.

API – Application Programming Interface. Ohjelmointirajapinta.

APK – Android Package. Android-sovelluksen paketti, joka julkaistaan Play-kauppaan.

DAO – Data Access Object. Olio, joka pääsee käsiksi tietokantaan.

UI – User Interface. Käyttöliittymä.

MVC - Model-View-Controller. Sovellusarkkitehtuurimalli.

MVP - Model-View-Presenter. Android-sovellusarkkitehtuurimalli.

MVVM - Mode-View-ViewMode. Android-sovellusarkkitehtuurimalli.

1 Johdanto

Insinöörityö tehtiin yksityiselle yritykselle ja työn tilaajana oli ko. yrityksen toimitusjohtaja. Opinnäytetyönä uudistettiin yrityksen julkaisemaa Android-sovellusta ja siihen pohjautuvia maksullisia sovelluksia.

Sovellus on Android-käyttöjärjestelmää käyttäville mobiililaitteille julkaistu ruokapäiväkirjasovellus. Sovellusta käytetään kartoittamaan jokapäiväisiä ruokailuja käyttäjän tekemien valokuvien ja muistiinpanojen avulla. Käyttäjä tutustuu itse omaan ruokavalioonsa ja ruokailutottumuksiinsa pystyessään katselemaan omia ruokailujaan usean päivän ajalta.

Sovelluksen tulevaisuuden tavoitteena on luoda siihen päivityksiä, joiden tarkoitus on tehdä sovelluksesta entistä sujuvampi käyttää ja tuoda pitkään toivottuja ominaisuuksia sen käyttäjille. Jotta haluttuja päivityksiä voitaisiin alkaa tekemään, sovellus vaati tietokantamigraation eli sovelluksen tietokannan sisäistä rakennetta tuli uudistaa. Tässä insinöörityössä keskitytään enimmäkseen tähän migraatioon ja sitä johtaneisiin arkkitehtuuripäivityksiin.

Tarkoituksena on hyödyntää Googlen Androidille tarjoamia arkkitehtuurikomponentteja sekä Room-tietokantamallia, jotta vanhasta tietokannasta saisi muokattavan ja helpomman käyttää. Käyttämällä arkkitehtuurikomponentteja sovelluksen arkkitehtuuria ja modulaarisuutta parannettiin ja sovelluksen laitteelle aiheuttamaa kuormitusta vähennettiin. Näin uusien työntekijöiden on helpompi perehtyä koodiin ja tulevien ominaisuuksien ohjelmointi vaatii vähemmän vaivaa.

2 Android-ohjelmointi

Android on yleinen mobiililaitteiden käyttöjärjestelmä. Sitä käytetään puhelimissa, tableteissa ja niille tarkoitetuissa oheislaitteissa. Androidille tehdyt sovellukset on useimmiten tarkoitettu toimimaan jokaisella käyttöjärjestelmää käyttävällä laitteella. Android-sovellusta tehtäessä tulee perustason tuotesuunnittelun lisäksi ottaa huomioon erilaisten laitteiden määrä, jolla sovellus tulee toimimaan. Käyttäjä voisi käyttää samaa sovellusta

joko pienellä puhelimella tai suurikokoisella tabletilla, ja sovelluksen tulee näyttää molemmissa tapauksissa hyvältä.

Toinen huomioon otettava asia sovelluksen ohjelmoinnissa on seuraava kysymys: Minkä tason Android-laitetta käyttäjä käyttää? Käyttöjärjestelmällä on käytössä useita eri API-tasoja, eli kuinka edistyksellinen rajapinta laitteella on saatavilla. Rajapinnan tason ilmoittaa laitteen käyttämä Android-versio, johon linkittyvät numerokoodi ja nimi.

Vanhemmat laitteet käyttävät alhaisempaa API-tasoa, eikä niitä voi päivittää uusimpaan versioon laitekohtaisten rajoitusten vuoksi. Uudemmat laitteet käyttävät puolestaan korkeampaa, edistyksellisempää API-tasoa, ja näille laitteille voidaan ohjelmoida uusia ominaisuuksia. Laitekohtaisen API-tason voi saada selville sen asetuksista tarkistamalla, mikä versio Androidista laitteeseen on asennettu. Jokaiseen Android-versioon on linkitetty sen API-tasoa vastaava numero. Esimerkiksi laitteet, joihin on asennettu Android 8.0, käyttävät tason 26 APIa. [1.]

Sovellukselle on määritetty sen sisäisissä tiedostoissa väli, millä API-tasolla sovellusta voidaan käyttää. Alhaisin mahdollinen API-taso määrittelee sen, kuinka vanha laite voi käyttää sovellusta tai toisin sanoen, mikä Android-versio laitteessa on vähintään oltava, jotta sovellus toimisi. Alhaisin taso laajentaa sovelluksen mahdollista käyttäjäkuntaa ottamalla mukaan myös vanhempia laitteita käyttävät henkilöt, mutta mahdollisesti rajoittaa ominaisuuksia, joita sovellukselle voidaan ohjelmoida.

Android-ohjelmoinnissa kielinä käytetään Javaa tai Kotlinia. Tässä insinöörityössä käytetty kieli on Java-kieli.

Yleisesti Android-ohjelmaan kuuluu kolme pääosaa: koodi, joka sisältää sovelluksen kaikki käytännön osat: sen logiikan, ominaisuudet, toiminnot yms. Toinen osa on resurssit, joihin sisältyy sovelluksen ulkoasut ja niiden osat sekä koodin käyttämät ulkoasulliset osat kuten värit, tekstisisältö, tekstin tyylit ja dynaamiset ulkoasukomponentit. Nämä ulkoasutiedostot käyttävät kielenään XML-kieltä. Kolmas osa sisältää testit, joiden avulla testataan sovelluksen toimintaa ja ominaisuuksia.

Android-sovelluksen toiminnallisuuteen ja ulkonäköön vaikuttavat usein käyttäjän ja laitteen tiedot sijainnista. Jokainen laite sisältää lokaalin, johon sisältyy laitteen sijaintimaa,

aikavyöhyke, kieli ja sijainnin käyttämä kellonaikaformaatti. Sovellusta suunniteltaessa pitää ottaa huomioon nämä lokaalin tiedot, jotta käyttäjän kokemus sovelluksen käytöstä olisi mahdollisimman tuttu ja häntä varten lokalisoitu. Android-ohjelmoinnissa lokaalin tiedot on tallennettu Locale-nimiseen olioon. Ominaisuutta, mikä tarvitsee lokaalin sisältämää informaatiota, sanotaan lokaaliherkäksi ominaisuudeksi. Tällainen ominaisuus käyttää Locale-oliota räätälöimään sovelluksen sisältöä käyttäjää varten [2]. Esimerkiksi sovellus saattaa pitää sisällään aikaa näyttävän kellon. Tämän kellon sisällä kellonaika pitää olla lokalisoitu systeemin aikavyöhykkeen mukaisesti.

2.1 Ohjelmat ja komponentit

Tässä insinöörityössä Android-ohjelmointiin käytetään Android Studio -ohjelmointiympäristöä. Sovelluksen versio on 3.1+, mikä sallii eräiden edistyksellisten ominaisuuksien käytön. Android Studio on nimensä mukaisesti Android-ohjelmointiin tarkoitettu ohjelma. Sen avulla voi ohjelmoida, suunnitella ja rakentaa Android-sovelluksia.

Android Studiossa tärkeänä tukipylväänä toimii Gradle-rakennustyökalu. Gradle automatisoi sovelluksen käyttämien moduulien, rajapintojen ja kehitystyökalujen asennuksen ja integroinnin sovellukseen. Gradle helpottaa myös sovelluspaketin eli APK:n rakentamista tekemällä siitä käyttäjäystävällisen, automatisoidun prosessin. Rakennussysteemi valitsee automaattisesti kaikki sovelluksen lähdetiedostot sekä Java- että XML-puolelta ja yhdistää ne toisiinsa luoden sovelluksen lopullisen APK-tiedoston, jota käytetään mobiililaitteella [3, ensimmäinen vastaus].

Android Studioon kuuluu sisäänrakennettu Android-emulaattori. Emulaattorin tarkoitus on toimia mobiililaitteen täydellisenä virtuaalisena kopiona. Emulaattoriin ladataan laitekuva, joka sisältää kaiken tarvittavan tiedon simuloitavasta laitteesta. Laitekuva sisältää mobiililaitteen käyttämän Android-version sekä laitteen ulkoasun ja käyttöliittymän. Laitteen simuloinnin avulla voidaan testata sovelluksen toimintaa eri Android-version omaavissa laitteissa ilman, että täytyy omistaa fyysistä versiota laitteesta. Myös erikokoisia ruutuja voidaan testata tällä.

Android Studio helpottaa sovelluksen graafisen käyttöliittymän suunnittelua siihen sisäänrakennetun design-ominaisuuden avulla. Tavallisesti sovelluksen ulkoasut kirjoitetaan koodina xml-tiedostoon, mutta design-näkymän avulla ulkoasuun voidaan liittää suoraan luettelosta vetämällä ulkoasukomponentteja ja suunnitella niiden asetelmaa ja ulkonäköä. Ulkoasukomponentteja ovat mm. tekstikentät, luettelot ja näppäimet.

Android-ohjelmoinnin perusteena on, että ulkoasuun ei suoraan kirjoiteta tekstiä, vaan sovelluksen käyttämä teksti tuodaan rakennussysteemin avulla suoraan resurssikansiosta tiedostosta, joka sisältää kaiken sovelluksen käyttämän tekstin. Tekstit ovat riveinä ja jokaisella tekstinpätkällä on nimi, jonka avulla se tunnistetaan ja haetaan resursseista. Tämä tapa säilöä tekstiä mahdollistaa sovelluksen käännöksen eri kielille, ja Gradle automatisoi sen. Käytännössä sovelluksen jokaiselle lokaalille on tehty oma arvokansionsa, joka sisältää saman nimisen xml-tiedoston sovelluksen teksteille. Tähän tiedostoon kirjoitetaan kyseisen lokaalin käännetyt tekstit, jotka silti käyttävät niille tarkoitettua samaa ID:tä. APK:t rakennetaan siten, että riippuen laitteen sisälle määritetystä lokaalista tekstit automaattisesti käännetään sen alueen kieleen. Tämä riippuu tosin siitä, onko kyseiselle lokaalille kirjoitettu käännöstiedostoa. Mikäli käännöksiä puuttuu tai käännöstiedostoa tietylle lokaalille ei ole olemassa, sovellus näyttää tekstin ensisijaisella kielellä, joka on yleensä englanti.

2.2 APK ja Google Play

Kun Android-sovellusta ja sen käyttöä halutaan testata puhelimella tai emulaattorilla, Gradle rakentaa siitä APK:n eli Android Packagen. Se tiivistää kaiken ohjelman sisällön yhteen sovelluspakettiin, joka voidaan suorittaa Android-laitteella.

APK:n suoritukseen liittyy tiettyjä rajoituksia. Ensinnäkin mobiililaitteen Android-tason pitää olla sillä tasovälillä, mikä sovellukselle on määritelty (ks. kpl 2). Toinen rajoitus liittyy Gradle-skriptin sisälle määriteltyihin sovelluksen julkaisukoodiin ja -nimeen. Julkaisukoodi on numerosarja, joka auttaa sovelluksen version tunnistamisessa eikä ole näkyvissä käyttäjille. Jotta sovelluksen voi päivittää uudempaan versioon laitteessa, jossa sovellus on jo asennettuna, tulee sovelluskoodin numeron olla edellisen version numeroa korkeampi. Android tarkistaa koodin APK:ta asennettaessa. Mikäli koodinumero on alempi kuin edellinen versio, päivitys ei tule onnistumaan, vaan sovellus pitäisi poistaa

laitteelta kokonaan uudelleenasennusta varten. Jos koodinumero sen sijaan on sama kuin edellinen versio, paketin tarjoaja ei välttämättä tunnista tätä päivitykseksi eikä käyttäjälle anneta vaihtoehtoa päivittää sovellusta.

Google Play on sovelluspakettien yleisin välittäjä. Käyttäjät asentavat sovelluksensa Google Play -sovelluksen kautta, ja sovelluskehittäjät julkaisevat sovelluksensa sekä sen päivitykset tämän palvelun kautta käyttäjille. Sovelluksen näkyvyys, sen asennus ja päivitykset sekä sovelluksen sisäiset ostokset suoritetaan kaikki Playn kautta, minkä vuoksi on erittäin tärkeää, että sovellus on Google Playn sääntöjä noudattava.

Google Play on tarkkaan määritellyt säännöt ja ehdot, joilla sovellusta voidaan markkinoida käyttäjille. Sovelluksen sisällön ja sen sisäisten ostosten ja mainosten tulee olla näiden sääntöjen mukaisia. Joka kerta, kun uusi APK laitetaan Play-kauppaan, sen sisältö tarkistetaan ennen julkaisua. Julkaisu voi epäonnistua, ja se voidaan estää, mikäli siinä on ehtojen vastaista sisältöä.

Kun sovelluksesta valmistuu uusi versio, joka halutaan julkaista käyttäjille, siitä tehdään APK, jolla on edellistä versiota korkeampi sovelluskoodi ja nimi. APK ladataan Playn kehittäjäpalvelun kautta kauppaan, ja se käy tarkastuksen läpi. APK:ta ladattaessa sille kirjoitetaan päivityksen sisältö lyhyenä tekstinä. Tekstin täytyy sisältää päivitykseen liittyvät tiedot ja uudet ominaisuudet kaikilla niillä kielillä, joille sovellus on julkaistu. Vaihtoehtoisesti sovellusversiolle voi kirjoittaa erillisen nimen.

Playn kehittäjäpalvelu on nimeltään Developers Console, joka sisältää sovelluksesta kaiken tärkeän julkaisuihin liittyvän tiedon. Konsolin kautta pääsee tarkastelemaan sovelluksen sisäisiä ostotapahtumia, virhetapahtumia, kaatumisia, asennus- ja päivitystapahtumia, poistotapahtumia sekä käyttäjien kirjoittamia asiakaspalautteita. Konsolin kaatumisraportit ovat tärkeitä sovelluksen kehityksen kannalta, sillä sen avulla pystytään huomaamaan koodin sisäisiä mahdollisia virheitä, jotka aiheuttavat sovelluksen kaatumisen. Kun sovellus kaatuu, Google laatii siitä automaattisen virheraportin, joka lähetetään konsolille. Raportti sisältää kaatumisen aiheuttaneen virheen nimen, olennaiset rivitiedot sekä sen laitteen, jolla sovellus kaatui, Android API-tason ja laitteen nimen tiedot. Tietyt ongelmat esiintyvät vain joissakin Android-versioissa, minkä vuoksi on tärkeää sovellusta kehittäessä ottaa huomioon ja testata kaikki mahdolliset versiot.

2.3 Tilaajayrityksen sovellus

Insinöörityössä uudistettu sovellus on Google Playn kautta ladattavissa oleva mobiilisovellus, joka toimii käyttäjän kuvallisena ruokapäiväkirjana. Sovelluksessa käyttäjällä on käytössään päivittäinen kalenteri, johon hän voi ottaa kuvia päivän aikana syömistään aterioista niitä vastaaviin ruutuihin. Jokaisella aterialla on oma ruutunsa, ja ruutuja on yhdelle päivälle 6 kappaletta.

Sovellus lähettää tiettyinä ajankohtina käyttäjälle ilmoituksen puhelimeen muistuttamaan siitä, että on aika syödä ateria ja ottaa siitä kuva. Käyttäjä saa vapaasti muokata muistutusten ajankohtia ja halutessaan laittaa niitä pois päältä.

Sovelluksessa voi katsoa vinkkejä liittyen terveelliseen ruokavalioon, ja se lähestyy syömistä filosofisemmalta kannalta eroten näin muista ruokavaliosovelluksista, jotka painottavat kalorien laskemiseen.

Sovelluksen voi sen sisällä päivittää Premium-versioon sovelluksen sisäisten ostosten avulla. Premium-versio sovelluksesta sisältää lisää toimintoja, enemmän historiaa ja tavan kirjoittaa lisää muistiinpanoja aterioihin liittyen.

Sovelluksen versiosta riippumatta käyttäjän tekemät merkinnät, kuten hänen ottamiensa valokuvien tiedostopolut ja aterioiden muistiinpanot, tallennetaan puhelimen sisäiseen muistiin käyttäen SQLite-tietokantamallia. Backend-palvelimen puuttuessa käyttäjä menettää datansa, mikäli hän poistaa sovelluksen tai vaihtaa laitetta. Tämän vuoksi kehityksen alla on ollut Backend-palvelin, joka tallentaa jokaisen käyttäjän tiedot ja datan pilveen henkilökohtaisen käyttäjätunnuksen nimiin, turvaten näin tiedon säilymisen.

2.4 SQLite-tietokanta

SQLite on avoimen lähteen SQL-tietokanta. Se tallentaa sovelluksen dataa laitteen sisäiseen tekstitiedostoon. Android sisältää oletuksena SQLite-implementaation. SQLite tukee kaikkia relaatiotietokantojen toimintoja, ja yhteys tietokantaan luodaan automaattisesti. [4.]

Sovellus käyttää nykyisessä julkaisussaan perinteistä SQLite-mallia, joka on tietyillä tavoilla vanhentunut. Yksi insinöörityön päätavoitteista on tämän tietokannan päivittäminen uuteen versioon.

Perinteinen SQLite-tietokanta on tasoltaan alhainen, ja sen käyttö vaatii paljon työtä ja aikaa. Sovelluksessa käytetään paljon raakoja SQL-kyselyitä tietokantaoperaatioiden tekemiseen. Tietokantamallia muutettaessa kaikki nämä kyselyt pitäisi käydä erikseen läpi, ja siksi tämänlainen kovakoodaus tekee tietokannan uudistamisesta erittäin työlästä ja virhealtista. SQL-tiedon muuttaminen sovelluksen lähdekoodissa käytettäväksi olioksi vaatii paljon koodia. Kehittäjät nykypäivänä suosittelevat voimakkaasti Room-tietokantamallin käyttöä, johon tässä insinöörityössä siirrytään. [5.]

2.5 Roomista yleisesti

Room on yksi Androidin arkkitehtuurikomponenteista, jotka on suunniteltu helpottamaan Android-ohjelmointia monesta eri näkökulmasta. Room on ORM eli Object Relational Mapping -kirjasto. Se siis luo mallin muuttaakseen tietokantaoliot Java-olioiksi. Room luo abstraktin tason SQLite-tietokannan ja sovelluksen koodin välille, mikä mahdollistaa sujuvan pääsyn tietokantaan.

Room mahdollistaa ajonaikaisen varmistuksen tietokantakyselyiden toiminnasta, ja helpottaa tietokannan rakenteen muuttamista ja testaamista huomattavasti. Tämän kirjaston avulla ei tarvitse kirjoittaa satoja rivejä raakaa koodia, jotta SQLite-tietokantaoliot voitaisiin muuttaa Java-olioiksi. Room on myös suunniteltu toimimaan yhdessä LiveDatan kanssa, mikä mahdollistaa yksinkertaisen dynaamisen käyttöliittymäpäivityksen tietokantamuutosten tapahtuessa. SQLite-mallin kanssa käyttöliittymäpäivitykset piti tehdä monimutkaisten kuuntelija-kutsujen avulla, jotka myös sotkivat sovelluksen arkkitehtuuria huomattavasti.

Jatkon kannalta on erittäin tärkeää, että sovellus jatkaa Room-tietokannan käyttämistä. Sovelluksen tulevaisuudessa sille voidaan tehdä muutoksia ja päivityksiä, jotka vaativat tietokannan rakenteen muuttamista. Esimerkiksi backend-integraatio voi vaatia tätä. Room helpottaa näiden muutosten tekemistä niin paljon, että perinteiseen SQLite-kyselymalliin ei kannata enää palata.

3 Toteutus

Työn toteutus alkoi tietokantamigraatiosta. Sovelluksessa, jonka tietokantarakenne ja tarvittavat muutokset ovat monimutkaisia, tuli migraation sisältää monta vaihetta. Migraation onnistuttua piti vielä tehdä tarvittavat käyttöliittymäpäivitykset, eli käytännössä koko sovellus uudistettiin käyttämään Androidin arkkitehtuurikomponentteja.

3.1 Tietokantarakenteet ennen ja jälkeen migraation

Sovelluksen alkuperäinen tietokanta on koodattu arkaaisella tavalla ja monet sovelluksen osat, jotka voisivat käyttää tietokantaa, käyttävät sen sijaan muita tiedon tallennusmenetelmiä. Esimerkiksi ateriataululle kuuluva kolumni, johon tallennetaan sovelluksen premium-versioon kuuluva toinen kuvateksti, tallennetaan samaan kolumniin kuin tavallinen kuvateksti (esimerkkikoodi1).

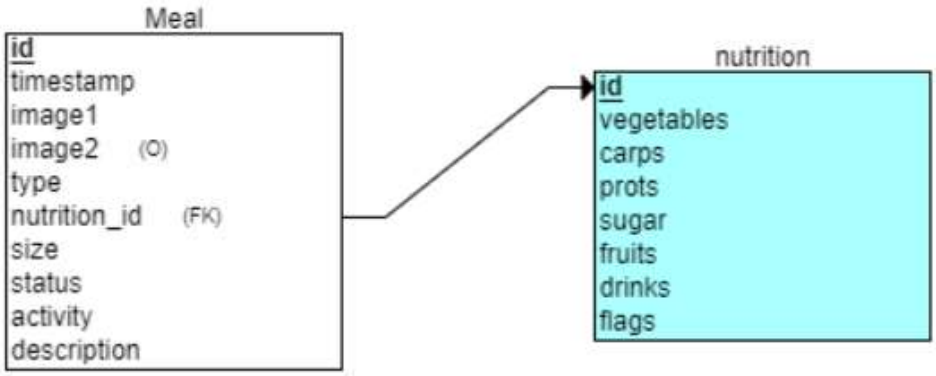
```
// We save premium notes in the same column as description.
// How we do this is, after description, we append this 'breakpoint' string
and then append the premium notes.
// DO NOT CHANGE THIS or user's saved descriptions and premium notes will
break :)
private final static String DESCRIPTION_PREMIUM_NOTES_BREAK = "tDzvEqcv";
```

Esimerkkikoodi 1. Alkuperäisestä koodista peräisin oleva selitys premium-muistiinpanojen tallennukselle.

Käyttäjän omat, henkilökohtaiset tiedot tallennetaan puhelimeen käyttämällä SharedPreferences-kirjastoa. SharedPreferences on laitteen sisäänrakennettu tapa tallentaa avain-arvopareja, ja jokaiselle sovellukselle on oma näille tiedoille tarkoitettu kansio. Jokaisella tietorivillä on avain, jota käytetään tiedon hakemiseen ja tallentamiseen. Yhtä avainta kohti voi olla vain yksi arvo, mutta tätä arvoa voi muuttaa vapaasti. [6.]

SharedPreferences on hyvä käyttää esimerkiksi silloin, kun tallennetaan tietoa, jossa on varmasti vain yksi arvo. Käyttäjä voi esimerkiksi sovelluksessa määrittää oman painonsa ja pituutensa, määrittää sen, haluaako hän kuulla ääniä muistutuksista ja mikä ääni siitä kuuluu. Nämä arvot tallennetaan joko tekstinä, numeroina tai boolean-arvoina.

Se, että käyttäjän antamat asetukset ja käyttäjän henkilökohtaiset tiedot tallennetaan SharedPreferences-kirjastoon tietokannan ulkopuolelle, on periaatteessa harmitonta, sillä käyttäjän tietoja ei yhdistetä tietokannan tauluihin mitenkään. Kuitenkin, kun backend integroidaan sovellukseen, voi olla, että taulu profiilitiedoille joudutaan tekemään.



Kuva 1. Tietokannan rakenne ennen migraatiota. Relaatiotietokantakaavio.

Kuten kuvassa 1 näkyy, alkuperäinen tietokantarakenne oli hyvinkin yksinkertainen sen sisältäessä vain kaksi taulua useilla riveillä. Nutrition- eli ravinnetaulun id tallennetaan ateriatauluun ulkoisena avaimena. Ateriataulun ja ravinnetaulun välillä on yksi-yhteen-suhde, eli yhtä ateriaa kohti on yksi ravinne. Aterialla on pakko olla ravinne, eikä ravinteita ole olemassa ilman ateriaa. Jos ateria poistetaan, ravinne poistuu sen mukana.

Ateriataulussa oleva kolumni, ”image2” on täysin käyttämätön kolumni. Se ohjelmoitiin alun perin tietokantaan sillä ajatuksella, että yhdelle aterialle voisi lisätä kaksi kuvaa. Myöhemmin sovelluksen rakennetta muutettiin kuitenkin siten, että kahden kuvaan sijaan yhdessä painikkeessa olisi kaksi erillistä ateriaoliota, toisin sanoen kaksi objektia erillisillä ID:llä. SQLite-tietokannan teknisten vaikeuksien takia tätä kolumnia ei pystytty poistamaan. Se, että premium-käyttäjien muistiinpanot tallennettiin samaan kolumniin kuin kuvan teksti, oli tarkoitettu väliaikaiseksi korjaukseksi sille, että uutta kolumnia kovakoodattuun tietokantaan ei pystytty lisäämään. SQLite-tyylinen migraatio olisi tässä vaiheessa ollut erittäin työlästä ja vienyt liikaa aikaa.

Kirjoittaja on tietoinen ravinnetaulussa olevasta kirjoitusvirheestä ja on ottanut tämän huomioon migraatiota tehdessään.

Uusien ominaisuuksien ohjelmointia varten sekä sovelluksen tulevaisuutta ajatellen sovellukselle oli rakennettava uusi tietokanta, jonka migraatio olisi mahdollisimman mutkatonta. Tietokantarakennetta pitäisi pystyä muuttamaan tarpeen vaatiessa. Ratkaisu tähän oli siirtyä käyttämään Googlen tarjoamaa Room-tietokantamallia, joka luo kehyksen SQLite-tietokannalle ja tekee tietokantaoperaatioista automatisoituja ja yksinkertaisempia.

3.2 Room ja migraatio

Room on yksi Androidille tehdyistä arkkitehtuurikomponenteista. Se tarjoaa abstraktin kehyksen SQLitelle, mikä mahdollistaa vahvemman pääsyn tietokantaan käyttäen SQLiten koko potentiaalia. Migraatiota helpottamaan Room tarjoaa Migration-luokan. Migration-luokalle kirjoitetaan tarvittavat toiminnot, joita tietokannalle pitää tehdä siirryttäessä yhdestä versiosta toiseen, kuten kolumnien lisäykset, nimien muutokset jne [7]. Kuten SQLite, sen tarkoitus on toimia sovelluksen tietokantana ja mahdollistaa tiedon tallennuksen sovellukselle ilman internetyhteyttä.

Roomin avulla migraatio toimii seuraavalla tavalla: tietokannan versio korotetaan, mahdolliset migraatiolausekkeet kirjoitetaan, uuden tietokannan entiteetit määritetään. Tietokantarakennetta muutettaessa tietokannan versionumero pitää kohottaa tai muuten sovellus kaatuu, samoin kuten tarvittava migraatiokoodi tulee tarjota Roomille silloin, kun versionumeroa kohotetaan. Riskinä on käyttäjän datan menettäminen, jos migraatio ei onnistu oikealla tavalla [7].

Kun tietokantaan otetaan ensimmäistä kertaa yhteys, Room-tietokanta rakentuu. Room luo sitten automaattisesti implementaation SQLiteOpenHelper-luokasta, jonka onUpgrade()-metodia se kutsuu laukaistakseen migraation. Tarvittavat operaatiot ja tiedonsiirrot suoritetaan. Tämän jälkeen tietokanta avataan käsittelyä varten. Migraation sisällä Room luo tietokannalle uniikin Hash-stringin eli avaimen tunnistamaan tietokan-

nan version. Room myös tunnistaa, jos rakennetta on yritetty muuttaa ilman version numeron muuttamista. Room käyttää json-muotoisia skeematiedostoja vertaillakseen kahden eri tietokantaversion rakennetta keskenään.

Yksinkertaisimmillaan migraatio SQLite APIsta Roomiin suoritetaan seuraavalla tavalla:

Gradleen lisätään tarvittavat kirjastot, jotka haetaan Googlen Maven-repositorion kautta (esimerkkikoodi 2). Samalla määritellään se Roomin versionumero, jota halutaan käyttää. Gradlelle myös määritetään tiedostosijainti, mihin se luo tietokantarakenteiden JSON-skeemat (esimerkkikoodi 3).

```
dependencies{
 …
implementation
   “android.arch.persistence.room:runtime:$rootProject.roomVersion”
annotationProcessor
   “android.arch.persistence.room:compiler:$rootProject.roomVersion”
androidTestImplementation
   “android.arch.persistence.room:testing:$rootProject.roomVersion”
}
```

Esimerkkikoodi 2. Tarvittavien kirjastojen lisääminen koodin dependensseihin.

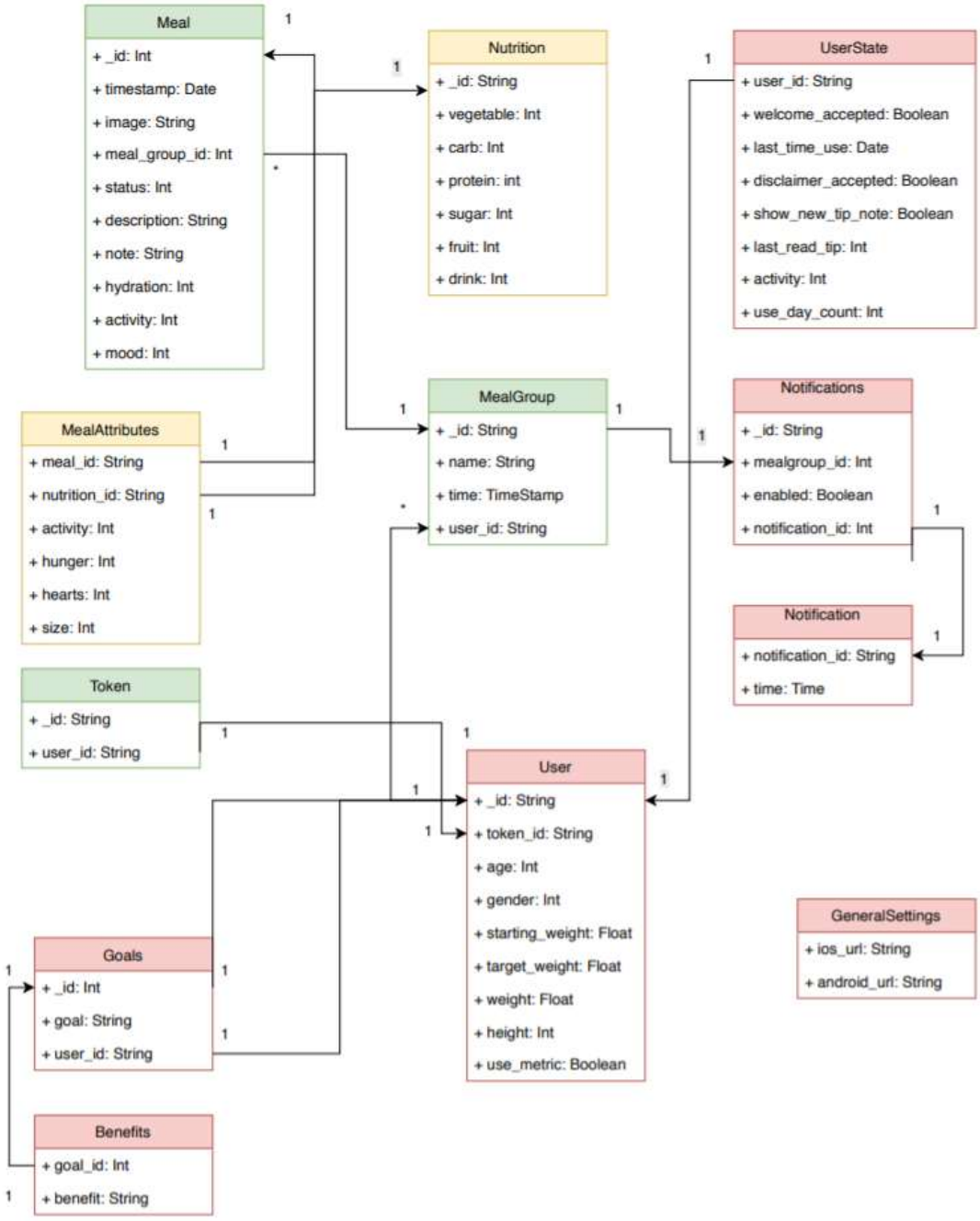
```
android {
    defaultConfig {
        ...
       // used by Room, to test migrations
        javaCompileOptions {
            annotationProcessorOptions {
                arguments = ["room.schemaLocation":
                                "$projectDir/schemas".toString()]
            }
        }
    }

    // used by Room, to test migrations
    sourceSets {
        androidTest.assets.srcDirs +=
                            files("$projectDir/schemas".toString())
    }
...
```

Esimerkkikoodi 3. Määritellään tiedostosijainnit json-skeemoja varten [9].

Kun sovelluksen build.gradle -tiedostoon on lisätty nämä tarvittavat koodit, Gradle automaattisesti tuo tarvittavat moduulit ja niiden koodit Googlen Maven-repositoriosta ja laittaa ne valmiiksi käyttöä varten. Tämän jälkeen voidaan siirtyä seuraavaan vaiheeseen, jossa aiemmin olemassa olleet Model-luokat päivitetään Roomin Entity-luokiksi.

Jotta uudet entiteettiluokat voitiin tässä insinöörityössä kirjoittaa, piti ensin selvittää rakenne, mihin tietokanta haluttiin muuttaa. Tulevaa tietokantarakennetta käsiteltiin Metropolian innovaatioprojektissa, jonka tekivät syksyllä 2018 Saini Patala, Aleksi Kesälahti, Tuomas Koivisto ja Otso Pohjola. Innovaatioprojektissa suunniteltiin sovellukselle backend-tietokanta, jossa sovelluksen tallennettavat tiedot tallennettaisiin automaattisesti pilvessä sijaitsevaan palvelimeen. Backend-tietokantamallin suunnitteli tilaajayritykseltä Sami Repo.

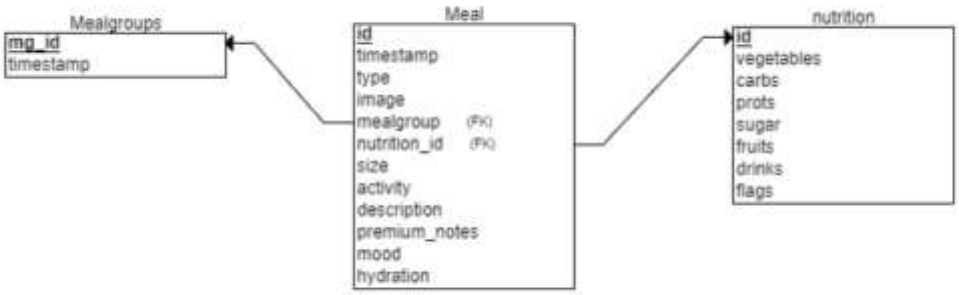


Kuva 2. Innovaatioprojektissa suunniteltu backend-tietokannan relaatiotietokantakaavio.

Backend-tietokanta on rakenteeltaan erilainen kuin sovelluksen sisälle suunniteltu lopullinen tietokanta. Tämän lisäksi osa kaavion sisällöstä on virheellistä. Esimerkiksi MealAttributes-taulu on välitaulu, jonka tiedot voidaan ihan hyvin tallentaa Meal-tauluun.

Suurin osa kuvan 2 tauluista on jätetty pois Room-tietokannasta siitä syystä, että ne on tallennettu sovelluksen SharedPreferenceihin. Taulut ovat näkyvillä kuvan 3 kaaviossa ja integroituina backendiin, jotta tiedot pystytään siirtämään SharedPreferenceistä sovelluksen palvelimelle. Jotta palvelimelta voidaan hakea oikean henkilön tiedot, on MealGroup-tauluun lisätty ulkoinen avain merkitsemään käyttäjän ID:tä. Tämä ulkoinen avain on kuitenkin yksinkertaisuuden vuoksi jätetty pois Room-tietokannasta, mutta tarvittaessa tämä muutos on helppoa tehdä tulevaisuudessa.

Lopullinen Room-tietokantamalli suunniteltiin ottaen huomioon sovelluksen toiminnan ja käyttötarpeen sekä mahdolliset tulevaisuuden suunnitelmat. Käytetty malli näytetään kuvassa 3, ja se on piirretty ERDplus-työkalulla, joka löytyy selaimesta osoitteessa https://erdplus.com/. Mikäli muutoksia tietokannan rakenteeseen halutaan tehdä, tulee niitä varten vain muokata Entity-luokkia, kirjoittaa uusi Migraatio-objekti Room-luokalle ja korottaa tietokannan versionumeroa.



Kuva 3. Relaatiotietokantakaavio, joka implementoitiin insinöörityössä tietokantaan.

Kuten kuvassa 3 näkyy, tietokanta on malliltaan melko yksinkertainen. Kaavioon ei liity Day- ja Week-objektit, joita käsitellään myöhemmin tässä dokumentissa.

Ateriataulusta on poistettu kolumnit image2 ja status. Aterian status merkitsi aiemmin sitä, onko ateria asetettu ruokailulle vai ei ja onko ateria ohitettu. Uudessa versiossa status määräytyy sillä, onko ateriaryhmään luotu vielä ateriaobjekteja, joten statuskolumnia ei enää tarvita. Toiselle kuvalle tarkoitettu kolumni on aina ollut ylimääräinen eikä sitä ole syytä säilyttää.

Uudet lisätyt kolumnit ovat mood ja hydration eli mieliala ja nesteytys. Nämä on tarkoitettu lisättävää ominaisuutta varten, jossa käyttäjä voi merkitä aterian yhteyteen ruokailun aikana juodut vedet sekä hänen mielialansa.

Ravinnetaulusta on korjattu siinä ollut kirjoitusvirhe, mutta rakenteeltaan se on samanlainen kuin aikaisemmin.

Ateriaryhmän taulussa sillä on ID ja aikaleima. Aikaleiman käyttö osoittautui migraatiossa ja sovelluksen toiminnassa ongelmalliseksi, sillä ateriaryhmien paikat määritetään nimenomaan aterian tyyppien mukaisesti. Aikaleima pitää koodissa konvertoida ateriatyypiksi käyttäen tyyppien oletusarvoja, mutta tämä ratkaisu voi aiheuttaa ongelmia etenkin, jos käyttäjä muokkaa voimakkaasti ateriatyyppien aikaleimoja. Tämän vuoksi on erittäin suositeltavaa, että jatkossa taulun rakennetta muutetaan siten, että sille määritetään aterian tyyppi eikä aikaleima.

Ateriataulussa ovat ulkoiset avaimet mealgroup ja nutrition_id, jotka liitetään niitä vastaavien taulujen id-kolumneihin. Aterian ja ateriaryhmän välillä on yksi-moneen suhde, jossa yhteen ateriaryhmään voi kuulua useampi ateria. Aterian ja ravinteen välillä on yksi-yhteen-suhde, jossa vain yksi ravinne voi olla olemassa yhtä ateriaa kohti. Ravinneobjekteja ei ole olemassa ilman niille kuuluvaa ateriaa.

3.2.1 Entity-luokat

Kuten Florina Muntenescu kirjoittaa artikkelissaan ”Incrementally migrate from SQLite to Room” [10], Room-migraatio kannattaa monimutkaisen datan kohdalla aloittaa yksinkertaisimmasta päästä, eli vanhassa tietokannassa olevat Model-luokat päivitetään Roomin Entity-luokiksi. Entity on luokka, joka määrittää tietokantataulun rakenteen ja sisällön. Siihen määritetään annotaatioiden avulla taulun kolumnit ja niiden datatyypit, taulun ulkoiset avaimet sekä taulun indeksit ja niiden tyypit.

Uusi Entity-luokka määritetään koodiin kirjoittamalla luokan nimen yläpuolelle annotaatio, joka määrittää sen taulukokonaisuudeksi. Annotaatiot merkitään @-merkillä niitä koskevien rivien yläpuolelle.

```
@Entity(tableName = "mealgroups")
public class MealGroup {
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = ShyeDatabase.MEALGROUP_KEY_ID)
    public long id;

    @ColumnInfo(name = ShyeDatabase.MEALGROUP_KEY_TIMESTAMP)
    public long mTimeStamp; // SQL DATE takes long and getTime() returns long

    public MealGroup(){

    }
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }

    public long getTimeStamp() {
        return mTimeStamp;
    }
    public void setTimeStamp(long mTimeStamp) {
        this.mTimeStamp = mTimeStamp;
    }

}
```

Esimerkkikoodi 4. Ateriaryhmätaulun entiteettiluokka.

Koodiesimerkissä 4 näkyy yksinkertainen entiteettiluokka, joka määrittää ateriaryhmien taulun rakenteen. Ylimpänä on määritelty @Entity-annotaatiossa taulun nimi, jonka jälkeen nimetään luokka. Luokan sisällä on julkisia muuttujia, jotka toimivat taulun kolumnien tunnisteina. Jokaiselle taulun kolumnille määritellään muuttuja. Kolumnin nimi ja informaatio merkitään annotaatioiden avulla muuttujien yläpuolelle. Kolumnin datatyyppi määräytyy muuttujan tyypin mukaisesti. Taulun pääavaimen eli koodiesimerkin 3 tapauksessa ateriaryhmän ID:n annotaatio @PrimaryKey(autoGenerate = true) kertoo Roomille tämän muuttujan toimivan taulun pääavaimena ja että sen arvo generoidaan automaattisesti. Automaattinen ID-generointi luo ID:n jokaiselle uudelle lisäykselle tietokantaan.

Meal- eli ateriataulun entiteetti sisältää yläviitteissään tiedot ulkoisista avaimista ja taulun indekseistä.

```
@Entity(tableName = "meals",
            indices = {@Index(value="mealgroup"),
            @Index(value="nutrition_id", unique=true)},
            foreignKeys = {@ForeignKey(entity = MealGroup.class,
            parentColumns = "mg_id",
            childColumns = "mealgroup"),
            @ForeignKey(entity=Nutrition.class,
            parentColumns="id",
```

```
            childColumns="nutrition_id",
            onDelete = CASCADE)
})
```

Esimerkkikoodi 5. Annotaatiot ateriataulun entiteettiluokalle.

Kuten koodissa 5 näkyy, tarvittava tieto ateriataulun rakenteesta ja sen yhteyksistä muihin tauluihin merkitään yläosan annotaatiossa. @Index kertoo taulun indeksien nimet ja sen, mihin kolumneihin ne liittyvät. Indeksit ovat tärkeitä Room-tietokannoissa silloin, kun taulujen välillä on yhteyksiä, ja ne helpottavat hakuoperaatioiden suorittamista.

Ravinnekolumniin viittaavaan indeksiin on merkitty unique=true, mikä tarkoittaa sitä, että jokaista ateriaa kohti ravinteen tunnisteen pitää olla uniikki. Tämä vahvistaa sitä, että näiden taulujen välillä on yksi-yhteen-suhde. Tätä merkintää ei ole lisätty ateriaryhmän kolumnin indeksiin, sillä useampi ateria voi liittyä samaan ateriaryhmään.

Ulkoiset avaimet on myös merkitty entiteetin annotaatioihin, kuten koodissa 4 näkyy. Ulkoisen avaimen merkintään kirjoitetaan yhdistettävän taulun nimi eli sen entiteetin luokan id, esim. Nutrition.class. Luokan id jälkeen kirjoitetaan haettavan kolumnin nimi ja sitä vastaavan kolumnin nimi ulkoisen avaimen omaavassa luokassa. OnDelete-operaatio on valinnainen lisäys merkintöihin ja ilmoittaa operaatiosta, joka suoritetaan silloin, kun ulkoisen avaimen omaavasta taulusta poistetaan merkintä. Tässä tapauksessa onDelete=CASCADE merkitsee sitä, että kun ateriataulusta poistetaan ateria, ravinnetaulusta poistetaan automaattisesti sille aterialle kuulunut ravinne. Ateriaryhmän ulkoiseen avaimeen ei tätä operaatiota ole merkitty, sillä se ateriaryhmä, josta ateria poistetaan, halutaan säilyttää. [8.]

Entity-luokat ovat annotaatioiden ulkopuolella samantyyppisiä kuin SQLiten model- eli malliluokat. Niissä on setterit ja getterit jokaista taulun kolumnia vastaaville muuttujille. Julkiset konstruktorit ovat suurimmalta osalta tyhjiä, mutta jos taulussa on pakollisia arvoja, ne ovat parametreina konstruktorissa (ks. koodi 6).

```
public Meal() {


}
public Meal(long timestamp, long nutritionId, long mealGroupId) {
            this.timeStamp = timestamp;
            this.nutritionId=nutritionId;
            this.mealGroupId = mealGroupId;
```

```
}
```

Esimerkkikoodi 6. Ateriataulun julkiset konstruktorit.

Entiteettiluokkaan voi myös kirjoittaa julkisia metodeja tiettyjä operaatioita, kuten lokalisaatiota varten. Esimerkiksi aterialuokkaan on kirjoitettu metodi tuomaan ateriatyypin nimen ID sitä varten, että nimen voi hakea sovelluksen resursseista lokalisoituna.

Vanhassa tietokannassa oli useita kolumneja, joihin oli määritetty oletusarvoksi null eli arvo, mitä ei ole olemassa. Roomin annotaatioihin ei voi merkitä null-oletusarvoa, vaan ne muuttujat, joiden tyyppi on int tai long, ovat automaattisesti datatyypiltään NOT NULL. Ne eivät siis salli null-arvoja ollenkaan.

Datatyypin ero migraatiokoodin ja entiteettiluokan välillä on otettava huomioon, sillä muuten migraatio ei onnistu ja ohjelma kaatuu. Kun Room yrittää verrata migraation käyneen tietokannan ja olemassa olevan tietokannan JSON-skeemoja, se huomaa eron jokaisen kolumnin datatyypeissä. Alkuperäisessä tietokantaskeemassa null-arvot ovat sallittuja, mutta entiteettiluokassa eivät.

Migraatiota kirjoittaessa tämä koitui ongelmaksi, sillä yleisesti Java-kielessä int- ja long-tyyppiset muuttujat eivät voi olla arvoltaan null. Ongelma korjautui siten, että kaikki kolumnit, joiden muuttujat olivat alun perin tyypiltään int tai long, muutettiin Integer- tai Long-objekteiksi. Jos muuttuja on tyypiltään numero-objekti eikä varsinaisesti numero, se sallii myös null-arvon. Ainoat pakolliset arvot on listattu luokkien julkisissa konstruktoreissa.

3.2.2 Converter- eli muuntajaluokka

Joskus entiteetin kolumnin arvo on tyypiltään muuta kuin numero tai teksti. Esimerkiksi ateriataululla on aterian tyypille kuuluva kolumni, joka on tyypiltään MealType. MealType on enum-luokka, johon kuuluu 6 arvoa, eli kaikki ateriatyypit. Enum-objektilla on koodi ja ID, joka kirjoitetaan tekstinä. Koodissa 7 näkyy tämä luokka kokonaisena.

```
public enum MealType {
            BREAKFAST(0), MORNING_SNACK(1), LUNCH(2), SNACK(3), DINNER(4),
            EVENING_SNACK(5);
```

```
            private int code;


            MealType(int code) {
                        this.code = code;
            }


            public static MealType getById(int code){
                        for (MealType e : values()){
                        if(e.code == code){
                                    return e;
                        }
                        }
                        return BREAKFAST;
            }


            public int getCode() {
                        return code;
            }

}
```

Esimerkkikoodi 7. Enum-tyyppinen luokka aterian tyypille.

Lyhyessä luokassa on konstruktori ja metodit hakemaan joko tyypin koodi tai ID.

Mikäli entiteetin arvo on tyypiltään enum-objekti, Room ei osaa itsestään tulkita sitä ja muuttaa sitä numeeriseksi arvoksi. Tätä varten on kirjoitettava TypeConverter-luokka, joka sisältää metodit tämän tyyppimuunnoksen tekoa varten. TypeConverter merkitään kolumnin määrityksen annotaatioihin, jotta Room osaa hakea metodit oikeasta luokasta.

```
@ColumnInfo(name = ShyeDatabase.MEAL_KEY_TYPE)
@TypeConverters(Converters.class)
public MealType type;
```

Esimerkkikoodi 8. Kolumni, jonka arvona on enum-tyyppinen objekti, ja muuntajaluokan ilmoittaminen.

Converters.class on tässä työssä se luokka, jonka sisällä muunnosmetodit on. Muunnosmetodit ovat yksinkertaisia operaatioita muuntamaan enum-objektin ID:n sitä vastaavaksi numeroksi ja toisin päin.

```
@TypeConverter
public static MealType toMealType(int mealType) {
            if (mealType == BREAKFAST.getCode()) {
                        return BREAKFAST;
            } else if (mealType == MORNING_SNACK.getCode()) {
```

```
                    return MORNING_SNACK;
            } else if (mealType == LUNCH.getCode()) {
                    return LUNCH;
            } else if (mealType == SNACK.getCode()) {
                    return SNACK;
            } else if (mealType == DINNER.getCode()) {
                    return DINNER;
            } else if (mealType == EVENING_SNACK.getCode()) {
                    return EVENING_SNACK;
            } else {
                    throw new IllegalArgumentException("Could not recog-
            nize Meal Type");
            }
}
@TypeConverter
public static Integer toInteger(MealType mealType) {
            return mealType.getCode();
}
```

Esimerkkikoodi 9. Muunnosmetodit aterian tyypille.

Esimerkkikoodissa 9 on metodit aterian tyypin muuntamiseksi numeroksi ja toisin päin. Metodien parametreinä on joko tyypin koodi tai ID. Metodin yläpuolelle on kirjoitettu annotaatio @TypeConverter, joka merkitsee tämän koodin juuri nimenomaan Roomin muuntajaksi.

3.2.3 DAO-luokat

Alkuperäinen sovelluksen koodi käytti kursoreita ja raakoja SQLite-kyselyitä tehdäkseen tietokantaoperaatioita. Yksinkertaisemmatkin tietokantalisäykset ja -haut olivat metodeja, jotka sisälsivät kymmeniä rivejä koodia. Niitä käyttäessä huomasi, miten vaikealukuista ja raskasta raakojen kyselyiden rakentaminen ja käyttö on.

```
    private Meal[] getMealsBefore(long timestamp) {
        Cursor cursor = mDb.rawQuery(mDataBaseModel.MealsBeforeSQL(timestamp),
new String[0]);
        Meal[] result = new Meal[cursor.getCount()];
        try {
            if (cursor.moveToFirst()) {
                do {
                    result[cursor.getPosition()] = Meal.fromCursor(cursor);
                }
                while (cursor.moveToNext() && !cursor.isAfterLast());
            }
        }
        catch(RuntimeException e) {
            e.printStackTrace();
        }
        finally {
            // Make sure to close the cursor
            cursor.close();
        }
```

```
        return result;
    }
```

Esimerkkikoodi 10. DataController-luokassa ollut metodi, joka haki ateriat ennen tiettyä aikaleimaa.

Esimerkkikoodissa 10 on yksinkertainen raaka SQLite-kysely, joka käyttää alkuperäisen tietokannan sisäistä metodia hakemaan tietoa tietokannasta. Tämä sisäinen metodi oli kovakoodattu yhdistämään parametreja valmiiseen SQLite-kyselytekstiin. Koodiesimerkissä kursori avataan, raaka kysely tehdään ja kursori käydään läpi silmukassa niin monta kertaa, kunnes viimeinen tulos on lisätty haluttuun listaan. Siinä on varauduttu RunTimeExceptioniin siltä varalta, että kysely tai kursorin lukeminen on epäonnistunut. Lopulta kursori suljetaan ja täytetty lista palautetaan.

Room on tuonut esille uuden tavan tehdä tietokantakyselyitä yksinkertaisesti ja kevyellä koodilla. Jokaiselle tietokannan taululle kirjoitetaan DAO- eli Data Access Object-luokka, joka sisältää kaikki metodit suoraa tietokantakyselyä varten. Valmiina olemassa olevien merkintöjen avulla lisäys, muokkaus ja poisto tietokannasta voidaan tehdä kahdella rivillä koodia ilman, että SQLite-kyselyä itseään tarvitsee kirjoittaa. DAO on abstrakti luokka, joka voi sisältää sekä abstrakteja metodeja ilman runkoa tai tavallisia, rungollisia metodeja.

```
@Dao
public abstract class MealDao {


@Insert(onConflict = OnConflictStrategy.REPLACE)
public abstract void insertMeal(Meal meal);


@Insert(onConflict = OnConflictStrategy.REPLACE)
public abstract long insertMealGetId(Meal meal);


@Update(onConflict = OnConflictStrategy.REPLACE)
public abstract void updateMeal(Meal meal);


@Delete
public abstract void deleteMeal(Meal meal);


public long insertMealIntoGroup(Meal meal, long group){
            meal.setMealGroupId(group);
            return insertMealGetId(meal);
}
```

Esimerkkikoodi 11. Aterian DAO-luokan lisäys- ja poisto-operaatiot.

Esimerkkikoodi 11 sisältää ateriataululle kirjoitetut lisäys- ja poisto-operaatiot. Abstraktien metodien yläpuolelle on kirjoitettu merkinnät, jotka ilmoittavat operaation tarkoituksen ja strategian, mitä Room käyttää konfliktin sattuessa. Metodi insertMealIntoGroup asettaa aterialle sen ateriaryhmän ID:n, johon se kuuluu ja käy sitten läpi abstraktin metodin insertMealGetId(meal) palauttaen juuri tietokantaan lisätyn aterian ID:n.

Metodin tyyppi ilmoittaa sen, mitä operaatio palauttaa. Esimerkiksi uutta ateriaobjektia lisätessä tietokantaan sen ID ei ole heti saatavilla, jos palauttaa lisätyn aterian sellaisenaan. Lisätyn aterian ID:tä voidaan tarvita muita tietokantaoperaatioita varten. Tätä varten insertMealGetId()-metodi sekä lisää objektin tietokantaan että palauttaa sille luodun ID:n koodille.

```
@Query ("SELECT * FROM meals WHERE id=:id")
public abstract Meal getMeal(long id);


@Query("SELECT id FROM meals WHERE mealgroup= :id")
public abstract List<Long> getMealsFromGroup(long id);


@Query("SELECT COUNT(*) FROM meals WHERE mealgroup= :id")
public abstract LiveData<List<Integer>> getMealsLiveFromGroup(long id);
```

Esimerkkikoodi 12. Ateriataulun hakuoperaatiot.

Esimerkkikoodissa 12 näytetään ateriataulun yksinkertaisimmat hakukyselyt. Nämä on alustettu merkinnällä @Query ja siihen kirjoitetaan se SQLite-lause, joka halutaan suorittaa. Koodin kolmas metodi palauttaa LiveData-objektin, johon palataan myöhemmin tässä raportissa.

DAO-luokan metodeja ei käytetä suoraan koodissa, vaan niiden käyttöä varten on kirjoitettu repositoriona toimiva luokka ShyeRepository, joka sisältää lähes kaikki sovelluksen tarvitsemat tietokantaoperaatiot ja palautukset. DAO-luokkaa kutsutaan tietokantaobjektin kautta erillisellä säikeellä.

3.2.4 Tietokannan ulkopuoliset rakenteet

Ennen kuin siirrytään kertomaan uudesta tietokannasta, sen migraatio-olioista ja sitä varten tehdyistä testeistä, tulee ymmärtää myös ne sovelluksen sisäiset rakenteet, jotka

eivät sisälly tietokantaan. Nämä rakenteet olivat olemassa jo ennen työn alkamista. Niiden suunnittelijat ovat asiakasyrityksen entisiä työntekijöitä, jotka olivat osallisina sovelluksen rakentamisessa.

Sovelluksen käyttöliittymä oli ennen insinöörityötä rakennettu suurimmalta osalta vanhan tietokantamallin mukaisesti, mutta asetuksia ja järjestystä helpottamaan oli tehty muitakin luokkia. Kuten aiemmin mainittiin, yhdellä sovelluksen sivulla on yhteensä 6 ruutua, joihin käyttäjä voi ottaa kuvia aterioistaan. Jokaisessa näistä ruuduista on yksi valmiiksi luotu ruokailuolio, vaikka ruudussa ei olisikaan kuvaa. Aterialla oli status-kolumni, joka määritti sen, onko käyttäjä ottanut ruutuun kuvaa tai merkintää vai ei. Jos ruokailun aika oli mennyt ohi, status muuttui automaattisesti ohitetuksi. Tuleva ruokailu on statukseltaan puolestaan Unset eli sitä ei ole vielä asetettu. Ruokailun paikka oikeassa ruudussa määriteltiin aterian tyypin mukaisesti, ja niille oli automaattisesti asetettu kellonajan sisältävä aikaleima.

Ruokailujen sijoittamisen helpottamiseksi oli luotu Day-malli, jossa yhden päivän ruokailut sijoitettiin yhteen päiväolioon. Päiväoliolla oli päivämäärä ja lista siihen kuuluvista aterioista objekteina. Sovelluksen käynnistyksen yhteydessä päiväoliot alustettiin ja niihin kuuluvat listat täytettiin ateriaolioilla. Näin yksikään ruutu ei käyttöliittymässä ole oikeasti tyhjä, ja käyttäjän lisätessä aterian hän vain muokkaa jo olemassa olevaa tietokantalisäystä. Käyttäjän valitessa ruokailun haluamaltaan päivämäärältä ohjelma hakee tarvitun aterian ensin päiväobjektista sen tyypin mukaisesti. Päiväobjekti helpotti näin hakuoperaatiota sijoittaen ruokailut jo valmiiksi omiin kohtiinsa jokaisella päivämäärällä.

Uudessa tietokantamallissa haluttiin ottaa mukaan ateriaryhmät, joiden tarkoitus selitettiin luvussa 3.1. Ateriaryhmien esittely ohjelmalle sekoitti päivämallin täysin, joten se piti suunnitella uudelleen. Uudessa päivämallissa ei ole enää valmiiksi tehtyjä tyhjiä ateriaolioita jokaisella ruudulla, vaan ruuduille sen sijaan sijoitetaan tyhjät ateriaryhmäoliot. Ryhmien sijoitus oikeille ruuduille tehtiin käyttäen niiden aikaleimoja. Jokaisen ryhmän aikaleimaan sisältyy päivämäärä ja kellonaika. Kellonaika määriteltiin sovelluksen sisään kirjoitettujen ateriatyyppien oletuskellonaikojen mukaisesti. Jälkikäteen ajatellessa olisi ollut yksinkertaisempaa, jos ateriaryhmillä olisi myös tyyppi, sillä tämä tekisi ruutuihin sijoittamisesta ja migraatiosta yksinkertaisempaa ja toimivaa vähemmällä koodilla. Kuitenkin suunnitelmien muutos tässä vaiheessa olisi ollut liian työlästä, joten sitä ei lähdetty tekemään.

Yhdessä päiväoliossa on siis kuuden ateriaryhmän ID:t listassa. Sovelluksen käynnistyksen yhteydessä koodissa tarkistetaan jokaisen käyttäjälle sallitun päivämäärän kohdalla ateriaryhmäobjektien olemassaolo ja oikeellisuus, ja olioiden ID:t sijoitetaan staattisiin listoihin. Valmiiksi täytetyt ja rakenteeltaan eheät päiväoliot sijoitetaan sitten staattiseen HashMap-karttaan, josta päivät haetaan käyttäen avaimena päivämäärää. Olioiden ollessa julkisia ja staattisia ne pystytään hakemaan mistä tahansa luokasta, niiden sisältö on aina sama, eikä niitä tarvitse alustaa uudelleen. Näin vältytään turhan monelta tietokantaoperaatiolta, jotka voisivat hidastaa sovelluksen käyttöä huomattavasti.

Päiväolioiden alustuksesta ja niiden rakenteen eheyden tarkistamisesta kerrotaan lisää luvussa 3.2.7.

3.2.5 ShyeDatabase-luokka

ShyeDatabase.java on abstrakti RoomDatabasen aliluokka. Se on koko tietokannan tukiranka. Room-tietokanta rakennetaan ja päivitetään tämän luokan avulla ja tämän luokan kautta tehdään DAO-kutsut. Se toimii ikään kuin tietokantaa kuvaavana objektina, jota ei tarvitse erikseen avata tai sulkea.

Ohjelman ollessa auki tietokanta on olemassa staattisena objektina. Se alustetaan pääaktiviteetissa heti käynnistyksen yhteydessä, jolloin se suorittaa myös tarvittavat migraatio-operaatiot. Tämän jälkeen luokan sisällä tietokannan instanssi merkitään luoduksi. Kun instanssi on luotu, sitä voidaan kutsua mistä tahansa luokasta ja sen ollessa staattinen sen ei tarvitse käydä läpi koko tietokannan luontia ja avaamista joka kutsulla.

Room-tietokantaa käytetään oliomaisesti, eli kaikissa luokissa, missä sitä tarvitaan, luodaan oma ShyeDatabase-olio, joka alustetaan staattisella getInstance()-metodilla (Esimerkkikoodi 13).

```
public static ShyeDatabase getInstance(Context context) {
    if (INSTANCE == null) {
        synchronized (ShyeDatabase.class){
            INSTANCE =
            buildDatabase(context.getApplicationContext());
            INSTANCE.updateDatabaseCreated(context);
        }
    }
    return INSTANCE;
}
```

Esimerkkikoodi 13. GetInstance()-metodi, jota käytetään tietokantaolion alustamiseen.

Mikäli instanssi on tässä vaiheessa null eli sitä ei ole vielä luotu, se rakentaa tietokannan ja asettaa sen luoduksi. Tämä rakennus tulee tehdä vain ohjelman käynnistyksen yhteydessä, ja muut viittaukset instanssiin tulisi tehdä käyttäen valmiina olevaa instanssia.

Kun olion instanssia kutsutaan ensimmäisen kerran, se luo tietokannan käyttämällä buildDatabase()-metodia. Se käyttää Roomin sisäistä metodia tietokannan rakentamiseen ja asettaa tietokannalle sovelluksen kontekstin. Alla olevassa koodiesimerkissä näytetään ShyeDatabase-luokan rakennusmetodi.

```
public static ShyeDatabase buildDatabase(final Context appContext){
    context = appContext;
    try {
        return Room.databaseBuilder(appContext, ShyeDatabase.class,
DATABASE_NAME)
        .addCallback(new Callback() {
            @Override
            public void onCreate(@NonNull SupportSQLiteDatabase db) {
                 super.onCreate(db);
            }
    }).addMigrations(MIGRATION_1_2, MIGRATION_2_3, MIGRATION_3_4).build();

    } catch (Exception e){
    e.printStackTrace();
    }
    return null;
    }
```

Esimerkkikoodi 14. Roomin databaseBuilder-metodia käyttävä staattinen buildDatabase()-metodi, joka palauttaa rakennetun tietokantaolion.

Esimerkkikoodissa 14 näkyy metodin sisällä Roomin databaseBuilder-metodin käyttö. Siihen lisätään kutsu SupportSQLiteDatabase-objektin luontiin, joka tukee Room-tietokannan operaatioita. Kutsun jälkeen tietokanta rakennetaan, ja siihen tulee lisätä kaikki luokan Migration-oliot. Nämä oliot sisältävät migraatiot yhdestä versiosta seuraavaan, ja koska tietokanta luodaan neljänteen versioonsa, migraatio versiosta kolme versioon neljä on lisätty edeltävien migraatio-objektien lisäksi.

Migraatio-objekti sisältää ne SQLite-operaatiot, joita tarvitaan suorittamaan migraatio versiosta toiseen. Tarvittavat operaatiot osoittautuivat työn toteutuksen aikana odotettua haastavammiksi suurien rakenne-erojen vuoksi. Monimutkaisuutensa vuoksi oli hyvä kirjoittaa testit sovelluksessa tehtäville migraatioille. Testeistä kerrotaan lisää seuraavassa kappaleessa.

3.2.6 Room-migraation testaus

Jotta Room-tietokannan toimivuus ja migraation onnistuminen voitaisiin varmistaa, migraatiolle oli hyvä tehdä testejä. Testit kirjoitettiin käyttäen rajapintana AndroidJUnit4-kirjastoa.

Yleisesti Android-yksikkötestausta varten on olemassa useita eri moduuleja ja kirjastoja, mitkä helpottavat testausta joko sovelluksen rakenteessa tai käyttöliittymässä. Koska migraatiota varten kirjoitetut testit eivät varsinaisesti testaa käyttöliittymää ollenkaan, ylimääräisiä moduuleja ei tarvinnut asentaa ja oli oltava tarkkana tietoa haettaessa siitä, millaista kirjastoa ja rajapintaa esimerkkitestit käyttivät. Mallina testien perusrakennetta varten käytettiin GitHubista löytyvää arkkitehtuurikomponenttien esimerkkiohjelmaa, joka on kirjoitettu nimenomaan esimerkkinä migraatiota SQLitestä Roomiin varten. [11.]

Testauksen aloittamista varten piti ensin tuoda projektiin build.gradle-tiedoston avulla tarvittavat kirjastot (esimerkkikoodi 15).

```
implementation 'junit:junit:4.12'
androidTestImplementation 'androidx.test:runner:1.1.1'
androidTestImplementation 'androidx.test:rules:1.1.1'
androidTestImplementation 'androidx.test.ext:junit:1.1.0'
androidTestImplementation group: 'androidx.test', name: 'core', version:
'1.1.0'
androidTestImplementation 'androidx.room:room-testing:2.1.0-alpha06'
```

Esimerkkikoodi 15. Projektin testaukseen liittyvät riippuvuudet build.gradle-tiedostossa.

Samaan tiedostoon oli merkittävä projektin konfiguraatiota varten käytettävä yksikkötestien ajaja (esimerkkikoodi 16).

```
defaultConfig{
..
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}
```

Esimerkkikoodi 16. DefaultConfig-lohkossa on projektin peruskonfiguraatio ja yksikkötestien ajaja merkitään sinne.

Tarvittavien JSON-skeematiedostojen, joita Room käyttää varmistaakseen tietokantarakenteen olevan oikea, tiedostosijainti piti myös merkitä tähän tiedostoon alla olevan esimerkkikoodin mukaisesti.

```
sourceSets{
..

androidTest.assets.srcDirs += files("$projectDir/schemas".toString())
}
```

Esimerkkikoodi 17. Skeematiedostojen sijainti konfiguroidaan sourceSets-lohkoon, josta Gradle löytää projektin käyttämät ulkoiset kansiot.

Kun Gradle oli konfiguroitu oikein, testien rakentaminen pystyttiin aloittamaan. Android-ympäristössä yksikkötestit kirjoitetaan yleisesti projektin lähdekansion sisällä olevaan androidTest-kansioon. Android Studio osaa hakea halutut testit tästä sijainnista niitä ajaessa.

Yksikkötestauksessa ei varsinaisesti käytetä ohjelman koodia itseään, vaan testejä varten luodaan tiedostoja, jotka toimivat kopioina oikeista luokista. Samaan tyyliin tietokantamigraation testaukseen piti tehdä uusia tiedostoja kuvailemaan alkuperäistä tietokantamallia. Näitä kutsuttiin Helper- eli avustajaluokiksi, ja ne sisälsivät metodit tietokannan luomiseen, alustamiseen ja sulkemiseen käyttäen tukenaan SupportSQLite-tietokanta-oliota. Avustajaluokat toimivat testeissä tietokannan tukena ilman, että varsinaiseen vanhaan koodiin täytyi puuttua.

Roomilla on myös oma luokka testausta varten, jotta testejä varten ei tarvitsisi kirjoittaa liikaa erillistä koodia. Tämä on nimeltään MigrationTestHelper, ja se määritellään testiluokan säännöksi käyttäen @Rule-annotaatiota.

```
    @Rule
    public MigrationTestHelper mHelper = new MigrationTestHelper(Instrumenta-
tionRegistry.getInstrumentation(),
            ShyePreDatabase.class.getCanonicalName(), new FrameworkSQLiteOpen-
HelperFactory());
```

Esimerkkikoodi 18. Testisäännön määritys.

Esimerkkikoodissa 18 näkyy MigrationTestHelper-luokan alustus. InstrumentationRegistry.getInstrumentation() on metodi hakemaan sovelluksen konteksti testausympäristössä. Android-ohjelmoinnin ollessa voimakkaasti kontekstipainotteinen konteksti pitää saada myös testiin käyttämättä varsinaista koodia, jota varten InstrumentationRegistry-luokka on tehty. Samalla alustuksessa annetaan Room-tietokantaluokan nimi sekä migraatiota varten käytettävän testiluokan alustus. [12.]

Testiavustaja ei käytä tietokantaluokan sisäistä koodia muuten kuin tarkistamalla sen DAO-luokat ja annotaatiot. Nämä annotaatiot ovat erittäin tärkeitä varmistamaan datan eheyden ja oikeellisuuden migraation tapahduttua.

Kun säännöt on määritelty, Room-testeille määritellään annotaatioiden avulla myös ne operaatiot, jotka suoritetaan ennen ja jälkeen testien ajamista. Tarvittavat annotaatiot ovat @Before ja @After, joiden alle tässä tilanteessa kirjoitettiin metodit setUp() ja tearDown(). Alustusmetodissa, joka suoritetaan ennen testien ajamista, alustetaan kaikki tarvittavat muuttujat ja rakennetaan testiversio tietokannasta. Lisäksi tässä insinöörityössä tänne alustettiin silloin käytetty kontrolleri datan lisäämistä ja käsittelyä varten. Kontrolleriin kirjoitettiin erilliset metodit testausympäristöön: aterian luonti ja lisääminen testitietokantaan.

Testin ajon jälkeen suoritetaan operaatio, jossa testitietokanta tuhotaan ja siellä ollut data poistetaan. Vanha data pitää poistaa testin jälkeen siltä varalta, että testi epäonnistuu tai testiä joudutaan muuttamaan. Tietokantaan lisättävät objektit sisältävät aina samat tiedot ja identtisiä rivejä halutaan välttää.

3.2.7 Migraatiotestin toteutus

Ensin tässä työssä tehtiin yksinkertainen testi, jossa testitietokantaan lisättiin uusi ateria tietoineen, tietokanta muutettiin Room-versioon ja tarkistettiin, olivatko aterian tiedot säilyneet eheinä. Roomin avustajaoliolla on metodi runMigrationsAndValidate(), jolle annetaan parametreiksi testitietokannan nimi, migraatiota varten kirjoitetut Migration-oliot sekä uusi versionumero. Mikäli varsinaisen tietokantamallin, joka määräytyy entiteetin kautta, sekä uuden tietokannan rakenteiden välillä on eroja, avustaja ilmoittaa virheestä testituloksissa. Joskus testit kaatuivat yksinkertaistenkin erojen, kuten Not null -sääntöjen ristiriidoista, ja Room ilmoitti virheistä tulostaen odotetut ja varsinaiset JSON-tulokset. [13; 14.]

Testien kanssa tuli siis olla tarkkana migraatiota kirjoittaessa. Migraatio itsessään oli rakenteeltaan monimutkainen ja raskas, sillä se jouduttiin kirjoittamaan perinteisellä SQLite-kielellä.

Yksi huomattavimmista ongelmista huomattiin siirtäessä suurempaa määrää dataa uuteen versioon, esimerkiksi yhden ateriaruudun kaksi ateriaa, tai ateriatyypin kellonajan ollessa muuttunut. Ensinnäkin piti poistaa kaikki ateriat, joiden status oli ohitettu tai ei-asetettu, sillä nämä olivat toimineet tyhjinä ateriaolioina sitä varten, että käyttäjä voisi itse lisätä niihin kuvan ja tekstin. Jäljelle jääneistä ateriaolioista piti selvittää, mihin ruutuun ne kuuluivat. Tyypillisesti ohjelman ajon aikana tämä selvitettiin aterian tyypin mukaisesti, mutta suunnittelun aikana tehtyjen virheiden vuoksi tämä ei ollut niin yksinkertaista.

Ensinnäkin johtuen aterian riippuvaisuudesta omistaa sille kuuluva ateriaryhmä olemassa olevia aterioita ei voitu suoraan siirtää uuteen tietokantaan, sillä ateriaryhmiä ei vielä ollut luotu. Ateriaryhmien taulu pitäisi luoda ja tietokanta siirtää siihen versioon, missä tämä taulu on luotu. Sitten vasta voitaisiin luoda ateriaryhmäoliot ja asettaa ne päivämäärien ja ateriatyypin oletuskellonaikojen mukaan oikeisiin ruutuihin. Operaation jälkeen ateriat itse voidaan siirtää niille kuuluviin ateriaryhmiin.

Tästä syntyi idea suorittaa migraatio kaksivaiheisena. Migraation alussa tietokannan versionumero oli 2. Ensimmäinen vaihe migraatiolle oli päivittää tietokanta versiosta 2 versioon 3. Tälle niin kutsutulle ”esimigraatiolle” tehtiin oma kansio sovelluksen koodissa, ja Roomin sääntöjen mukaisesti jokaiselle taululle tehtiin entiteettiluokka ja tietokannalle rakennusluokka. Tietokannan rakennukseen liitettiin migraatio versiosta 2 versioon 3, joka oli rakenteeltaan yksinkertainen: siinä tarvitsi vain luoda ateriaryhmille oma taulunsa ja määrittää sen rakenne (esimerkkikoodi 19).

```
public static Migration MIGRATION_2_3 = new Migration(2, 3) {
        @Override
        public void migrate(@NonNull SupportSQLiteDatabase database) {
            database.execSQL("DROP TABLE IF EXISTS mealgroups");
            database.execSQL("CREATE TABLE mealgroups (mg_id INTEGER NOT NULL
PRIMARY KEY, " +
                    "timestamp INTEGER NOT NULL DEFAULT 0);");

        }
    };
```

Esimerkkikoodi 19. Migraatio versiosta 2 versioon 3

Migraation ensimmäiselle vaiheelle kirjoitettiin oma, erillinen testinsä. Siinä ensin luotiin kaksi testiateriaa, jotka syötettiin versiossa 2 avattuun testitietokantaan. Testiaterioilla oli omat tyyppinsä, päivämääränsä ja kellonaikansa. Luonnin jälkeen migraatio ajettiin ja

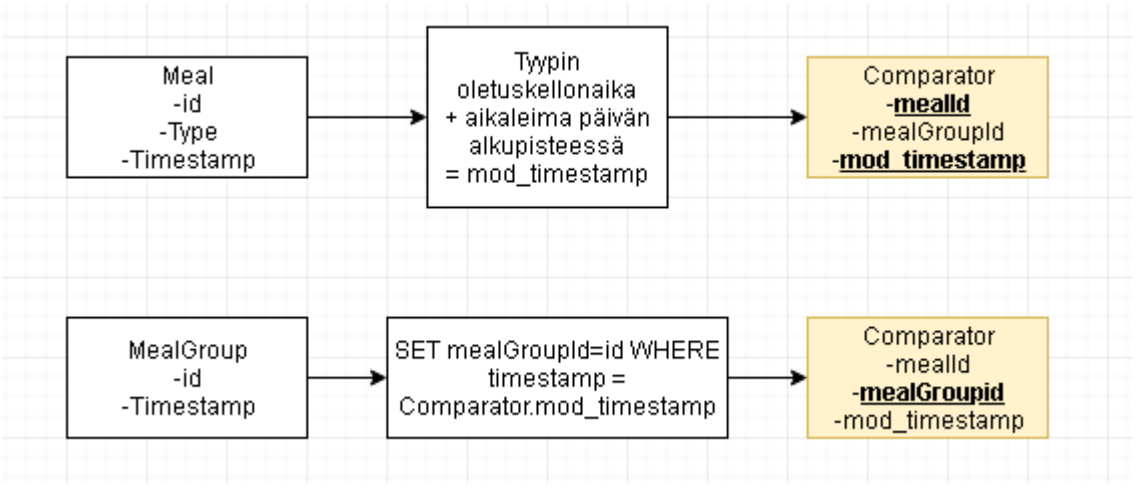
validoitiin, ja tyhjä ateriaryhmätaulu oli valmiina käytettäväksi. Testissä pystyttiin käyttämään valmiiksi kirjoitettua metodia ateriaryhmätaulun täyttämistä varten. Kun kaikki ateriaryhmät oli luotu, tarkistettiin siirrettyjen aterioiden tietojen eheys sekä se, että ateriaryhmiä oli luotu oikea määrä (esimerkkikoodi 20).

```
/**
 * Test first stage of migration: create the
 * correct amount of mealGroups in our database.
 * check if the amount of Days is correct,
 * and check if the Day objects have the right amount of MealGroups,
 * and check if the MealGroups within Day are unique.
 * @throws IOException
*/
@Test
public void a_testMigrationfrom2to3() throws IOException{
        Date date = new Date();
        long h = date.getTime();
        DateTime dt2 = new DateTime(2019,2,21,18,30,38);
        long j = dt2.toDate().getTime();
        fi.octo3.shye.deprecated.Meal m = mController.addNewMeal-
Test(fi.octo3.shye.deprecated.Meal.Type.BREAKFAST, h);
        fi.octo3.shye.deprecated.Meal m2 = mController.addNewMeal-
Test(fi.octo3.shye.deprecated.Meal.Type.DINNER, j);
        m.setStatus(3);
        m2.setStatus(3);
        mController.dbCreateMealTest(m, db);
        mController.dbCreateMealTest(m2, db);
        long mealId = m.getId();
        System.out.println(mealId);
        mHelper.runMigrationsAndValidate(TEST_DB_NAME, 3, true, MIGRATION_1_2,
MIGRATION_2_3);
        System.out.println(m.getStatus());
        dayCreatorHelper = new DayCreatorTest(getMigratedRoomDatabase(), Ap-
plicationProvider.getApplicationContext());
        dayCreatorHelper.loadAllDaysLite_test(dt.toDate().getTime());
        fi.octo3.shye.controllers.pre_migration_db.models.Meal ughMeal =
getMigratedRoomDatabase().mealDao().getMeal(mealId);
        assertThat(ughMeal.getType(), equalTo(MealType.BREAKFAST));
        assertThat(dayCreatorHelper.getDaysList(), correctAmountOfDays());
        for(Day day : dayCreatorHelper.getDaysList()){
            assertThat(day, hasUniqueMealGroups());
            assertThat("Day object size", day.getMealGroups().size(),
equalTo(6));
        }
    }
```

Esimerkkikoodi 20. Testi, joka kirjoitettiin tietokannan nostamiseen versiosta 2 versioon 3.

Vasta, kun ateriaryhmäoliot oli luotu ja niiden oikeellisuus varmistettu, pystyttiin varsinainen migraatio suorittamaan. Sille tehtiin erillinen migraatio-olio, joka sisälsi tarvittavat operaatiot tietokannan siirtämiseen versiosta kolme versioon neljä. Tähän liittyi suunnitelmavaiheen virheen takia vaikeuksia.

Ateriaryhmälle ei tietokannan alkuperäisen suunnitelman mukaan asetettu tyyppiä. Sen sijaan ateriaryhmällä on tyypin oletuskellonaikaa vastaava aikaleima, joka ei välttämättä korreloidu itse tyypin ajan kanssa, sillä käyttäjä on voinut vaihtaa tyypin kellonaikaa. Jotta ateria pystyttiin varmasti asettamaan oikeaan ruutuun, piti aterioista selvittää niiden päivämäärä ja niihin kuuluvan tyypin oletusaika sekä verrata tästä saatua aikaleimaa ateriaryhmän aikaleimaan. SQLite ei salli sisäisiä WHERE-lauseita, kuten SQL normaalisti sallisi, joten tämänlainen tarkka hakeminen piti tehdä eri tavalla. Ratkaisu kehitettiin luomalla migraation sisällä tiedon vertailua varten väliaikainen taulu.



Kuva 4. Havainnollistava kuva väliaikaisen taulun täyttämisestä

Kuva 4 havainnollistaa operaatiot, jotka suoritettiin väliaikaisen vertailutaulun luomista varten. Ensin selvitettiin sovelluksen koodista aterian tyypin oletuskellonaika. Koodin metodien avulla pystyi palauttamaan aikaleiman, joka vastasi kellonaikaa ilman päivämäärää. Koska ateriaryhmän aikaleima sisältää päivämäärän ja kellonajan, niiden yhdistämistä varten ateriallakin pitää olla päivämäärän ja kellonajan sisältävä aikaleima. Aterian aikaleimasta selvitettiin päivämäärä muokkaamalla aikaleima Jodan LocalDateTime-luokan avulla: aikaleimasta tehtiin LocalDateTime-objekti, jonka tunnit, minuutit ja sekunnit asetettiin nollaksi, jonka jälkeen saatu objekti muutettiin taas aikaleimaksi. Tuloksena saadun aikaleiman pystyi yhdistämään tyypin oletuskellonajasta saatuun aikaleimaan yksinkertaisesti lisäämällä. Tulokseksi saatiin päivämäärän ja tyypin oletuskellonajan sisältävä aikaleima, jota pystyy suoraan vertaamaan ateriaryhmän aikaleiman kanssa. Aterian ID ja muokattu aikaleima siirrettiin vertailutauluun.

Samalle riville vertailutaulussa piti lisätä vielä aterialle kuuluvan ateriaryhmän ID. Tätä varten muokattiin vertailutaulua siten, että mealGroupId eli ateriaryhmän ID asetettiin sillä ehdolla, että ateriaryhmän aikaleima oli sama kuin aterian muokattu aikaleima.

```
List<Long> timestamps = new ArrayList<>();
List<Long> mealIds = new ArrayList<>();
//since meal doesnt have the timestamp we need by default, we need to modify
it in according to its type
while(cursor.moveToNext()){
        long timeStamp = cursor.getLong(cursor.getColumnIndex-
OrThrow("timestamp"));
        int typeCode = cursor.getInt(cursor.getColumnIndexOrThrow("type"));
        MealType type = MealType.getById(typeCode);
        timeStamp=TimeSettings.getModifiedTime(context, timeStamp, type);
        timestamps.add(timeStamp);
        mealIds.add(cursor.getLong(cursor.getColumnIndexOrThrow("id")));
 }
cursor.close();
for(int i = 0 ; i < mealIds.size(); i++){
        database.execSQL("INSERT INTO comparator (meal_id, mod_timestamp)VAL-
UES (" + mealIds.get(i) +" , " +
                        timestamps.get(i) +");");
}

database.execSQL("INSERT OR REPLACE INTO comparator SELECT comparator.meal_id,
comparator.mod_timestamp, mealgroups.mg_id" +
                    " FROM comparator" +
                    " JOIN mealgroups ON comparator.mod_timestamp=meal-
groups.timestamp");
```

Esimerkkikoodi 21. Koodin sisäinen operaatio väliaikaisen taulun kirjoittamiselle

Kaikki väliaikaisen vertailutaulun luontiin tarvittavat operaatiot näkyvät esimerkkikoodissa 21. Ne sisältyivät migraatioon versiosta kolme versioon neljä. Kun vertailutaulun sisältö oli oikea, pystyttiin lopullinen datan siirto vanhasta ateriataulusta uuteen tekemään.

```
            //join the 2 tables correctly with the help of the temporary com-
parator table.
            database.execSQL("INSERT OR IGNORE INTO 'meals_new'(id, timestamp,
image, type, nutrition_id, size, activity, description, mealgroup)" +
                    "SELECT " +
                    "meal.id, " +
                    "meal.timestamp, " +
                    "meal.image1, " +
                    "meal.type, " +
                    "meal.nutrition_id, " +
                    "meal.size, " +
                    "meal.activity, " +
                    "meal.description," +
                    "mealgroups.mg_id " +
                    "FROM comparator"
                    + " JOIN mealgroups ON comparator.mealgroup_id = meal-
groups.mg_id" +
                    " JOIN meal ON comparator.meal_id=meal.id");
```

Esimerkkikoodi 22. Datan siirtäminen vanhasta ateriataulusta uuteen

Esimerkkikoodissa 22 näkyy operaatio, mikä tehtiin, kun siirrettiin data vanhasta ateriataulusta uuteen. Koska SQLitessä ei sinänsä ole toimivaa UPDATE-toimintoa, jossa taulu voitaisiin päivittää ja siihen lisätä uusia kolumneja, jouduttiin tätä varten tekemään uusi taulu lopullisen taulun rakenteella. Uuteen tauluun siirrettiin data käyttäen JOIN-toimintoa apuna selvittämään, minkä ateriaryhmän ID aterialle asetetaan. Aterian ja ateriaryhmän ID haettiin vertailutaulusta, ja jos molemmat ID:t olivat samalla rivillä, kuuluivat nämä kaksi oliota yhteen.

Kun siirto-operaatio oli onnistunut, poistettiin vanha taulu tietokannasta ja uusi taulu uudelleennimettiin vastaamaan haluttua taulua. Vertailutaulu poistettiin myös, sillä se ei kuulunut entiteetteihin eikä skeemoihin.

Lopulliseen migraatio-olioon kuului noin 130 riviä koodia, mikä vahvistaa ajatusta siitä, että perinteinen SQLite-kieli on liian raskasta käyttää tyypillisessä koodissa.

Migraatiolle versiosta kolme versioon neljä kirjoitettiin myös oma testinsä. Se sisälsi samat operaatiot kuin edellinen testi, eli testikantaan luotiin pari ateriaoliota ja tietokanta päivitettiin versiosta kaksi versioon kolme. Sitten luotiin ateriaryhmät oikeine aikaleimoineen ja vietiin tauluun. Sen jälkeen alustettiin uusi tietokanta, ja migraatio versiosta kolme versioon neljä ajettiin. Objektien eheys varmistettiin testin onnistumisen ehtona.

Kaikki kirjoitetut testit eivät suinkaan riittäneet kattamaan migraation kaikkia eri mahdollisuuksia, ja virheitä löydettiin myöhemmin käytännön kokeiluissa. Luvussa 4 kerrotaan jatkokehitysideoista, joiden avulla testejä voidaan vielä parantaa.

3.2.8 Migraatio käytännössä

Kun sovellus käynnistetään ensimmäistä kertaa, RoomDatabase ajetaan, ja se suorittaa tarvittavat migraatiot. Tietokannan muutokset tehdään pysyvästi ja versionumeroa kohotetaan.

Ajonaikaisessa migraatiossa oli riskinsä johtuen monitasoisesta versionkohotuksesta. Kun tietokanta oli avattu versiossa 4, sitä ei pystynyt enää avaamaan versiossa 2, mikä

johti ohjelman kaatumiseen, jos sovellus yritti käynnistyksen yhteydessä suorittaa migraation ensivaihetta. Sovelluksen tulisi siis tarkistaa, onko tietokanta jo päivitetty uusimpaan versioonsa, jolloin se voisi suoraan avata uusimman tietokannan, vai pitäisikö kaksivaiheinen migraatio suorittaa, jolloin se aloittaisi avaamalla migraation välivaiheisen tietokannan. Tämä pystyttiin ratkaisemaan Androidin SharedPreferences-kirjaston avulla.

Kun käyttäjä avaa ensimmäistä kertaa sovelluksen päivityksen jälkeen, sovellus tarkistaa käynnistyksen yhteydessä sisäisen tietokantansa. SharedPreferences-kannassa on avain, jonka arvopariksi kuuluu boolean-arvo, joka kertoo sen, onko päivitys tehty. Boolean-muuttuja needMigration on oletusarvoltaan tosi, eli migraatio tarvitaan. Jos arvo on true, sovellus voi turvallisesti olettaa, että käyttäjä on vasta päivittänyt tai asentanut sovelluksensa ja siirtyy suorittamaan kaksivaiheista migraatiota. Avaimen arvo asetetaan tämän jälkeen epätodeksi, ja se pysyy sellaisena koko sen ajan, kun sovellus on asennettuna eikä sen sisäistä tietokantaa pyyhitä. Jos käynnistyksen yhteydessä arvo on jo epätosi, sovellus olettaa migraation jo tapahtuneen ja siirtyy suoraan uusimman version avaamiseen. Esimerkkikoodissa 23 näkyy tämä tarkistus ja sen aikana suoritettavat operaatiot.

```
//hasMigrated() is a static constant that is set to True after this operation
is done to avoid
//doing it more than once.
     if(!hasMigrated){
          //NeedMigration is a static class in utils/NeedMigration.java that
     checks current database
          //version
          if(NeedMigration.needMigrate(this)){
               /*
               2-STEP MIGRATION: if the user has recently updated, control-
               lers/pre_migration_db/ShyePreDatabase.java
               will be created first. new Instance will be instantiated in
               getInstance(context).
               DayRepo is instantiated and it will create all of the meal
               groups for every day and store them into the static
               dayInstance object,
               so that existing meals can be moved into the corresponding
               meal groups during migration.
                */
               pdb=ShyePreDatabase.getInstance(this);
               dRepo = new DayRepo(pdb, this, dayInstance);
               createDays();
               /*
               After first step of the migration, second step proceeds by in-
               stantiating controllers/database_controller/ShyeDatabase.java.
               All meal groups need to exist in order for this to work.
               Make sure no meals exist that are older than 42 days.
                */
               mdb=ShyeDatabase.getInstance(this);
```

```
            mRepo = ShyeRepository.fromContext(this);
        } else {
            /*
            If the user already has updated and the database exists in its
            newest state, skip straight to ShyeDatabase.java
            instantiation and load Day objects from database.
             */
            mdb=ShyeDatabase.getInstance(this);
            dRepo = new DayRepo(mdb, this, dayInstance);
            createDays();
        }

        /*
        Since db was created, now we want to load the async task to delete
        data older than 42 days
        */
        mgr = new StartupTaskManager(mdb);
        mgr.loadStartUpTasks();
        hasMigrated=true;
    }
```

Esimerkkikoodi 23. Sovelluksen käynnistyksen yhteydessä suoritettavat operaatiot ja tarkistus

Kuten esimerkkikoodissa 23 mainitaan, sovelluksella on staattinen boolean-muuttuja hasMigrated, joka on ensi käynnistyksessä epätosi. Tietokannat alustetaan ja migraatiot suoritetaan vain, jos hasMigrated on epätosi. Muuttuja on staattinen, joten sen arvo ei nollaannu ajon aikana. Operaatioiden jälkeen muuttuja asetetaan todeksi, eikä tähän lohkoon enää sovelluksen ajon aikana siirrytä. Näin voidaan välttää turha ajonaikainen tietokannasta hakeminen, mikä hidastaisi suoritusta huomattavasti.

HasMigrated-muuttujan ollessa epätosi sovellus käy tarkistamassa aiemmin mainitun avain-arvoparin arvon ja sen mukaan suorittaa vastaavat operaatiot. DayRepo-luokka, joka sisältää päiväobjektien luontimetodit, alustetaan sen tietokantaolion avulla, joka ensimmäisenä avataan. Tämä pitää tehdä siksi, että jos migraatio pitää tehdä kaksivaiheisena, päiväolioiden ja niihin kuuluvien ateriaryhmien pitää olla jo olemassa, jotta toinen vaihe menisi läpi ilman kaatumista tai datan menettämistä. Päiväolioiden alustaminen on muussa tapauksessa vain sitä, että niihin kuuluvat ateriaryhmät haetaan ja syötetään niiden päivämääriä vastaavien päiväolioiden listoihin.

3.2.9 Repositorio ja tietokantaoperaatiot

Koska DAO-luokkien tietokantaoperaatioita ei tule käsitellä sovelluksen käyttöliittymäpuolella, tuli tätä varten kirjoittaa repositorioluokka. Tämä luokka sisältää metodit kaikkiin sovelluksen tarvitsemiin tietokantaoperaatioihin.

Jotta sovelluksen ylikuormittumista voitaisiin välttää, tietokannan DAO-luokkiin ei kosketa sovelluksen pääsäikeellä. Vaikka tietyt operaatiot tulee suorittaa synkronisesti ohjelman toiminnan kanssa, on hyvä harjoittaa taustasäikeiden käyttöä tietokantaoperaatioiden suorittamiseen.

```
public void getMealObjectsFromGroup(final long mealGroup, final OnMealsLoaded-
Listener listener){
    listener.setBoxIntoLoading();
    exe.diskIO().execute(() ->{
    final WeakReference<OnMealsLoadedListener> listenerWeakReference =new
WeakReference<OnMealsLoadedListener>(listener);
    List<Long> tempList;
    Meal tempMeal;
    List<Meal> tempMealList = new ArrayList<>();
    OnMealsLoadedListener listener1 = listenerWeakReference.get();
    tempList= mDb.mealDao().getMealsFromGroup(mealGroup);
    for(long i : tempList){
        tempMeal = mDb.mealDao().getMeal(i);
        tempMealList.add(tempMeal);
    }
    exe.mainThread().execute(()->{
        if(listener1 == null){
            return;
        }
        //NO NULLS!
        if(tempMealList.size()!=0 && !tempMealList.contains(null)){
            listener1.setMeals(tempMealList);
            mealList = tempMealList;
        }
    });
    });
}
```

Esimerkkikoodi 24. Metodi hakemaan lista ateriaolioista, jotka kuuluvat yhdelle ateriaryhmälle.

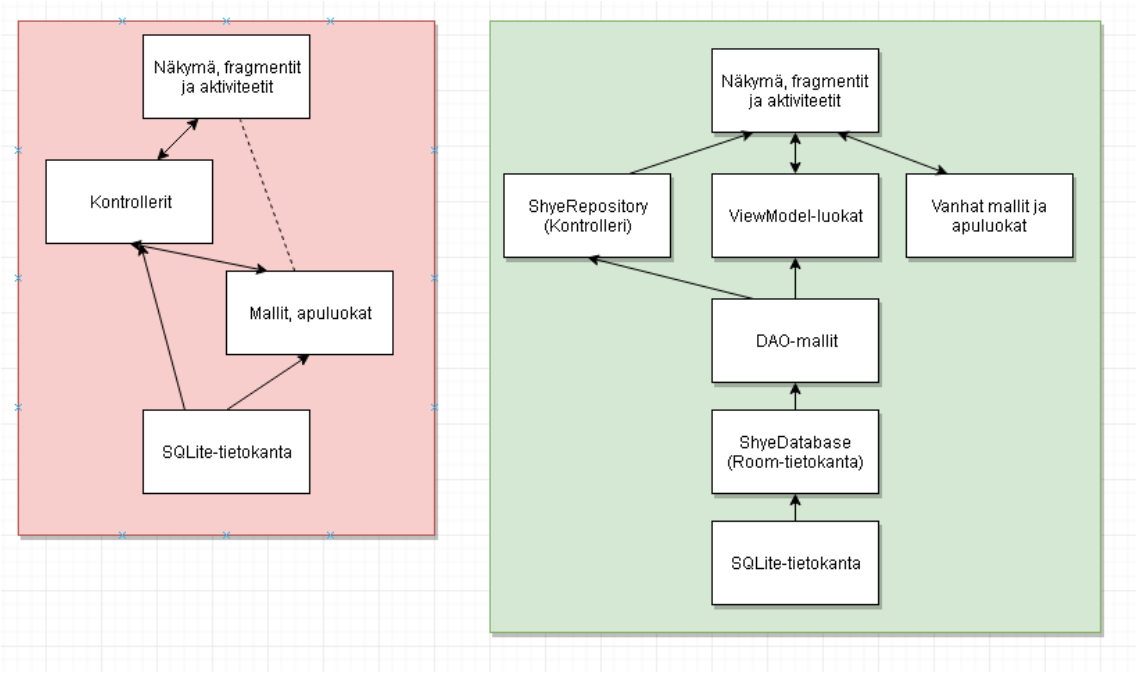
Koodiesimerkissä 24 näkyy metodi, jossa käytetään kommunikaatiota tausta- ja pääsäikeiden välillä. Säieoperaatiot suorittavat Executor-oliot, jotka haetaan niille kirjoitetusta luokasta, ja ne ovat valmiiksi alustettuja. Ensin lähetetään ilmoitus kuuntelijalle, joka asettaa ruudun käyttöliittymässä latautumisasemaan, eli se valmistautuu lataamaan aterian tietoja ruudun sisälle. Sen jälkeen diskIO() kutsuu Executor-oliota, joka laukaisee yhden taustasäikeen sovelluksen käyttöön. Tietokantakutsu tehdään taustasäikeen sisällä ja kyselyn tulokset talletetaan väliaikaiseen listaan. Samalla alustetaan metodin kuuntelijalle eli sille luokalle, joka tätä operaatiota tarvitsee, WeakReference-olio. WeakReference helpottaa säikeiden välistä kommunikaatiota ja estää tietovuotojen tapahtumista. [15.]

Kun väliaikainen lista on täytetty tietokantakyselyn tuloksilla, luokka laukaisee pääsäiettä kutsuvan olion, eli yhteys pääsäikeeseen avataan taustasäikeen sisällä. Tämä on pakollinen operaatio sitä varten, että käyttöliittymämuutoksia voitaisiin tehdä, sillä taustasäikeillä ei ole oikeutta koskea näkymän elementteihin. Pääsäikeen kautta kuuntelijalle lähetetään täytetty lista UI-käyttöä varten sekä repositorion staattinen lista täytetään nopeaa hakua varten. Samalla kuuntelija saa tiedon siitä, että operaatio on valmis, joten se pystyy lataamaan aterian tiedot käyttöliittymään käyttäjän nähtäväksi.

Periaatteessa arkkitehtuurin sääntöjen mukaisesti jokaiselle entiteetille pitäisi tehdä oma repositorioluokkansa, eli yhteen entiteettiin liittyvät metodit ovat vain yhdessä luokassa. Jatkokehityksessä ShyeRepository-luokan voi muuttaa abstraktiksi, jolloin sen kautta voidaan kutsua entiteettien repositorioita, tai entiteettien repositorioluokat voivat olla ShyeRepositoryn aliluokkia. Liitos ShyeRepositoryyn tulisi olla olemassa, sillä tällä hetkellä kaikki viittaukset tietokantaan tehdään joko ShyeRepositoryn tai ViewModel-luokkien avulla. ViewModel-luokista kerrotaan tarkemmin luvussa 3.5.

3.3 Sovelluksen arkkitehtuuri

Insinöörityön tekoon siirtyessä sovelluksen sisäinen arkkitehtuuri ei noudattanut erityisesti mitään mallia tai sääntöä. Vaikka jotkin luokat oli merkitty malli- tai kontrollerikansion luokiksi, niiden rakenne ei välttämättä vastannut mallia, näkymää tai kontrolleria.



Kuva 5. Kaavio sovelluksen arkkitehtuurista ennen insinöörityössä tehtyjä muutoksia ja niiden jälkeen

Yksinkertainen MVC-mallia noudattava yhteys löytyi sovelluksen tietokannasta ja sen yhteydestä koodiin - tietokanta ja sen entiteetit olivat malleja, DataController-luokka oli kontrolleri, joka muutti mallit Java-olioiksi ja suoritti tietokantaoperaatioita. Fragmentit ja aktiviteetit olivat näkymiä, jotka kommunikoivat molemminpuolisesti DataController-luokan kanssa. Arkkitehtuurimallin ulkopuolisia luokkia olivat esimerkiksi TimeSettings-luokka, joka sisältää ateriatyyppien aikaleimat ja käyttäjän asettamat asetukset. Kuvassa 5 näkyy sovelluksen vanha arkkitehtuuri vasemmalla ja uusi arkkitehtuuri oikealla.

Uudessa versiossa arkkitehtuuri on tehty jokseenkin selkeämmäksi. Tosin sovellus ei vieläkään noudata täysin mitään selkeää mallia, kuten MVC- MVP- tai MVVM-mallia. Tämä päätös tehtiin, sillä koko sovelluksen koodin refaktorointi arkkitehtuurimallin mukaiseksi vaatisi liikaa työtä, mitä ei pystytty priorisoimaan insinöörityön aikana. Kuitenkin Android-arkkitehtuurikomponenttien käyttö on tehnyt sovelluksen arkkitehtuurista huomattavasti selkeämpää. Joillekin fragmenteille, jotka ovat tiheässä yhteydessä tietokannan kanssa, kuuluu oma ViewModel-luokkansa, ja nämä fragmentit tottelevat MVVM-mallia. ViewModel-luokista kerrotaan tarkemmin luvussa 3.5. Toisaalta repositorioluokan käyttäminen tottelee enemmän MVC-mallia.

3.4 Aktiviteetit ja fragmentit

Androidin arkkitehtuurikomponentit, mitkä tässä insinöörityössä otettiin käyttöön, ovat hyvin riippuvaisia niitä käyttävien isäntien elinkaaresta. Nämä isännät ovat aktiviteetteja tai fragmentteja, jotka ovat Android-ohjelman käyttöliittymä. Aktiviteetit muodostavat pohjan näkymän toiminnolle, ja aktiviteetin päälle kiinnitetään fragmentteja. Aktiviteetti ei itsessään välttämättä sisällä käyttöliittymäkomponentteja, vaan ne on sijoitettu fragmenttiin sen yläpuolella. Se näkymä, minkä käyttäjä näkee laitteensa ruudulla ajaessaan sovellusta, on yleensä fragmentti tai aktiviteetin sisällä oleva, osan näytöstä kattava fragmentti.

Fragmentit tekevät sovelluksesta modulaarisemman, ja niiden käyttö keventää sitä työtä, mitä sovellus tekee vaihtaessaan aktiviteetista toiseen. Tätä voisi verrata esimerkiksi yhden sivun verkkosivuihin, joissa linkkejä klikkaamalla käyttöliittymä muuttaa itseään dynaamisesti sen sijaan, että lataisi kokonaan uuden sivun. Samaan tyyliin aktiviteetin yllä näkyvät fragmentit vaihtuvat sen mukaan, mitä käyttäjä käyttöliittymällä tekee.

Fragmenttien käyttö myös parantaa kommunikaatiota käyttöliittymän eri osien välillä. Fragmentilta voi esimerkiksi lähettää rajapinnan kautta kuuntelijan ylemmän tason fragmentille silloin, kun se luodaan, ja kuuntelija on koko tämän fragmentin elinkaaren ajan aktiivinen. Kun ylemmän tason fragmentilla tehdään toiminto, jota kuuntelija tarkkailee, ylempi fragmentti lähettää kuuntelijalle kutsun, ja kuuntelijafragmentti päivittää käyttöliittymänsä. Androidin arkkitehtuurikomponentit esittelivät ViewModel-mallin tekemään tämänkaltaisesta kommunikaatiosta vielä yksinkertaisempaa.

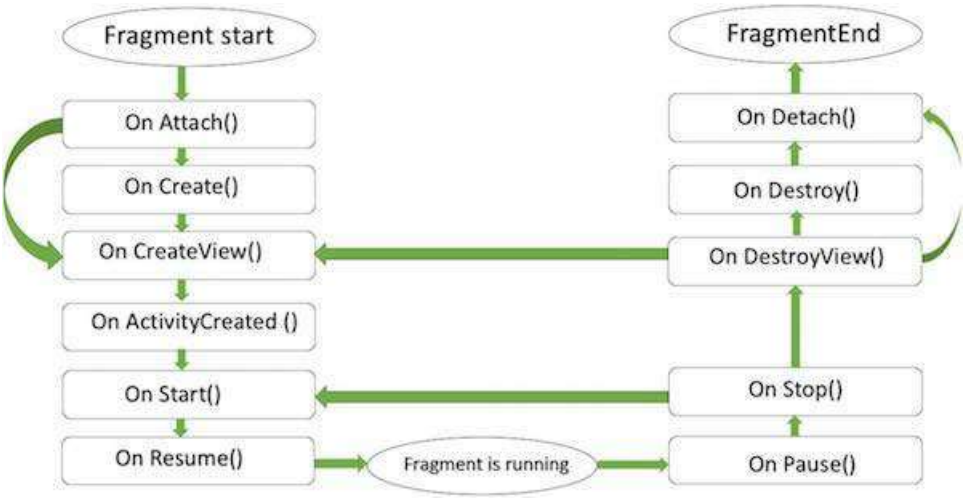
Fragmentit ovat hyviä myös kyvyssään näkyä sisäkkäin. Yhden fragmentin sisällä voi olla esimerkiksi lista sisäisiä fragmentteja, joita käyttäjä voi käydä läpi pyyhkäisemällä tai vetämällä ruutua. Toisaalla tietyn näppäimen painaminen voi avata puhekuplan, joka itsessään on pienen käyttöliittymän sisältävä fragmentti.

Kun Android-sovellus käynnistetään, se avaa käyttäjälle aktiviteetin, joka toimii pohjana kaikille tarvittaville fragmenteille. Tämä aktiviteetti on olemassa koko sovelluksen ajon ajan. Insinöörityön sovelluksessa avautuu ensimmäisenä MainActivity ja sen kautta ruudulle ilmestyy useita eri moduuleja:

- Sivun alareunassa oleva navigaatiovalikko on osa aktiviteettia,
- Kalenterin, päivämäärän ja ruudut sisältävä fragmentti valetaan aktiviteetin päälle,
- Ruokailujen ruudut itsessään kuuluvat kalenterifragmentin päällä olevaan fragmenttiin.

Etusivun ruutujen fragmentin alapuolella oleva fragmentti pitää sisällään ViewPager-objektia, jonka avulla käyttäjä pystyy navigoimaan sovelluksella pyyhkäisemällä sormella jompaankumpaan suuntaan. Pager luo jokaiselle sivulle oman fragmentin, joka pohjautuu samaan koodiin, mutta laitaa näkyviin erilaista dataa riippuen päivämäärästä ja kellonajasta. Uusi fragmentti luodaan sen luokan newInstance()-metodilla, joka paitsi luo fragmentin olion, myös antaa sille tarvittavat tiedot näyttämään oikeaa dataa.

Kun käyttäjä siirtyy fragmentilta toiselle, piiloon mennyt fragmentti ei tuhoudu kokonaan, vaan sen näkymä tuhoutuu. Se kuitenkin säilyttää sisällään datan sitä varten, kun se tulee taas näkyviin. Toisaalla jos fragmentti oli näkymän päällimmäisenä ja käyttäjä poistuu sivulta, päällä ollut fragmentti voidaan tuhota kokonaan, sillä sen sisältämää dataa ei tarvita muualla. Riippuen siitä, miten navigointi Android-sovelluksella toteutetaan ja mikä niiden välinen yhteys on, eri fragmentit käyvät kukin läpi niiden elinkaaren toimintoja.



Kuva 6. Fragmentin elinkaari ja sen aikana käytävät toiminnot [18]

Fragmentin elinkaari alkaa siitä, kun se alustetaan newInstance()-metodilla, jonka sisällä uusi fragmenttiolio luodaan. Se käy läpi metodit onAttach(), jossa fragmentti kiinnitetään

aktiviteetin kontekstiin, ja onCreate() ennen näkymän lataamista. OnAttach() on harvemmin käytetty metodi, sillä sen sisällön voi hyvin laittaa onCreate()-metodiin. OnCreate()-metodissa tehdään tarvittava haku tietokannasta dataan liittyvien käyttöliittymäobjektien lataamista varten ja alustetaan käyttöliittymän ulkopuoliset muuttujat.

Kun fragmentti lataa käyttöliittymän ja tulostaa sen näkyviin, se käy läpi metodit onCreateView(), onActivityCreated(), onStart() ja onResume(). Näistä ensimmäisessä alustetaan yleensä käyttöliittymäobjektit, kuten näppäimet ja tekstikentät, ja niiden toiminnallisuuksiin liittyvät metodit ajetaan tämän metodin kautta. OnActivityCreated() voidaan yleensä jättää tyhjäksi, mutta Androidin arkkitehtuurikomponentteja käytettäessä sen sisällä kannattaa alustaa esimerkiksi ViewModelin tarkkailijat. OnResume() käyttää yleensä dataa, joka lähetetään eri fragmentilta ja päivittää käyttöliittymää muutoksen tapahtuessa.

Näkyvillä oleva fragmentti kuuntelee käyttäjän tekemiä toimintoja, kuten napin painalluksia, näytön pyyhkäisyä jne. ja suorittaa niihin liittyvät toiminnot. Jos toiminto avaa uuden fragmentin tai siirtyy taaksepäin, näkyvistä poistuva fragmentti käy läpi elinkaaritoimintoja johonkin pisteeseen asti. Joissain tapauksissa fragmentin näkymä vain pysähtyy, jolloin se menee onPause()- ja onStop()-metodien läpi. Jos näkymä vaihtuu kokonaan, mutta fragmentin tila säilyy, se menee onDestroyView()-metodiin asti.

Fragmentin vaihtumisessa käytävät toiminnot riippuvat fragmentin transaktiosta. Aktiviteetille eli fragmenttien kontekstille kuuluu fragmenttien hallintaan tarkoitettu FragmentManager. Managerin avulla voi avata, muokata ja sulkea fragmenttien transaktioita. Sovelluksessa transaktiossa on pino fragmentteja, jossa alemmat fragmentit ovat odottavassa tilassa ja päällimmäiset aktiivisina. Riippuen niistä metodeista, joita transaktiossa käytetään, vaihdettava fragmentti käy johonkin pisteeseen asti elinkaaren lopettamisen metodit. Fragmenttien alle jäävää tilaa, jossa ne odottavat uutta kutsua, sanotaan backstackiksi eli fragmenttipinoksi.

Siirryttäessä transaktion avulla fragmentilta A fragmenttiin B on kolme erilaista skenaariota [19]:

- Fragmentti B lisätään fragmenttiin A transaktion add()-metodilla. Fragmentti A säilyy transaktion fragmenttipinossa ja sen näkymää ei tuhota, vaikka fragmenttia itseään ei olisi näkyvissä. Fragmentti A jää suorittamaan

metodejaan ja B käy läpi alkumetodinsa. Kun fragmentista B poistutaan, A ei kutsu alkumetodejaan, sillä se on säilyttänyt tilansa.

- Fragmentti B korvaa fragmentin A transaktion replace()-metodilla. Tällöin fragmentti A tuhoutuu täysin ja käy läpi kaikki elinkaarensa poistometodit. Tällöin poistuminen fragmentista B pitää tehdä myös korvauksen avulla, sillä edellinen fragmentti on irrotettu kontekstistaan eikä ole enää olemassa.
- Fragmentti B korvaa fragmentin A transaktion replace()-metodilla, mutta fragmentti A säilytetään fragmenttipinossa. Fragmentin A näkymä tuhoutuu, mutta sen instanssi ja data säilyy. Tällöin se menee metodiin onDestroyView() asti. Fragmentti B käy läpi elinkaareensa tavalliseen tapaan. Siitä poistuttaessa Fragmentti A palauttaa näkymänsä käyttämällä siihen säilynyttä dataa, jatkaen elinkaartaan onCreateView()-metodista.

3.5 LiveData ja ViewModel

Edellisessä luvussa kerrottiin, miten aktiviteetit ja fragmentit käyvät elinkaarensa eri vaiheet monta kertaa läpi ajon aikana ja kuinka monivaiheinen fragmenttien elinkaari on. Aiemmin fragmentin käyttämä data ja sen tila säilytettiin instanssin tilaan savedInstanceState()-metodilla ja onStateRestored()-metodilla tallennettu data haettiin. Dataa käytettäessä piti olla tietoinen näkymän elinkaaren vaiheista ja tasoista. ViewModel kehitettiin sitä varten, että käyttöliittymän tarvitsevan datan, kuten tietokannasta haettavan ja käyttäjän syöttämän datan, ei tarvitse tuntea tai hallita näkymän elinkaaren tasoa.

ViewModelin tarkoitus on tarkkailla datan kulkua ja tarjota näkymälle yhteys ulkopuoliseen dataan dynaamisesti. Tässä insinöörityössä ViewModelit tarkkailevat Room-tietokantaan tehtäviä kyselyitä ja siellä liikkuvaa dataa. Malli tarjoaa näkymälle tarkkailijan, joka tarkkailee kyselyä ja päivittää käyttöliittymän dynaamisesti muutoksen tapahtuessa. Tarkkailija on elossa aktiviteetin tai fragmentin elinkaaren ajan. [20.]

ViewModelin ja Room-tietokannan kanssa yhteensopiva datatyyppi on LiveData, joka mahdollistaa tietokantakyselyn jatkuvan tarkkailun tarkkailijan ollessa avoin. LiveDatan tarkkailija on tietoinen paitsi oman näkymänsä elinkaaren tilasta, myös muiden komponenttien elinkaaresta. Elinkaaritietoisuus varmistaa sen, että LiveData päivittää vain niitä tarkkailijoita, jotka ovat sillä hetkellä aktiivisia. [21.]

Yksinkertainen LiveData-olio sisältäisi esimerkiksi listan ruokailuista ateriaryhmän sisällä. Kyselyä tämän listan hakemiseen pystytään tarkkailemaan LiveData-olion kautta, ja ViewModel tarjoaa LiveData-objektin näkymän käytettäväksi. ViewModel sisältää myös metodit lisäämään aterian ateriaryhmään tai poistamaan sen. Näiden metodien käynnistyessä data muuttuu ja tarkkailija huomaa tämän, jolloin se laukaisee tarkkailijan sisällä olevan onChanged()-metodin ja päivittää näin käyttöliittymää halutulla tavalla.

```
public class AllMealsViewModel extends AndroidViewModel {
    private LiveData<List<Meal>> liveMealList;
    private ShyeDatabase mDb;

    public AllMealsViewModel(@NonNull Application application) {
        super(application);
        mDb=ShyeDatabase.getInstance(application);

        liveMealList=mDb.mealDao().loadMeals();
    }

    public LiveData<List<Meal>> getLiveMealList(){
        return liveMealList;
    }
}
```

Esimerkkikoodi 25. ViewModel-luokka tarjoaa tarkkailtavan listan tietokannan aterioista

Koodiesimerkki 25 sisältää esimerkin ViewModel-luokan alustamisesta. Luokka alustuksessaan hakee Room-tietokannasta LiveData-listan tietokannan aterioista ja palauttaa sen LiveData-objektina. Tätä objektia voidaan tarkkailla aktiviteetissa tai fragmentissa.

SHYE-sovelluksessa yksinkertainen yhden kyselyn LiveData ei riittänyt kaikkiin tarkoituksiin. Esimerkiksi silloin, kun tarkkaillaan päivän aterioiden määrää ja päivitetään ViewPageria määrän muuttuessa, näkymämallin pitäisi tarkkailla päivän jokaisen ateriaryhmän sisäistä aterialistaa, eli erillisiä tietokantakyselyitä päivän ateriaryhmiä kohtaan. Tällaisissa tapauksissa hyödynnettiin MediatorLiveDataa. MediatorLiveDatan avulla voidaan tarkkailla montaa erillistä LiveData-objektia samanaikaisesti ja myös päivittää niiden arvoja. Erilliset LiveData-objektit lisätään yhden LiveDatan lähteiksi, ja näitä tarkkaillaan jatkuvasti MediatorLiveDatan ollessa olemassa. Tätä on hyvä käyttää esimerkiksi sovelluksen tapauksessa silloin, kun tarkastellaan yksi-moneen-suhdetta päivän ateriaryhmien ja aterioiden välillä (esimerkkikoodi 26).

```
public LiveData<List<Long>> getMealIdsInDay(){
        liveDataMerger = new MediatorLiveData<>();
        //create a for loop for each id within day object to get ALL the meals
of the day
```

```
        for(long id : day.getMealGroups()){
            LiveData<List<Long>> newMealsInGroup = db.mealDao().loadMealsWith-
inGroup(id);
            groupsAndMealsMap.append(id, newMealsInGroup);
            liveDataMerger.addSource(groupsAndMealsMap.get(id), live-
DataMerger::setValue);
        }
        return liveDataMerger;
    }
```

Esimerkkikoodi 26. Metodi, joka luo yhdistelmän LiveData-olioita päivän jokaista ateriaryhmää kohden.

Esimerkkikoodissa 26 haetaan lista päiväolion sisällä olevista ateriaryhmistä ja sen läpi iteroidaan. Jokaista ateriaryhmää kohti haetaan LiveData-olio, joka sisältää listan sen ateriaryhmän aterioiden IDstä. Näin ollen LiveDatan tyyppi on List<Long>, koska IDt ovat long-tyyppisiä numeroita. Ryhmät ja ateriat säilytetään kartassa, jonka avulla listojen päivittäminen on helpompaa. MediatorLiveDatan addSource-metodilla listan sisältävälle LiveData-oliolle luodaan tarkkailija ja se liitetään lähteeksi välittäjälle. Lopullisen välittäjäolion sisällä on aktiivisena kuusi tarkkailijaa, ja muutoksen tapahtuessa missä tahansa näistä tarkkailijoista välittäjä lähettää tarkkailijalleen huomautuksen.

```
mViewModel.getMealIdsInDay().observe(getViewLifecycleOwner(), longs -> {

    if(isResumed){
        int index = mViewPager.getCurrentItem();
        myAdapter.changeTheItems(longs, mCurrentIndexDay, index);
        mViewPager.setCurrentItem(myAdapter.getPageIndexForMealId(mViewMo-
del.getMealId()));
        isResumed=false;
    } else {
        myAdapter.changeItems(longs, mCurrentIndexDay);
        mViewPager.setCurrentItem(myAdapter.getPageIndexForMealId(mMealId));
    }
    updateIndicator(mViewPager);
});
```

Esimerkkikoodi 27. Välittäjäolion tarkkailija.

MediatorLiveDatan eli LiveDatan välittäjän tarkkailija käynnistetään fragmentin onActivityCreated()-metodissa eli sen jälkeen, kun näkymä ja data on alustettu. Tarkkailijan omistaja on fragmentin elinkaaren omistaja, eli se on olemassa ja tarkkailee sisäisiä olioitaan niin kauan, kun fragmentin elinkaari on käynnissä.

Minkä tahansa näkymämallin tarkkailtavan olion tarkkailija rekisteröidään metodissa onActiviyCreated(), sillä fragmentit voivat usein olla ohjelman taustalla epäaktiivisina.

Epäaktiivisen fragmentin tarkkailu kuormittaa sovellusta turhaan, ja siksi onActivityCreated() on paras paikka rekisteröidä tarkkailijat, jotka kommunikoivat näkymän kanssa.

Fragmenttien kanssa on myös otettava huomioon tarkkailijan tila silloin, kun fragmentin näkymä ei ole aktiivisena, mutta sen instanssi on olemassa. Jos tarkkailijan elinkaareksi määritetään fragmentin elinkaari, tarkkailija tulee olemaan aktiivinen koko fragmentin instanssin olemassaolon ajan. Tällöin taustalla ollut fragmentti, joka tuodaan uudestaan esille ja se kutsuu onActivityCreated()-metodia, rekisteröi turhan kopion tarkkailijasta, ja fragmentilla on näin ollen useampi aktiivinen tarkkailija tarkkailemassa samaa LiveData-objektia. Ylimääräiset tarkkailijat kuormittavat koodia ja aiheuttavat metodien moninkertaisen käynnistyksen heikentäen sovelluksen vakautta.

Fragmentin näkymän ja fragmentin instanssin elinkaarta voidaan ajatella erillisinä. Jos tarkkailija rekisteröidään näkymän elinkaaren omistajalle getViewLifeCycleOwner()-metodia käyttämällä, on tarkkailija aktiivisena silloin, kun näkymän elinkaari on aktiivinen. Näin ollen fragmentin tuhotessa näkymänsä ja mennessä epäaktiiviseen tilaan tarkkailija tuhoaa itse itsensä, ja ylimääräisten tarkkailijainstanssien olemassaololta vältytään. Muussa tapauksessa ViewModelin tarkkailija pitäisi irrottaa fragmentin onViewDestroyed()-metodissa. [23.]

Esimerkkikoodin 27 tarkkailija päivittää fragmenttilistan sisältöä silloin, kun aterioita lisätään tai poistetaan käyttöliittymän kautta. Listan adapterille annetaan päivitetty lista aterioista ja sen sisältö päivitetään näiden mukaisesti. Samalla tämänhetkinen sivu asetetaan juuri luodun aterian indeksiin, tai nykyiseen indeksiin riippuen siitä, lisättiinkö uusi ateria vai muokattiinko nykyistä dataa. Kun listoja päivitetään eli sinne lisätään tai sieltä poistetaan esine, lista pitää ladata uudestaan tietokannasta. Vanha LiveData-olio poistetaan välittäjän lähteistä ja sen tilalle asetetaan uusi, samaa listaa tarkkaileva LiveData-olio. Poisto ja uudelleenasettaminen pitää tehdä, sillä välittäjä ei itse yhdistä dataa luonnissa, vaan yksinkertaisesti välittää sisältönsä tarkkailijalle. [22]

Sovellukselle vaarallinen paikka rekisteröidä näkymämallin tarkkailija on fragmentin sisällä silloin, kun fragmentti kuuluu toisen fragmentin tai aktiviteetin sisältämään ViewPageriin. Sellainen ViewPager, jota adaptoi FragmentStatePagerAdapter, pitää yhtäaikai-

sesti aktiivisena kolmea siihen kuuluvaa fragmenttia. Kun sivuilla navigoidaan, aktiivisena olevat fragmentit vaihtuvat. Jokaisella sivulla on erillinen instanssi samasta fragmentista, ja näiden elinkaarien tilat vaihtuvat sivuja pitkin navigoidessa. Jos fragmentin näkymässä rekisteröidään näkymämallin tarkkailija, kopioita tästä tarkkailijasta voi olla yhtäaikaisesti aktiivisina monessa eri sivujen fragmenteissa. Ongelmia voi syntyä esimerkiksi silloin, kun käyttäjä pyyhkäisee sivujen läpi nopeasti, jolloin fragmenttien instanssit eivät ehdi luoda itseään kokonaan ennen kuin ne taas tuhoutuvat aiheuttaen sovelluksen kaatumisen.

Ongelma ratkaistiin siten, että sen sijaan, että näkymämalli alustettaisiin fragmentin onCreate-metodissa, jokaisen sivun fragmentille lähetetään parametrina jo alustettu näkymämallin olio. Näkymämallilla on tiedossa taulu tarkkailtavista LiveData-olioista, jossa avaimena on tarkkailijafragmentin instanssi. Kun fragmentti luodaan, adapteri kutsuu sille kirjoitettua setViewModel()-oliota ja asettaa näin fragmentin näkymämalliksi sen instanssin, jonka adapterin omistaja sille lähettää. Fragmentilla on näin tiedossa näkymämalli, jonka sisäistä LiveData-oliota se voi tarkkailla. Näkymämallin tarkkailija rekisteröidään kuten tavallisesti fragmentin onActivityCreated()-elinkaarimetodissa ja kyseinen tarkkailija on aktiivisena vain silloin, kun siihen sivuun kuuluva fragmentti on näkyvissä.

Toinen esiintynyt ongelma liittyy informaatioon, jota sovellus tarvitsee silloin, kun käyttäjä painaa etusivulla olevaa ateriaruutua. Etusivun fragmentti sisältää ViewPagerin, jonka sivuina on instanssit ruudut sisältävistä fragmenteista. Ruuduista painamalla sovellus siirtyy joko kameranäkymään tai aterian yksityiskohtaiseen näkymään. Kameranäkymään siirryttäessä aktiviteetti vaihtuu ja alla olevat fragmentit tuhotaan, mutta aterianäkymään siirryttäessä etusivun fragmentit jäävät aktiivisina fragmenttipinoon. Sivut sisältävä fragmentti on ruutujen isäntäfragmentti.

Ruuduille kuuluvat tarkkailijat ovat aktiivisina silloinkin, kun käyttäjä on aterianäkymässä, sillä elinkaaren tuhoamismetodeja ei käydä läpi, ellei niiden isäntä käy läpi samoja metodeita. Sovellus hidastuu ja muuttuu epävakaaksi sen joutuessa pitämään yllä useita sivuja, näkymämalleja ja tarkkailijoita. Aterianäkymän alla olevia fragmentteja ei voi täysin tuhota, sillä niiden välillä on oltava tiivis kommunikaatio.

Ruutujen ulkonäkö pitää päivittää sitä mukaan, mitä muutoksia käyttäjä aterioihinsa tekee ja mitä toimintoja hän käyttää. Toiminnot on saatu vakaiksi lukuisten kuuntelija-kutsuyhteyksien ja rajapintojen avulla, mutta jatkokehittäjän kannattaa selvittää, miten kaiken saisi toimimaan varmemmin arkkitehtuurikomponenttien avulla. Ainakin koodin määrä ruutujen fragmentilla voisi vähentyä huomattavasti niitä käyttämällä.

4 Jatkokehitysideoita

Vaikka migraatio on nyt toimiva ja sovellus toimii pääosin oikein, se ei ole vielä täysin vakaa ja sen sisäistä arkkitehtuuria voidaan korjata.

Etusivulla oleva ViewPager voidaan kirjoittaa syöttämään yksi valmiiksi määritelty näkymämallin instanssi jokaiseen siinä olevaan fragmenttiin parametrin avulla. Näkymämalliluokkaan alustetaan taulu tarkkailijoista ja tarkkailtavista, jotta se voi pitää yllä monta tarkkailijaa yhdessä ViewPagerissa. FragmentDayTiles-luokkaan kirjoitetaan yksinkertainen setter eli asetusmetodi näkymämallia varten. Adapteri luo uuden FragmentDayTiles-instanssin jokaiselle selaajan sivulle. Tämän jälkeen voi kutsua sen fragmentin asetusmetodia ja asettaa näin jokaiselle fragmentille saman näkymämallin kutsumalla fragment.setViewModel(mViewModel). Fragmentin tarkkailijaa ei enää rekisteröidä onActivityCreated()-metodissa vaan setViewModel()-metodissa.

Repositorioluokka kannattaa jakaa modulaarisesti eri osiin. Jokaisella tietokannan taululla olisi oma repositorioluokkansa, jolloin yhteen tauluun liittyvät tietokantakutsut löytyvät yhdestä luokasta. Kannattaa myös selvittää, voivatko erilliset repositoriot yhdistää ShyeRepository-luokkaan abstraktion avulla, jolloin pääkoodia ei tarvitsisi muuttaa liikaa ja ShyeRepository-luokkaa voidaan edelleen käyttää eri repositorioiden isäntänä.

Tietokannan nykyisessä tilassa sovellukseen voidaan implementoida esimiehen toivomat uudet toiminnot. Jos kuitenkin suunnitelmiin tulee muutoksia ja tietokannan mallia pitäisi muuttaa, se tehdään päivittämällä ShyeDatabase-luokkaa. Yksinkertaisesti DAO-luokat muutetaan haluttuun muotoon, tietokannan versionumeroa korotetaan ja Roomille kirjoitetaan tarvittavat migraatiot. Vanhat migraatio-oliot kannattaa säilyttää varmuuden varalta, mutta uutta migraatiota varten voidaan vain kirjoittaa uusi migraatio-olio.

Uusia ominaisuuksia ohjelmoitaessa kannattaa suunnitella ja tutkia, miten ne voidaan toteuttaa modulaarisesti ja tyylikkäästi ja miten Androidin arkkitehtuurikomponentteja voitaisiin hyödyntää niissä. Dynaaminen UI-päivitys tulee varmasti olemaan tarpeellista, jolloin MVVM-mallin ja LiveDatan käyttö on vahvasti suositeltua.

Uusille ominaisuuksille, migraatioille ja käyttöliittymämetodeille tulisi kirjoittaa jatkossa kattavat testiryhmät. Koska tämän työn lopussa migraation testit eivät vielä kata kaikkia mahdollisuuksia, niitä tulisi kirjoittaa enemmän. Varsinkin datan siirto ja eheys silloin, kun ateriaryhmässä on kaksi ateriaa ja kun premium-käyttäjä on kirjoittanut lisämuistiinpanoja, tulisi testata kattavammin.

Sovelluksen testaus yleisesti on puutteellista, joten jatkokehittäjän kannattaa miettiä yksikkötestien implementointia sovelluksen eri ominaisuuksille. Tämä on ehdottomasti tärkeä vaihe sovelluksen kehityksessä, sillä laajojen testien avulla voidaan ennaltaehkäistä ja korjata käyttöliittymään liittyviä bugeja. Nykyään kaikki testaus on sovelluksella käyttäjäkohtaista, eli käyttäjä testaa sovellusta laitteellaan käytännössä. Tämä asettaa sovelluksen alttiiksi huolimattomuusvirheille sekä mahdollisille kaatumisille, jotka tapahtuvat sovelluksen taustaprosesseissa tai vain joissain lokaaleissa.

5 Yhteenveto

See How You Eat -sovellus uudistettiin sekä tietokanta- että käyttöliittymäpuolelta. Sovelluksella oli vanhentunut SQLite-tietokanta, jonka rakenteen muokkaaminen ja päivittäminen oli erittäin työlästä ja koodin sen hetkisellä tasolla lähes mahdotonta. Sovelluskehityksen ylläpitämiseksi ja jatkokehityksen helpottamiseksi tietokanta päivitettiin käyttämään Room-tietokantaa. Room on Googlen tarjoama Androidin arkkitehtuurikomponentti, joka tarjoaa abstraktin tason SQLite-tietokannan ylle ja muuttaa SQLite-kyselyiden tulokset Java-olioiksi. Se siis käyttää ORM eli Object Relational Mapping -tyyliä tehdäkseen muutokset Java-olioiden ja SQLite-tietokannan välillä helposti ja automaattisesti.

Roomin migraatiotoimintoja apuna käyttäen sovelluksen tietokanta päivitettiin uusimpaan versioonsa, jolloin siinä on olemassa paikat myös tulevia ominaisuuksia varten.

Tietokantarakenne vastaa nykyään toivottuja toimintoja. Migraation yhteydessä varmistetiin, että käyttäjä ei menetä dataa päivittäessään sovelluksensa ja että ajonaikaisilta kaatumisilta vältyttäisiin.

Koska tietokanta uudistettiin, piti uudistaa myös loput koodista käyttämään Room-tietokantaa ja sen DAO-luokkia tietokannan kanssa kommunikointiin. Tätä varten tehtiin repositorioluokka, joka sisältää metodit kaikkien tarvittavien tietokantatoimintojen tekemiseen. Olemassa oleva koodi korvattiin käyttämään vanhan SQLite-kontrolleriluokan sijaan uutta repositorioluokkaa.

Osa sovelluksen toiminnoista vaativat dynaamista käyttöliittymäpäivitystä. Tämä tehtiin ennen useiden kuuntelija-kutsuyhteyksien avulla. Dynaaminen käyttöliittymäpäivitys uudistettiin käyttämällä Androidin arkkitehtuurikomponentteja ViewModel ja LiveData. ViewModel mahdollistaa tietokantakyselyn jatkuvan tarkkailemisen ja tarjoaa näkymälle tarkkailtavan LiveData-olion. Kun LiveDataan tehdään muutoksia, tarkkailija huomaa tämän ja ilmoittaa näkymälle, että käyttöliittymäpäivitys tulee tehdä. Näin yhteys tietokannan ja näkymän välillä on entistä tehokkaampi ja koodista tuli siistimpi.

SHYE-sovelluksen uudistaminen on onnistunut sovelluksen lähes joka osalta. Julkaisukelpoista versiota varten pitää vielä korjata testauksen yhteydessä havaittuja bugeja. Kun sovelluksen stabiiliudesta varmistutaan, päivitys voidaan julkaista ja sovellus on valmis jatkokehitystä varten.

Lähteet

1 Build numbers. Androidin Source-dokumentaatio. Verkkoaineisto. https://source.android.com/setup/start/build-numbers Luettu 25.5.2019.

2 Locale. Androidin Developers-dokumentaatio. Verkkoaineisto. https://developer.android.com/reference/java/util/Locale Luettu 25.5.2019.

3 Singh, Gagandeep. 18.7.2014, Stack Overflow-vastaus. https://stackoverflow.com/questions/16754643/what-is-gradle-in-android-studio Luettu 25.5.2019.

4 SQLite Database. Tutorialspoint-dokumentaatio. Verkkoaineisto. https://www.tutorialspoint.com/android/android_sqlite_database.htm Luettu 25.5.2019.

5 Save Data using SQLite. Androidin Developers-dokumentaatio. Verkkoaineisto. https://developer.android.com/training/data-storage/sqlite Luettu 25.5.2019.

6 Obut, Orhan. Android 101: Shared Preferences. 3.1.2018. ProAndroidDev-artikkeli. Verkkoaineisto. https://proandroiddev.com/shared-preferences-101-aee26c13e4 Luettu 25.5.2019.

7 Muntenescu, Florina. Understanding migrations with Room. 18.6.2017. Medium.com -artikkeli. Verkkoaineisto. https://medium.com/androiddevelopers/understanding-migrations-with-room-f01e04b07929 . Luettu 25.5.2019.

8 Szklarska, Paulina. Android Architecture Components: Room – Relationships. 1.11.2017. Medium.com -artikkeli. Verkkoaineisto. https://android.jlelse.eu/android-architecture-components-room-relationships-bf473510c14a Luettu 25.5.2019.

9 Muntenescu, Florina. 7 Steps To Room. 7.7.2017. Medium.com-artikkeli. Verkkoaineisto. https://medium.com/androiddevelopers/7-steps-to-room-27a5fe5f99b2 Luettu 25.5.2019.

10 Muntenescu, Florina. 7 Pro-tips for Room. 2.11.2017. Medium.com-artikkeli. Verkkoaineisto. https://medium.com/androiddevelopers/7-pro-tips-for-room-fbadea4bfbd1 Luettu 25.5.2019.

11 Muntenescu, Florina. Incrementally migrate from SQLite to Room. 18.12.2017. Medium.com-artikkeli. Verkkoaineisto. https://medium.com/androiddevelopers/incrementally-migrate-from-sqlite-to-room-66c2f655b377 Luettu 25.5.2019.

12 Room Migration Sample. 2017. Android Open Source Project Inc. Apache Software Foundation. Koodiesimerkki. Verkkoaineisto. https://github.com/googlesamples/android-architecture-components/tree/master/PersistenceMigrationsSample Luettu 25.5.2019.

13 Muntenescu, Florina. Testing Room migrations. 24.7.2017. Medium.com-artikkeli. Verkkoaineisto. https://medium.com/androiddevelopers/testing-room-migrations-be93cdb0d975 Luettu 25.5.2019.

14 humazed. expected vs found in exception. 7.9.2017.GitHub-foorumiviesti. https://github.com/humazed/RoomAsset/issues/1 Luettu 25.5.2019.

15 humazed. 11.1.2018. Github-foorumivastaus. https://github.com/humazed/RoomAsset/issues/7 Luettu 25.5.2019.

16 WeakReferece. Androidin Developers-dokumentaatio. Verkkoaineisto. https://developer.android.com/reference/java/lang/ref/WeakReference. Luettu 25.5.2019.

17 Communicate with other Fragments. Androidin Developers-dokumentaatio. Verkkoaineisto. https://developer.android.com/training/basics/fragments/communicating Luettu 25.5.2019.

18 Android Fragment Lifecycle. 16.8.2017. CSDevBin Android Coding Experiences. Verkkoaineisto. http://csdevbin.blogspot.com/2017/08/android-fragment-lifecycle.html Luettu 26.5.2019.

19 Rofe Haim, Hanan. Understanding Fragment's lifecycle calls during transaction. StackOverFlow-vastaus. 14.2.2017. Verkkoaineisto. https://stackoverflow.com/questions/42218546/understanding-fragments-lifecycle-methods-calls-during-fragment-transaction Luettu 26.5.2019.

20 Fujiwara, Lyla. ViewModels: A Simple Example. 28.6.2017. Medium.com-artikkeli. Verkkoaineisto. https://medium.com/androiddevelopers/viewmodels-a-simple-example-ed5ac416317e Luettu 26.5.2019.

21 Elye. Understanding LiveData Made Simple. 21.10.2018. Medium.com-artikkeli. Verkkoaineisto. https://medium.com/@elye.project/understanding-live-data-made-simple-a820fcd7b4d0 Luettu 26.5.2019.

22 Alcérecca, Jose. LiveData beyond the ViewModel – Reactive patterns using Transformations and MediatorLiveData. 16.7.2018. Medium.com-artikkeli. Verkkoaineisto. https://medium.com/androiddevelopers/livedata-beyond-the-viewmodel-reactive-patterns-using-transformations-and-mediatorlivedata-fda520ba00b7 Luettu 26.5.2019.

23 Beyls, Cristophe. Architecture Component Pitfalls – part 1. 24.10.2017. Medium.com-artikkeli. Verkkoaineisto. https://medium.com/@BladeCoder/architecture-components-pitfalls-part-1-9300dd969808 Luettu 26.5.2019.

24 Redaelli, Tiago. Architecting FragmentStatePagerAdapter, ViewModel and a Fragment List. 18.6.2018. StackOverFlow-vastaus. Verkkoaineisto. https://stackoverflow.com/questions/50817238/architecting-fragmentstatepageradapter-viewmodel-and-a-fragment-list Luettu 26.5.2019.