LAUREA
AMMATTIKORKEAKOULU
Yhdessä enemmän

Prototyypin luonti iOS-sovellukseen

Kirkkola, Konsta

Laurea-ammattikorkeakoulu

Prototyypin luonti iOS-sovellukseen

Kirkkola, Konsta
Tietojenkäsittelyn koulutus
Opinnäytetyö
Kesäkuu, 2019

Laurea-ammattikorkeakoulu **Tiivistelmä**
Laurea Leppävaara
Tietojenkäsittelyn koulutusohjelma

Konsta Kirkkola

Prototyypin luonti iOS-sovellukseen

Vuosi 2019 Sivumäärä 71

Tämän toiminnallisen opinnäytetyön kehittämistehtävänä oli suunnitella ja toteuttaa mobiilisovellus. Työn tavoitteena oli luoda sovellus, jossa käyttäjä asettaa itselleen aikaan sidotun budjetin ja jakaa sen eri kategorioihin. Käyttäjä kirjaa sovellukseen tekemänsä ostokset ja saa sekä budjetin että kategorioiden ylityksistä ilmoituksen. Työn toimeksiantajana oli liiketalouden opiskelija Haaga-Heliasta.

Työn tietoperustan muodostivat työssä hyödynnettyjen teknologioiden dokumentaatiot ja suunnitteluvaiheessa myös designiin liittyvä kirjallisuus ja tutkimustieto. Suunnitteluvaiheessa toteutus rajattiin kattamaan sovelluksen olennaisimmat toiminnot teknisistä haasteista johtuen. Sovelluksen prototyyppi toteutettiin iOS-alustalle.

Tekninen toteutus suoritettiin käyttäen Applen Xcode-kehitysalustaa ja Swift-ohjelmointikieltä. Muita työssä käytettyjä teknologioita olivat muun muassa Firebase, Core Data, Realm ja Cocoapods. Teknisessä toteutuksessa hyödynnettiin viiteaineistona Stack Overflow'n kaltaisia ohjelmointiin painottuvia sivustoja ja aiemmin hankittua omaa kokemusta.

Työn tuloksena oli toimiva prototyyppi sovellusideasta. Olennaisimmat toiminnot toteutettiin, kuten sovelluksen rakenne, välilehdet, navigointi, tietokantojen alustus ja tiedon tallentaminen tietokantoihin. Merkittävin toteuttamatta jäänyt toiminto oli dynaamiset kategoriat, joiden lisääminen ei ollut mahdollista toteuttaa suunnitteluvaiheessa luodun mallin mukaisessa ratkaisussa. Jatkokehitystä varten suunniteltiin uusi alustava malli, jonka avulla on mahdollista toteuttaa dynaamiset kategoriat ja niihin liittyvät toiminnot. Toinen kohde jatkossa on tutkia työn siirtämistä React Nativen kaltaiselle Cross Platform- alustalle, jolla on mahdollista kehittää sovellusta samanaikaisesti sekä iOS- että Android-alustalle vain yhdellä koodikannalla.

Asiasanat: iOS, Swift, Xcode, mobiilisovellus, kehittäminen

Laurea University of Applied Sciences **Abstract**
Degree Programme in Business Information Technology
Bachelor's Thesis

Konsta Kirkkola

Creation of a Prototype of an iOS Application

Year 2019 Pages 71

The development task of this functional thesis was to design and implement a mobile application. The goal of this work was to create an application, in which the user sets a time restricted budget for himself and then divides the budget into subcategories. The user then registers the purchases he or she has made to the application and receives a notification from the application, if the sum for the budget or one of the categories has been exceeded. The work was commissioned by a business student from the Haaga-Helia University of Applied Sciences.

The knowledge base of the work consisted of documentation of the technologies used in the application development and design-related literature and research data for the design phase of the work. At the design phase, the technical implementation phase was limited to cover the most essential functionalities of the application due to technical challenges. The prototype of the application was developed for the iOS platform.

The technical implementation phase was conducted using Apple's Xcode development platform and the Swift programming language. Other technologies used in the work include Firebase, Core Data, Realm and Cocoapods. In the technical implementation stage, sites such as Stack Overflow and previous experience with iOS development were also used as reference material.

The result of the work was a working prototype of the original application idea. The most essential functionalities were implemented such as application structure, tabs, navigation, database initialization and data storage in the databases. The most noticeable missing feature was the dynamic categories, which could not be implemented due to the limitations of the design model that was created during the design phase. For the purpose of further development, a new preliminary model was designed to allow the implementation of dynamic categories and functionalities related to it. Another objective in the future is to explore the transfer of the work to a Cross Platform environment, such as React Native, which allows development of the application simultaneously on both iOS and Android platforms with just one code base.

Keywords: iOS, Swift, Xcode, mobile application, development

Sisällys

1 Johdanto

Nykypäivän yhteiskunnassa henkilökohtainen taloudenhallinta ja rahankäyttö ovat alati kasvavia ongelmia. Näiden ongelmien ratkaisemiseen toimeksiantajani oli kehittänyt idean, joka mahdollistaa helpon tavan seurata henkilökohtaista rahankäyttöä ja budjetin raameissa pysymistä. Työn toimeksiantajana toimii entuudestani tuntema Haaga-Helian liiketalouden opiskelija, joka tarvitsi apua hänen ideaansa perustuvan mobiilisovelluksen suunnitteluun ja kehittämiseen. Sovelluksen prototyyppi toteutettiin iOS-alustalle, johtuen positiivisista aikaisemmista kokemuksistani iOS-kehitystyöstä verrattuna Android-kehitystyöhön.

Sovelluksen ideana on antaa käyttäjän määritellä itselleen budjetin, budjetin kategoriat, rahasummat kategorioille ja seuranta-ajan jokaiselle kategorialle. Tämän jälkeen käyttäjä aina tehdessään ostoksen kirjaa tämän sovellukseen, jossa sen arvo vähennetään sekä kokonaisbudjetista että valitusta alakategoriasta. Kun budjetissa tulee ylityksiä seuranta-ajan puitteissa joko kokonaisbudjettiin tai johonkin sen alakategorioista, antaa sovellus tästä varoituksen. Budjetin seuranta-aikana on myös tarkoituksena sallia käyttäjän tekevän muutoksia esim. seuranta-ajan pituuteen. Sovellusta kutsutaan alustavasti Mon- nimellä, ja se oli sovelluksen nimi tämän työn aikana.

Työn toteutus on hyödyllinen molemmille osapuolille. Minä saan kehitettyä omaa osaamistani iOS-sovelluskehittäjänä ja pääsen myös ensimmäistä kertaa osallistumaan täydessä roolissa ammattimaisen mobiilisovelluksen ideointiin ja suunnitteluun. Työssä käytettiin myös monia edistyneitä teknologioita, joiden opettelu ja omaksuminen ovat erittäin hyödyllisiä taitoja mobiilisovelluskehittäjälle. Työnantaja taasen hyötyy saadessaan alkuperäisestä ideastaan konkreettisen tuotteen, jonka avulla on mahdollista lähteä toteuttamaan markkinatutkimusta sovelluksen kysynnästä. Jos tuotteelle löytyy riittävästi kysyntää, voidaan sen jälkeen aloittaa tuotteen markkinointi ja viimeistely tuotantotason tuotteeksi, joka voidaan laittaa levitykseen iOS:n App Store- sovelluskauppaan.

Tässä opinnäytetyössä kehitettävällä sovelluksella ei ole vielä konkreettisia loppuasiakkaita, koska yksi työn jatkokehityksen päätavoitteista on tehdä tutkimusta tuotteen kysynnästä. Opinnäytetyön asiakas on aiemmin mainittu Haaga-Helian liiketalouden opiskelija, joka kehitti alkuperäisen idean sovellukselle ja häneen viitataan työssä termillä toimeksiantaja. Työssä kehitettyyn sovellukseen viitataan taasen termillä sovellus.

2 Työn lähtökohdat

Tässä osiossa käsitellään työn lähtökohtia sen alkuvaiheessa. Osiossa käydään läpi sekä työn lähtökohtainen tilanne, että jo olemassa oleva tietoperusta aiheesta. Tämän lisäksi käydään myös läpi menetelmät, joilla tietoperustaa voisi kerryttää suunnittelua ja kehitystyötä varten. Näiden lisäksi osiossa käsitellään työn tarkoitus ja tavoitteet.

2.1 Tilanteen kartoitus

Työn alussa konkreettista materiaalia sovelluksesta oli vain alustava perusidea ja sen tuoma arvonlisäys potentiaalisille asiakkaille. Tätä lukuun ottamatta alussa ei ollut muuta konkreettista sisältöä työhön liittyen. Työn tavoitteena oli käydä sovelluksen kehitysprosessi läpi aina määrittely- ja suunnitteluvaiheista toimivan prototyypin luomiseen asti, joten lähtökohtainen pohjamateriaali oli riittävä työn aloittamiseen.

Tilanteen kartoitus muilta osin oli positiivinen. Osaamista sekä työssä käytetystä Xcode- kehitysalustasta että Swift- ohjelmointikielestä löytyi jo vahvan perusosaamistason verran. Tämän lisäksi teknistä ja tietopohjaa laajentavaa materiaalia työn aiheista löytyi internetistä runsaasti. iOS on käyttöjärjestelmänä laaja, ja se käsittää merkittävän määrän erilaisia teknologioita. Siitä johtuen työn aikana tiedonhaku toimi ongelmanratkaisukeskeisesti, eli tietoa etsittiin ja löydettyä tietoa sovellettiin ratkomaan työn aikana ilmenneitä ongelmia.

Muilta osin tilanne työn alussa oli myös hyvä. Tietoperusta eri teknologioista ja käytettävistä menetelmistä oli hyvällä tasolla opintojeni kautta kertyneen kokemuksen ja osaamisen ansiosta. Käytännön kokemusta kaikista teknologioista ei ollut, mutta työssä oli sovelluksen tuottamisen lisäksi tavoitteena hioa sovelluskehitysosaamiseni puutteita käytännön työn kautta, joten kokemuksen puute kaikista teknologioista ei ollut ongelma. Työn tavoitteena oli myös saada luotua selkeä toimintamalli siitä, kuinka vastaavanlaisia projekteja tulisi lähestyä tulevaisuudessa.

2.2 Työn tarkoitus ja tavoitteet

Työn tarkoituksena oli ensiksi käyttämällä erilaisia tutkimus- ja ideointimenetelmiä generoida perusta sovelluksen rakenteelle, ulkoasulle, toiminnoille ja ominaisuuksille. Tämän jälkeen hyödyntäen erilaisia rautalankamallityökaluja toteutettiin havainnollistava malli sovelluksesta, joka demonstroi sen käyttäjäkokemuksen ilman ohjelmointia. Tämä vaihe iteroitiin niin monta kertaa, kunnes rautalankamallilla saatiin toteutettua halutunlainen malli sovelluksesta.

Seuraavaksi tarkoituksena olisi ryhtyä kehittämään sovelluksen prototyyppiä edellisten vaiheiden tuotosten pohjalta. Määrittely- ja suunnitteluvaiheessa tuotettiin kuitenkin hyvin paljon

erilaisia sovellusmalleja ja toimintoja. Tämän vuoksi työn totetusvaiheeseen tehtiin rajauksia, jottei työn mittakaava olisi liian laaja.

Lopullisena tavoitteena työssä olisi saada aikaan prototyyppi sovelluksesta, jossa on olennaisimmat toiminnot toteutettuna. Prototyypin avulla olisi tarkoitus ryhtyä esimerkiksi markkinoimaan sovellusta. Tämän lisäksi voitaisiin tehdä myös markkinatutkimusta siitä, onko työn tuotokselle kysyntää riittävästi, että sen myynti ja levitys olisi taloudellisesti kannattavaa. Tavoitteena olisi myös saada työ vietyä siihen pisteeseen asti, että se olisi hyvä pohja muutoksille tuleviin versioihin. Lisäksi prototyypin avulla pitäisi pystyä myös havainnollistamaan käytännön käytön avulla mahdollisia puutteita sen toiminnoissa ja rakenteessa.

3 Työn rajaus ja aiheen merkitys toimeksiantajalle/loppukäyttäjälle

Tässä osiossa käydään läpi, kuinka työn mittakaava rajattiin, ja mitä työssä oli tavoitteena saada toteutettua. Työhön tuli suunnitteluvaiheen päätteeksi paljon erilaisia toimintoja ja toteutusmalleja. Tämän vuoksi toteutusvaiheeseen jouduttiin tekemään rajauksia, jotta työn mittakaava ei olisi liian laaja. Toinen olennainen asia, jota tässä osiossa käsitellään, on työn merkitys sen toimeksiantajalle ja potentiaalisille loppukäyttäjille.

3.1 Työn rajaus

Työssä kehitettävä sovellus oli laaja kokonaisuus. Tästä johtuen sovelluksen prototyyppiin tehtiin rajauksia. Erilaisista suunnitteluista malleista tuli yli 10 erillistä välilehteä sovellukselle, joissa kaikissa oli erilaisia toimintoja. Tämän lisäksi suunnittelu- ja ideointivaiheissa löydettiin merkittävä määrä mahdollisia kokeellisia toimintoja ja ominaisuuksia. Näistä syistä sovelluksen määrittelyvaiheessa oli tärkeää luokitella eri ominaisuudet tärkeysjärjestyksen ja teknisen haastavuuden mukaan, jotta toteutuksen aikana olisi mahdollista saada olennaisimmat ominaisuudet toteutettua.

Työ rajattiin eri vaiheisiin seuraavasti. Ensimmäisenä oli määrittelyvaihe, jossa ideoitiin sovelluksen ulkoasu, rakenne, toiminnot ja ominaisuudet. Tarvittaessa tähän vaiheeseen voitiin palata työn aikana myöhemmin. Tämän jälkeen toinen vaihe, joka toteutettiin kokonaan työn aikana iteroituna, oli suunnitteluvaihe. Siinä valikoitiin parhaimmat ideat alkuperäisestä ideoinnista ja pyrittiin luomaan havainnollistava rautalankamalli sovelluksesta.

Sovellus suunniteltiin rautalankamallina, jolla pyrittiin myös havainnollistamaan sen käyttäjäkokemus. Iteroimalla rautalankamallia toimeksiantajan kanssa tarkennettiin mallia, kunnes malli oli halutunlainen. Näin luodun rautalankamallin avulla rakennettiin sovelluksen prototyyppi.

Toteutusvaiheessa oli työn ensimmäinen vaihe, jossa tehtiin rajauksia toteutukseen. Tavoitteena toteutuksessa oli luoda sovelluksen visuaalinen ulkoasu ja navigointi. Toisena tavoitteena oli myös tietokantojen alustus siten, että voitiin sekä lisätä tietoa ja muokata tietoa kantaan että myös tarvittaessa poistaa tietoa kannasta. Tämän lisäksi myös tiedon vastaanottaminen loppukäyttäjältä ja datan tallennus tietokantaan tuli saada toteutettua.

3.2 Työn merkitys toimeksiantajalle ja loppukäyttäjille

Aiheen merkitys on työn toimeksiantajalle hyvin olennainen. Työn toimeksiantaja on Haaga-Helian liiketalouden linjan opiskelija, ja hän on kehittänyt sovelluksen alkuperäisen idean. Hän toimi myös pääasiallisena ideoijana työn aiheen luonnissa. Työn merkitys on täten hänelle kahdensuuntainen. Toisaalta työssä hän pääsee näkemään IT-alan tuotekehitysprosessin alusta loppuun, ja täten saamaan selkeän kokonaiskuvan siitä, kuinka vastaavanlaisia tuotteita tehdään ja kehitetään. Hän pääsee myös käytännössä viemään eteenpäin bisnesideaansa ja siihen perustuvaa sovellusta osallistumalla sen kehitysprosessiin.

Tämän lisäksi työn lopputuotteen on tarkoituksena tukea häntä opinnoissaan. Sovelluksen valmistuttua on tarkoituksena tämänhetkisten suunnitelmien mukaan alkaa tekemään markkinointityötä työlle ja luomaan sille näkyvyyttä esim. sosiaalisessa mediassa. Tämän lisäksi tavoitteena jatkokehityksessä on ryhtyä tekemään markkinatutkimusta, jolla kartoitetaan tuotteelle löytyvää kysyntää. Markkinatutkimuksen tulosten perusteella tehdään päätös, onko kannattavaa ryhtyä kehittämään liiketoimintaa tuotteen ympärille. Jos tutkimuksen tulokset ovat positiivisia, niin tarkoituksena on perustaa yritys, aloittaa tuotteen markkinointi ja tehdä siitä lopullinen tuotantoversio, joka laitetaan levitykseen App Storeen.

Koska työn varsinainen idea oli vielä ideatasolla työn alussa, ei sillä ollut vielä konkreettista asiakaskuntaa. Tästä huolimatta oli kuitenkin mahdollista sanoa, miten potentiaaliset asiakkaat hyötyisivät sovelluksesta ja millaisia asiakkaita sillä voisi olla. Potentiaalisille asiakkaille työ tuo lisäarvoa sallimalla heidän yksinkertaisella ja käyttäjäystävällisellä tavalla pitävän kirjaa menoistaan. Tätä kautta he voivat hallinnoida paremmin omaa talouttaan ja välttää ylimääräisiä menoja ja liiallista rahankulutusta. Työn alkuvaiheessa toimeksiantaja identifioi potentiaalisiksi kohderyhmiksi sovellukselle etenkin nuoret aikuiset ja opiskelijat, joilla yleensä talouden tarkka seuraaminen on välttämätöntä pienistä tuloista johtuen.

4 Tutkimusmenetelmät ja tietoperusta

Tässä osiossa käsitellään työssä käytettyjä tutkimusmenetelmiä ja työn tietoperustaa. Aluksi käydään läpi työn tutkimusmenetelmät, joita hyödynnettiin työn alussa kerryttämään tietoa

erilaisista design-ratkaisuista, joita voisimme käyttää sovelluksen suunnittelussa. Tämän jälkeen käydään läpi työn tärkeimmät kirjalliset lähteet, jotka painottuivat pääsääntöisesti työn määrittely- ja suunnitteluvaiheessa. Lopuksi käydään läpi sähköiset lähteet, joita käytettiin etenkin työn toteutusvaiheessa.

Brainstorming oli työn alussa merkittävin tutkimusmenetelmä ja suurin tietoperustan kerryttäjä. Brainstormingia käytettiin keksimään erilaisia ratkaisumalleja, joiden avulla voisimme toteuttaa toimintoja, joita tarvitaan sovellusidean mukaiseen sovellukseen. Brainstorming oli myös menetelmä, jonka avulla saatiin paljon käyttökelpoisia kokeellisia ja työssä toteutettuja toimintoja. Esimerkkinä näistä toimi idea kaksinkertaisesta sovellusrakenteesta ja kahdesta eri tietokannasta. Tämä mahdollisti sekä sovelluksen käytön offline-tilassa että budjetin seurannan useilla laitteilla samanaikaisesti hyödyntäen pilvitietokantapalvelua. Brainstorming on menetelmänä hyvä tuottamaan ideoita, joita brainstorming-sessioon osallistuvat eivät olleet edes tulleet ajatelleeksi etukäteen (Garrett 2011, 66).

Benchmarking oli toinen tärkeä tutkimusmenetelmä työn suunnitteluvaiheessa etenkin sovelluksen rakenteen luonnin kanssa. Benchmarkingia hyödynnettiin etsimällä samantyylisiä mobiilisovelluksia sovelluskaupoista, ja niiden design- ratkaisujen pohjalta kehitimme erilaisia ideoita sovellukseemme. Esimerkiksi sovelluksen navigointi ja sen toteutus perustuivat pitkälti benchmarkingilla johdettuun tietoon standardin mukaisista navigointiratkaisuista, joita hyödynnetään myös isomman luokan mobiilisovelluksissa.

Työssä hyödynnettiin suunnitteluvaiheen päätteeksi myös vapaamuotoista vertailuanalyysia, jonka avulla pystyimme arvioimaan oman sovellusmallimme toimivuutta käyttäjäkokemuksen näkökulmasta. Kun olimme saaneet suunnitteluvaiheen sovellusmallin valmiiksi, teimme vertailua muun muassa designin toimivuudesta ja Time To Completion/Number Of Steps- lukemasta vertailemalla niitä muihin sovelluksiin. Tällä menetelmällä saimme todettua, että käyttäjäkokemuksen puolesta suunnittelemamme malli oli toimiva myös verrattuna ammattilaissovelluksiin.

Havainnointia käytettiin tutkimusmenetelmänä määrittely- ja suunnitteluvaiheessa. Havainnointia käytettiin kuvaamaan tiettyjä ilmiöitä, jotka havaittiin määrittelyn aikana. Näitä ilmiöitä käytettiin osittain suunnitellessa sovelluksen väriskeemaa ja ulkoasua.

Kirjallisuutta käytettiin työssä enimmäkseen design- puolen teoreettisuuden kasvattamiseen ja uusien ideoiden generointiin. Esimerkiksi suunnitteluvaiheen loppupuolella tehty ratkaisu sovelluksen design- periaatteen vaihdosta niin sanotusta skeuomorfisesta mallista (Baker 2017; Garrett 2011, 106) Flat design- malliin perustui kirjalliseen tietoon. Muut merkittävät kirjallisuudesta johdetut teoreettiset ideat designissa olivat Number Of Steps (Garrett 2011,

92; Banga 2014, 68) ja Time To Completion (Banga 2014, 69). Näillä mitataan sovelluksen käytettävyyttä mittaamalla sovelluksen käyttöön kuluvaa aikaa tai sen käyttöön vaadittavia painallusten määriä.

Työn teknisessä puolessa ei juurikaan hyödynnetty kirjallisuutta lähteenä. Tähän on syynä Applen vuonna 2014 iOS-ekosysteemiin tekemät muutokset, joissa iOS:n ensisijainen ohjelmointikieli vaihdettiin Objective-C:stä Swiftiin (New Gen Apps 2014). Tästä syystä johtuen kaikki löytämäni iOS-kehitykseen keskittyvät kirjat käsittelivät sovelluskehitystä Objective-C:llä, ja siitä johtuen ne eivät olleet käyttökelpoisia tähän työhön. Swift- ohjelmointikieli on muutenkin hyvin nuori ohjelmointikieli, etenkin verrattuna Javan ja C++:n kaltaisiin pitkäikäisiin ohjelmointikieliin. Tästä johtuen suurin osa siihen liittyvistä materiaaleista on pelkästään internetissä, koska kieltä kehitetään koko ajan ja Apple julkaisee siihen säännöllisesti uusia versioita ja frameworkeja.

Yksi merkittävimmistä tietoperustan osista teknisellä puolella oli Applen viralliset iOS-dokumentaatiot (Apple 2019a). Etenkin kartoittaessa iOS:n teknologioita, dokumentaatio oli erittäin hyödyllinen. Siellä oli kattavat kuvaukset eri teknologioista ja usein niiden sivut sisälsivät myös syventäviä artikkeleja ja esimerkkikoodia. Eri teknologioiden kartoittamisen lisäksi dokumentaatiota käytettiin myös selvittämään tarkemmin tuntemieni teknologioiden sisäänrakennettuja toimintoja ja niiden käyttömahdollisuuksia eri tilanteissa.

Koska työssä hyödynnettiin myös useampia kolmansien osapuolien kehittämiä teknologioita, käytettiin tietoperustana näiden teknologioiden dokumentaatioita. Dokumentaatioita käytettiin sekä referenssinä ohjelmoinnissa että perehdytysmateriaalina uusien teknologioiden kohdalla. Etenkin työn toteutusvaiheen puolivälissä nämä dokumentaatiot olivat erittäin keskeisessä roolissa tehdessä päätöksiä eri teknologioiden käyttöönotosta.

Applen omista dokumentaatioista hyödynnettiin iOS-dokumentaation lisäksi Applen designiin painottuvaa HIG(Human Interface Guideline)- dokumentaatiota (Apple 2019b). HIG painottuu nimensä mukaisesti enimmäkseen käyttöliittymän suunnitteluun ja siinä esitetään erilaisia suosituksia ja vaatimuksia, jotka tulee ottaa huomioon käyttöliittymää suunnitellessa. HIG:n huomiointi on myös erittäin olennainen tekijä työn jatkokehityksen kannalta, koska sovellus ei välttämättä läpäise App Storen laaduntarkistusta, jos sen design ei täytä HIG:n vaatimuksia (McWherter 2012, 193).

Muista työssä hyödynnetyistä teknisistä lähteistä tärkeimmäksi muodostui Stack Overflow- sivusto. Stack Overflow on ohjelmointiin keskittyvä sivusto, jossa käyttäjät voivat esittää kysymyksiä ohjelmointiin ja ohjelmoidessa kohdattuihin ongelmiin liittyen. Sivun käyttäjät voivat osaamisensa mukaan avustaa näiden kysymyksien ja ongelmien ratkomisessa. Stack Overflow

oli erittäin hyödyllinen useamman ongelman ratkaisemisessa, koska monesti jo joku toinen sivuston käyttäjä oli kohdannut vastaavanlaisen ongelman, ja täten ongelman pystyi ratkaisemaan hyödyntäen keskustelussa esitettyjä ratkaisuja.

Muina sähköisinä lähteinä työssä käytettiin YouTubea ja erinäisiä ohjelmointiin ja designiin liittyviä sivustoja, kuten Medium.comia. YouTubea hyödynnettiin etsimällä havainnollistavia esimerkkiprojekteja uusista teknologioista. Mediumin kaltaisia sivustoja taasen käytettiin laajentamaan ymmärrystä eri teknologioista ja perehtymään uusiin aiheisiin etenkin toteutusvaiheen puolella.

5 Prototyypin määrittely ja suunnittelu

Tässä osiossa käydään läpi sovelluksen suunnitteluvaihe ja sen välivaiheet. Aluksi käydään läpi määrittelyvaihe ja sen tuotokset. Tämän jälkeen käydään yksi kerrallaan lävitse sovelluksen eri osa-alueiden suunnittelu. Ensimmäiseksi käydään läpi sovelluksen visuaalisen ulkoasun suunnittelu, sen jälkeen käyttöliittymän suunnittelu ja viimeiseksi käydään läpi toimintojen suunnittelu. Osion viimeisessä kappaleessa käydään läpi vielä Marvel- ohjelmalla tehdyn rautalankamallin toteutus. Tässä osiossa esitellään myös lomittain alkuperäisen suunnitteluvaiheen vaiheiden ja tuotosten ohella jälkikäteen tehtyjä uusia iteraatioita kyseisistä vaiheista. Iteraatioita suunnitteluvaiheen jälkeen tehtiin pääasiassa erinäisten teknisten ongelmien ratkomiseksi, jotka syntyivät alkuperäisen suunnittelumallin puutteista.

5.1 Määrittelyvaihe

Määrittelyvaiheessa ensimmäisenä vaiheena oli sovelluksessa tarvittujen toimintojen määrittely. Tämän vaiheen toteutukseen käytimme toimeksiantajan kanssa brainstormingia tutkimusmenetelmä. Kuten työn alussa todettiin, sovelluksen oli tarkoituksena antaa käyttäjän määrittää itselleen budjetti, sen kategoriat, kategorioiden summat ja seuranta-ajan kaikille kategorioille. Tämä edellytti sitä, että sovelluksessa tarvitaan ainakin seuraavat toiminnot: budjetin määrittely, kategorioiden summien määrittely ja seuranta-ajan määrittely. Tämän lisäksi tarvittiin myös toiminnot muun muassa kategorioiden lisäämiselle ja poistamiselle, uusien ostosten lisäämiselle ja ostoksen summan vähentämiselle kategorioiden summista. Tukevina toimintoina tarvittiin myös budjetin tämänhetkisen rahamäärän näyttäminen ja varoituksen antaminen budjetin ylityksistä.

Koska sovelluksen sisältämää tietoa oli tarkoitus seurata pidemmällä aikavälillä, tarvittiin sovellukseen myös jonkinlainen tapa tallentaa siihen syötettyä tietoa. Sovelluksen tuli myös pystyä seuramaan sitä pyörittävän laitteen aikaa, jotta voitaisiin tarkkailla seuranta-aikojen etenemistä. Näiden toimintojen määrittelyn jälkeen lähdimme toimeksiantajan kanssa miettimään, kuinka nämä toiminnot voitaisiin toteuttaa käytännössä.

Aluksi piti määritellä rakenne sovellukselle. Toimintoja sovelluksessa oli sen verran paljon, että niitä ei pystyisi sisällyttämään yhteen välilehteen, joten päätimme jaotella toiminnot eri sivuille. Ainakin budjetin asettamiselle, budjetin tilanteen seuraamiselle ja uusien ostosten kirjaamiselle tulisi kaikille oma sivunsa. Näin saatiin toiminnot jaettua loogisesti useammalla sivulle, ja vältyttiin täyteen ahdetuilta välilehdiltä.

Näiden ideoiden perusteella aloimme jalostamaan toimintoja eteenpäin ja kehitimme alustavia malleja siitä, kuinka ne voisivat käytännössä toimia. Budjetin asetus- välilehdellä käyttäjällä olisi valmiita kategorioita, joita olisi mahdollista poistaa ja lisätä tarpeen mukaan. Tämän lisäksi tarvittiin syöttökenttä rahamäärälle ja tapa syöttää seuranta-aika. Myös ostosten kirjaamisen sivulle tarvittaisiin syöttötapa sekä ostoksen summalle että kategorialle. Budjetin seurantasivulla taasen ei olisi toimintoja, vaan siinä pystyisi pelkästään katsomaan budjetin tämänhetkistä tilannetta.

Muita hahmoteltuja sivuja tässä vaiheessa olivat muun muassa sivu, jossa näytettäisiin luettelomaisesti kaikki tehdyt ostokset ja niiden summat. Toinen hahmoteltu sivu olisi asetussivu, joka tulisi sovellukseen, jos siihen toteutettaisiin sisäänkirjautuminen. Tällä sivulla käyttäjä pystyisi muuttamaan tarvittaessa erinäisiä asetuksia liittyen tiliinsä. Eri sivujen lisäksi sovelluksessa tulisi myös olla päävalikko navigointia varten.

Koska sovelluksen perusrakenne alkoi olemaan selkeä tässä vaiheessa, päätimme seuraavaksi alkaa ideoimaan sovelluksen ulkoasua. Päädyimme ideoinnin päätteeksi pariin erityyppiseen alustavaan malliin. Ensimmäisessä mallissa oli tarkoitus olla taustakuva välilehtien taustana, ja painikkeet esitettäisiin symbolien ja tekstin yhdistelmänä. Toisaalta kehitimme myös pelkistetymmän ratkaisun, jossa taustana olisi taustakuva, mutta painikkeet olisivat yksinkertaisia ja ne olisivat väritetty taustakuvan värien erilaisilla sävyillä. Näin ne erottuisivat taustasta, mutta olisivat kuitenkin samaan aikaan samaa värityyppiä, antaen kokonaisuudelle yhtenäisen vaikutelman.

Tässä vaiheessa olimme saaneet tuotettua alustavia ideoita sekä myös kehitettyä erilaisia teemoja visuaaliseen ulkoasuun, joten päätimme siirtyä suunnitteluvaiheeseen. Määrittelyvaiheessa saimme kuitenkin myös jonkin verran kokeellisia toimintoja keksittyä, jotka kirjasimme ylös mahdollista jatkokehitystä varten. Näihin kuului muun muassa idea käyttää kännykän kameraa skannaamaan ostoksia suoraan kuiteista ilman erillistä manuaalista syöttämistä sovellukseen.

Toinen esimerkki kokeellisista ideoista oli integroida sovellus pankkien järjestelmiin, josta saataisiin suoraan ostosten tiedot sovellukseen. Näiden toimintojen tarkempi suunnittelu jätettiin kuitenkin jatkokehitystä varten, koska ne olisivat erittäin haastavia toteuttaa. Pankkitilien linkitykseen liittyvän idean kohdalla tarvittaisiin myös yhteistyötä pankkien kanssa.

5.2 Suunnittelu – Visuaalisuus

Määrittelyvaiheen jälkeen siirryimme suunnittelemaan sovellusta ideoinnin tuotosten perusteella. Aloitimme toimeksiantajan kanssa työn suunnittelun visuaalisesta suunnittelusta. Visuaalisuuden suunnittelu koostui pääasiassa sovelluksen väriskeemasta (Informed Illustrator 2012), koska suunnittelun aikana päätimme kehittää sovelluksen ulkoasun minimalistiseksi. Värivalinnat nousivat tärkeimmäksi elementiksi, koska flat design- tyyppisessä ratkaisussa on haastavaa saada monimutkaisia visuaalisia elementtejä designiin. Flat designin periaatteena on luoda käyttöliittymiä, jotka käyttävät pääasiassa yksinkertaisia elementtejä ja kirkkaita värejä (Interaction Design Foundation).

Edellämainituista syistä oli tärkeää, että sovellukseen valittiin oikeanlainen väriskeema, jolla oli mahdollista saada luotua identiteetti sovellukselle. Tehdessämme benchmarkingia eri värivaihtoehdoista, havaitsin, että väriskeemoja oli tyypeiltään karkeasti kahta erilaista tyyppiä. Ensimmäinen tyyppi oli komplementoiva skeema, jossa kaikki sovellukset värit ja sävyt olivat yhtä väriä ja tämän värin komplementoivia värejä. Toinen skeematyyppi oli palettiskeema, jossa värit eivät keskittyneet yhden värin ympärille, vaan siinä saattoi olla samanaikaisesti sekä komplementoivia että vastakkaisia sävyjä. Edellä mainitut käsitteet eivät ole vakiintuneita käsitteitä, vaan ne perustuivat työn aikana tekemiini havaintoihin.

Minulla ja toimeksiantajalla ei ollut aluksi selkeää visiota sovelluksen potentiaalisesta väriskeemasta ja ulkoasusta. Päätimme lähteä etsimään inspiraatiota ulkoasulle Benchmarkingin avulla. Benchmarkingin tavoitteena oli löytää hyviä väriskeemoja, joita voisimme käyttää alustamaan työn värimaailmaa.

Benchmarkingin lopuksi päädyimme valitsemaan alustavaksi väriskeemaksi Premier Leaguen (PL) virallisen mobiilisovelluksen värit, koska sen väriskeema koettiin kokonaisuutena hyvin toimivaksi. PL- sovelluksen värit olivat palettiskeema- tyyppisiä, eli ne edustivat useampaa eri sävyä väripyörän eri kohdista (Draw Paint Academy 2018). Niiden vahvuutena oli sopiva kirkkaus, eli ne eivät ollet liian häiritseviä, mutta ne olivat kuitenkin riittävän värikkäitä, jotta ne olisivat miellyttäviä silmille pidemmänkin käytön aikana. Työn kehityksessä niiden monimuotoisuus salli myös oman identiteetin luonnin jokaiselle välilehdelle, sillä jokaiselle sivulle voitiin antaa oma värinsä paletin väreistä.

Alkuperäisen design-mallin värit perustuivat suoraan Premier Leaguen sovelluksen väreihin, joten päätimme, että työn loppupuolella palaisimme vielä uudestaan väriskeeman suunnitteluun. Emme pitäneet hyvänä loppuratkaisuna designille, että se perustuisi suoraan toisen tuotteen väreihin. Tämä hankaloittaisi pidemmällä aikavälillä sovelluksen oman brändin luontia ja identiteetin syntyä.

Kun alustava väriskeema oli saatu valittua, ryhdyimme seuraavaksi suunnittelemaan skeeman implementointia hyödyntäen erilaisia visuaaliseen designiin liittyviä materiaaleja. Näiden avulla identifioimme lopulta 2 tärkeää käsitettä, joiden ympärille sovelluksen lopullinen ulkoasu rakennettiin. Ensimmäinen näistä oli sovelluksen ulkoasun design-periaate, jonka pohjalta ulkoasua lähdettiin kehittämään. Aluksi päädyimme käyttämään skeuomorfisesta mallia, jonka perusperiaatteena on tehdä visuaalisesti laajoja visuaalisia ulkoasuja, joihin kuuluu muun muassa varjojen, monimuotoisten taustojen ja monimutkaisten elementtien käyttö (Banga 2014, 106).

Skeuomorfinen malli todettiin kuitenkin ensimmäisen rautalankamallin toteutuksen yhteydessä toimimattomaksi sen monimutkaisen ulkoasun vuoksi. Malli oli sekä hankala toteuttaa että se oli myös häiritsevä käyttää pidemmän päälle. Tämän vuoksi päätimme lopulta design-periaatteen valinnassa tehdä päinvastaisen ratkaisun. Päätimme, että pyrimme luomaan minimalistisen design-mallin. Valitsin design-periaatteeksi Flat designin, jonka periaatteena on luoda mahdollisimman yksinkertainen ulkoasu käyttäen yksivärisiä taustoja ja pelkistettyjä visuaalisia elementtejä.

Design- periaatteiden lisäksi toinen tärkeä lähde visuaalisuuden ja väriskeeman luonnissa oli HIG(Human Interface Guideline), joka on Applen designiin painottuva dokumentaatio. Siinä on sekä suosituksia että vaatimuksia siitä, miten sovelluksia tulisi suunnitella iOS:lle. Tämän lisäksi siinä on myös ohjeita siitä, kuinka tiettyjä elementtejä, kuten navigoinnin Back-painiketta, tulisi hyödyntää käyttäjäkokemuksen parantamisessa sovelluksessa (Apple 2019e). Kaikkia HIG:n ohjeistuksia ei ole tarvitse noudattaa täsmällisesti, mutta jos sovellus kokonaisuutena rikkoo liian monia sen käytäntöjä, saatetaan sovelluksen lataaminen App Storeen estää lataamista edeltävässä tarkistusprosessissa (McWherter 2012, 193).

HIG:ta hyödynnettiin sovelluksen väriskeeman ideoinnissa työn lopussa. Olin käyttänyt HIG:ta työn alkuperäisessä suunnitteluvaiheessa navigoinnin ja käyttöliittymän suunnittelun kanssa. Palatessamme työn lopuksi suunnitteluun päätin myös tutustua HIG:n suosituksiin liittyen sovellusten väreihin ja niiden käyttöön. Tämä oli erittäin hyödyllistä, koska HIG:ssa oli hyviä suosituksia värien käytöstä sovellusdesignissa. Parhaimmat näistä, joita päädyttiin myös hyödyntämään myöhemmin, olivat ehdotus rakentaa sovelluksen väriskeema logon sävyjen ympärille (Apple 2019d) ja ehdotus käyttää ensisijaisesti komplementoivia värejä (Apple 2019d),

koska niiden avulla saatiin yhtenäinen ulkoasu, joka ei veisi huomiota sovelluksen toiminnoista.

Flat designin vuoksi päätimme lopulta, että sovelluksessa olisi loogisinta olla hyvin minimalistinen visuaalisuus. Koska Flat designissa kaiken on tarkoitus olla yksinkertaista ja yksiväristä (Interaction Design Foundation), oli täten loogista, että väriskeema olisi myös yksinkertainen. Tätä periaatetta noudattaen päätimme myös, että sovellukseen ei tule myöskään mitään ylimääräisiä visuaalisia elementtejä sovelluksen logoa lukuun ottamatta. Tämä oli osasyy myös siihen, minkä vuoksi päätimme luopua Premier Leaguen sovelluksen väriskeeman käytöstä lopulta.

Vaikka Premier Leaguen väriskeema oli hyvä ja monipuolinen, sen värien käyttö oli kuitenkin melko haastavaa, koska eri värejä oli niin monta. Tämä antoi sovelluksen ulkoasusta mielestämme kaoottisen vaikutelman. Tämän vuoksi päätimme, että tulisimme käyttämään HIG:n suosittelemaa mallia, jossa sovelluksen väriskeema perustuu sovelluksen logon värejä komplementoivaan sävymaailmaan (Apple 2019d). Näin saatiin sovellukselle yhtenäinen ulkoasu, joka on etenkin sovelluksessa navigoidessa käyttäjäkokemusta parantava tekijä.

Vaikka tämän työn tavoitteena oli ensisijaisesti suunnitella sovellus ja sen toiminnot, sivusimme myös suunnittelussa jonkin verran brändin luontia. Työn toimeksiantaja, jolla oli meistä vahvempi ymmärrys ja osaaminen brändäämisestä, oli päävastuussa tämän vaiheen ideoinnissa. Hänen näkemystensä mukaan sovelluksen ulkoasulla oli mahdollista luoda brändiä 2:lla eri tekijällä: logolla ja väriskeemalla.

Sovellukselle luotiin alustava prototyyppi logosta. Koska sovelluksen muussa designissa oli hyvin vahva minimalistinen tyyli, oli sen vuoksi myös sovelluksen logon oltava hillitty, jotta se ei erottuisi liikaa muusta ulkoasusta. Päätimme ideoinnin päätteeksi luoda sovelluksen alustavaksi logoksi pienen m- kirjaimen, joka viittasi sovelluksen Mon- nimeen. Logon väriksi valittiin vaaleanvihreä, koska se toimi logon muodon kanssa kaikkein parhaiten kokeilemistamme värivaihtoehdoista.

Logo yritettiin asettaa osaksi sovellusta työn lopussa, mutta toistaiseksi tuntemattomasta bugista johtuen tämä ei onnistunut. Logo toteutettiin Adobe Photoshop kuvankäsittelyohjelmalla, ja se siirrettiin Photoshopista png- formaatissa. Png- formaatista huolimatta logolla oli selkeä erottuva tausta, joka ei ollut läpinäkyvä. Logo jouduttiin täten jättämään toistaiseksi pois työstä.

Logon värin valinta oli hyvin tärkeä tekijä brändäyksen luonnin suhteen. Päätimme työn lopussa lähteä tekemään uutta väriskeemaa sovellukselle HIG:n suositusten mukaan. Valitsimme

aluksi Xcoden Color Picker- työkalulla alustavan logon värin heksakoodin. Sen jälkeen valikoimme hyödyntäen Color Picker- työkalun slidereita muita samankaltaisia, mutta toisistaan erottuvia, vihreän sävyjä. Niistä päätimme rakentaa sovelluksen uuden väriskeeman ja yksilöllisen ulkoasun. Käyttämällä vihreän eri sävyjä logon lisäksi välilehtien taustoissa, painikkeissa ja tietyissä teksteissä, saimme luotua sovellukselle yhtenäisen teeman, mikä oli erittäin tärkeää etenkin työn jatkokehitykselle.

5.3 Suunnittelu – Käyttöliittymä

Tässä osiossa käydään läpi työn suunnitteluvaiheen osa, jossa suunniteltiin sovelluksen käyttöliittymä. Osiossa käydään aluksi suunnittelun tukena käytetty teoreettinen tietoperusta. Sen jälkeen käydään läpi jokaisen sovelluksen välilehden suunnittelu sivu kerrallaan ja tämän lisäksi käydään läpi myös navigointiin ja painikkeiden sijoitteluun liittyvät ratkaisut. Lopuksi käydään läpi erinäisiä pieniä käyttäjäkokemusta parantavia tekijöitä, joita sovellukseen suunniteltiin tämän vaiheen aikana.

Yksi tärkeimmistä teoreettisista tekijöistä käyttöliittymän suunnittelussa oli web-kehityksen puolelta tuttu 3-click Rule- eli kolmen klikkauksen sääntö. Kolmen klikkauksen säännöllä tarkoitetaan sitä, että verkkosivun sisällä mihin tahansa paikkaan sivulla pitäisi päästä maksimissaan kolmella klikkauksella (Tanzon-Corre). Toinen samankaltainen käsite oli Number Of Steps. Siinä käyttöliittymän toimivuutta mitataan laskemalla klikkausten tai muiden käyttöliittymätoimintojen (esim. tekstin syöttäminen) lukumäärä, jota tarvitaan yhden tehtävän suorittamiseen sovelluksessa. Pidin näitä kahta käsitettä hyvänä viitteenä suunnittelulle. Navigointirakennetta luodessa päätin tarkkailla klikkausten määrää ja muita välivaiheita, joita tarvittiin sovelluksen käyttämiseen.

Kolmen klikkauksen säännön lisäksi toinen olennainen teoreettinen käsite, jota käytettiin käyttöliittymän suunnittelussa, oli Time To Completion. Time To Completion on samankaltainen käsite kuin kolmen klikkauksen sääntö, koska myös siinä mitataan, kuinka helppoa sovelluksen käyttäminen on. Time To Completionilla mitataan välivaiheiden tai askelien sijaan aikaa, joka käyttäjällä kuluu suorittaessa yhden toiminnon sovelluksen sisällä. Tämä oli hyvin olennainen tekijä huomioida. Sovelluksen sisällä on mahdollista liikkua vain muutamalla painalluksella, mutta esimerkiksi liian täyteen ahdestusta käyttöliittymästä johtuen voi yhden toiminnon suorittamiseen kulua kuitenkin merkittävästi aikaa. Tämä on käyttäjäkokemukselle huono asia, joten tästä johtuen Time To Completionin tarkkailu oli hyvin tärkeää.

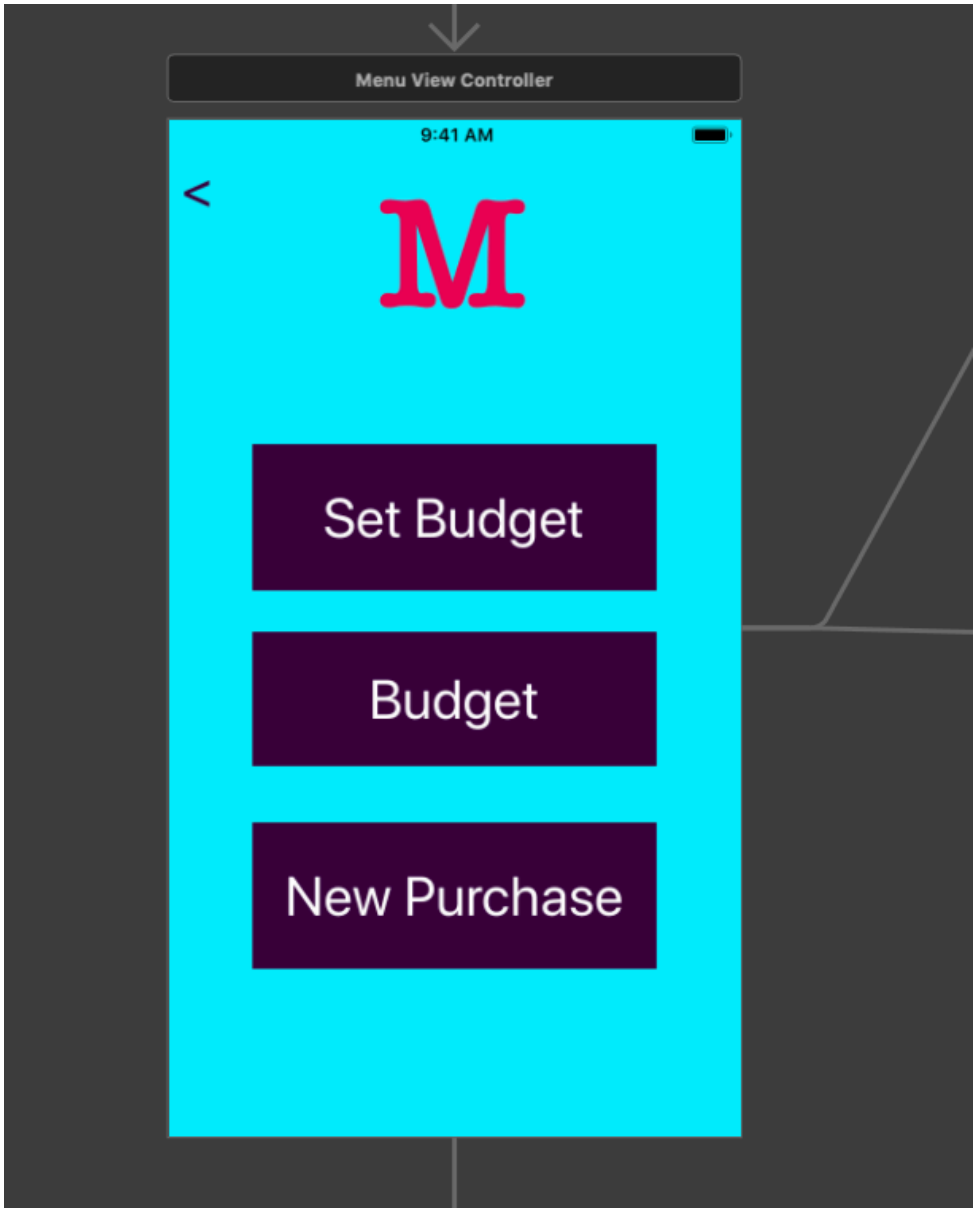
Edellä mainittujen tekijöiden lisäksi hyödynsimme käyttöliittymän suunnittelussa myös tutkimustietoa, jota oli kerätty mobiilidesigniin painottuvissa tutkimuksissa. Näistä tärkeimpänä pidin tietoa sovellusten keskimääräisestä käyttöajasta. Erään tutkimuksen mukaan keskiverto uuden sovelluksen ladannut käyttäjä käyttää sovellusta alle 5 minuuttia kerralla (Banga 2014,

159). Tämän vuoksi oli erittäin tärkeää tehdä sovelluksen käyttöliittymästä niin selkeä, että käyttäjä pystyisi omaksumaan sen nopeasti.

Viimeisenä merkittävä lähteenä käyttöliittymän suunnittelussa käytettiin HIG:ta. HIG:n merkitys korostui varsinkin navigoinnin suunnittelussa ja painikkeiden sijoittelussa. HIG:ssa suositellaan, että navigointiin liittyvien painikkeiden pitää selkeästi informoida käyttäjälle, mihin sovellus siirtyy painiketta painaessa ja saman painikkeen pitäisi aina tilanteesta riippumatta toteuttaa sama siirtymä (Apple 2019e). Tämä oli merkittävä tekijä työn teknisen toteutuksen aikana tehdessä päätöstä siirtymisestä Navigation Controller- teknologian käyttöön. Navigation Controllerin avulla oli erittäin helppoa toteuttaa navigointi siten, että sovellus automaattisesti informoi käyttäjää siitä, mille sivulle siirrytään Back- painiketta painaessa.

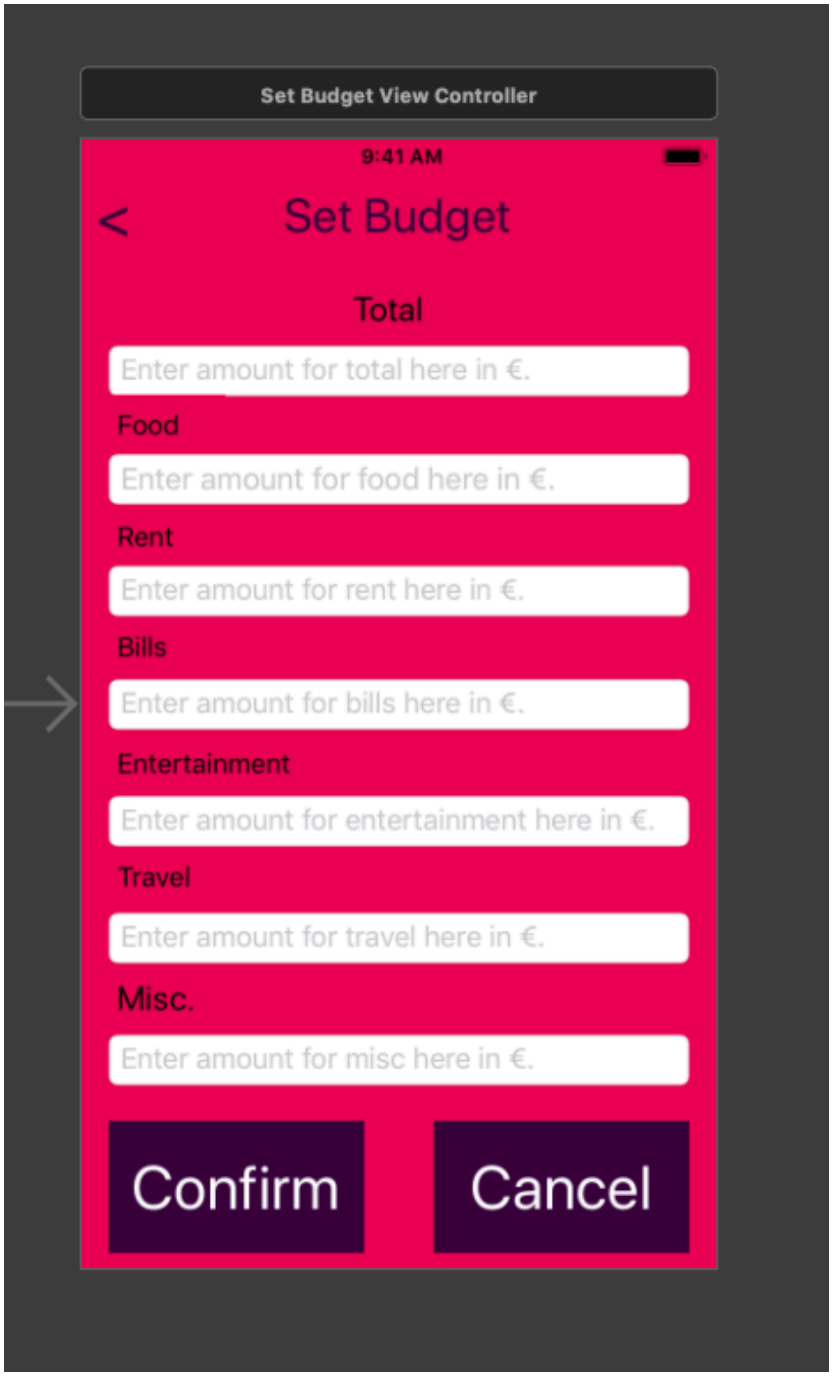
Toinen HIG:n olennaisimmista suosituksista liittyi painikkeiden sijoitteluun. HIG suosittelee, että painikkeet sijoitetaan ja skaalataan riittävän isoiksi, että niitä on helppo painaa riippumatta siitä, miten käyttäjä pitää laitetta käsissään (Apple 2019f). Tämän vuoksi sovelluksen pienemmät painikkeet sijoitettiin laitoihin, jotta niihin ylettyy myös käytettäessä laitetta yhdellä kädellä. Keskelle laitetuista painikkeista tehtiin sen verran leveitä, että niitä on mahdollista painaa myös laitetta pitävän käden sormilla.

Aloitimme käyttöliittymän suunnittelun sovellusrakenteen ja välilehtien suunnittelusta. Huomioiden Number Of Steps/3-Click Rule- periaatteet, päätimme, että sovelluksessa täytyy olla matala rakenne, jotta sen sisällä pystyisi liikkumaan vain muutamalla painalluksella. Tämän vuoksi päädyimme tekemään sovelluksen rakenteen menu-pohjaisesti. Sovellukseen tehtiin päävalikko, josta pystyy siirtymään kaikkiin välilehtiin. Välilehdistä taasen pystyisi siirtymään takaisin päävalikkoon Back- painikkeen avulla.



Kuvio 1: Päävalikko Xcoden Storyboardissa.

Tämä malli toimi erinomaisesti huomioiden asetetut tavoitteet navigoinnille. Välilehdet taasen jaettiin kolmeen eri välilehteen, joista jokainen vastasi yhdestä toiminnosta, jotka määriteltiin ideointivaiheessa. Ensimmäinen näistä oli Set Budget- välilehti, jossa käyttäjä määrittää budjetin ja sen kategoriat. Toinen näistä oli Budget- välilehti, jossa käyttäjä voi tarkkailla budjettinsa tämänhetkistä tilannetta. Kolmas näistä oli New Purchase- välilehti, jossa käyttäjä kirjaa uudet ostokset sovellukseen. Edellämainitut termit olivat myös jokaisen painikkeen tekstinä, jotta niistä pystyisi nopeasti päättelemään, mille sivulle kukin painike tekisi siirtymän.

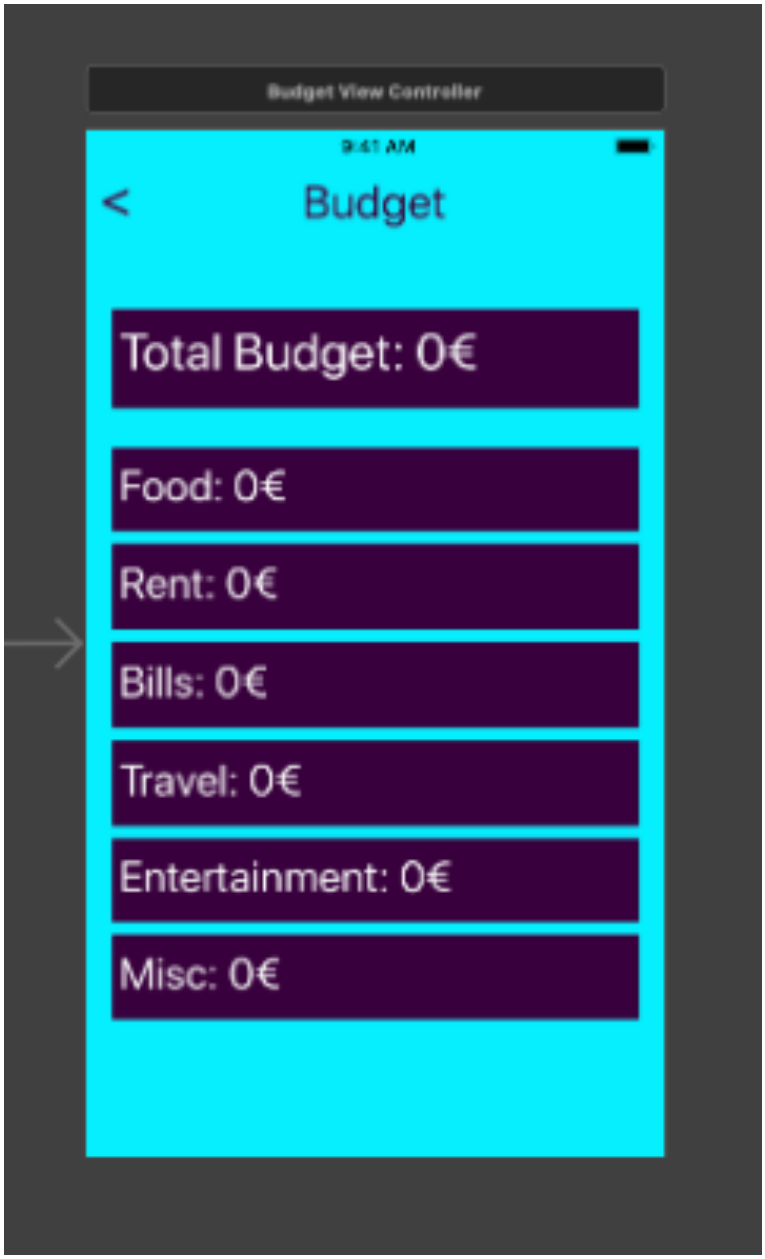


Kuvio 2: Set Budget- välilehti Xcoden Storyboardissa.

Yksittäisten välilehtien rakenteet koitettiin pitää yksinkertaisina, jotta sovelluksen käyttöönotto olisi helppoa ja Time To Completion- lukema pysyisi alhaisena. Set Budget- välilehdessä oli otsikko kategorialle, ja sen alla oli sitä vastaava syöttökenttä. Syöttökenttiä oli valmiiksi asetettuna yhteensä 7, joista yksi oli budjetin kokonaissummalle ja loput 6 olivat valmiiksi asetetuille kategorioille. Sivun alalaidassa olivat painikkeet sekä budjetin tallennukselle että

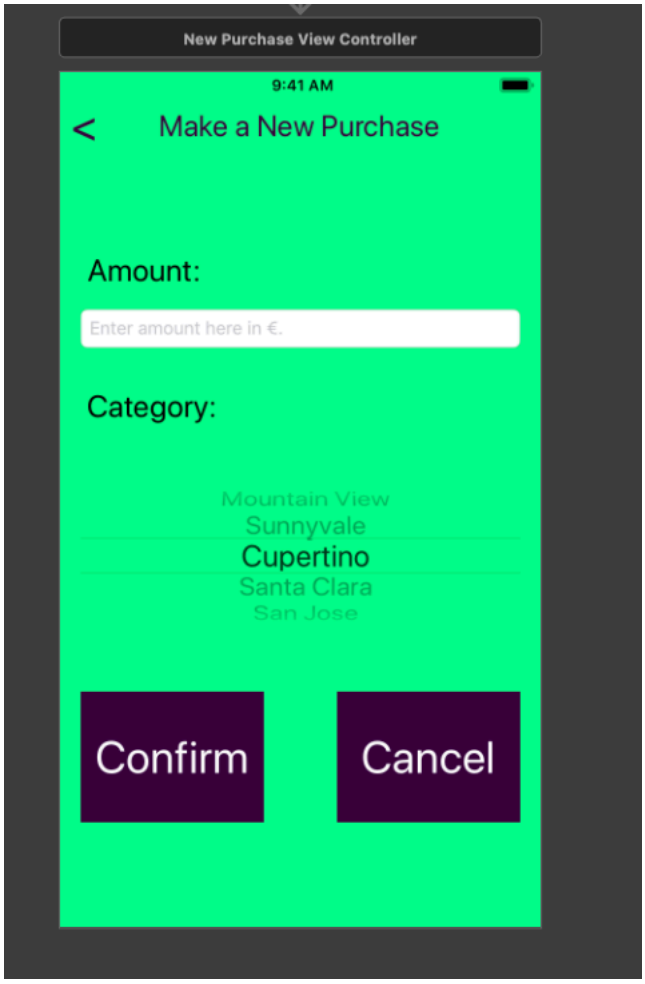
syöttökenttien tyhjentämiselle. Painikkeet uusien kategorioiden lisäämiselle, olemassa olevien kategorioiden poistamiselle ja seuranta-ajan lisäämiselle päätettiin toteuttaa myöhemmin.

Dynaamisten toimintojen osalta päätin tehdä selvityksen kahdessa osassa. Aluksi testaisin tämän toteutusmallin toimivuuden staattisena versiona ja sen jälkeen tekisin selvityksen dynaamisuuden lisäämisestä. Tässä mallissa tarkoituksena olisi lisätä kategorian summan syöttökentän alle toinen syöttökenttä, johon käyttäjä syöttäisi seuranta-ajan. Kategorioiden lisäämistä varten tulisi lisätä esimerkiksi sivun ylälaitaan erillinen painike tälle, jossa käyttäjä lisää kategorian nimen. Tämän jälkeen sivulle generoituisi dynaamisesti syöttökentät uuden kategorian tiedoille, samalla lailla kuin valmiiksi asetetuissa kategorioissa. Tässä mallissa oli vielä epäselvyyksiä teknisen toteutuksen puolesta, joten sen vuoksi siitä tehtiin pelkistetty versio sovelluksen prototyyppiin.



Kuvio 3: Budget- välilehti Xcoden Storyboardissa.

Budget- välilehti taasen oli yksinkertaisempi. Koska siihen ei tarvittu mitään erillisiä toimintoja, oli siinä sen vuoksi vain Back- painike päävalikkoon palaamista varten. Tämän lisäksi sivulla oli vain tekstikentät budjetin ja sen kategorioiden summien näyttöä varten. New Purchase- sivulla oli taasen 1 syöttökenttä ostoksen summaa varten ja Picker View- valintarulla ostoksen kategoriaa varten. Näiden lisäksi sivulla oli Confirm- ja Cancel- painikkeet ostoksen hyväksymistä ja syötettyjen arvojen nollausta varten.



Kuvio 4: New Purchase- välilehti Xcoden Storyboardissa.

Toteutusvaiheessa päätin toteuttaa sovellukseen kaksinkertaisen sovellusrakenteen, joka mahdollistaisi tiedon tallennuksen sekä paikallisesti että pilvipalvelun avulla. Tämä lisäsi sovellukseen yhden navigaatiokerroksen lisää, koska sovelluksen käynnistyessä käyttö alkaisi aloituspäävalikosta, jossa käyttäjä voi siirtyä rekisteröintisivulle, kirjautumissivulle tai paikallisen mallin päävalikkoon. Rekisteröinti/kirjautumisvaiheen jälkeen sovelluksen rakenne oli identtinen paikalliseen version rakenteeseen.

Välilehtien suunnittelun jälkeen suunniteltiin sovelluksen navigointirakenne. Myös navigoinnin tuli olla yksinkertainen, jotta Time To Completion ja Number Of Steps- lukemat saataisiin pidettyä alhaisina. Sovelluksessa eteenpäin liikkuminen tapahtui pääasiassa päävalikkojen painikkeiden kautta. Taaksepäin liikkuminen taasen tapahtui <- ja Back- merkittyjen painikkeiden kautta. Tämä malli toimi muuten hyvin, mutta lisätessä sovellukseen rekisteröinti/kirjautuminen- vaiheen ja uloskirjautumis-toiminnon, jouduttiin miettimään vaihtoehtoista ratkaisua. Tällä navigointimallilla ei pystyttäisi uloskirjautuessa siirtymään takaisin päävalikkoon, vaan siirtymä tapahtuisi kirjautumissivuun. Tämän ongelman kiertämiseksi otettiin käyttöön

Navigation Controller, jonka avulla oli mahdollista toteuttaa siirtymä suoraan aloituspäävalikkoon. Navigation Controllerin avulla pystyi myös korvaamaan käsin tehdyt Back-painikkeet ja sen avulla saatiin navigointipainikkeiden ulkoasu iOS- standardien mukaiseksi ilman erillistä konfigurointia. Täten Navigation Controller päätettiin ottaa käyttöön lopulta koko sovellukseen.

Tämän jälkeen tuli vielä suunnitella painikkeiden ja muiden visuaalisten elementtien asettelu sovelluksessa. Päävalikoissa päätin standardinomaisesti sijoittaa painikkeet allekkain keskelle näyttöä. Painikkeista tehtiin sen verran leveitä, että niitä oli mahdollista painaa myös käytettäessä laitetta yhdellä kädellä. Set Budget- ja New Purchase-sivuilla käytettävät Confirm- ja Cancel-painikkeet sijoitettiin näytön alalaitaan, jotta niitä pystyisi painamaan mahdollisimman helposti. Back- painikkeet sijoitettiin vasempaan yläkulmaan, missä ne tavallisesti ovat iOS- sovelluksissa. Tärkein syy iOS-standardien seuraamiseen painikkeiden sijoittelussa oli sovelluksen käyttöönoton helpottaminen käyttämällä käyttäjille tuttuja navigointiratkaisuja (Banga 2014, 174).

Syöttökentät Set Budget- ja New Purchase- sivuissa levennettiin koko näytön leveydelle, jotta niiden painaminen olisi mahdollisimman helppoa. New Purchase- sivussa taasen ostoksen kategorian valikointiin käytettävä Picker View- valintarulla levennettiin myös koko näytön leveydelle. Tällä varmistettiin, että sen käyttäminen olisi mahdollisimman helppoa.

Lopuksi suunnittelin vielä tässä vaiheessa muutamia toimintoja käyttöliittymään, jotka parantsivat käyttäjäkokemusta. Ensimmäinen näistä oli placeholder- tekstit, jotka laitettiin syöttökenttiin kertomaan käyttäjälle, mitä niihin pitää syöttää ja missä muodossa. Toinen tekijä oli yleinen tyylittely, johon kuului muun muassa Spotify- tyyliset soikeat painikkeet. Näiden avulla sovelluksen ulkoasusta saataisiin kokonaisuutena modernimpi ja ammattilaisempi. Niiden toteutus jätettiin kuitenkin rajausten vuoksi jatkokehitykseen.

5.4 Suunnittelu – Toiminnallisuus

Tässä osiossa käydään läpi suunnitteluvaiheen viimeinen osio, jossa suunniteltiin sovelluksen toiminnot ja toiminnallisuus. Aluksi tässä osiossa käydään lyhyesti läpi tietoperusta, jota hyödynnettiin toiminnallisuuden suunnittelussa. Sen jälkeen käydään läpi yksi kerrallaan sovelluksen toimintojen suunnittelu. Lopuksi käydään läpi toteutettavien toimintojen rajaus ja myös työn loppuvaiheessa tehty uusi sovellusmalli. Tämän mallin tavoitteena oli korjata dynaamisiin toimintoihin liittyvät ongelmat alkuperäisessä mallissa.

Tietoperusta perustui pitkälti Time To Completion- käsitteen ympärille tässä vaiheessa. Tavoitteena toimintoja suunnitellessa oli saada jokainen sovelluksen sisäinen toiminto toteutet-

tua mahdollisimman yksinkertaisesti, jotta Time To Completion- lukema pysyisi mahdollisimman pienenä. Muuten tietoperustana käytettiin tässä osiossa pääsääntöisesti omaa tietämystäni iOS-kehityksestä, jonka avulla oli mahdollista tehdä päätöksiä parhaista mahdollisista teknisistä ratkaisuista.

Aloitin välilehtien toimintojen suunnittelusta. Koska Budget- sivu oli vain budjetin tarkkailua varten, keskityttiin tässä vaiheessa pääasiassa Set Budget- ja New Purchase- välilehtiin. Budget- sivun ainoa dynaaminen ominaisuus päivittyvien arvojen lisäksi oli dynaamiset kategoriat, joiden tekstikentät sijoitettaisiin allekkain valmiiksi määritettyjen kategorioiden tekstikenttien alle. Set Budget- välilehden toiminnoiksi kuuluivat budjetin kategorioiden summien määrittely, seuranta-ajan määrittely ja uusien kategorioiden lisääminen. Kuten edellisessä osiossa mainittiin, päätettiin tämä toteuttaa luomalla erilliset syöttökentät summille ja seuranta-ajoille. Uudet kategoriat lisättäisiin erillisen pluspainikkeen kautta, jota painaessa käyttäjä määrittää kategorian nimen ja sen jälkeen sivulle generoituu dynaamisesti kategorian syöttökentät.

En ollut kuitenkaan täysin varma tämän mallin toimivuudesta teknisen toteuttamisen näkökulmasta, joten päätin, että aluksi luodaan vain kategorioiden otsikot ja summien syöttökentät. Tähän oli syinä se, että päivämäärien lukeminen syöttökentistä todettiin haastavaksi ja dynaamisten kategorioiden lisääminen ei ollut täysin selvää teknisen totetuksen puolesta. Päätin täten aluksi luoda pelkistetyn version sivusta. Jos teknisen analyysin päätteeksi malli todettaisiin tomivaksi, niin sen jälkeen sivut muokattaisiin suunnitellun mallin mukaisiksi.

Toimintojen suunnittelun jälkeen siirryttiin suunnittelemaan tiedon tallentamista ja tiedon käsittelyä tietokannoilla. Tietokantojen toiminta kohdistuisi toimintojen toteuttamista varten kaikille sovelluksen välilehdille. Set Budget- välilehdellä tietokantojen tulisi ottaa vastaan ja tallentaa käyttäjän syöttämät tiedot Confirm- painikkeen painalluksen yhteydessä. Budget-välilehdellä tietokannan tulisi ladata tallennetut tiedot ja näyttää ne halutussa formaatissa. New Purchase- välilehdellä tietokantojen tulisi tallentaa tiedot kirjatuista ostoksista ja kategoriatietoa hyödyntäen poistaa syötetty summa kohdistetuista kategorioista.

Aluksi suunnittelin sovelluksen niin, että siinä olisi vain paikallinen tietokanta, eli tietokanta tallentaisi tietonsa sovellusta pyörittävän laitteen muistiin. Työn puolenvälin jälkeen päätin kuitenkin, että työhön tulisi toteuttaa kaksinkertainen sovellusrakenne, jossa olisi paikallisen tietokannan lisäksi myös pilvitietokantapalvelu. Tämän myötä sovellukseen lisättiin aloitus-päävalikko, rekisteröinti- ja kirjautumissivu. Rekisteröinti/kirjautumissivujen jälkeiselle puolelle lisättiin samat välilehdet kuin paikallisen tietokannan puolelle.

Perustelut tämän päätöksen tekemiselle oli vaihtoehtojen ja lisäarvon tarjoaminen sovelluksen käyttäjille. Kahden tietokannan mallilla pystyttiin tarjoamaan käyttäjälle mahdollisuus käyttää sovellusta sekä offline-tilassa että verkkoyhteyden kautta. Tämän lisäksi käyttäjä voisi myös käyttää sovellusta samaan aikaan usealla laitteella, koska pilvitietokannan tiedot siirtyvät laitteelta toiselle reaaliaikaisesti pilven kautta.

Toiminallisuuden suunnittelusta oli tämän jälkeen vielä jäljellä satunnaisten toimintojen suunnittelu, jotka pyrkivät käyttäjäkokemuksen parantamiseen. Ensimmäinen näistä toiminnoista oli Alert-popupit. Alert- popupikkunat ovat yleinen iOS-sovelluksissa käytetty toiminto. Ne ovat väliaikaisia ikkunoita, jotka ilmestyvät ruudulle jonkin tietyn toiminnon yhteydessä. Alerteja käytettiin tässä sovelluksessa tiedottamaan käyttäjälle onnistuneesta ostoksen kirjaamisesta ja epäonnistuneesta kirjautumisesta/rekisteröitymisestä.

Alertien lisäksi toinen tärkeä tekijä oli tutkia Time To Completion- lukeman minimoimista. Testatessamme toimeksiantajan kanssa sovelluksen ensimmäistä toimivaa prototyyppiä totesimme, että Time To Completion- lukeman minimointi oli onnistunut hyvin. Etenkin sovellukseen kirjautuminen, rekisteröinti, navigointi ja ostosten kirjaaminen toimivat nopeasti. Ainoa hitaampi toiminto oli Set Budget- sivun kategorioiden täyttäminen.

Isoimpana ongelmana oli Set Budget- välilehti, jossa Number Of Steps- lukemalla tarkisteltuna saattoi kulua hyvinkin paljon aikaa. Jos käyttäjä tarvitsisi laajaa budjettiseurantaa, valmiiksi määritellyt kategoriat olivat erittäin hyödyllisiä. Jos taas käyttäjä ei tarvitse laajaa seurantaa, niin silloin valmiiksi määritellyt kategoriat olivat enimmäkseen haitaksi. Sivulla kaikkiin syöttökenttiin joutui käytettyjen teknologioiden vuoksi syöttämään jokin arvon, koska kokonaan tyhjiä arvoja ei pystytty hyväksymään koodissa.

Tässä mallissa oli myös käytössä vain yksi syöttökenttä, lopullisessa mallissa olisi tarkoitus myös olla toinen syöttökenttä päivämäärää varten. Tämä laajentaisi sekä Time To Completionia, että Number Of Stepsiä. Päätin tästä johtuen aloittamaan vaihtoehtoisten ratkaisumallien kartoittamisen Set Budget- sivun toiminnoille, sillä alkuperäinen malli vaati liikaa välivaiheita.

Kun toimintojen suunnittelu oli saatu valmiiksi, ryhdyin sen jälkeen rajaamaan toimintoja teknistä toteutusta varten. Sovellukseen kehitettiin paljon erilaisia kokeellisia ominaisuuksia ja edistyneitä ominaisuuksia, joten tämän vuoksi oli tarpeellista rajata toimintoja niin, että ne oli realistista saada toteutettua työn aikataulun puitteissa (Stevens 2011, 141). Päätin lopulta keskittyä toteutuksessa olennaisimpiin toimintoihin, joita pystyttäisiin käyttämään jatkokehityksessä ilman lisätyöstöä. Sovelluksen toiminnoista päätettiin toteuttaa sovelluksen rakenne, välilehdet, navigointi, tiedon syöttö tekstinsyöttökenttiin ja syöttökenttien lukeminen. Tämän

lisäksi tulisi myös toteuttaa tietokantojen alustus ja tiedon tallentaminen kantoihin. Tämän lisäksi tarkoituksena oli myös tutkia dynaamisten toimintojen toteuttamista.

Työn lopussa tulin siihen johtopäätökseen, että alkuperäisen mallin mukainen dynaamisuus ei ollut mahdollista toteuttaa. Ryhdyin sen vuoksi suunnitelemaan uutta toteutusmallia, jonka avulla olisi mahdollista toteuttaa dynaamiset toiminnot. Ongelmat tulivat Set Budget- sivun uusien kategorioiden lisäämisestä ja Budget- sivun tekstikenttien lisäämisestä. Näiden mainittujen asioiden dynaaminen generointi ei ollut mahdollista toteuttaa toivotulla tavalla, joten päätin tämän vuoksi suunnitella molemmat sivut kokonaan uusiksi.

Set Budget- sivun valmiiksi asetetut kategoriat poistettiin kokonaan. Niiden sijaan sivu jätetiin tyhjäksi ja sivun ylälaitaan asetettiin pluspainike, jota painamalla käyttäjä saisi asetettua uuden kategorian. Kategorian asetuksessa määriteltiin kerralla kategorian nimi, rahamäärä ja seuranta-aika. Asetetut kategoriat ilmestyisivät tyhjään tilaan, jossa niitä voisi koskettamalla muokata. Näytön alalaidassa oli Confirm- painike, joka kirjaisi kaikki asetetut kategoriat budjettiin.

Budget- välilehdellä olisi 2 erilaista ratkaisumallia sille, kuinka siinä olevaa dataa voitaisiin näyttää toivotulla tavalla. Ensimmäinen ratkaisu olisi käyttää TableView- näkymää, jossa nykyinen välilehti korvataan TableViewillä. TableView sallii datan näyttämisen taulukkotyyppisesti käyttäen TableViewCell- soluja. Näiden solujen avulla saadaan kierrettyä ongelmat uusien tekstikenttien lisäämisen kanssa. Soluja voi lisätä rajattoman määrän, ja taulukkonäkymä skaalautuu automaattisesti solujen määrän mukaan. Toinen ratkaisu olisi käyttää Charts API-Cocoapodia, jonka avulla olisi mahdollista tehdä erilaisia taulukoita syötetystä datasta. Tällöin datan esittäminen voitaisiin toteuttaa esimerkiksi piirakkakaaviolla tai palkkikaaviolla, jolloin ei olisi tarvetta dynaamisille tekstikentille tai taulukkosoluille.

5.5 Marvel

Marvel on Marvel App- yrityksen luoma web-pohjainen ohjelma, jonka avulla on mahdollista luoda rautalankamalliversioita mobiilisovelluksista. Siinä pystyy myös toteuttamaan mockup-versioita sovelluksen toiminnoista ilman ohjelmointia (Marvel 2019). Marvelin avulla edellisissä osioissa tehdyt päätökset testattiin toiminnallisella rautalankamallilla. Tämän avulla pystyttiin tarkemmin havainnollistamaan, miten eri design-valinnat ja toiminnot toimisivat käytännössä.

Marvelin käyttö oli tässä työssä iteratiivinen prosessi. Aluksi testasimme toimeksiantajan kanssa alkuperäistä designiamme luomalla siitä rautalankamallin. Tämän jälkeen siirryimme takaisin suunnitteluvaiheeseen, jossa Marvelin tulosten avulla arvioimme kriittisesti rautalan-

kamallin tuloksia. Tämän jälkeen teimme muutoksia huonoiksi todettuihin ratkaisuihin, ja toteutimme toisen iteraation rautalankamallista. Kuin saimme lopulta luotua toimivan mallin, siirryin tämän jälkeen Xcode- kehitysalustalle toteuttamaan työn toteutusvaihetta.

Marvelilla toteutettu ensimmäinen malli perustui alkuperäiseen designiimme, jossa sovelluksen tausta olisi taustakuva, joka toistuu kaikissa välilehdissä. Tämä design perustui Benchmarkingin tuloksiin, koska tämä malli todettiin alustavasti sopivaksi tähän työhön. Tässä mallissa myös painikkeiden suunnittelu ja ulkoasu olivat hyvin merkittävässä roolissa. Painikkeiden tuli erottua taustakuvaa vasten, ja samalla niiden tuli myös viestiä selkeästi käyttäjälle niiden tarkoitus.

Sovelluksen rakenne oli kokonaisuutena melko samanlainen, kuin viimeistelyssä designissa. Välilehtiä oli päävalikon lisäksi neljä: Set Budget, Budget, New Purchase ja Account. Account-välilehti oli alun perin suunniteltu välilehti, jossa käyttäjä pääsisi muokkaamaan pilvitietokantaan liittyviä asetuksiaan. Pilvitietokannan suunnitellut toiminnot eivät kuitenkaan olleet riittävän monimutkaisia, että erilliselle asetussivulle olisi tarvetta, joten asetussivusta päätettiin luopua.

Ensimmäisessä mallissa emme toteuttaneet minkäänlaista mockup- toiminnallisuutta rautalankamalliin, sillä tärkeintä oli saada luotua visuaalinen representaatio designistamme. Tämän pohjalta voisimme arvioida, olisiko se riittävän hyvä, vai iteroisimmeko suunnitteluvaiheen. Sovelluksen rakenteeseen olimme tyytyväisiä tässä vaiheessa, joten emme tehneet siihen suuria muutoksia. Isoin työn rakenteeseen tehty muutos tämän vaiheen jälkeen olivat kaksinkertainen sovellusrakenne ja kahdet tietokannat. Nämä lisäsivät lopulta 7 välilehteä sovellukseen, mutta ne olivat enimmäkseen kopioita jo olemassaolevista sivuista.

Toinen toteutettu malli perustui uudistettuun suunnitteluun, jossa taustakuva korvattiin ja muu sovelluksen visuaalinen sisältö Flat designiin perustuvilla yksivärisillä taustoilla ja pelkistetyillä designeilla. Tarkoituksena designissa oli pitää sovelluksen ulkoasu hyvin pelkistettynä, jossa taustoina toimisi yksittäiset värit. Muut elementit, kuten painikkeet ja syöttökentät, olivat yksivärisiä tekstiä lukuunottamatta. Jokaisella välilehdellä oli tässä mallissa omalla taustavärinsä, jotka perustuvat väliaikaisratkaisun väriskeemaan. Tämä johti myös muihin pelkistettyihin ratkaisuihin ulkoasussa, koska esimerkiksi alkuperäisen mallin käyttämiä painikeikoneja ei enää tarvittu, vaan ne saatiin korvattua yksivärisillä painikkeilla.

Toiseen malliin toteutimme visuaalisen rakenteen ja ulkoasun lisäksi myös esimerkkitoiminnon hyödyntäen Marvelin mockup- toimintoja. Aluksi asetin siirtymän päävalikon Set Budget- painikkeelle Set Budget- sivulle. Tämän jälkeen Set Budget- sivulla täytettiin syöttökentät ja

Confirm- painikkeen painalluksen jälkeen siirryttiin Budget- sivulle. Budget-sivulla kaikki kategoriat oli esitetty laatikkomaisina tekstikenttinä. Tämä havainnollisti sovelluksen perustoimintoja, ja työn toimeksiantaja pti sitä hyvänä lähtökohtana työn toteutusvaiheelle.

Toimeksiantajan arvion mukaan toisen iteroidun mallin ominaisuudet olivat riittävän hyvät. Tästä syystä tein päätöksen päättää työn suunnitteluvaiheen. Tämän jälkeen siirryttiin tekemään sovelluksen teknistä prototyyppiä Xcode- sovelluskehitysalustalla. Työssä ei enää tämän vaiheen jälkeen palattu tekemään uusia malleja sovelluksesta Marvelilla.

Toteutusvaiheen loppupuolella Firebasen lisäämisen ja suunnitellun uuden mallin myötä oli mahdollista, että Marvelilla tehtäisiin vielä iteraatioita, joissa nämä muutokset olisi otettu huomioon. Tätä ei kuitenkaan koettu tarpeelliseksi, koska molemmista muutoksista oli jo olemassa riittävästi havainnoivaa materiaalia muissa muodoissa. Firebase- malli oli enimmäkseen kopio toteutetusta toisesta mallista. Sen vuoksi siihen ei tarvittu muita uusia sivuja kuin rekisteröinti- ja kirjautumissivut, jotka olivat erittäin yksinkertaisia designinsa puolesta. Uusiksi suunnittelussa jatkokehitysmallissa taasen suurimmat erot tulivat TableView- välilehdistä, joista minulla oli jo samantyyppinen toteutus tehtynä yhdessä vanhassa esimerkkiprojektissa. Käytin sitä havainnoillistamaan toimeksiantajalle ideaani uuden mallin tavasta toteuttaa dynaamiset toiminnot.

6 Prototyypin tekninen toteutus

Tässä osiossa käydään läpi sovelluksen teknisen toteutuksen vaihe. Osio alkaa käyttöliittymän ja välilehtien luonnista. Tämän jälkeen käydään läpi sekä painikkeiden luonti että niiden toimintojen toteutus. Viimeiseksi käydään läpi työn tietokantaratkaisut, päätökset eri tietokantojen valinnoista ja niiden tekninen toteutus.

Teknisessä toteutuksessa kehitysalustana käytettiin Applen Xcode- kehitysalustaa. Xcode on Applen virallinen sovelluskehitysalusta, jossa pystyy iOS:n mobiilisovellusten lisäksi kehittämään myös esimerkiksi sovelluksia Apple Watchille ja MacOS:lle. Xcode on iOS- kehityksessä erittäin hyödyllinen alusta, koska siinä on valmiiksi asennettuna kaikki iOS:n eri lisäosat ja teknologiat. Xcodeen on myös sisäänrakennettuna muun muassa visuaalinen rakennustyökalu käyttöliittymille ja visuaalinen hallintajärjestelmä Core Data- SQLite- frameworkille.

Teknisen toteutuksen ohjelmointipuoli taasen toteutettiin Swift- ohjelmointikielellä. Swift on Applen kehittämä moderni ohjelmointikieli, joka julkaistiin alun perin vuonna 2014 (New Gen Apps 2014). Swift on multiparadigmakieli, joka tukee olio-ohjelmoinnin lisäksi muun muassa funktionaalista ohjelmointia (Manferdini). Työn aikana ohjelmoinnissa käytettiin enimmäk-

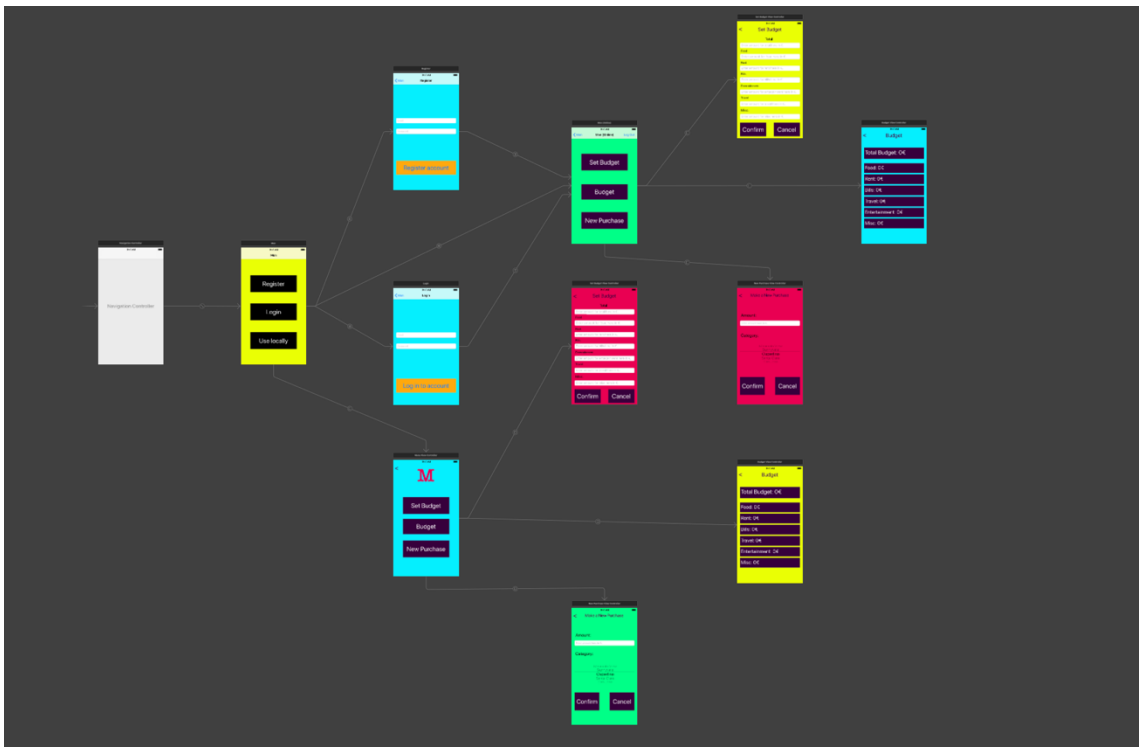
seen proseduraalista ohjelmointia, mutta tietokantojen kanssa käytettiin myös olio-ohjelmointia. Työssä käytettiin Swiftin versiota 5.0, johon työ päivitettiin 5.0-version julkaisun yhteydessä.

6.1 Tekninen toteutus - Käyttöliittymä ja välilehdet

Tässä osiossa käydään läpi sovelluksen käyttöliittymän ja välilehtien toteutus. Aluksi käydään läpi Xcoden Storyboard- ominaisuuden käyttö ja sovelluksen rakenteen luonti. Tämän jälkeen käydään läpi Interface Builder- työkalun käyttö sovelluksen toiminnallisten elementtien, kuten painikkeiden, luomisessa. Lopuksi käsitellään Interface Builderiin kuuluvia työkaluja, kuten Constraintsia ja Auto Layoutia. Näiden avulla toteutettiin elementtien symmetrisyys ja myös niiden sijainnin tasaisuus eri näyttökooilla.

6.1.1 Storyboard

Sovelluksen käyttöliittymä toteutettiin hyödyntämällä Xcoden sisäänrakennettua Storyboard-työkalua, jonka avulla on mahdollista toteuttaa sovelluksen rakenne ja siirtymät rakenteen välilehtien välillä ilman ohjelmointia (McWherter 2012, 197). Storyboard on toiminnaltaan yksinkertainen. Aluksi avataan sovelluksen projektikansio, ja sieltä valitaan main.storyboard-tiedosto, joka sisältää aluksi yhden tyhjän välilehden. Tähän näkymään on mahdollista liittää drag-and-drop- menetelmällä lisää välilehtiä tarpeen mukaan.



Kuvio 5: Kuva sovelluksen lopullisesta rakenteesta Storyboardin sisällä.

Sovellukseen tehtiin aluksi 4 välilehteä, jotka määriteltiin suunnitteluvaiheessa. Jälkikäteen päätettiin tehdä myös pilvitietokantaa hyödyntävä toteutus sovellukselle, ja sen myötä välilehtien määrä kasvoi lopulta kahteentoista. Nämä välilehdet olivat sovelluksen aloituspäävalikko, josta pystyi siirtymään kirjautumissivulle, rekisteröintisivuille ja paikallisen mallin päävalikkoon. Kirjautumis- ja rekisteröintisivujen jälkeen sovellusrakenteessa tuli pilvitietokantapuolen päävalikko, joka oli identtinen paikallisen mallin päävalikkoon. Päävalikoissa oli siirtymät Set Budget-, Budget- ja New Purchase- välilehtiin.

Käyttöliittymän kaikki sivut olivat UIView- näkymiä, joissa ei ole mitään erityistoimintoja. Tämän vuoksi ne soveltuivat hyvin sovelluksen käyttötarkoitukseen, koska erityistoimintoja sisältäviä näkymiä ei tarvittaisi tässä työssä alkuperäisen suunnittelumallin mukaisessa toteutuksessa. Jatkokehityksen puolella kuitenkin Budget- välilehti tullaan korvaamaan TableView- näkymällä, ja sen lisäksi suunniteltu Purchases- välilehti toimii myös TableView- näkymällä.

Kun kaikki välilehdet oli saatu alustettua Storyboardille, tuli seuraavaksi linkittää välilehdet toisiinsa. Sovelluksen toiminnan kannalta oli luonnollisesti tärkeää, että sivujen välillä pystyisi liikkumaan. Välilehtien linkittäminen tehtiin käyttämällä segue- siirtymiä.

Segueita hyödynnettiin työssä sekä alustamaan sovelluksen navigointirakenne että toteuttamaan siirtymät sivujen välillä. Segueita käytettiin myös aluksi siirtämään tietoa sovelluksen sisällä, mutta tästä mallista luovuttiin siihen liittyvien teknisten ongelmien vuoksi. Segueiden toteutus Storyboardissa oli yksinkertaista. Aluksi valittiin haluttu välilehti ja tämän jälkeen vedettiin ruudulle muodostuva viiva välilehdestä toiseen. Seguet vedettiin toisiinsa siinä järjestyksessä, kuin niiden välillä tulisi siirtyä sovellusta käytettäessä. Järjestys eteni vasemmalta oikealle, eli aloitussivusta vedettiin ensimmäiset seguet. Sovellusrakenteen viimeisimpiin sivuihin vedettiin viimeiset seguet. Segueiden luonnin yhteydessä niille määriteltiin myös nimi, jota käytettiin niiden yksilöivänä tunnuksena koodissa.

6.1.2 Interface Builder

Kun välilehdet ja segue- yhteydet välilehtien välillä oli saatu valmiiksi, siirryttiin seuraavaksi toteuttamaan välilehtien sisältämät elementit ja niiden tyylittely käyttämällä Xcoden Interface Builder(IB)- työkalua. Interface Builder on Storyboardin tapaan visuaalinen työkalu, jonka avulla on mahdollista luoda sivujen visuaalisuus ja elementit ilman ohjelmointia (McWherter 2012, 193). Interface Builderin käyttö keskittyi pääasiassa elementtivalikkoon, josta drag-and-drop- tyylillä sai tiputettua välilehtiin tarvittavat elementit, kuten painikkeet ja syöttökentät. Toinen tärkeä osa IB:tä oli Attribute Inspector- näkymä, jossa on mahdollista muokata eri ominaisuuksia valituista elementeistä.

Ensimmäiseksi toteutettiin sovelluksen välilehtien taustavärit design- vaiheen mallin mukaisesti. Tämä tapahtui avaamalla aluksi haluttu välilehti Attribute Inspectorissa. Attribute Inspectorissa valittiin Background- paneeli, josta avautui näkymä, jossa oli mahdollista syöttää halutun värin heksadesimaali- arvo tai RGB- arvo. Koska valitsimme alustavaan väriskeemaan värit toisen sovelluksen väriskeemasta, tuli nämä värit aluksi saada siirrettyä Xcodeen. Tämä tehtiin käyttämällä HTML Color Picker- työkalua, jonka avulla saatiin valittua tarkat heksadesimaaliarvot. Tämän jälkeen arvot syötettiin Attribute Inspectoriin.

Kun taustat oli saatu oikean värisiksi, toteutettiin seuraavaksi välilehtiin niiden sisältämät visuaaliset elementit ja objektit. Näiden lisääminen toimi hyvin samalla lailla, kuin välilehtien lisääminen Storyboardissa. Aluksi IB:n objektipaneelista haettiin halutut objektit, ja tämän jälkeen ne vedettiin Storyboardiin haluttuun välilehteen. Näihin objekteihin kuuluivat muun muassa painikkeet, tekstin syöttökentät ja tekstiä näyttävät tekstikentät.

Kun kaikki halutut IB- objektit oli asetettu välilehtiin, tuli sen jälkeen määrittää niille niiden ominaisuudet. Tämän toteutettiin kahdella eri tavalla, riippuen siitä, mitä ominaisuuksia objekteissa tuli muokata. Objektien kokoa muokatessa tämä tehtiin Storyboard- näkymän sisällä valitsemalla objekti ja sen jälkeen venyttämällä sitä objektia sen kulmista. Muihin ominaisuuksiin käytettiin Attribute Inspectoria. Pääasiallisesti siellä muutettiin tekstin fonttia, kokoa ja painikkeiden väriä.

Viimeisenä ominaisuutena IB:ssä käytettiin sijoittelutyökalua, jonka avulla oli mahdollista saada sivuille sijoitetut objektit aseteltua symmetrisesti. Tämä toimi hyödyntäen automaattista sijoitustyökalua Xcodessa. Sijoitustyökalu piirsi objektin ympärille rajaviivat objektin ollessa näkymässä symmetrisesti muihin objekteihin verrattuna. Tätä työkalua hyödynnettiin toteuttamaan objektien alustava sijoittelu. Myöhemmin sen lisäksi käytettiin myös edistyneempiä työkaluja.

6.1.3 Constraints, Auto Layout ja Stack View

Kaikkien sovelluksen välilehtien objektien sijoitteluun hyödynnettiin työssä myös muita Interface Builderiin kuuluvia menetelmiä. Näihin teknologioihin kuuluivat Constraints, Auto Layout ja Stack View. Constraintsia käytettiin luomaan painikkeiden responsiivinen sijoittelu. Niiden avulla objektien suhteellinen sijainti pysyi samana riippumatta sovellusta pyörittävän laitteen näytön koosta. Auto Layoutia ja Stack Viewiä taasen käytettiin varmistamaan objektien symmetrinen sijainti suhteessa muihin objekteihin.

Constraintsit ovat ominaisuus Xcodessa, jonka avulla visuaalisia objekteja on mahdollista sitoa paikalleen niin, että kaikki elementit säilyttävät suhteellisen sijaintinsa erikokoisilla näytöillä. Koska sovellusta olisi tarkoitus pystyä käyttämään kaikilla tuetuilla iOS-laitteilla, oli

erittäin tärkeää, että constraintsit olivat asetettuna kaikkiin sovelluksen visuaalisiin objekteihin. Constraintsien tekninen toteutus tapahtui Storyboard- näkymän alalaidassa olevan painikkeen kautta, jossa voi asettaa constraintsit valittuun objektiin.

Aukeavassa näkymässä annettiin 4 eri suuntaa, joihin constraintseja voi asettaa: ylös, alas, vasen ja oikea. Kaikkiin suuntiin ei tarvinnut aina asettaa constraintseja, koska constraintsien kinnittäminen muihin objekteihin aiheutti usein ongelmia skaalautumisen kanssa. Tämän vuoksi constraintseja asetettiin enimmäkseen näytön laitoihin.

Contstraintsien käytön kanssa ei esiintynyt ongelmia Set Budget- sivua lukuunottamatta. Set Budget- sivulla constraintsien asettelu muodostui hankalaksi johtuen välilehden suuresta objektimäärästä. Ongelmat constraintsien asettelun kanssa oli osatekijä päätöksessä suunnitella Set Budget- sivu uudelleen.

Constraintsien lisäksi käyttöliittymän suunnittelussa hyödynnettiin myös Auto Layoutia, jonka avulla oli mahdollista kohdentaa Xcoden pikseliyksikön tarkkuudella objektien sijainteja. Etenkin määritellessä constraintseja oli hyvin olennaista, että niihin sidoksissa olevat objektit olivat symmetrisesti sijoiteltuja. Jos objektit eivät olisi symmetrisesti sijoiteltuja, saattaisi se aiheuttaa ongelmia skaalautumisen kanssa. Tämän lisäksi epäsymmetrinen sijoittelu antaisi sovelluksesta epäammattimaisen vaikutelman.

Auto Layoutia hyödynnettiin sekä sijoittelutyökalun että ohjelmoinnissa tapahtuvan sijoittelun kautta. Sijoittelutyökalun avulla saatiin esimerkiksi päävalikoiden painikkeiden sijainnit symmetrisiksi asettamalla aluksi keskimmäisin painike näytön keskelle. Objektin ollessa keskellä näyttöä Auto Layout piirsi sen ympärille katkoviivoja, jotka menivät sovelluksen kaikkiin laitoihin. Tämä indikoi painikkeen olevan täsmälleen näytön keskellä.

Tämän jälkeen varmistaakseni painikkeiden tasaiset välimatkat toisistaan, käytin Auto Layoutin ohjelmoinnillista puolta. Aluksi määriteltiin näytön koko pikseleissä. Tämän jälkeen Auto Layoutia käytettiin laskemalla yhtä pitkät välimatkat sekä ylimmälle että alimmalle painikkeelle sivun ylä- ja alalaidasta. Tämän tiedon avulla väliin jääville painikkeille laskettiin yhtä pitkät välimatkat toisistaan ja reunimmaisista painikkeista. Tietoa näytön koosta käytettiin varmistamaan, että kaikkien objektien välillä oli samat etäisyydet näytön laitoihin.

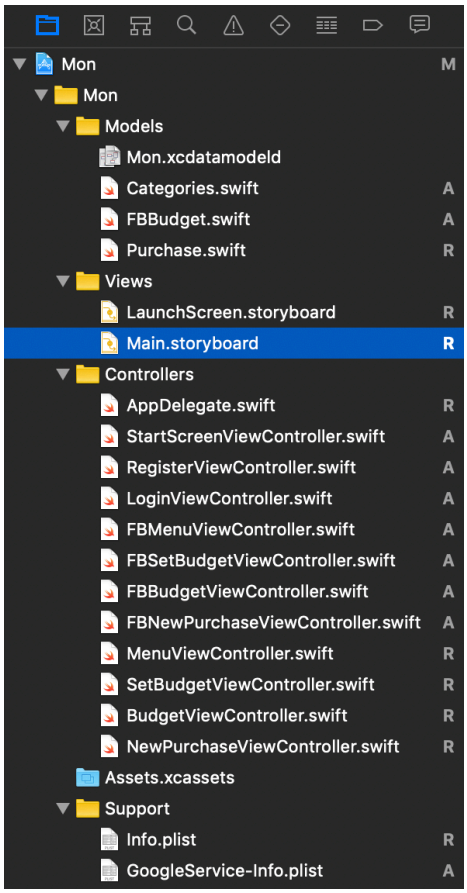
Toinen sijoitteluun liittyvistä merkittävistä teknologioista oli Stack View. Stack View oli toiminnoiltaan erikoistuneempi kuin Auto Layout, sillä sen käyttötarkoitus oli käsitellä useita visuaalisia objekteja yhtenä ryppäänä. Stack View toimi lisäämällä aluksi halutut objektit siihen. Tämän jälkeen siihen asetettiin ominaisuuksia, kuten objektien koko ja niiden etäisyys toisistaan.

Tässä työssä Stack Viewia oli tarkoitus käyttää ratkaisemaan sijoitteluun liittyviä ongelmia Set Budget- ja Budget- välilehdillä. Molemmat sivut sisälsivät suuren määrän visuaalisia objekteja, joten olisi ollut loogista käyttää näiden asetteluun Stack Viewiä. Stack Viewiä ei kuitenkaan lopulta otettu käyttöön, koska molemmat sivut, joissa sitä olisi tullut käyttää, päätettiin suunnitella uusiksi teknisistä ongelmista johtuen. Stack Viewia saatetaan kuitenkin käyttää jatkokehityksessä molempien tietokantaversioiden päävalikkojen kanssa. Niihin olisi tarkoituksena lisätä yksi uusi välilehti, mikä kasvattaisi painikkeiden määrän neljään. Neljän painikkeen kanssa sijoittelu toimiisi helpommin Stack Viewin kanssa. Tämän vuoksi nämä sivut tullaan myöhemmin vaihtamaan Stack Vieweihin.

6.2 Tekninen toteutus - Toiminnot

Tässä osiossa käsitellään sovelluksen teknisen toteutuksen toimintojen luonti. Aluksi käydään lyhyesti läpi sovelluksen MVC- arkkitehtuuri, jota käytettiin selkeyttämään projektihierarkian rakennetta sovelluksen kasvaessa. Tämän jälkeen käydään läpi sovelluksen navigoinnin toteutus. Tämän jälkeen käydään läpi Cocoa Touch- tiedostot ja koodin linkitys käyttöliittymän objekteihin, kuten painikkeisiin. Sen jälkeen käydään läpi muut toiminnot, jotka olivat osa sovellusta. Lopuksi käsitellään vielä toteutusvaiheen aikana ilmenneitä teknisiä ongelmia ja debuggausta Xcodessa. Lopuksi käydään läpi ongelmat dynaamisten toimintojen lisäämisen kanssa ja ongelmien ratkaisuun ehdotetut ratkaisumallit.

6.2.1 MVC- malli



Kuvio 6: MVC-malli Xcoden projektinäkymässä.

Sovelluksen hierarkia perustui sovelluskehityksessä yleiseen MVC(Model View Controller)- malliin. MVC:ssä sovelluksen tiedostot olivat jaettuna kolmeen osaan. Ensimmäinen osa oli Model, johon kuului tietoa käsittelevät tiedostot. Toinen osa oli View, johon kuului visuaaliset storyboard- tiedostot. Kolmas osa oli Controller, johon kuuluu toimintoja hallinnoivat tiedostot. MVC:tä käyttämällä oli mahdollista varmistaa paremmin organisoidumpi rakenne sovelluksen projektihierarkialle. MVC mahdollisti koodin tehokkaamman hallinnan ja organisoinnin.

Model- osio MVC:stä viittasi sovelluksen dataa käsitteleviin tiedostoihin ja niihin määritellyihin datamalleihin. Model- osioon sisältyi tässä työssä paikallisten tietokantojen tiedostot. Sovelluksen käyttämät pilvitietokannat eivät tarvinneet kuin yhden tiedoston. Tämä tiedosto oli .plist- tyyppinen tiedosto, joten sitä ei sisällytetty Model-osioon. .plist-tiedostot sijoitettiin MVC:stä erilliseen Supporting Files- kansioon.

Tiedostoja tässä osiossa oli yhteensä 3. Ensimmäinen niistä oli Core Data- tietokannan data model- tiedosto, jossa määriteltiin sen käyttämät dataobjektit. Toinen tiedosto oli Core Datan alustustiedosto, jossa määriteltiin Core Datan aloituskonfiguraatiot. Kolmas Model-osion

tiedosto oli Realm- tietokannan käyttämä dataobjekti New Purchase- sivulla tehtäville ostoksille.

MVC:n View-osio viittasi sovelluksen välilehtiin ja sen muihin visuaalisiin ominaisuuksiin. Tässä työssä View- osioon kuului 2 tiedostoa. Ensimmäinen tiedosto oli launch.storyboard, joka oli sovelluksen latausruutu. Toinen tiedosto oli main.storyboard, jossa oli sovelluksen kaikki välilehdet ja niiden välille asetetut segue- siirtymät. View- osio oli pienin kaikista MVC:n alakansioista, koska sovelluksen koko rakenteen sai sisällytettyä yhteen storyboard- tiedostoon.

MVC:n Controller- osio viittasi tiedostoihin, jotka määrittelivät ja hallinnoivat sovelluksen toimintoja välilehdissä ja sovelluksen elinkaaressa. Tämä osa MVC:tä oli selkeästi isoin johtuen välilehtien suuresta määrästä. Jokaisella välilehdellä oli oma Cocoa Touch- tiedosto, johon määriteltiin muun muassa funktiot kaikille niihin linkitetyille objekteille, kuten painikkeille. Cocoa Touch- tiedostojen lisäksi tässä kansiossa oli myös AppDelegate- tiedosto, jonka tehtävä oli hallinnoida sovelluksen toimintaa sen elinkaaren eri vaiheissa. AppDelegate esimerkiksi toteutti funktiot, jotka suoritettiin aina sovelluksen käynnistyessä. Tässä työssä AppDelegatea käytettiin Realm- ja Firebase-tietokantojen alustamiseen tarvittavien funktioiden suorittamiseen sovelluksen käynnistymisen yhteydessä.

Viimeisenä osana projektiarkkitehtuuria oli Supporting Files- kansio. Se ei ollut osa MVC- mallia, mutta se oli hyödyllinen sovelluksen tiedostojen organisoimisessa. Supporting Files- kansioon tuli pääasiassa .plist- tiedostot, jotka vastasivat sekä itse sovelluksen että Firebasen konfiguroinnista. Info.plist oli valmiiksi iOS-sovelluksissa oleva tiedosto, ja se vastasi sovelluksen perustiedoista ja asetuksista. GoogleService-Info.plist taasen vastasi Firebasen tarvitsemista asetuksista ja tiedoista.

MVC:n kansioiden ulkopuolelle jäi sovelluksen Assets- kansio, jonne sijoitettiin sovelluksen käyttämät kuvat, logot ja muut graafiset elementit. Tähän kansioon sisältyivät myös sovellusikonit. Nämä ikonit toimivat iOS-laitteiden sovellusvalikossa sovelluksen pikakuvakkeena. Tämän vuoksi ne olivat pakollisia, jotta sovelluksen saisi ladattua App Storeen. Sovellukseemme ei kuitenkaan tämän työn puitteissa luotu ikoneja, koska sovellusta ei vielä ladattu App Storeen.

6.2.2 Navigointi

Painikkeista ensimmäisinä toteutettiin sovelluksen päävalikon painikkeet, jotka mahdollistavat siirtymisen sivujen välillä. Tämän jälkeen tehtiin Back- painikkeet kaikille sivuille, joiden avulla oli mahdollista siirtyä takaisinkaikilta sivuilta pääsivulle. Painikkeiden toiminnallisuuden toteutus oli yksinkertainen. Päävalikon painikkeille määriteltiin IBAction- funktio. Funkti-

ossa määriteltiin haluttu segue- siirtymä. Siirtymä kohdennettiin asettamalla IBAction- funktioon halutun segue-siirtymän tunnus eli sen nimi, joka annettiin seguelle sen luomisen yhteydessä.

Back- painikkeiden kohdalla käytettiin funktiota, joka peruutti sillä hetkellä käynnissä olevan segue-siirtymän. Peruutuksen yhteydessä sovellus palasi alkuperäiseen välilehteen, eli tässä tapauksessa päävalikkoon. Jälkikäteen lisättyjen sovelluksen aloituspäävalikon ja pilvitietokantapuolen päävalikon kanssa kaikki toiminnot toteutettiin samalla tavalla.

Kun perusnavigointi oli saatu toteutettua, toteutettiin seuraavaksi rekisteröintiin, kirjautumiseen ja uloskirjautumiseen liittyvä navigointi. Tämä toimi samalla lailla kuin sovelluksen muu navigointi. Alkupäävalikosta siirryttiin joko rekisteröintisivulle tai kirjautumissivulle segue-siirtymällä ja paluu alkupäävalikkoon toimi saman palautumisfunktion kautta.

Onnistuneen kirjautumisen tai rekisteröinnin yhteydessä suoritettiin segue-siirtymä, joka siirtää käyttäjän pilvitietokantapuolen päävalikkoon. Uloskirjautumista ei voitu kuitenkaan toteuttaa tällä mallilla. Tämä johtui siitä, että Log Out- painikkeen painaminen siirtäisi käyttäjän kirjautumissivulle alkupäävalikon sijaan. Tämän ongelman ratkaisemiseksi sovelluksen navigointirakenne vaihdettiin käyttämään Navigation Controller- teknologiaa.

Navigation Controller (NC) on iOS:iin sisäänrakennettu toiminto, joka helpottaa navigoinnin toteutusta ja vähentää siihen tarvittavaa koodia. Navigation Controller otettiin käyttöön vetämällä se aluksi Storyboardissa olevasta objektipaneelista. Sen jälkeen NC asetettiin sovelluksen hierarkian ensimmäiseen sivuun, joka muuttui NC:n root- sivuksi. Root-sivu viittaa NC:n ensimmäiseen sivuun. Tämän jälkeen NC:n pystyi yhdistämään muihin välilehtiin valikoimalla halutut segue-siirtymät Storyboardista. NC mahdollisti sulavan siirtymisen sivujen välillä. NC generoi muun muassa automaattisesti Back- painikkeet välilehtiin ja nimesi nämä painikkeet sivujen otsikoiden mukaan. Tämän avulla pystyttiin myös noudattamaan HIG:n vaatimuksia liittyen navigointiin ja käyttäjän informointiin.

NC:n mukana kaikkiin linkitettyihin sivuihin tuli NavBar- palkki sivujen ylälaitaan. NavBar-palkkiin tuli muun muassa edellä mainitut Back- painikkeet ja sivujen otsikot. Uloskirjautuminen saatiin toteutettua NC:n asettamalla pilvitietokantapuolen päävalikko- sivun NavBar-palkkiin Log Out- painike. Tähän painikkeeseen linkitettiin IBAction- funktion avulla siirtymäfunktio, jonka kohteena oli NC:n root- näkymä. Näin saatiin tehtyä siirtymä pilvitietokantapuolen päävalikosta aloituspäävalikkoon.

6.2.3 Cocoa Touch- tiedostot ja IBOutlet/IBAction

Tässä osiossa käydään läpi sovelluksen toimintojen kannalta 2 tärkeää teknologiaa, Cocoa Touch ja IBOutlet/IBAction. Cocoa Touch on uuden luokan määrittelevä tiedosto, jota käytetään etenkin välilehtien toimintojen hallinointiin. IBOutlet/IBAction ovat termejä, joilla viitataan koodiin linkitettyihin IB- objekteihin. IBOutlet on passiivinen objekti, joka muun muassa vastaanottaa ja näyttää tietoa. IBAction on aktiivinen IB-objekti, jolla voidaan toteuttaa erinäisiä toimintoja, jotka määritetään IBAction-funktion sisälle.

Cocoa Touch- tiedostot poikkesivat muista Swift- tiedostoista, koska niissä määriteltiin aina uusi luokka. Tavallinen Swift- tiedosto voisi esimerkiksi vain alustaa tietokannan hyödyntämän dataobjektin. Cocoa Touch- tiedostoja käytettiin työssä rajapintana välilehtien ja niiden toimintojen välillä. Cocoa Touch- tiedostot yhdistettiin ensiksi välilehteen. Tämän jälkeen niihin oli mahdollista kirjoittaa koodia ja linkittää välilehden objekteja IBOutletin/IBActionin kautta.

IBOutlet/IBAction olivat linkkejä käyttöliittymän ja toiminnallisuuden välillä. Ne luotiin vetämällä yhteys Storyboardissa IB- objektista Cocoa Touch- tiedostoon. Uudelle yhteydelle määritettin nimi ja sen tyyppi, eli Outlet tai Action. Jos yhteys määritettiin Actioniksi, generoitui linkitettyyn Cocoa Touch- tiedostoon uusi IBAction- funktio.

Työssä IBOutleteja käytettiin kaikissa välilehdissä, koska sekä tekstiä näyttävät tekstikentät että syötettyä tekstiä vastaanottavat tekstinsyöttökentät olivat tyypiltään IBOutleteja. IBActionit vastasivat työssä lähes kaikesta toiminnallisuudesta. Kaikki painikkeet ja niiden toiminnot määritettiin IBAction- funktioiden sisälle. IBAction- funktioiden ulkopuolelle jäivät vain viewDidLoad- funktiot ja koodin lukemista helpottavat apufunktiot.

6.2.4 viewDidLoad

Tässä osiossa käsitellään työn toteutusvaiheessa hyödynnettyä viewDidLoad- funktiota. Swiftissä funktio määritellään func- avainsanalla. Sen jälkeen syötetään funktion nimi ja lopuksi funktion sulkeiden sisään kirjoitetaan funktion koodi. Funktio voidaan kutsua halutussa paikassa koodissa formaatilla funktionNimi(). Swift ei vaadi lausekkeiden tai funktiokutsujen loppuun puolipistettä.

ViewDidLoad on Cocoa Touch- tiedostoihin valmiiksi määritelty funktio. Nimensä mukaisesti se suoritetaan aina välilehden lataamisen yhteydessä. ViewDidLoad oli työssä hyödyllinen ja monipuolinen funktio, koska sen avulla pystyttiin toteuttamaan lukuisia erilaisia määrittelyjä sivujen latautumisen yhteydessä.

Työssä viewDidLoadia käytettiin New Purchase- sivulla lataamaan sen hyödyntämän Picker View- valintarullan datalähde. Datalähde määrittää kaikki valintarullan näyttämät tiedot. ViewDidLoadia käytettiin myös Budget- sivun toimintojen alustavaan toteutukseen. ViewDidLoadilla määriteltiin, mitä dataa budjetin kategorioiden tekstikentät näyttivät sivun ladatessa.

Visuaalisuuden toteutuksessa viewDidLoadia käytettiin määrittämään NavBarin tyylittelyjä. Esimerkiksi NavBarin väriä ei pystytty vaihtamaan Attribute Inspectorin kautta. Tämän vuoksi määrittelyt näille ominaisuuksille tehtiin koodin kautta. Määrittelyjen tulisi olla käytössä heti sivun ladatessa, joten tämän vuoksi ne sijoitettiin koodissa viewDidLoad- funktioihin.

6.2.5 Muut toiminnot

Tässä osiossa käydään läpi muita työssä hyödynnettyjä teknologioita. Aluksi käydään läpi käyttöliittymän teknologiat, kuten Picker View- valintarullat ja Alert- popupit. Tämän jälkeen käydään läpi työssä käytetty Cocoapod- pakettienhallintajärjestelmä ja Cocoapodien käyttö. Lopuksi käsitellään vielä ohjelmointiin liittyviä teknologioita ja menetelmiä.

Picker View- valintarulla oli yksi merkittävimmistä toimintoihin liittyvistä teknologioista, joita työssä käytettiin. Sen tarkoituksena työssä oli toteuttaa kategorian valinta ostosta tehdessä New Purchase- sivulla. Picker View lisättiin välilehtiin kuten muutkin käyttöliittymäobjektit, eli vetämällä se Storyboardin objektipaneelista haluttuun välilehteen. Tämän jälkeen valintarullalle määriteltiin koodiin array- muodossa sen datalähde, joka määrittää valintarullan näyttämät tiedot. Picker Viewin toiminta toteutettiin käyttämällä Extension- menetelmää.

Extension iOS- kehityksessä viittaa erilliseen koodinosioon, jossa on mahdollista tehdä määrityksiä erillään muusta koodista. Tämän tarkoitus tässä työssä oli siistiä koodikantaa ja sallia tehokkaampi koodinhallinta. Sen avulla Picker Viewin määritykseen vaadittava koodi saatiin irralleen muusta sivun koodista. Extensiossa määritettiin Picker Viewin toimintaan vaadittavat ominaisuudet, kuten sen käyttämä datalähde ja rivien määrä. Rivien määrä jätettiin yhteen, koska Picker Viewillä ei näytetty työssä muuta kuin budjetin määritellyt kategoriat.

Cocoapodit olivat hyvin tärkeä osa työn toteutusvaihetta. Cocoapods on paketinhallintajärjestelmä (Cocoapods 2019), jonka avulla on mahdollista asentaa ja käyttää kolmansien osapuolien tuottamia toimintoja iOS:lla. Tässä työssä niitä käytettiin Firebase- pilvitietokannan ja Realm-tietokannan asentamiseen. Asennus toimi Cocoapodeissa kahdessa osassa. Aluksi Xcodessa avattiin sovelluksen Podfile- tiedosto, joka sisälsi sovelluksen Cocoapodeihin liittyvät tiedot. Siinä oli luetteloituna asennetut Cocoapodit ja niiden asennettavat versiot. Kun nämä

olivat määriteltyjä, niin seuraavaksi MacOS:n Terminalilla navigoitiin sovelluksen projektikansioon ja ajettiin pod install- komento, joka asensi Podfile- tiedostossa määritellyt Cocoapodit sovellukseen.

Datan validointi oli yksi tärkeimmistä ohjelmointiin liittyvistä ominaisuuksista, joita työssä käytettiin. Datan validoinnilla varmistettiin, että käyttäjän syöttämä data oli halutussa muodossa. Datan validointi tapahtui kahdessa osassa. Aluksi syöttökenttien sisältämä data luettiin funktioilla. Tämän jälkeen luettu data muunnettiin oikeaan datatyyppiin. Datayypin muuntaminen oli pakollinen vaihe summien käsittelyn yhteydessä, koska Xcode tulkitsee syöttökenttien datan oletuksena string- merkkijonoksi. Työssä haluttu datatyyppi summien kohdalla oli float- desimaaliluku. New Purchase- sivun kategorioiden kohdalla muuntoa ei tarvinnut tehdä, koska kategorian kuului olla datatyypiltään merkkijono.

Data saatiin muunnettua oikeaan muotoon käyttäen Swiftin sisäänrakennettua datakonversiofunktota. Tällä saatiin varmistettua vastaanotetun tiedon datatyyppi vastasi tietokantaan määritettyjä datatyyppejä. Datan validointia varten tarvittiin myös toinen menetelmä,jolla voitiin varmistaa, että syöttökenttiin oli syötetty dataa. Swiftissä syöttökenttien täytyy aina sisältää dataa, koska sovellus kaatuu, jos se löytää syöttökentästä tyhjän arvon.

Tämän ongelman ratkaisemiseen oli kaksi erilaista lauseketta, jotka pystyivät tarkistamaan tekstikentät tyhjien arvojen varalta ja varmistamaan, ettei sovellus kaadu. Nämä lausekkeet olivat Guard Let ja Let If. Molemmille määriteltiin aluksi If- lauseke, jossa tarkistettiin, oliko kohdennettu tekstinsyöttökenttä tyhjä. Jos se oli tyhjä, niin suoritettiin If- ehtoa seuraava koodi, joka antoi huomautuksen käyttäjälle, että syöttökentät olivat tyhjiä. Jos taasen If-ehto meni läpi, niin silloin lauseke ohitettiin. Yksinkertaisemman syntaksin vuoksi käytin Guard Let- lauseketta työssä.

Alert- popupit olivat työssä tärkeä teknologia. Niitä käytettiin kahteen tarkoitukseen. Set Budget- ja New Purchase- sivuissa alerteja käytettiin antamaan käyttäjälle ilmoitus onnistuneista kirjauksista tietokantaan. Näillä sivuilla rekisteröinti -ja kirjautumissivujen lisäksi alerteja käytettiin myös huomauttamaan käyttäjää virheellisestä datan syötöstä. Varoituksia annettiin muun muassa liian lyhyistä salasanoista, väärässä formaatissa olevista sähköpostiosoitteista ja tyhjistä syöttökentistä. Alertit tehtiin If/Else- lausekkeilla, joissa aluksi validoitiin data, ja jos validointi epäonnistui, niin silloin kutsuttiin alertin näyttävä koodi.

Viimeinen merkittävä työssä käytetty ohjelmointiin liittyvä teknologia oli Closure- koodiblokit. Closuret olivat sisäkkäisiä funktioita, jotka pystyivät vastaanottamaan ja käsittelemään ulomman funktion sisältämiä arvoja (Swift Docs). Toiminnoltaan Swiftin closuret toimivat samanlaisesti kuin esimerkiksi C#- ja Python-kielien Lambda-ilmaisut. Closureja käytettiin muun

muassa Firebase- pilvitietokannan valmiissa koodiblokeissa datan edelleenlähettämiseen. Tässä esimerkkitapauksessa ulommassa funktiossa syötetty data vastaanotettin, ja sisäkkäisessä funktiossa data lähetettiin eteenpäin Firebasen tietokantaan.

6.2.6 Tekniset haasteet ja debugging

Tässä osiossa käydään läpi työn aikana kohdattuja teknisiä haasteita ja niihin kehitettyjä ratkaisuja. Aluksi käydään läpi merkittävimmät työn aikana koodatut bugit ja niiden ratkaisut. Sen jälkeen osiossa käsitellään Xcoden Debugger- työkalun käyttöä lyhyesti. Lopuksi käydään läpi havainnollistava esimerkki Debuggerin käytöstä.

Alun perin sovelluksessa oli tarkoituksena siirtää Set Budget- välilehdessä tehtyjen kirjausten tiedot segue- siirtymällä Budget- välilehteen. Tämä tuotti suurimman teknisen ongelman, mikä kohdattiin työn toteutusvaiheessa. Tämä toteutus ei toiminut siksi, että segueta käyttämällä navigointi ei toimisi enää normaalisti. Siirtymisessä jouduttaisiin ensiksi siirtyä takaisin Set Budget- sivuun ja vasta sieltä voitaisiin siirtyä päävalikkoon. Tämän lisäksi datan tallennukseen käytettävää funktiota ei pystytty kohdentamaan oikein. Jouduin tämän vuoksi tutkimaan asiaa syvällisemmin. Päädyin lopulta johtopäätökseen, että tiedonsiirto sivujen välillä tulisi toteuttaa toisella tavalla. Set Budget- sivulla data lähetettäisiin tietokantaan. Tämän jälkeen aina käyttäjän siirtyessä Budget- välilehteen tietokantaan tallennetut tiedot ladataan sivun tekstikenttiin.

Toinen isompi ongelma sovelluksen tiedonkäsittelyn kanssa tuli Firebase- pilvitietokannan kanssa. Siinä ongelmaksi muodostui Firebase-tietokannan katoaminen aina sovelluksen käynnistyessä iOS Simulatorissa. Sovellus aluksi poisti Firebasen tietokannat ja sen jälkeen kirjoitti sen tilalle tekstin ”Test 2: Message Received”. Tämä oli hämmentävä bugi, koska en onnistunut löytämään Xcoden konsolin ja debuggerin avulla mitään, mikä voisi aiheuttaa tämän ongelman. Lopulta ongelma ratkesi, kun muistin jättäneeni AppDelegate- tiedoston didFinishLaunchingWithOptions- funktioon print-lausekkeen, joka tulosti Firebaseen kyseisen tekstin. Print-lausekkeen poiston myötä ongelma ratkesi.

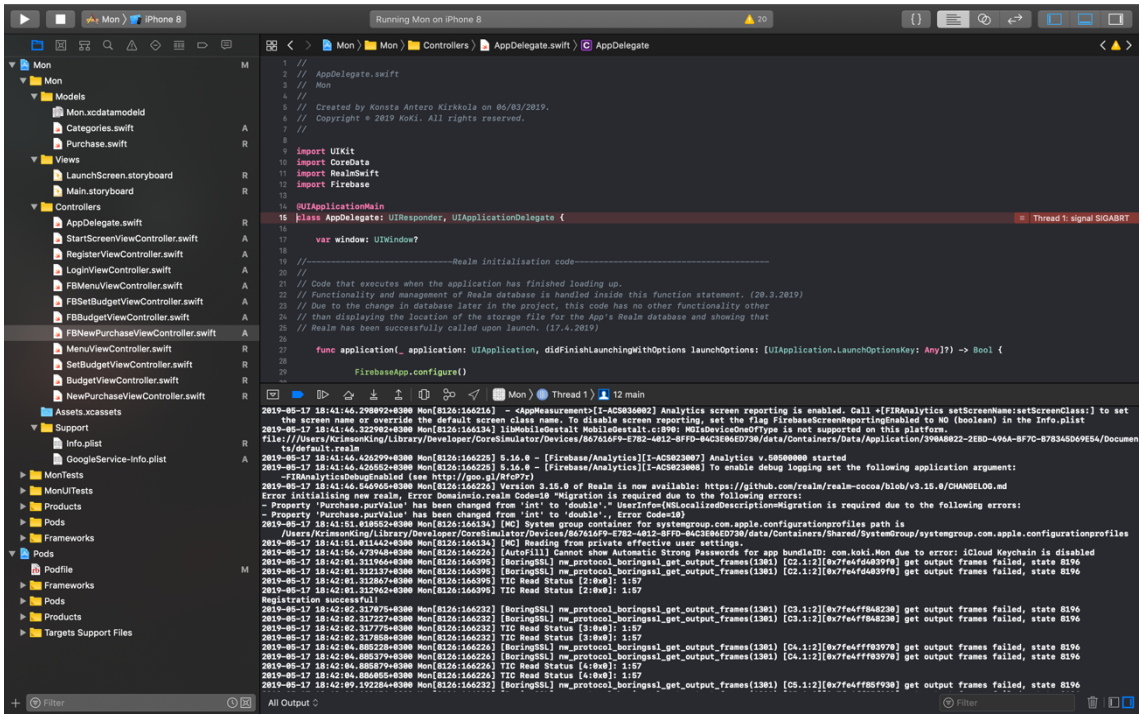
Yksi useamman kerran havaittu bugi työssä oli key-value pair not compliant- bugi. Tämä tuli kääntäessä ohjelman koodia, ja se esti sovelluksen käynnistämisen. Olin aiemmin kohdannut kyseisen bugin, joten tiesin mistä se johtui ja osasin kohdentaa ongelman tutkinnan oikeaan kohtaan koodia. Tämä bugi syntyi tehdessä IBOutlet/IBAction- linkityksiä Cocoa Touch- tiedostoihin. Jälkikäteen poistaessa näitä yhteyksiä esimerkiksi refactoring- prosessissa saattoi koodiin jäädä viittauksia vanhoista yhteyksistä. Tämä kaatoi sovelluksen, koska kääntäjä ei pystynyt löytämään kyseistä yhteyttä. Tämä kävi minulle kerran koodin refactoring- prosessia teh-

dessä. Huomasin, että joidenkin IBOutletien nimet eivät olleet käyttämieni nimeämiskäytäntöjen mukaisia. Vaihtaessa nimiä minulta jäi vanhat viittaukset koodiin. Tämä korjaantui lopulta käymällä koodi läpi erittäin huolellisesti ja poistamalla vanhat viittaukset.

Toinen monen kertaan kohdattu kriittinen bugi oli Found nil while unwrapping Optional- bugi. Tämä viittaa IBOutlet- tekstinsyöttökenttien käyttämään datamuotoon, eli Optionaliin. Optional viittaa Swiftin arvotyyppiin, joka sallii jonkin arvojen olevan joko nolla tai tyhjä arvo (AppCoda). Tämä saattoi kaataa sovelluksen, jos Optional-arvoja lukevalle funktiolle ei ollut erikseen määritelty, mitä sen tulee tehdä kohdatessaan tyhjän Optional-arvon. Sovellus kaatui aina testatessa Set Budget- sivun syöttökenttiä, jos jokin arvoista oli jätetty tyhjäksi. Ongelma ratkesi lopulta Guard Let- lausekkeiden käytöllä.

Xcoden Debugger oli hyödyllinen työkalu, jonka avulla oli mahdollista tutkia ja analysoida teknisiä ongelmia. Ongelmia pystyi ratkaisemaan Debuggerilla parilla tavalla. Nämä tavat olivat debugger-lokin analysointi ja Breakpoint- tarkistuspisteiden käyttö. Debugger- loki generoitui automaattisesti Xcoden konsoliin, jos Simulatorissa pyörivä sovellus kaatui. Generoitu loki oli tavallisesti pitkä. Sen olennaisin osuus oli kuitenkin yleensä lokin alussa, jossa näkyi virheilmoitus sovelluksen kaataneesta bugista. Esimerkiksi Found nil while unwrapping Optional-bugi löydettiin näin alunperin.

Breakpoint- tarkistuspistenanalyysi toimi niin, että aluksi Debuggerilla asetettiin Breakpointeja koodiin, ja sen jälkeen koodi suoritettiin. Koodin suoritus pysähtyi aina breakpointin kohdalla. Tämä oli erittäin tehokas tapa tehdä debuggausta. Sen avulla oli mahdollista löytää tehokkaasti bugin aiheuttava koodi. Breakpointeja ei kuitenkaan testikäyttöä lukuunottamatta käytetty työssä. Tämä johtui siitä, että työssä kohdatut bugit olivat entuudestaan minulle tuttuja, joten niiden aiheuttajaa ei tarvinnut erikseen selvittää.



Kuvio 7: Kuva Xcoden Debuggerin generoimasta lokista.

Käytännön esimerkki Debuggerin käytöstä tuli juuri Optional-arvojen käsittelyssä. Testatessa sovellusta Simulatorilla jätin yhden syöttökentistä tyhjäksi ja painaessani Confirmia sovellus kaatui. Xcode generoi lokin kaatumisesta. Siirryin lokissa analysoimaan suoraan sen ensimmäisiä rivejä, koska arvelin tietäväni ongelman syyn. Ilmoitus oli odotetusti Found nil while unwrapping Optional-bugi. Aloin tämän jälkeen selvitellä, kuinka kyseisen bugin voisi ratkaista. Jos virheilmoitus olisi ollut entuudestaan tuntematon, niin aluksi olisi selvitetty ongelman syy ja siihen liittyvät bugit.

6.2.7 Dynaamisuus ja sovellusmallin ongelmat

Työn toteutusvaiheen puolivälissä tulin siihen johtopäätökseen, että suunnitellulla toteutusmallilla ei ollut mahdollista toteuttaa kategorioiden dynaamista lisäämistä. Ongelmia oli sekä Budget- sivulla että Set Budget- sivulla. Budget-sivulla ongelmaksi muodostui sivulla käytettyjen tekstikenttien lisääminen dynaamisesti. En tutkimalla onnistunut löytämään mitään keinoa tehdä tätä Swiftiä käyttäen. Samoin Set Budget- sivulla ei pystytty lisäämään uusia kategorioita, koska se hajoittaisi sivun constraintit. Sen lisäksi en löytänyt myöskään syöttökenttien dynaamiselle lisäämiselle mitään toimivaa menetelmää.

Toiseksi ongelmaksi Set Budget-sivulla muodostui myös valmiiden kategorioiden seuranta-ajan syöttämisen puute. Tässä haasteena oli se, että sivu oli jo täynnä visuaalisia objekteja. Si-

vusta tulisi erittäin täyteen ahdettu, jos jokaiselle kategorialle lisättäisiin vielä toinen syöttökenttä. Tämän vuoksi päätin, että sivu olisi parasta suunnitella uudestaan dynaamisten toimintojen lisäämiseksi.

Budget- välilehdessä dynaamisuuden pääasiallinen haaste muodostui tekstikentistä. Niitä ei pystynyt generoimaan koodin avulla dynaamisesti lisää. Tämän lisäksi haasteita tuotti uusien tekstikenttien saanti osaksi sivun symmetristä sijoittelua. Tämän vuoksi päädyin myös tämän sivun kohdalla suunnittelemaan sivun kokonaan uusiksi.

New Purchase- välilehdessä dynaamisuuden kanssa suurimmaksi ongelmaksi koitui Set Budget- välilehden dynaamisuuden puute. New Purchase- sivulla oli vähemmän dynaamisuutta kuin muilla sivuilla. Siinä oli kuitenkin Picker View- valintarulla, jonka datalähteenä toimi budjetin kategoriat. Tarkoituksena olisi saada datalähteestä dynaaminen, joka päivittyisi käyttäjän lisäämien ja poistamien kategorioiden mukaan.

Kehitin työn lopussa vaihtoehtoisen ratkaisumallin, jolla voidaan ratkaista ongelmat dynaamisuuden lisäämisen kanssa. Tässä mallissa Budget- sivun nykyinen rakenne korvataan Table-View- välilehdellä. TableView viittaa oletuksena iOS:issa olevaan välilehtityyppiin, jonka rakenne koostuu Tableview- soluista. Näistä soluista jokainen on oma objektinsa, ja jokaisessa solussa voi näyttää erillisen tiedon. Esimerkiksi arrayn jokaisen merkinnän voi näyttää Table-Viewissä omana solunaan. TableViewillä olisi mahdollista saada tarvittava määrä tilaa uusille kategorioille, koska soluja voi olla rajaton määrä.

Tässä mallissa myös Set Budget- sivu määriteltiin uusiksi, koska sivun tämänhetkisen rakenteen syöttökentissä on sama ongelma kuin Budget- sivun tekstikentissä. Niitä ei pystynyt generoimaan dynaamisesti lisää. Tämän vuoksi lähdin suunnittelemaan sivua uudestaan erilaisesta näkökulmasta. Uudessa designissa sivu olisi lähtöön täysin tyhjä, ja siinä olisi vain yläkulmassa painike, jolla voisi lisätä uuden kategorian.

Painaessa painiketta sivulle tulisi alert-näkymä, johon käyttäjä syöttää kategorian nimen, rahamäärän ja seuranta-ajan. Hyväksyessä kategorian, sen syötetyt tiedot tulostuisivat tyhjään tilaan välilehdessä. Tässä tilassa käyttäjä voi painamalla kohdetta tehdä vielä muokkauksia siihen. Hyväksyessä budjetin kategorian nimet, rahamäärät ja seuranta-ajat tallentuisivat tietokantaan. Sieltä tieto siirrettäisiin näytettäväksi Budget- välilehden soluihin sivun auetessa. New Purchase- sivun datalähde päivittyisi myös tietokannan kautta, mutta siihen luettaisiin vain kategorioiden nimet. Valintarullan datalähde ei tarvitse toimintaansa varten muita tietoja.

6.3 Tekninen toteutus - Tietokannat

Tässä osiossa käydään läpi sovelluksessa käytetyt tietokantatyypit. Sen lisäksi käydään myös läpi tietokantatyyppien valinta ja valittujen ratkaisujen toteutus. Osio on jaettu kahteen osaan. Ensimmäisessä osassa käydään läpi tietokantateknologiat, joiden avulla paikallinen tiedonkäsittely voidaan toteuttaa. Tämän jälkeen vertaillaan parhaimpia löydettyjä ratkaisuja. Sen jälkeen eritellään, minkä vuoksi päädyttiin lopulta valittuihin vaihtoehtoihin. Lopuksi käydään vielä läpi valittujen ratkaisujen toteutus. Osion toisessa osassa käydään läpi sama prosessi pilvitietokannoille.

Tietokantojen teknisen haastavuuden vuoksi niiden totetukseen jouduttiin tekemään rajauksia. Toiminnoista työn aikana oli tavoitteena saada toteutettua tietokantojen alustus, tiedon vastaanotto ja tallentaminen. Pilvitietokannan kohdalla tulisi myös toteuttaa käyttäjien rekisteröinti palveluun ja sisäänkirjautuminen rekisteröidyillä tunnuksilla.

6.3.1 Paikalliset tietokannat

Sovelluksen tietokantaratkaisu päätettiin aluksi toteuttaa hyödyntäen paikallisia tietokantaratkaisuja. Ensimmäinen vaihe toteutuksessa oli kartoittaa ja valikoida teknologiat, joiden avulla paikallinen tiedonkäsittely voitaisiin toteuttaa. Tähän sisällytettiin sekä iOS:n sisäänrakennetut että Cocoapod- lisäosina ladattavat tietokantaratkaisut. Tutkinta suoritettiin referoimalla aikaisemmin tekemiäni esimerkkiprojekteja ja tutkimalla aiheeseen liittyviä dokumentaatioita. Tutkimuksen päätteeksi löydettiin 4 mahdollista ratkaisua: User Defaults, Codable- protokolla, Core Data ja Realm.

User Defaults on osa iOS:n perustoimintoja, jotka ovat sisällettynä kaikkiin iOS- projekteihin. User Defaults on oletustietokanta, joka hyödyntää iOS:in sisäänrakennettua Defaults- tietokantaa. Defaults- järjestelmä vastaa iOS:issa sovelluksen oletustilan asetusten (esimerkiksi UI-teema, äänenvoimakkuus jne.) tietojen tallentamisesta.

User defaultsin ongelma oli sen heikko soveltuvuus oliopohjaiseen tiedonkäsittelyyn. Tämän työn tiedonkäsittelyyn tarvittiin oliopohjaista datamallia. User Defaultsin kaltaisen oletustietokannan kautta on hankala hallinnoida dataobjekteja, joita tarvittaisiin työn tiedonkäsittelyssä. User Defaultsin ensisijainen käyttötarkoitus onkin tallentaa käyttäjän henkilökohtaisia asetuksia sovelluksissa, kuten esimerkiksi taustamusiikin äänenvoimakkuus musiikkia sisältävässä sovelluksessa. Näiden tietojen tallennus tapahtuu info.plist- konfigurointitiedostoon, joka vastaa myös sovelluksen verifikointiin tarvittavista tiedoista.

Codable- protokolla on iOS:in sisäänrakennettu järjestelmä, jonka avulla on mahdollista tallentaa tietoa laitteen muistiin. Codable on edistyneempi ratkaisu kuin User Defaults, koska

sen avulla pystyy käsittelemään useita kohteita kerralla. Codable ottaa vastaan syötetyn tiedon, ja kirjoittaa sen laitteen Documents- kansioon hyödyntäen encoding-tomintoa. Ladatessa tietoja Codable käyttää decodingia. Decoding vastaa toiminnaltaan encodingia päinvastaisessas järjestyksessä.

Codablen avulla oli mahdollista toteuttaa osa vaadituista tietokantatoiminnoista. Se jäi kuitenkin vajavaiseksi suunnittelessa ratkaisuja päivitettyjen budjettiarvojen laskemiseen. Codablen avulla ei pystynyt kohdentamaan tietoa yksittäisten dataobjektien tasolla. Tämä tekisi muun muassa kategorioiden summien päivittämisestä haastavaa, koska ostotapahtumia ei pystyttäisi kohdentamaan valittuhin kategorioihin.

Tämän vuoksi tietokantaratkaisua varten tarvittiin edistynyt tietokantajärjestelmä. Tietokannan tulisi tallentaa ja käsitellä dataa yksittäisten tietokantakenttien tasolla. Tätä varten identifioin kaksi vaihtoehtoa: Applen Core Data- framework ja kolmannen osapuolen kehittämä Realm.

6.3.2 Core Data vs Realm

Core Data on Applen framework iOS- käyttöjärjestelmälle, joka mahdollistaa edistyneen tiedonkäsittelyn iOS-sovelluksissa. Core Data ei ole tietokanta, vaan se on framework, jolla hallinnoidaan taustalla toimivaa SQLite-tietokantaa. SQLite- tietokantaan voi määrittää uusia dataobjekteja, joita tallennetaan tietokantaan. Nämä dataobjektit mahdollistavat myös tiedon käsittelyn objektien yksittäisten ominaisuuksien tasolla. Tämä oli kriittinen ominaisuus sovelluksen tietokantatomintojen toteutukseen.

Realm on kolmannen osapuolen mobiililaitteille kehitetty tietokantaratkaisu, joka mahdollistaa tiedonkäsittelyn mobiilisovelluksissa hyödyntäen dataobjekteja. Realm toimii samanlaisesti kuin Core Data. Aluksi sille määritellään dataobjekti. Tämän jälkeen koodin avulla dataobjektin avulla voidaan lisätä, muokata ja poistaa dataa sovelluksessa. Huolimatta kyseisten teknologioiden samankaltaisuuksista, Realm hyödyntää NoSQL- teknologiaa (Avantica 2017), kun taas Core Data perustuu SQLite- teknologiaan (Jacobs 2017).

Core Datan ja Realmin välinen vertailu suoritettiin tutkimalla Core Datan dokumentaatiota (Apple 2019c) ja Realmin omaa dokumentaatiota (Realm 2019). Tämän lisäksi vertailua tehtiin myös analysoimalla esimerkkiprojekteja YouTubesta ja Githubista. Analysoinnissa huomioitiin erilaiset tominnot ja ominaisuudet, joita molemmissa teknologioissa oli käytettävänä. Toimintojen laajuuden lisäksi myös saatavilla olevan teknisen materiaalin määrä teknologioille oli tärkeää. Mahdollisten teknisten ongelmien ratkaisemiseksi oli välttämätöntä, että esimerkiksi Stack Overflow’ssa olisi paljon esimerkkiratkaisuja, joita voitaisiin hyödyntää kohdattujen ongelmien kanssa.

Core Datan vahvuuksiin kuului sen laaja dokumentaatiotuki. Se on Applen virallinen tietokantaratkaisu iOS:ille, joten siitä löytyi paljon materiaalia iOS:n dokumentaatioista. Tämän lisäksi Core Datalle löytyy myös paljon tukimateriaalia esim. YouTubesta ja Stack Overflow'sta. Core Datassa oli myös vahvuutena Xcoden editorinäkymä, jossa oli mahdollista käsitellä sekä dataobjekteja että niiden ominaisuuksia visuaalisella hallintatyökalulla. Core Datan vahvuudeksi luettiin myös muutama löytämäni esimerkkiprojekti. Nämä projektit esittivät vaihtoehtoisia menetelmiä alustaa Core Data ja sen käyttö. Nämä menetelmät sallivat Core Datan käyttämisen yksinkertaistamisen etenkin käytetyn koodin määrässä.

Realmin vahvuuksiin kuului myös vahva dokumentaatiotuki. Realmin sivuilta löytyi kattavat dokumentaatiot, joissa käytiin läpi alustan olennaiset toiminnot ja ominaisuudet. Realmille löytyi myös paljon tukevaa materiaalia muista lähteistä. Etenkin YouTuben puolelta löytyi havainollistavia esimerkkiprojekteja. Realmissa oli myös vahvuutena hyvin yksinkertainen syntaksi ja koodin määrä, jota tarvitaan olennaisimpien toimintojen implementointiin. Tämän lisäksi siinä oli myös valmiita funktioita, kuten filter- funktio, jonka avulla oli hyvin yksinkertaista lajitella dataa toivotulla tavalla. Tämä oli etenkin työn jatkokehitykseen suunniteltuihin toimintoihin erittäin hyödyllinen ominaisuus.

Päädyin lopulta valitsemaan Core Datan työn paikalliseksi tietokannaksi. Syinä tähän oli Core Datan objektieditori- näkymä, mikä teki dataobjektien luonnista ja muokkaamisesta helppoa. Tämän lisäksi aiemmin mainitut vaihtoehtoiset alustusmenetelmät olivat tärkeä tekijä päätöksessä. Niiden avulla Core Datan käyttöä varten tarvittu koodin määrä oli merkittävästi pienempi. Minulla oli myös aikaisempaa kokemusta Core Datan käytöstä. Tämä teki sen käytöstä helppoa, etenkin kun Core Datan syntaksia saatiin yksinkertaistettua löydetyillä alustamismenetelmillä.

Realmissa oli kuitenkin ominaisuuksia, joille en löytänyt Core Datasta vastaavia toimintoja. Tämän vuoksi Realm päätettiin asentaa ja ottaa käyttöön työssä. Realmia käytettiin New Purchase- sivulla toteuttamaan ostosten kirjaaminen. Realmin avulla jokainen ostos kirjataan omaksi NoSQL- dataobjektiksi. Tämän lisäksi Realmissa oli vahvuutena filter- funktio, jonka avulla oli mahdollista yksinkertaisesti näyttää dataa halutussa järjestyksessä. Tämä sallisi toteuttaa suunnitellun Purchases- välilehden toiminnot. Lopulta tarkoituksena olisi käyttää vain yhtä tietokantaa paikallisessa tiedonkäsittelyssä. Toistaiseksi Realmia kuitenkin käytetään, kunnes saadaan selvitettyä, kuinka nämä toiminnot voitaisiin toteuttaa Core Datalla.

6.3.3 Pilvitietokannat

Tässä osiossa käydään läpi pilvitietokannat ja niiden toiminta. Tämän lisäksi osiossa käsitellään myös syitä pilvitietokantojen käyttämiseen. Lyhyesti käydään myös läpi niiden tuottama

lisäarvo sovelluksen toiminnalle ja loppukäyttäjille. Tämän lisäksi käydään läpi vielä kartoitus parhaimmista pilvitietokannoista, jotka valittiin jatkotutkimukseen pilvitietokannan valintaa varten.

Pilvitietokannat viittaavat tietokantoihin, jotka eivät fyysisesti sijaitse laitteessa, jossa sovellus pyörii. Pilvitietokantojen palvelimet sijaitsevat yleensä suurissa datakeskuksissa, joista käsin pilvitietokantojen dataa käsitellään. Tässä työssä niiden tarkoituksena oli parantaa käyttäjäkokemusta tarjoamalla käyttäjälle vaihtoehtoja sovelluksen käyttämiseen. Paikallisen tiedonkäsittelyn heikkoutena oli se, että sen toiminnot olivat rajoitettu yhteen laitteeseen.

Jos käyttäjä kirjaa ostoksia puhelimellaan ja tabletillaan, niin paikallisessa mallissa ei ole mahdollista siirtää näitä tietoja laitteiden välillä. Tämän vuoksi päätin lisätä pilvitietokantatoiminnot työhön, koska niiden avulla käyttäjä pystyy käsittelemään ja tarkkailemaan budjettiaan usealla eri laitteella. Tiedot pystytään myös reaaliaikaisesti siirtämään laitteiden välillä, joten tarvittaessa budjettia voisi käsitellä usealla laitteella lyhyen aikavälin sisällä.

Kartoitus eri vaihtoehdoista oli helppoa tehdä, koska olin jo entuudestaan työskennellyt pilvitietokantojen parissa iOS- sovelluskehityksessä. Täten tiesin jo muutaman hyvän pilvitietokantapalvelun, joita voitaisiin hyödyntää sovelluksessa. Tein vielä kartoitusta muista mahdollisista pilvitietokannoista, mutta en onnistunut löytämään sellaisia vaihtoehtoja, joita olisi voinut hyödyntää tässä työssä. Valikoimani tietokantavaihtoehdot olivat Google Firebase ja Amazon AWS Amplify.

6.3.4 Firebase vs AWS Amplify

Googlen Firebase ja Amazonin AWS Amplify valikoituivat lopulta viimeisiksi ehdokkaiksi sovelluksen pilvitietokantaratkaisuiksi. Tässä osiossa esitellään molemmat palvelut ja käydään läpi niiden olennaisimmat ominaisuudet. Lopuksi käydään läpi erinäiset, joiden pohjalta päädyin lopulta valitsemaan toisen palveluista sovellukseen.

Firebase on Googlen mobiililaitteille optimoima pilvitietokanta. Siihen sisältyy paljon erilaisia palveluja aina pilven kautta pyörivistä koneoppiteknologioista reaaliajassa päivittyviin tietokantoihin. AWS Amplify on taasen Amazonin mobiilikäyttöön optimoitu pilvipalvelu, jonka toimintoihin kuuluu mm. reaaliaikainen tietokanta ja tietokannan käytön seurannan analytiikkatyökalut (Amazon 2019a). Firebase pyörii Google Cloud- pilvipalvelulla ja AWS Amplify on osa Amazon Web Services(AWS)- pilvipalvelua, joten molemmat olivat hyvin optimoituja toimimaan suurempienkin käyttäjämäärien kanssa.

Aloitin palvelujen vertailun hintatasojen vertailusta, koska etenkin työn jatkokehityksessä sovelluksen ylläpitökustannukset ovat tärkeä tekijä. Kartoituksessa pidettiin myös positiivisena

tekijänä ilmaisversioita, jotta kehitystyön aikana ei joutuisi maksamaan palvelun käytöstä. Firebasen osalta hinnoittelu oli erittäin hyvä. Harrastus- ja testauskäyttöön tarkoitettu ilmainen Spark- maksusuunnitelma tarjoaa 1 gigan tallennustilaa, 100 samanaikaista yhteyttä ja 10 gigan edestä latauksia tietokannasta. Tämä oli soveltuva ratkaisu kehityskäyttöön.

2 laajempaa maksusuunnitelmaa olivat Flame ja Blaze. Flame maksaa kiinteänä maksuna 25€ per kuukausi ja Blaze veloittaa käytön mukaan (Google 2019a).. Nämä ovat kilpailukykyisiä hintoja tuotantotason käyttöä varten. AWS:n osalta hinnoittelu jäi melko epäselväksi, sillä en onnistunut löytämään Amplifyille omaa maksusuunnitelmaa AWS:n hinnastosivuilta Amazon 2019b). Ilmaiseksi tarjotut kapasiteetit esitellyistä tietokantatyypeistä olivat riittäviä kehityskäyttöä varten. Hinnaston epäselvästä asettelusta johtuen tuotantotason käytön hinnoista ei pystynyt tekemään tarkempaa analyysiä.

Dokumentaatiossa Firebase oli erittäin hyvällä tasolla. Dokumentaation etusivulta löytyi runsaasti valmiita oppaita tietokannan alustukseen kaikille eri tuetuille alustoille. Sen lisäksi myös Firebase API:sta löytyi hyvin kattavasti artikkeleja eri toiminnoille (Google 2019b). Amplifyn kohdalta dokumentaatiota löytyi myös runsaasti. Amplifyn etusivulla oli selkeät ohjeet, kuinka Amplify alustetaan iOS:ille. Muutenkin dokumentaatiosta löytyi paljon tietoa palvelun toiminnoista kaikille tuetuille alustoille (Amazon 2019c).

Muussa teknisessä materiaalissa, kuten YouTube- videoissa ja Stack Overflow- keskusteluissa Firebase löytyi selkeästi enemmän materiaalia. Esimerkiksi YouTubesta löytyi runsaasti erilaisia käytännön esimerkkejä Firebasen käytöstä iOS:llä. YouTuben puolella löytyi myös Amplifyille esimerkkiprojekteja mutta ne olivat pääsääntöisesti toteutettu web-sovelluksiin. Stack Overflow'n puolelta löytyi myös hakemalla Firebasea iOS:illa selkeästi enemmän aiheita kuin Amplifyille iOS:iin liittyen.

Päätöstä tehdessä pidin myös tärkeänä kriteerinä aikaisempaa kokemusta alustojen käytöstä. Firebasen kanssa olin jo entuudestaan tehnyt muutaman pienemmän projektin. Amplify taasen oli minulle täysin uusi teknologia kartoituksen aikana. En ollut käyttänyt sitä aikaisemmin ollenkaan.

Päädyin lopulta valitsemaan Firebasen sovelluksen pilvitietokannaksi. Tärkeimmät tekijät päätöksen teossa olivat aikaisempi kokemus Firebasen käytön kanssa, kattavampi tukimateriaali internetissä ja selkeä hinnasto Firebasen sivuilla. Tukimateriaali oli etenkin jatkokehityksen kannalta tärkeä. Edistyneempien toimintojen toteutuksessa tullaan tarvitsemaan tukimateriaalia ja sitä oli laajemmin saatavilla Firebaselle. Hinnaston selkeys Firebasen sivuilla oli myös tärkeä tekijä päätöstä tehdessä. Sen avulla on helpompi suunnitella jatkokehityksessä sovelluksen bisnesmallia, kun on tarkka tietämys sovelluksen ylläpidon kustannuksista.

6.3.5 Tietokantojen toteutus – Core Data & Realm

Core Datan alustus tapahtui kahdessa eri vaiheessa. Ensimmäisenä alustettiin koodi, jota tarvitaan Core Datan tallennustoimintoa varten. Kyseinen koodiblokki on alustettuna valmiiksi iOS-projekteissa. Siihen kuitenkin tehtiin muutoksia aiemmin mainittujen esimerkkiprojektien mallin perusteella.

Aluksi koodiblokki siirrettiin omaan tiedostoonsa, jonka avulla Core Datan käyttöä voitaisiin helpottaa merkittävästi tallennusfunktiota käyttäessä. Tässä tiedostossa sille luotiin uusi luokka nimeltä Categories, jonka avulla pystyttiin ohjelmoitaessa Core Datan funktioita tekemään funktiokutsuja muodossa Categories.x. Ilman tätä luokkamäärittelyä funktioiden syntaksi olisi paljon pidempi ja hankalampi lukea.

Alustuksen jälkeen ryhdyttiin konifguroimaan Core Datan datamalli-tiedostoa luomalla siihen Core Datan dataobjektit. Tiedostoa ei tarvinnut luoda erikseen, koska se oli valmiiksi generoitu työssä. Dataobjekti luotiin aluksi luomalla Entity, joka vastaa tyypiltään olio-ohjelmoinnin olioita. Siihen määritettiin attribuutteja, jotka vastaavat olion ominaisuuksia. Entity sai nimen Budget, koska sen alle tallennettaisiin kaikki budjettitiedot. Entityn attribuutit olivat sovellukseen valmiiksi asetetut kategoriat, ja ne ottivat dataa vastaan Float-desimaalilukuina.

Realmin alustus aloitettiin asentamalla sen käyttöä varten tarvittavat Cocoapodit, koska Realm ei ole osa iOS:n natiivitoimintoja. Cocoapodien osalta riitti yhden Cocoapodin asennus, jonka nimi oli RealmSwift. Realmin onnistuneen alustamisen testaamista varten käytettiin AppDelegate- tiedoston didFinishLaunchingWithOptions- elinkaarifunktiota. Sen sisälle luotiin Do/Catch- lauseke, joka tulosti konsoliin viestin sovelluksen käynnistyessä normaalisti. Virheen sattuessa konsoliin tulostui virheilmoitus.

Seuraavaksi määritettiin Realmin dataobjekti, joka sisälsi kategoriat ja niiden datatyypit. Aluksi Model- kansioon luotiin uusi tiedosto nimeltä Purchase, johon määriteltiin uusi objekti. Objektille määriteltiin 3 ominaisuutta, joita tarvittiin sovelluksen ostojen kirjaamista varten. Ominaisuudet olivat Value, joka sisälsi ostolle määritellyn summan, Category, joka sisälsi ostolle määritellyn kategorian ja Date, joka oli oston kirjausaika. Datatyypeiltään nämä olivat Integer, String ja Date.

Alustuksen jälkeen seuraavana vaiheena oli toteuttaa tietokantojen käyttö koodissa. Set Budget- sivulle luotiin aluksi funktio, joka kävi läpi kaikki syöttökentät, ja tarkisti Guard Let- lausekkeiden avulla, ettei niissä ollut tyhjiä arvoja. Tämän jälkeen arvot muunnettiin String- tyypistä Float- tyyppiin. Kun tämä oli tehty, niin sen jälkeen syötetyt tiedot tallennettiin Core Datan SQLite-tietokantaa saveContext- funktio avulla.

Realmia hyödynnettiin New Purchase- sivulla tallentamaan käyttäjän tekemät ostokset. Aluksi ostoksen summan arvo tarkistettiin Guard Let- lausekkeella ja ostoksen kategoria otettiin valitsemalla valintarullan datalähteestä sillä hetkellä valittuna oleva arvo. Tieto tallennettiin Realmiin luomalla Do/Catch- lauseke, jossa Do- osiossa luotiin uusi Realm-instanssi, jonka arvoiksi tulivat alustuksessa luodun dataobjektin kategoriat. Työssä vain Value ja Category- arvot tallennettiin, koska järjestelmän ajan lukemiseen ei ehditty perehtymään työn aikana. Lausekkeen Catch- osio tulosti virheilmoituksen, jos tallennuksessa tapahtui virheitä.

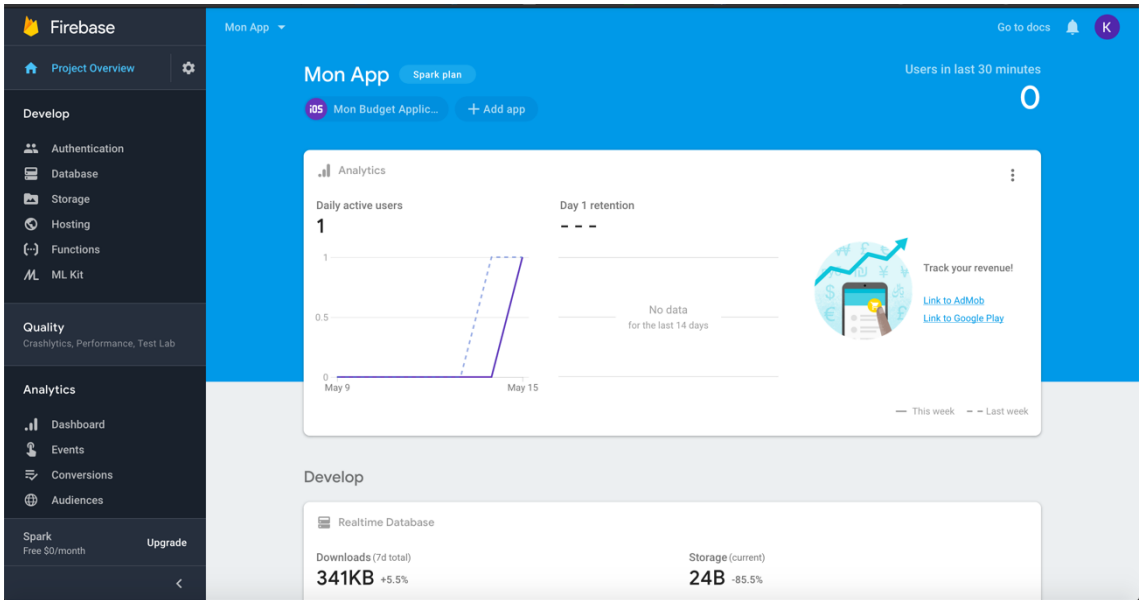
Molempien tietokantojen tiedon onnistuneeseen verifiointiin käytettiin samanlaisia menetelmiä. Tiedon siirtymisen toteuttavan koodin yhteyteen laitettiin print-lauseke, joka tulostaa viestin Xcoden konsoliin, jos kyseinen koodi suoritettiin onnistuneesti. Tällä pystyttiin verifioimaan, että tiedon tallentamisen suorittava koodi toteutettiin. Tätä tarkemmin tiedon tallentumista ei pystytty verifioimaan.

Kummankaan tietokannan kohdalla ei ole mahdollista tarkistella dataa ilman erillistä ohjelmaa. Realmin datan tarkisteluun käytetään Realm Browseria ja Core Datan tarkisteluun käytetään Datum- ohjelmaa. Kyseisten ohjelmien käyttöönottoa ei kuitenkaan ehditty tekemään, koska molemmissa ohjelmissa tarvitaan polku tiedostoihin. Nämä polut ovat syvällä Xcoden projektikansioissa, ja niiden etsimistä ei ehditty toteuttamaan työn aikana.

6.3.6 Tietokantojen toteutus – Firebase

Firebasen toteutus tapahtui karkeasti jaoteltuna kolmessa vaiheessa. Aluksi projektikansioon asennettiin tarvittavat Cocoapod-lisäosat Firebasen käyttöä varten. Sen lisäksi tehtiin myös tarvittavat alustustoimenpiteet Firebasen hallintakonsolissa. Tämän jälkeen tehtiin tarvittavat alustustoimet sovelluksen koodipohjaan, jotta sovelluksen saisi yhdistettyä Firebaseen. Viimeisenä vaiheena toteutettiin kaikki Firebasen tiedonkäsittelyn toiminnot, kuten rekisteröinti, kirjautuminen ja käyttäjän syöttämän tiedon vastaanottaminen.

Ensimmäisenä Firebasen alustuksessa asennettiin vaadittavat Cocoapodit. Firebase oli laaja kokonaisuus, joten sen asentamiseen tarvittiin 3 Cocoapodia. Esimerkiksi Realmin asennukseen riitti 1 Cocoapod. Kyseiset Cocoapodit olivat Firebase, joka sisälsi kaikki perustoiminnot ja valmiiksi määritellyt funktiot. Toinen näistä oli Firebase/Auth, joka sisälsi toiminnot kirjautumiselle ja rekisteröitymiselle. Kolmas oli Firebase/Database, joka sisälsi tietokannan käsittelyyn tarvittavat toiminnot.



Kuvio 8: Kuva Firebasen hallintakonsolista.

Seuraavaksi alustuksessa tuli alustaa sovelluksen tietokanta Firebasen hallintakonsolissa. Tämä tehtiin menemällä Firebasen sivuille ja kirjautumalla sisään. Tämän jälkeen valittiin Firebasen konsolista Add New Project. Tämän jälkeen seurasi 5-vaiheinen alustusvaihe. Siinä verifioitiin aluksi sovellus syöttämällä sen Bundle ID, joka määriteltiin sovellukselle projektikansion luonnissa. Tämän jälkeen ladattiin Firebasen generoima config- tiedosto, joka asetettiin projektihierarkiassa Supporting Files- kansioon. Tämän jälkeen alustusprosessi ehdotti Cocoapodien asentamista, mutta koska ne olivat asennettu etukäteen, voitiin siirtyä suoraan seuraavaan vaiheeseen.

Seuraavaksi toteutettiin Firebasen linkittäminen sovellukseen sovelluksen käynnistyessä. Koodi asetettiin AppDelegate- tiedoston didFinishLaunchingWithOptions- funktioon, joka suoritetaan aina sovelluksen käynnistyessä. Alustaminen oli yksinkertaista. Sitä varten tarvittiin vain yksi rivi koodia, jossa kutsuttiin Firebasen Configure- funktio. Kun tämä oli tehty, siirryttiin sen jälkeen alustuksen viimeiseen vaiheeseen. Siinä sovellus piti käynnistää kerran, jotta Firebase voisi verifioida sovelluksen ottamalla siihen yhteyden. Tämä suoritettiin käyttäen Xcoden iOS Simulatoria.

Toiminnoista toteutettiin ensimmäisenä rekisteröityminen ja navigointi. Navigoinnilla tarkoitettiin tässä kontekstissa siirtymistä Firebasen päävalikkoon onnistuneen rekisteröinnin jälkeen. Sen lisäksi toisena osana navigointia oli uloskirjautuminen sovelluksesta. Uloskirjautumisessa tulisi siirtyä Firebasen päävalikosta sovelluksen aloituspäävalikkoon. Tätä varten sovelluksen navigointirakenne tuli vaihtaa käyttämään Navigation Controller- teknologiaa.

Navigation Controllerin avulla uloskirjautumisen siirtymän toteutus oli helppoa. Log Out- painikkeen IBAction- funktioon määritettiin kaksi funktiota, joista ensimmäinen katkaisee yhteyden kirjautuneen tunnuksen yhteyden Firebaseen. Toinen funktio oli Navigation Controllerin sisäänrakennettu popToRootViewController- funktio. Tässä funktiossa toteutettiin siirtymä senhetkisestä näkymästä Navigation Controllerin root- näkymään, joka oli sovelluksessa määritetty aloituspäävalikkoon.

Kun rekisteröintiä varten tarvittava infrastruktuuri oli toteutettu, toteutettiin seuraavaksi itse rekisteröityminen. Tämä oli helppoa toteuttaa, koska tähän vaaditut toiminnot olivat sisäänrakennettu Firebasen Cocoapodeihin. Rekisteröintisivun Register- painikkeeseen liitetyyn IBAction- funktioon luotiin If/Else- lauseke. Tämä lauseke tarkisti, ettei käyttäjän syöttämissä tiedoissa ollut virheitä, kuten esimerkiksi liian lyhyttä salasanaa tai väärässä muodossa olevaa sähköpostiosoitetta. Jos tämä tarkistus meni läpi, niin sen segue- siirtymällä siirryttin Firebasen päävalikkoon.

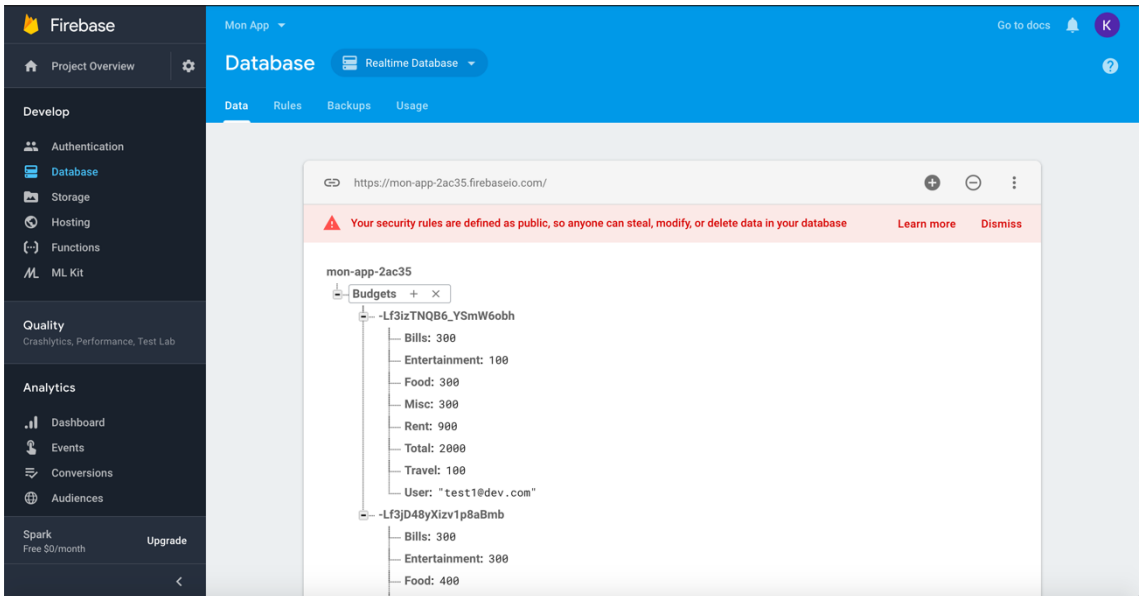
Rekisteröinnin toteuttamisen jälkeen siirryttiin toteuttamaan sovellukseen kirjautuminen luoduilla tunnuksilla. Kirjautuminen toimi samalla lailla kuin rekisteröinti. Aluksi tehtiin If/Else-lauseke, joka tarkisti käyttäjän syöttämän sähköpostin ja salasanan. Jos Firebasesta löytyi syötettyjä tietoja vastaava tunnus, niin sen jälkeen toteutettiin segue-siirtymä Firebasen puolen päävalikkoon.

Tämän jälkeen siirryttiin toteuttamaan käyttäjän syöttämän datan tallentamista tietokantaan. Tämä toteutettiin sekä Set Budget- että New Purchase- välilehdessä, koska tiedon tallentamisen pystyi toteuttamaan lähes identtisellä koodipohjalla molemmilla sivuilla. Set Budget- välilehden tallennusfunktiossa käytiin aluksi läpi kaikkien syöttökenttien arvot Guard Let-lausekkeilla, joilla varmistettiin, ettei syöttökentissä ollut tyhjiä arvoja.

Arvojen validoinnin jälkeen määriteltiin yhteys Firebaseen. Yhteyden alustuksessa määriteltiin myös tietokannan nimi, johon tietoa tulisi tallentaa. Set Budget- sivulla tietokannan nimi oli Budgets ja New Purchase- sivulla tietokannan nimi oli Purchases. Tehdessä tietokantaan ensimmäisen tallennuksen Firebase generoi automaattisesti tietokannan rakenteen hallintakonsoliin. Seuraavilla tallennuskerroilla Firebase lisäsi automaattisesti uudet kirjaukset aikaisempien kirjausten perään.

Teknisestä näkökulmasta Firebasen tiedon tallentaminen toimi kolmessa osassa. Ensimmäisessä osassa määriteltiin array. Tämän arrayn arvoiksi määriteltiin sovellukseen kirjautuneen käyttäjän sähköposti ja tallennettava dataobjekti. Set Budget-sivulla dataobjekti oli budjetinkategorioihin syötetyt summat ja New Purchase- sivulla dataobjekti oli ostoksen kategoria ja

summa. Lopuksi array tallennettiin Firebaseen JSON- formaatissa käyttäen Firebasen sisäänrakennettua childByAutoID().setValue- funktiota. Tämä funktio lisäsi arrayn tiedot funktiossa määritettyyn tietokantaan. Set Budget- sivulla tämä tietokanta oli Budgets ja New Purchase-sivulla tietokanta oli Purchases. Ennen tiedon lähettämistä suoritettiin vielä If/Else-lauseke, joka tarkisti, ettei tiedon siirtymisen aikana tapahtunut virheitä.



Kuvio 9: Esimerkki tallennetuista JSON- objekteista Firebasessa. Tietokannan turvallisuussäännöt määritellään työn jatkokehityksessä.

Viimeisenä vaiheena Firebasen toteutuksessa oli verifioida, että tieto oli siirtynyt onnistuneesti tietokantaan. Tämä toteutettiin kahdessa osassa. Ensimmäisessä osassa asetettiin tiedon siirtymisen toteuttavan koodin yhteyteen print-lauseke, joka tulostaa konsoliin viestin koodin suorituessa. Tällä pystyttiin tarkistamaan, että sovelluksen koodipohjassa ei ollut ongelmia ja koodi suoritettiin oletetusti.

Tämän jälkeen siirryttiin Firebasen hallintakonsoliin, jossa Database- välilehdessä voitiin tarkkailla kaikkia tietokantaan tehtäviä tallennuksia. Nollausbugin ratkaisemisen jälkeen sekä kirjatut ostokset että budjetit tallentuivat tietokantaan reaaliaikaisesti. Kaikki kirjatut objektit myös säilyivät kannassa myös sovelluksen käynnistämisen ja sammuttamisen yhteydessä, joten tiedon tallenus oli saatu toteutettua onnistuneesti.

7 Työn arviointi

Tässä osiossa käydään läpi ensimmäisenä oma arvioni työn tuloksista ja kehityskohteista. Sen jälkeen tulee työn toimeksiantajan arvio työstä ja sen tuloksista. Lopuksi käydään läpi sovelluksen testaus ja testauksen tulokset.

7.1 Oma arvio

Työn aikana saatiin hyvin tuloksia aikaan, ja suurin osa suunnitelluista tavoitteista saavutettiin. Toiminnoista saatiin toteutettua lähes kaikki suunnittelut toiminnot. Tämän lisäksi sain myös luotua selkeän suunnitelman sille, miten tällä hetkellä puuttuvat toiminnot voidaan toteuttaa. Ideoinnin, määrittelyn ja designin puolesta sain kaiken tehtyä. Saimme tuotettua myös hyvin kokeellisia toimintoja ja jatkokehitysideoita, jotka voivat potentiaalisesti tuottaa lisäarvoa sovellukselle. Toteutetut toiminnot työssä olivat visuaalisen ulkoasun suunnittelu ja toteutus, välilehdet, navigointi, MVC-malli projektihierarkialle, tietokantojen alustus, Firebaseen rekisteröityminen ja kirjautuminen, tietokantoihin syötetyn tiedon vastaanotto ja tallennus. Huolimatta saavutuksista työhön jäi kuitenkin joitakin puutteita verrattuna alkuperäiseen suunnitelmaan.

Toiminnoissa sovelluksessa on vielä jonkin verran puutteita. Isona puutteena on järjestelmän ajan lukeminen ja sen hyödyntäminen funktioissa. Tätä toimintoa tultaisiin hyödyntämään kaikissa erilaisissa tällä hetkellä suunnitteluissa ratkaisuissa.Tämän vuoksi se on yksi tärkeimmistä jatkokehityksen kohteista. Toinen iso puute on dynaamisuuden puute kategorioiden lisäämisessä. Tällä hetkellä sovelluksessa ei pysty lisäämään uusia kategorioita. Tähän on kuitenkin tehty jo selvitys ja kehitysehdotus, kuinka toiminto voitaisiin toteuttaa hyödyntäen taulukkokenttiä ja niiden soluihin pohjautuvaa rakennetta.

Tämän lisäksi vielä yhtenä puutteena sovelluksessa on käyttäjän tallentamien tietojen näyttäminen käyttöliittymässä. Tällä hetkellä sovellukseen syötettyä tietoa pystyy katselemaan vain joko Firebasen hallintakonsolissa tai SQLite- dataa näyttävällä erillisellä ohjelmalla. Tämä ei kuitenkaan ollut suuri ongelma työssä, koska aluksi suunniteltu tiedon näyttämisen malli ei toiminut halutulla tavalla. Täten sen olisi muutenkin joutunut toteuttamaan uudestaan.

Muihin puutteisiin sovelluksessa kuuluu tällä hetkellä muun muassa visuaalisen ulkoasun ja designin keskeneräisyys Set Budget- ja Budget-välilehdissä. Sivujen tarkkaa ulkoasua ei ole vielä suunniteltu. Tämän lisäksi visuaalisuuteen liittyvä puute työssä on visuaalisen ulkoasun epätasaisuus. Tällä hetkellä vasta pilvitietokantaa käyttävä puolisko sovelluksesta on muunnettu käyttämään uutta väriskeemaa ja Navigation Controller- navigointia. Aloituspäävalikko ja kaikki paikallisen puolen välilehdet käyttävät tällä hetkellä yhä vanhaa väriskeemaa. Paikallisen puolen välilehdet käyttävät myös yhä käsin tehtyä navigointirakennetta.

Kokonaisuutena työssä onnistuttiin hyvin tavoitteiden saavuttamisessa. Olennaisimmat toiminnot ja sovelluksen rakenne saatiin luotua, ja tietokannat saatiin alustettua toimivaan tilaan. Onnistuin myös kartoittamaan ja testaamaan työn aikana tehdyn mallin dynaamisuuden toimi-

vuuden. Sain tämän lisäksi myös tehtyä selvityksen siitä, kuinka työn aikana luodun mallin ongelmat voidaan korjata muilla teknologioilla. Myös toteuttamatta jääneiden toimintojen toteutukseen on tehty kartoitukset tarvittavista teknologioista.

7.2 Toimeksiantajan arvio

Tässä osiossa on lyhentämättömänä sovelluksen toimeksiantajan oma arvio työstä ja sen tuloksista. Arviossa on huomioitu sekä ideointi- että suunnitteluvaiheen työ, että myös tekninen toteutus ja sen tulokset. Tämän lisäksi on myös arvioitu jatkokehitysehdotukset ja suunnitelmat työn jatkosta ONT:n jälkeen.

Ajatus Konstan opinnäytetyön ohessa kehittämästä mobiilisovelluksesta syntyi syksyn 2017 aikana. Tuolloin itselleni heräsi ajatus mobiilisovelluksesta, jonka avulla kuluttaja pystyy seuraamaan omaa ostokäyttäytymistään, sekä rahankäyttöään helposti reaaliajassa niin viikko-, kuin kuukausitasollakin. Ajatuksena oli näet tulevaisuudessa perustaa muutaman hengen startup-yritys, jonka ydinliiketoimintaa olisi kyseisen sovelluksen kehittäminen, sekä myynti ja markkinointi.

Esittelin Konstalle alkuperäistä ideaa kyseisestä sovelluksesta syyskuussa 2017. Tuolloin sovellus oli vielä puhtaasti ajatuksen tasolla, mutta piakkoin alkoi ajatusten oheen syntyä konkretiaakin. Syksyn ja talven aikana kehittelimme sovellusta eteenpäin, ja vaihdoimme ajatuksia ulkonäöstä, sovelluksen rakenteesta, kuin keskeisimmistä ominaisuuksistakin. Kehitystä syntyi vuorovedoin, joskin olin itse enemmän vastuussa ideoiden kehittämisestä sekä niiden jalostuksesta, Konstan kehittäessä sovellusta parhaaksi näkemällään tavalla kehitysehdotuksieni pohjalta.

Konstan opinnäytetyön tuotoksena syntynyt prototyyppisovellus tuntuu lupaavalta, siinä on erittäin paljon potentiaalia mahdollisen tulevan tuotekehityksen kannalta. Keskeisimmät ominaisuudet on implementoitu prototyyppiin toimivalla tavalla, ja ne vastaavat alkuperäisiä suunnitelmia varsin hyvin. Kehittäessämme sovellusta eteenpäin mietimme monia erilaisia ominaisuuksia ja niiden toimivuutta, ja onkin ilo nähdä, kuinka ne nyt toteutuvat silmien edessä varsin mallikkaasti. Opinnäytetyöprojekti on synnyttänyt toimivan sovelluksen rungon, jonka pohjalta tulevaa tuotekehitystä on mahdollista jatkaa äärimmäisen vahvalla pohjalla.

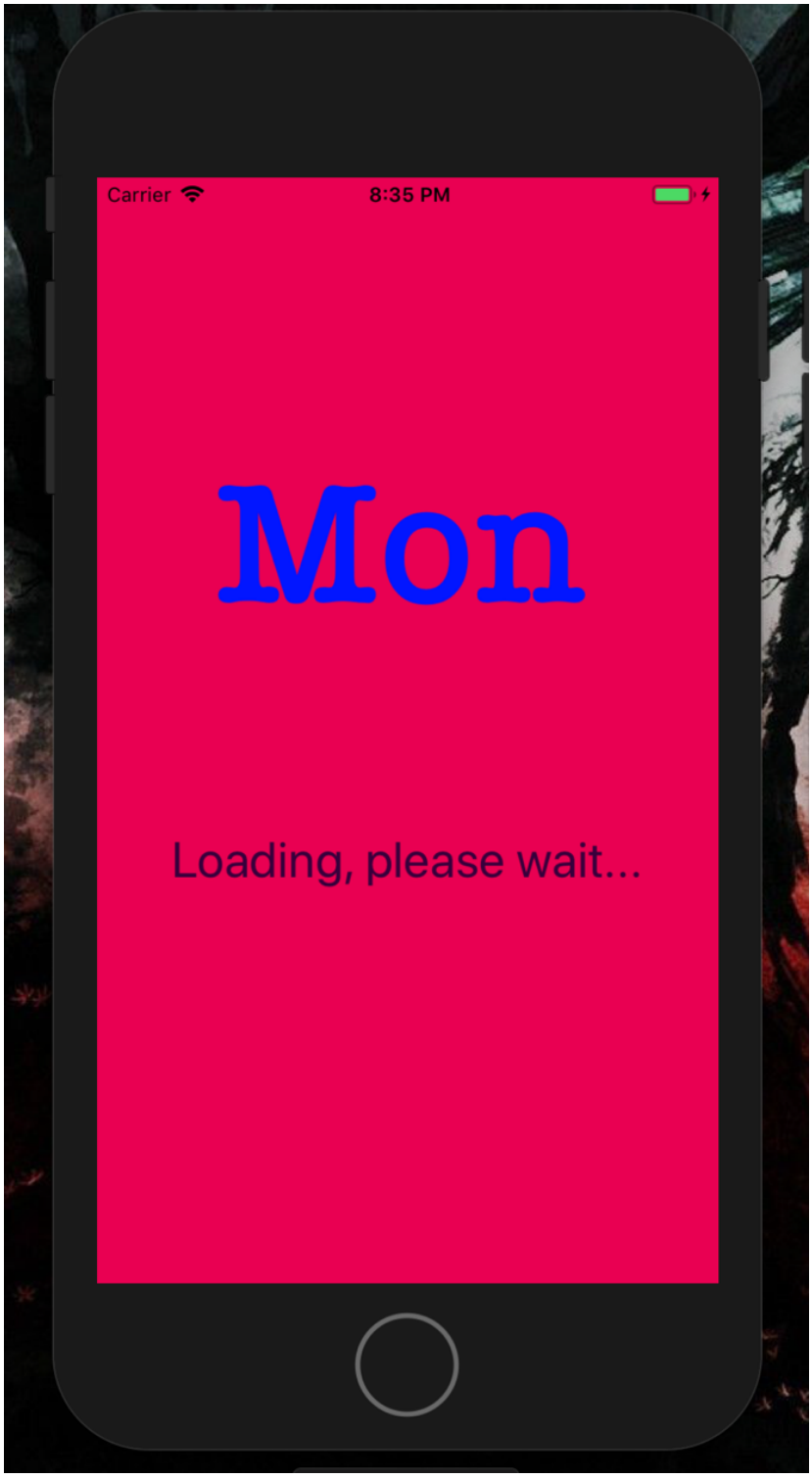
Erityisen upeaa oli huomata, kuinka toimivalle pohjalle sovelluksen keskeisimmät ominaisuudet oli perustettu. Sovellusta oli kehitetty siten, että sen toiminnoille on mahdollista perustaa laajempiakin kokonaisuuksia, kuten käyttäjätietokannan rakentaminen pilvipalveluun. Tämä puolestaan mahdollistaa sovelluksen antamien mahdollisuuksien laajentamisen kaupallista lanseerausta silmällä pitäen. Toimivan liiketoimintamallin rakentaminen on helpompaa, kun keskeisimmät ominaisuudet on rakennettu luotettavalle pohjalle jo alkuvaiheesta lähtien.

Upeaa oli myös nähdä Konstan joustavuus ja avoimuus sovelluksen laajamittaista kehitystä silmällä pitäen. Uudet ajatukset ja ideat on viety oikeastaan lähes sellaisenaan sovellukseen, eikä niiden toteuttamisessa ole ollut juuri minkäänlaisia hankaluuksia. Tämä puolestaan omalta osaltaan puhuu hänen ammattitaidostaan, ja sitoutumisestaan tietoteknistä alaa kohtaan.

7.3 Testaus

Sovelluksen kohtuullisen yksinkertaisen rakenteen vuoksi automatisoitua testaamista ei käytetty työssä, vaikka sovelluksen projektikansioon alustettiinkin testaustyökalut. Näiden avulla olisi mahdollista luoda algoritmeja ja skriptejä, jotka automaattisesti testaavat kohdennettuja toimintoja. Etenkin laajemman mittakaavan projekteissa nämä työkalut ovat erittäin hyödyllisiä, sillä niiden avulla pystyy testaamaan osia sovelluksista. Työssä toimintoja ei kuitenkaan ollut niin paljoa, että niihin olisi tarvinnut automatisoitua testaamista. Totesin manuaalisen testaamisen olevan riittävää tämän työn tarpeisiin.

Yksinkertaisuus oli myös toinen tekijä päätöksessä, koska automatisoidut testaustyökalut tulee ohjelmoida erikseen jokaista käyttötarkoitusta varten. Tämän vuoksi testaaminen suorittettiin käyttämällä kahta Xcodessa valmiina olevaa teknologiaa: Sideloading ja iOS Simulator. Näitä kahta menetelmää hyödyntäen sovelluksen toimivuus testattiin manuaalisesti. Sovellusta testattiin luomalla uusia käyttäjätunnuksia, kirjautumalla niillä sovellukseen ja kirjaamalla ostoksia ja budjetteja sekä Firebaseen että Core Dataan.



Kuvio 10: Esimerkkikuva Simulatorista pyörittämässä sovellusta iPhone 8-simulaatiolla.

iOS Simulator on Xcodeen sisäänrakennettu simulaatiotyökalu, jonka avulla on mahdollista simuloida mitä tahansa iOS- laitetta. Simulatorin toimintaprosessi on kokonaisuutena yksinker-

tainen. Aluksi Xcodessa valitaan haluttu simuloitava laite, jolla sovellusta pyöritetään. Laitteen valinnan jälkeen Xcoden kääntäjä kääntää koodin, ja sovellus käynnistyy tämän jälkeen simuloidussa laitteessa.

Sideloadingilla tarkoitetaan Xcodessa olevaa metodia, jolla voidaan ladata julkaisemattomia iOS-sovelluksia iOS:ää käyttäviin laitteisiin. Prosessissa iOS-laite yhdistetään tietokoneeseen, josta laitteseen ladataan sovelluksen pyörittämiseen tarvittavat tiedostot. Tämän jälkeen sovellusta pystyy käyttämään laitteella kuin mitä tahansa App Storesta ladattua sovellusta.

Testauksen tulokset olivat positiivisia. Testaus suoritettiin iOS Simulatorilla, sillä testausta varten ei ollut saatavilla soveltuvia iOS- laitteita. Sovelluksen jatkokehityksen aikana tullaan kuitenkin tarkoituksena hyödyntää sideloadingia muutaman tekijän vuoksi. Ensimmäinen syy on saada tarkka käsitys siitä, kuinka helppoa sovelluksen käyttö on fyysisellä laitteella käyttäen sormia. Simulatorilla käytetään sovellusten käyttöön hiirtä ja näppäimistöä.

Toinen syy on Simulatorin mahdollinen vääristymä simuloitujen laitteiden tehoissa. iOS Simulator ja sen pyörittämät mallinnukset ovat tarkka kopio iOS-laitteista. On kuitenkin mahdollista, että Simulator käyttää sitä pyörittävän tietokoneen laskentatehoja sovellusten pyörittämiseen. Applen dokumentaatio ei antanut selkeätä vastausta aiheeseen. Tämän mahdollisen ongelman vuoksi on olennaista, että sovellusta päästään testaamaan myös oikealla iOS-laitteella. Näin voidaan varmistaa, että sovellus pyörii yhtä hyvin iOS-laitteilla kuin Simulatorin mallinnuksilla.

Firebasesta ei myöskään onnistuttu löytämään ongelmia sovelluksen testaamisen aikana. Firebasen testaamiseen luotiin useita testitunnuksia, joita käytettiin sovellukseen kirjautumiseen. Kaikki testit menivät läpi, eikä niiden autentikoinnissa syntynyt ongelmia. Kohtuullisen pienestä toimintojen määrästä johtuen oli helppoa käydä läpi kaikki toiminnot ja testata niiden toimivuus.

Tietokannan testaamista varten Set Budget- ja New Purchase- välilehtien tekstikentät muokattiin niin, että niihin syötetty data siirtyi Firebaseen Confirm- painiketta painaessa. Uudet kirjaukset näkyivät aina nollausongelman korjaamisen jälkeen onnistuneesti Firebasessa, joten tiedon tallennus toimi. Tämän myötä voitiin todeta, että Firebasen käyttäjänhallintaan liittyvä infrastruktuuri ja perustoiminnot olivat kunnossa. Täten Firebasen kohdalla voidaan jatkokehityksessä siirtyä toteuttamaan edistyneempiä toimintoja.

8 Johtopäätökset ja työn tulokset

Tässä osiossa käydään aluksi läpi työn tulokset. Seuraavaksi käydään läpi johtopäätökset, mitä työn toteutusvaiheen tuloksista johdettiin. Viimeisessä osiossa käydään läpi johtopäätöksiä omasta ammatillisesta kehittymisestäni työn aikana.

8.1 Työn tulokset

Suunnittelu- ja määrittelyvaihe onnistui hyvin, sillä prosessi iteroitiin useamman kerran työn aikana. Jokaisella iteraatiolla saatiin luotua uutta lisäarvoa tuottavaa sisältöä sovellukseen. Tämän vaiheen tulokset olivat merkittäviä koko työlle. Siinä sain yhdessä työn toimeksiantajan kanssa hahmoteltua alkuperäisen sovellusidean pohjalta sovelluksen rakenteen ja sen visuaalisen designin.

Sen lisäksi saimme määriteltyä sovelluksen ydintoiminnot, joiden pohjalta oli mahdollista rakentaa sovelluksen prototyyppiä Xcodessa. Sovellusrakenteen ja ydintoimintojen lisäksi saimme luotua myös toimintoja, joita voidaan sisällyttää sovellukseen jatkokehityksessä. Esimerkkinä näistä toimii taulukkovälilehti, johon tulisi kirjattuna kaikki käyttäjän sovellukseen kirjaamat ostokset.

Näiden lisäksi ideoimme myös kokeellisia toimintoja, joita voi mahdollisesti lisätä sovellukseen tulevaisuudessa. Esimerkkinä näistä toimii iOS:in CoreML-frameworkin käyttö tekoälyalgoritmin kehittämiseen, jolla voisi skannata puhelimen kameralla ostosten kuitteja. Tekoäly klasssifioisi automattisesti ostoksen oikeaan kategoriaan ja kirjaisi sen sovellukseen.

Työn lopussa kävimme myös läpi työn senhetkisen tilanteen ja sen tärkeimmät kehityskohteet. Kävimme läpi tärkeimpiä muutoksia, mitä sovellukseen tulisi toteuttaa. Saimme kehitettyä niihin ratkaisuja hyödyntäen teknistä osaamistani ja toimeksiantajani visioita sovelluksen kehittämisestä.

Sovelluksen rakenne oli onnistunut kokonaisuus. Työhön saatiin sisällytettyä pilvitietokantaversio ja SQLiteen pohjautuva paikallinen versio. Molempiin versioihin saatiin myös toteutettua kaikki välilehdet. Näiden lisäksi saatiin toteutettua rekisteröintisivu, kirjautumissivu ja aloituspäävalikko. Ainoa puuttuva välilehti sovelluksessa on tällä hetkellä Purchases- välilehti, johon tulisi luettelo kaikista käyttäjän tekemistä ostoksista.

Navigoinnin toteutuksessa onnistuttiin myös hyvin. Sovellusrakenteen toteuttamisen lisäksi onnistuin myös luomaan selkeän navigaation kaikkien välilehtien välille. Navigoinnissa pystyy

liikkumaan kaikkiin välilehtiin. Myös teknisesti haastavimmat navigointivaiheet saatiin toteutettua. Nämä olivat rekisteröinnin/kirjautumisen jälkeen tapahtuva päävalikkoon siirtyminen ja uloskirjautumisessa tapahtuva aloituspäävalikkoon siirtyminen. Navigation Controllerin ja NavBarin onnistunut integrointi sovellukseen oli myös merkittävä onnistuminen.

Työssä saatiin myös toteutettua sekä SQLite-pohjaisten paikallisten tietokantojen että pilvipohjaisen Firebase- tietokannan alustaminen. Molemmat tietokannat olivat työn päätteeksi toimivia. Niiden käyttöön vaadittavat alustukset olivat toteutettu onnistuneesti. Firebaseen onnistuttiin lisäksi toteuttamaan sekä rekisteröityminen että kirjautuminen. Näiden lisäksi molempiin tietokantoihin onnistuttiin työn aikana tallentamaan syötettyä tietoa onnistuneesti. Tiedon tallentuminen pystyttiin myös verifioimaan molemmissa tietokannoissa.

Isoimpana puutteena työssä oli dynaamisten kategorioiden puute. Dynaamisia toimintoja ei muutenkaan onnistuttu työssä toteuttamaan. Työssä onnistuttiin kuitenkin tekemään kattava selvitys siitä, kuinka puuttuvat toiminnot voitaisiin toteuttaa. Set Budget- ja Budget-sivuille saatiin suunniteltua alustavat mallit, joilla dynaamisuus pystytään toteuttamaan. Dynaamisuuden puuttumisen lisäksi muut puutteet olivat järjestelmän ajan lukemisen puuttuminen ja sovelluksen uudisteutun ulkoasun viimeistely.

8.2 Toteutusvaiheen johtopäätökset

Toteutusvaiheen johtopäätöksistä olennaisin oli tekstikenttien ja syöttökenttien dynaamisen generoinnin mahdottomuus. En löytänyt tutkimalla dokumentaatioita ja Stack Overflow'ta ratkaisua, jolla olisi ollut mahdollista luoda dynaamisesti uusia teksti- ja tekstinsyöttökenttiä. Tämän vuoksi käytin työn lopussa merkittävästi aikaa vaihtoehtoisen ratkaisun kehittämiselle. Sain työn päätteeksi alustettua TableView- teknologiaan pohjautuvan ratkaisun, jonka avulla ongelmat dynaamisten kategorioiden kanssa voidaan ratkaista.

TableViewien avulla on mahdollista näyttää dataa ja esittää sitä erilaisissa formaateissa. TableViewin hallinta on myös helpompaa, kuin manuaalisesti ohjelmoida vastaavanlaisia toimintoja. Näin hyödyntäen iOS:n sisäänrakennettuja toimintoja saatiin ratkaistua ongelmat dynaamisten toimintojen kanssa.

TableViewin toteuttamista varten jouduttiin suunnittelemaan myös uusiksi sekä Set Budget- että Budget- välilehtien rakenne ja toiminnot. Tällä hetkellä Set Budget- välilehden funktiot lukevat läpi kaikkien syöttökenttien arvot. Tämän jälkeen funktiot validoivat datan ja kirjaavat tiedot tietokantaan. Tämä toimii hyvin valmiiksi asetettujen syöttökenttien kanssa, mutta dynaamisessa mallissa se ei olisi toimiva menetelmä.

Ohjelmoinnin puolesta joudutaan tekemään isoja muutoksia koodiin. Valmiiden tekstikenttien lukemisen sijaan tarvitaan For Each- luuppilauseke, joka käy läpi kaikki Set Budget- sivulla asetetut kategoriat. Jokainen kategoria lisätään luupin aikana tietokantaan. Tämän lisäksi tarvitaan erillinen funktio, joka kerää kategorioiden nimet, ja lähettävät ne eteenpäin New Purchase- sivun valintarullan datalähteeseen.

Visuaalisuuden puolesta molemmat sivut tuli suunnitella uudestaan. Set Budget- sivulle ehdotuteuksi malliksi tuli tyhjä TableView- näkymä, jossa olisi vain pluspainike uusien kategorioiden lisäämiselle. Uudet kategoriat lisätään allekkain kustomoiduilla TableView- soluilla. Tämä mahdollistaisi kategorioiden muokkaamisen niiden luonnin jälkeen. Budget- välilehdessä käytettäisiin TableViewiä näyttämään taulukkomuodossa asetetut kategoriat. Vaihtoehtoisesti sivulla voitaisiin näyttää kategoriat erilaisten graafien avulla hyödyntäen Charts API- lisäosaa.

Viimeiseinä johtopäätöksenä työstä oli Cross Platform- alustojen käytön selvittäminen. Jos tulevaisuudessa päätämme kehittää sovelluksesta Android- version, jouduttaisiin silloin samanaikaisesti käsittelemään kahta eri koodikantaa. Tämän vuoksi pyritään tutkimaan, onko Cross Platform parempi ratkaisu natiivikehitykseen verrattuna. Koodikannan hallinnan puolesta se tekisi työn jatkamisesta huomattavasti helpompaa ja se sallisi myös Android-version toteutuksen luonnin ilman erillistä ohjelmointia.

8.3 Ammatillinen kehittyminen

Työ oli opettavainen kokemus iOS-sovellusten kehittämisestä. Ennen tätä työtä olin tehnyt iOS-sovelluksia mallin mukaan seuratessa ohjevideoita. Tässä työssä pääsin hyödyntämään osaamistani ja soveltamaan taitojani oikean sovellusidean kehittämisessä. Sovelluksen tekeminen ilman valmista mallia kehitti myös merkittävästi ongelmanratkaisukykyjäni.

Työ opetti myös hallinnoimaan laajoja kokonaisuuksia sovelluskehitystyössä. Tämän lisäksi työn aikana myös työskentelytapani tehostuivat. Työssä pääsin toteuttamaan sovelluskehitysprojektin aina alun määrittelyvaiheista lopulliseen tekniseen toteutukseen asti. Tämän myötä sain selkeän kokonaiskuvan siitä, kuinka sovelluskehitysprojekteja tulee toteuttaa. Työn alussa aikaa kului turhaa erilaisiin virheisiin työtavoissa, koska silloin minulla ei ollut vielä riittävää kokemusta kokonaisen projektin hallinnoinista. Tulevaisuudessa uskon parjääväni vastaavanlaisten töiden kohdalla paremmin. Tähän vaikuttaa sekä nykyinen tietämykseni sovelluskehitysprosessista että tehostuneet työtapani.

Työ opetti myös sovelluskehityksen haastavuudesta ja osoitti, että päteväksi sovelluskehittäjäksi kehittyminen vaatii paljon työtä, harjoittelua ja pitkäjänteisyyttä. Minulla oli työn alussa lähtöön hyvä perusosaamistaso iOS-sovellusten kehittämisestä. Huomasin työn aikana

kuitenkin useamman kerran, että kokemuksestani huolimatta minulla oli vielä merkittävästi kehitettävää osaamisessani.

9 Kehityskohteet ja työn jatko

Tässä osiossa käydään yksitellen läpi erinäiset kehityskohteet, joita sovelluksessa oli työn päätyttyä. Tämän lisäksi käydään läpi ratkaisuja, joiden avulla nämä kohteet voidaan toteuttaa. Ensimmäiseksi käydään läpi visuaalisuuteen ja designiin liittyvät parannukset. Sen jälkeen käydään läpi teknisen puolen kehityskohteet. Tämän jälkeen käsitellään erilaisia suunniteltuja lisätoimintoja. Viimeiseksi esitellään myös erilaisia työmenetelmiä ja teknologioita, joita voitaisiin hyödyntää työn jatkokehityksessä.

Visualisuuden yhtenäistäminen on ensimmäinen kehityskohde työssä, koska tällä hetkellä aivan projektin lopussa tehdyssä päätöksessä muokata työn ulkoasua päädyimme vaihtoehtoiseen ratkaisuun ulkoasun värimaailman kanssa. Työn puitteissa ehdin kuitenkin muuntaa vasta noin puolet sovelluksen sivuista uuden design-ratkaisun mukaiseksi. Tällä hetkellä vielä kirjautumis – ja rekisteröintisivut, päävalikko ja kaikki paikallisen mallin sivut ovat yhä alkuperäisen designin mukaan tyyliteltyjä. Ensisijaisena jatkokehitystehtävänä onkin muuttaa kaikki loput sivut uuden designin mukaisiksi, jotta sovelluksen visuaalinen ulkoasu saadaan yhtenäistettyä.

Visuaalisuuden viimeistelyn jälkeen tärkeimpänä kehityskohteena on toteuttaa sovelluksen puuttuvat toiminnot. Ensimmäiseksi toteutetaan dynaamiset kategoriat ja sen vaatimat toiminnot työn lopussa kehitetyn mallin mukaisesti. Dynaamisten kategorioiden lisäksi toteutetaan laitteen ajan seuraaminen ja sen hyödyntäminen funktioissa. Sovelluksen toiminnan kannalta on välttämätöntä, että pystytään seuraamaan kategorioiden päättymispäiviä ja laitteen aikaa.

Puuttuvien perustoimintojen toteutuksen jälkeen seuraava kehityskohde työlle on Purchases-välilehden luonti. Tavoitteena on lisätä sekä Firebasen että paikallisen tietokantamallin päävalikoihin 4. välilehti, joissa näytetään kaikki käyttäjän kirjaamat ostokset. Aluksi sivulla tulee olemaan kaikki ostokset yhtenä luettelona järjestettynä uusimmasta ostoksesta vanhimpaan. Myöhemmin sivun tulisi näyttää ostokset luettelossa samalla lailla kuin esimerkiksi pankkien mobiilisovelluksissa. Niissä on merkittynä välipalkeilla kuukaudet, mikä helpottaa ostosten läpikäymistä siinä vaiheessa, kun niitä on kertynyt suurempia määriä.

Sovelluksesta puuttui työn lopussa Firebasen käyttämiseen liittyen pieniä käyttäjäkokemusta parantavia tekijöitä. Tällä hetkellä sovellus ei anna käyttäjälle tietoa kirjautumisen tai rekisteröinnin epäonnistumisen syistä. Jos käyttäjä esimerkiksi asettaa liian lyhyen salasanan tai

sähköpostiosoite on väärässä formaatissa, sovellus antaa tästä geneerisen virheilmoituksen. Tämä tulisi korvata jatkokehityksen aikana kustomoiduilla virheilmoituksilla, jotka kertovat ongelman tarkan syyn käyttäjälle. Niiden tulisi myös antaa ehdotuksia käyttäjälle ongelman ratkaisemiseen. Jos esimerkiksi käyttäjän syöttämä salasana on liian lyhyt, silloin annetaan virheilmoitus, jossa todetaan, että salasanan täytyy olla vähintään 6 merkkiä pitkä.

Viimeinen merkittävä lisätoiminto on widget-toiminnallisuus. Widgetillä tarkoitetaan iOS-laitteen etusivulle lisättäviä pieniä ikkunoita, joiden avulla voidaan toteuttaa yksinkertaisia toimintoja sovelluksesta. Tarkoituksena olisi lisätä widgetinä käyttäjälle mahdollisuus kirjata uusia ostoksia suoraan laitteen etusivulta. Etenkin joutuessa kirjaamaan useita ostoja lyhyen ajan sisään, ei käyttäjäkokemus ole paras mahdollinen, jos jokaisen ostoksen kohdalla joutuu aina siirtymään New Purchase-välilehteen.

Time To Completionin minimointi oli työn lopussa kehityskohteita miettiessä tärkeä asia. Sen vuoksi suunnittelin työn loppuvaiheessa vaihtoehtoisen navigaatioratkaisun, jonka avulla saadaan nopeutettua liikkumista sovelluksen sisällä. Päävalikon sijaan kaikki sivut ovat käyttäjän saatavilla ruudun alalaidassa sijaitsevan valikon kautta. Tämä auttaa käyttäjää sovelluksen käytössä ja parantaa sovelluksen käyttäjäkokemusta, sillä sen avulla jokaiseen yksittäiseen toimintoon kuluva aika ja painallusten määrä saadaan laskettua. Päävalikon ja back-painikkeiden kautta tapahtuva navigointi saadaan näin poistettua. Tämä malli laskee merkittävästi sovelluksen käyttöön vaadittavien painallusten määrää.

Työn aikana ei hyödynnetty minkäänlaista versionhallintajärjestelmää. Tämä oli ongelma etenkin työn puolivälissä, jossa tuntemattomasta syystä johtuen sovellusta ei saanut avattua lainkaan Xcodessa. Gitin kaltaisen versionhallintajärjestelmän avulla tämä ongelma olisi ollut ratkaistavissa palauttamalla sovelluksen aikaisempi versio. Etenkin työn jatkokehityksessä versionhallinnan käyttö on välttämätöntä. Sen aikana testataan erilaisten kokeellisten toimintojen lisäämistä sovellukseen. Jos näiden toteuttamisessa tulee ongelmia, joita ei pystytä korjaamaan, voidaan sovellus palauttaa takaisin alkuperäiseen tilaansa.

Sovelluksen iOS-version valmistuttua sovelluksesta kehitetään Android-versio, jos iOS-versiolle löytyy riittävästi kysyntää. Android-version luonti on kuitenkin teknisesti haastavaa, koska minulla on vain pintapuolista kokemusta Android-kehitystyöstä. Android- version kehitys vaatisi Android- käyttöjärjestelmän ja Kotlin- ohjelmointikielen opettelun. Tästä syystä olen tutkinut, olisiko kannatavampaa siirtää sovelluksen kehitys jatkossa Cross-platform- alustalle. Cross-platform- kehitystyön etuna on sovelluksen samanaikainen kehittäminen molemmille käyttöjärjestelmille.

Cross-platform- alustoista valikoin jatkotutkimukseen kolme vaihtoehtoa: Xamarin Forms, Flutter ja React Native. Kaikissa kolmessa alustassa on hyvät ja huonot puolensa. Esimerkiksi Xamarin Formsissa on erittäin laaja valikoima kolmansien osapuolien kehittämiä UI- kirjastoja. React Native taasen käyttää Javascriptiä kehityskielenään (Github 2019), josta minulla on jo aikaisempaa käyttökokemusta. Flutter hyödyntää Googlen Material Designia, mikä sallii laadukkaiden käyttöliittymien luonnin. Pikaisen tutustumiseni perusteella siinä on myös monipuoliset animointityökalut.

Cross-platformille siirtyminen on iso päätös. Sen vuoksi on tärkeää tehdä tarkka kartoitus parhaasta alustasta. Tämän lisäksi täytyy myös tehdä selvitys siitä, onko tehokkaampaa siirtyä Cross-platformille vai jatkaa työn kehitystä natiivialustoilla. Molemmissa vaihtoehdoissa täytyy opetella kokonaan yksi uusi alusta ja siihen liittyvät teknologiat. Työn aikana kehityin Swiftin ja iOSin käytössä merkittävästi. Tämän vuoksi natiivikehityksessä joutuisin myös opettelemaan vain yhden uuden alustan, eli Androidin.

10 Lähteet

10.1 Kirjalliset lähteet

Banga, C. & Weinhold, J. 2014. Essential Mobile Interaction Design. New York City: Pearson Education

Garrett, JJ. 2011. User-Centered Design For Web And Beyond. New York City: Pearson Education

Mcwherter, J. & Cowell, S. 2012. Professional Mobile Application Development. New York City: John Wiley & Sons

Stevens, C. 2011. Designing for the iPad: Building applications that sell. New York City: John Wiley & Sons.

10.2 Sähköiset lähteet

Baker, J. 2017. Skeuomorphic Design – A controversial UX approach that is making a comeback. Viitattu 10.3.2019.
https://medium.muz.li/skeuomorphic-design-a-controversial-ux-approach-that-is-making-a-comeback-a0b6e93eb4bb

Scott, D. 2018. A Guide To The Artist's Color Wheel. Viitattu 15.3.2019.
https://drawpaintacademy.com/artists-color-wheel/

Apple(a). 2019. Apple developer Documentation. Viitattu 12.2.2019.
https://developer.apple.com/documentation

Apple(b). 2019. Human Interface Guidelines – iOS Design Themes. Viitattu 20.3.2019
https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/

Apple(c). 2019. Core Data. Viitattu 10.4.2019.
https://developer.apple.com/documentation/coredata

Apple(d). 2019. Human Interface Guidelines – Color. Viitattu 19.4.2019.
https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/color/

Apple(e). 2019. Human Interface Guidelines – Navigation Bars. Viitattu 27.4.2019.
https://developer.apple.com/design/human-interface-guidelines/ios/bars/navigation-barAs/

Apple(f). 2019. Human Interface Guidelines – Adaptivity and Layout. Viitattu 15.4.2019.
https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/adaptivity-and-layout/

Realm. 2019. Realm Swift Latest. Viitattu 10.4.2019.
https://realm.io/docs/

New Gen Apps. 2014. Apple WWDC 2014 – A new programming language- Swift. Viitattu 1.6.2019.
https://www.newgenapps.com/blog/apple-wwdc-2014-a-new-programming-language-swift

Arday, D. 2012. Color Schemes Defined. Viitattu 10.4.2019.
http://www.theinformedillustrator.com/2012/10/color-schemes-defined_16.html

Interaction Design Foundation. What is Flat Design? Viitattu 7.3.2019.

https://www.interaction-design.org/literature/topics/flat-design

Marvel. Marvel – Everything you need to bring ideas to life. Viitattu 10.3.2019.
https://marvelapp.com/why-marvel/

Manferdini, M. Functional Programming in Swift: An Unsual yet Powerful Paradigm. Viitattu 5.6.2019.
https://matteomanferdini.com/swift-functional-programming/

Cocoapods. 2019. Cocoapods – Get on with building your app, not duplicating code. Viitattu 5.4.2019.
https://cocoapods.org/about

Swift Docs. Closures. Viitattu 6.6.2019.
https://docs.swift.org/swift-book/LanguageGuide/Closures.html

AppCoda. 2014. A Beginner's Guide To Optionals in Swift. Viitattu 5.5.2019.
https://www.appcoda.com/beginners-guide-optionals-swift/

Avantica. 2017. NoSQL mobile databases with Realm. Viitattu 22.4.2019.
https://www.avantica.net/blog/realm-mobile-database

Jacobs, B. 2017. What is the difference between Core Data and SQLite. Viitattu 21.4.2019.
https://cocoacasts.com/what-is-the-difference-between-core-data-and-sqlite/

Google(a). 2019. Firebase Pricing Plans. Viitattu 24.4.2019
https://firebase.google.com/pricing/

Google(b). 2019. Firebase Documentation. Viitattu 23.4.2019.
https://firebase.google.com/docs

Amazon(a). 2019. AWS Amplify. Viitattu 23.4.2019.
https://aws.amazon.com/amplify/

Amazon(b). 2019. AWS Amplify Documentation. Viitattu 23.4.2019.
https://docs.aws.amazon.com/amplify/?id=docs_gateway

Amazon(c). 2019. AWS Free Tier. Viitattu 23.4.2019.
https://aws.amazon.com/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc&awsf.Free%20Tier%20Types=categories%23featured

Github. 2019. React Native 0.59. Viitattu 29.5.2019.
https://facebook.github.io/react-native/

Kuviot

Taulukot

Liitteet