Patrik Heinonen

Bitcoin-säästösovelluksen toteutus Flutter-ohjelmistokehyksen avulla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

30.5.2022

Tiivistelmä

Tekijä:	Patrik Heinonen
Otsikko:	Bitcoin-säästösovelluksen toteutus Flutter-ohjelmistokehyksen avulla
Sivumäärä:	36 sivua + 1 liite
Aika:	30.5.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Vesa Ollikainen Head of Engineering Aki Teliö

Opinnäytetyön tavoitteena oli toteuttaa bitcoin-säästösovelluksesta demoversio, jotta sitä voidaan esitellä mahdollisille asiakkaille, jotka ovat kiinnostuneita bitcoin-säästösovelluksen ostamisesta. Demosovellukseen valittiin olennaisimmat säästösovelluksen toiminnot, jotta säästösovelluksen idea selkeytyisi tahoille, joille sitä esitellään. Tavoitteena oli myös kuvata Flutter-ohjelmistokehystä ja sen käyttömahdollisuuksia sekä dokumentoida osa bitcoin-säästösovellusdemon toiminnallisuuksista.

Demosovellus toteutettiin iteratiivisesti eli valittiin aina noin kahden viikon mittaiseen jaksoon tietty määrä toimintoja, jotka haluttiin toteuttaa. Kehitysprosessin lopussa ja sen aikana sovellusta testattiin ja siinä esiintyneet ohjelmointivirheet korjattiin.

Demosovellus toteutettiin Flutter-ohjelmistokehyksen avulla. Flutter valittiin demosovelluksen ohjelmistokehykseksi, koska sillä on nopea kehittää sovelluksia ja sillä kehitettyjä sovelluksia voidaan integroida jo olemassa oleviin sovelluksiin. Sovelluksen sivujen graafisen ilmeen suunnitteli yrityksen ulkopuolinen suunnittelija.

Insinöörityön lopputuloksena syntynyttä demosovellusta hyödynnetään niin, että sitä esitellään mahdollisille asiakkaille, jotka ovat kiinnostuneita bitcoin-säästösovelluksen ostamisesta. Jos asiakas haluaa ostaa sovelluksen, niin siitä tehdään kustomoitu tuotantoversio, joka vastaa heidän vaatimuksiansa.

Avainsanat:	Flutter, mobiilikehitys, alustariippumaton ohjelmistokehitys, Bitcoin

Abstract

Author:	Patrik Heinonen
Title:	Implementation of Bitcoin Savings Application Using Flutter Framework
Number of Pages:	36 pages + 1 appendix
Date:	30 May 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Software Engineering
Supervisors:	Vesa Ollikainen, Senior Lecturer Aki Teliö, Head of Engineering

---

The aim of the study was to implement a demo version of a bitcoin savings application so that it could be presented to potential customers interested in buying a bitcoin savings application. The most essential functions of the saving application were selected to be implemented in the demo application to clarify the idea of the savings application to the customers it is presented to. Another aim was to depict Flutter framework and how it can be used and to document some of the functionalities of the demo application.

The demo application was developed using an iterative process which means that a number of functionalities that needed to be implemented were selected and implemented in cycles lasting two weeks. During and after the development process the application was tested and all the software bugs were fixed.

The demo application was implemented using the Flutter framework. Flutter was chosen as the framework because Flutter applications are fast to develop, and they can be integrated into existing applications. The graphical user interface was designed by an outsourced designer.

The demo application will be utilized in presenting the bitcoin savings application to potential customers. If the customer wants to buy the application, it will be made into a customized production version that meets the customer’s requirements.

Keywords:	Flutter, mobile development, cross-platform development, Bitcoin

Sisällys

1 Johdanto 1

2 Flutter-ohjelmistokehys 3

- 2.1 Flutterin edut 3
- 2.2 Flutterin huonot puolet 8
- 2.3 Dart-ohjelmointikieli 9
- 2.4 Flutter perusteet 12

3 Bitcoin-säästösovellus 18

4 Bitcoin-säästösovelluksen toteutus 24

- 4.1 Sovelluksen sivut 24
- 4.2 Navigoiminen 29
- 4.3 Tilanhallinta 30

5 Yhteenveto 33

Lähteet 35

Liitteet

Liite 1: Esimerkki StatefulWidget-komponentin käytöstä

1 Johdanto

Insinöörityön tavoitteena on kehittää demosovellus, jossa käyttäjät voivat ostaa ja säilyttää bitcoin-virtuaalivaluuttaa. Käyttäjät voivat myös säilyttää euroja sovellukseen luomallansa käyttäjätilillä. Työn tilaaja on Oivan Group Oy, ja he tilasivat työn, jotta demosovellusta voidaan esitellä asiakkaille. Jos asiakas haluaa ostaa sovelluksen, heille kehitetään kustomoitu tuotantoversio sovelluksesta, jossa on enemmän toimintoja kuin demosovelluksessa. Sovellusta ei siis ole tarkoitus laittaa App Store- tai Google Play -sovelluskauppoihin vaan myydä sitä yrityksille, jotta ne voivat tarjota sovellusta asiakkaillensa. Sovellus toteutetaan Flutter-ohjelmistokehystä käyttäen. Työn tavoitteena on myös kuvata Flutter-ohjelmistokehystä ja sen käyttömahdollisuuksia sekä dokumentoida osa bitcoin-säästösovellusdemon toteutuksesta.

Flutter on alustariippumaton ohjelmistokehys eli sillä voi kehittää Android-, iOS- ja web-sovelluksia niin, että kehittäjien tarvitsee kehittää vain yhtä lähdekoodia, josta voidaan rakentaa sovellus edellä mainituille alustoille. Flutterilla voidaan myös kehittää macOS-, Windows- ja Linux sovelluksia. Sovellusten kysyntä on kasvanut viimevuosina huomattavasti, ja useat yritykset haluavat tarjota palveluitansa sovelluksien kautta. Yksi alustariippumattoman kehityksen eduista on se, että kehitys on halvempaa, koska tarvitaan vain yksi lähdekoodi ja näin ollen vähemmän kehittäjiä. Alustariippumattoman kehityksen etuihin kuuluu myös se, että tuotteet saadaan nopeammin markkinoille, ylläpito on helppoa ja nopeaa, koska Flutter tarjoaa runsaasti valmiita komponentteja ja pub.dev-sivustolta löytyy runsaasti avoimen lähdekoodin kirjastoja, joita kuka tahansa voi käyttää projekteissaan. (1; 2)

Edellä mainituista syistä nähdään, miksi Flutterin suosio on kasvamassa ja kuinka nyt on hyvä hetki kehittyä Flutter-osaajana ja hankkia yrityksiin Flutter-osaajia. Flutter-osaaminen mahdollistaa palveluiden myynnin myös pienemmille asiakkaille, joilla ei välttämättä ole rahaa kahden natiivin mobiilisovelluksen kehitykseen.

Tässä työssä tullaan havainnollistamaan Flutterin hyödyllisyyttä kehittämällä bitcoin-säästösovellus. Työssä esitellään Flutter-ohjelman perusrakennetta ja ohjelmistokehyksen olennaisia komponentteja. Työn alussa kerrotaan Flutter-ohjelmistokehyksen eduista ja huonoista puolista. Huonojen puolien esittelyn jälkeen esitellään ohjelmistokehyksen perusteet ja kerrottaan Dart-ohjelmointi-kielestä, jolla Flutter-sovellukset kehitetään. Työ sisältää myös luvun, jossa kerrotaan bitcoin-säästösovelluksesta käyttäjän ja sovelluksen omistajan näkökulmasta, ja luvun, jossa dokumentoidaan osa säästösovelluksen toiminnoista.

2 Flutter-ohjelmistokehys

Flutter on alustariippumaton ohjelmistokehys, jolla voidaan kehittää sovellus, joka toimii Android- ja iOS-mobiilikäyttöliittymillä sekä macOS-, Windows- ja Linux-käyttöliittymillä (3). Tässä luvussa käsitellään Flutterin etuja kuten esimerkiksi sen taloudellisia hyötyjä. Luvussa käsitellään myös Flutterin huonoja puolia sekä havainnollistetaan Flutter-sovelluksen toimintaa yksinkertaisen esimerkkiohjelman avulla. Luvun lopussa kerrotaan tilanhallintamenetelmä Providerista ja tarkastellaan Flutter-ohjelmistokehyksen perusteita teknisestä näkökulmasta.

Flutterin hyviä ja huonoja puolia käsiteltävistä luvuista tulee ilmi, milloin projektin ohjelmistokehykseksi kannattaa valita Flutter. On useita syitä valita Flutter-ohjelmistokehys kuten sen taloudelliset hyödyt, nopea kehitys ja syy, mistä harvemmin puhutaan, joka on se, että kehittämällä sovellus yhdellä lähdekoodilla takaa sen, että käyttöliittymä on saman näköinen kaikilla käyttöliittymillä.

2.1 Flutterin edut

Flutter-sovelluksia toimivat useilla käyttöliittymillä, ja niiden suoritusteho on verrattavissa natiivisovelluksiin. Kilpailullisessa ympäristössä kehitysajalla on merkitystä, koska tuotteen menestys voi olla kiinni siitä, saadaanko se käyttäjien käytettäväksi ennen kilpailijaa. Alustariippumattomien ohjelmistokehysten etu on siinä, että ne tarjoavat runsaasti valmiita komponentteja ja kirjastoja kehittäjien käytettäväksi, joka näin ollen nopeuttaa kehitystä sekä tekee siitä halvempaa, koska kaikkia komponentteja ja kirjastoja ei tarvitse itse kehittää alusta asti. Nopeasti kehitettävässä tuotteessa on myös se etu, että asiakkailta tuleva palaute tuotteesta saadaan myös nopeasti ja näin ollen tuotteen jatkokehitys nopeutuu. (2.)

Flutteria voidaan hyödyntää pienissä sekä isoissa yrityksissä. Pienen yrityksen tulevaisuus voi olla kiinni siitä, saako se sovelluksen tarpeeksi nopeasti markkinoille, ja siitä, ovatko tuotteen kehityskustannukset tarpeeksi halvat. Isommat yritykset taas voivat säästää merkittäviä summia rahaa, kun puhutaan valtavista

sovelluksista, koska Flutterin avulla heidän ei tarvitse kehittää kahta eri lähdekoodia. Flutterista on siis hyötyä sekä isoille että pienille yrityksille. (2)

Sovelluksen julkaisu on vasta alkuvaihe tuotteen kehityksessä. Sovellusta tarvitsee huoltaa ja jatkokehittää. Sovellusten huoltaminen vie aikaa ja on kallista. Flutterin etuihin kuuluu uusien toiminnallisuuksien nopea lisääminen sekä huoltamisen helppous. Jos kehitystiimiin tarvitsee lisätä, uusi työntekijä on sekin helppoa, koska ei tarvitse esimerkiksi palkata kahta työntekijää kehittämään sekä iOS- ja Android-sovellusta. Palkkaamisen helppous tietenkin helpottaa henkilöstöhallinnon tehtäviä ja se, että joudutaan palkkaamaan noin puolet vähemmän kehittäjiä, vähentää rekrytointiin liittyviä riskejä kuten epäpätevän työntekijän palkkaamista. (2.)

Flutterin käyttöön voidaan myös hiljalleen siirtyä jo olemassa olevassa sovelluksessa. Flutter-koodia voidaan helposti integroida Android- ja iOS-sovelluksiin, mikä mahdollistaa sen, että osa natiivisovelluksesta kirjoitetaan Flutterilla. Useat yritykset kirjoittavatkin jo osan sovelluksesta Flutterilla ja integroivat sen heidän sovellukseensa (2). Edellä mainittu on järkevää varsinkin silloin, kun halutaan muuttaa sovellus vaiheittain Flutter-sovellukseksi, tai jos kehitettävä ominaisuus on hyvin kevyt eikä sen kehittäminen natiivina tarjoa parempaa ruudunpäivitysnopeutta. Integraation helppous myös luo markkinaraon yrityksille, jotka haluavat kehittää sovelluksia, jotka voidaan integroida osaksi toisten yritysten sovelluksia. Esimerkki tästä on vaikkapa tässä työssä raportoitava Bitcoin-säästösovellus, joka voidaan integroida esimerkiksi pankkien mobiilikäyttöliittymiin.

Google kehittää jatkuvasti Flutteria, ja Flutter-yhteisö kasvaa sen ansiosta, että Google pitää tapahtumia, joissa opetetaan pieniä ja isoja yrityksiä Flutterin eduista (2). Google tarjoaa kehittäjille runsaasti ohjevideoita, dokumentaatiota ja kirjastoja. Pub.dev-sivustolta löytyy useita avoimen lähdekoodin kirjastoja, joita vapaaehtoiset kehittäjät kehittävät ja parantavat. Nämä ilmaiset kirjastot edesauttavat nopeaa ohjelmistokehitystä. Kehittäjän työtä helpottavat nämä useat avoimen lähdekoodin kirjastot. Vaikka kehittäjä ei voisi suoraan käyttää

kirjastoa, voi hän ottaa sen lähdekoodin ja tehdä tarvitsemansa muokkaukset siihen ja sitten ottaa sen käyttöön.

Flutter-sovellusten testaaminen on yksinkertaisempaa verrattuna natiivisovellusten testaamiseen, koska Flutterin avulla voidaan luoda ohjelmistotuote, jolla on vain yksi lähdekoodi. Näin ollen se tarvitsee testata vain kerran, joka säästää valtavasti aikaa ja rahaa. Tämä testaamisen helppous näkyy selvästi säästetyssä rahassa sekä myös siinä, kuinka paljon ihmisiä tarvitsee rekrytoida testaustiimeihin. (3.)

Sovelluksen suoritusteho on välttämätöntä hyvän käyttäjäkokemuksen takaamiseksi. On vaikea sanoa tarkkoja lukuja, mutta voidaan kuitenkin todeta, että Flutter-sovellusten suoritusteho on useissa tapauksissa täysin verrattavissa natiivisovellusten suoritustehoihin. Flutterin suoritusteho on jopa natiivisovelluksia parempi, kun toteutetaan monimutkaisia animaatioita. Tämä kaikki johtuu siitä, että Flutter ei ole riippuvainen koodin tulkitsemisesta toisin, kuten useat alustariippumattomat ohjelmistokehykset. Flutter-sovelluksia ei tulkita sovelluksen ajon aikana, vaan ne käännetään konekieleksi, joka parantaa suoritustehoa. (4.)

Alustariippumattomat ohjelmistokehykset tarjoavat tavan käyttää samaa lähdekoodia kaikilla kohdealustoilla. Flutterin lisäksi ei kuitenkaan ole ohjelmistokehystä, joka muuntaa käyttöliittymä koodin näytölle sopivaan esitysmuotoon yhtä yksinkertaisesti kuin Flutter. Tätä muuntamisprosessia kutsutaan kuvantamiseksi. Kuvassa 1 näkyy, miten kuvantaminen on React Nativessa toteutettu. Tämä toteutustapa on hyvin samanlainen muissa alustariippumattomissa ohjelmistokehyksissä. Kuvassa esitetyn kaltainen ohjelmistokehys on riippuvainen alustakohtaisista komponenteista, ja jokainen tapahtuma kuten napin painallus on välitettävä kohdealustan piirrettäväksi. (4; 5.)

Kuvan 1 kaltaisessa tilanteessa kuvantaminen on riippuvainen alustakohtaisista komponenteista, mikä tarkoittaa sitä, että ohjelmistokehyksen jokainen kompo-

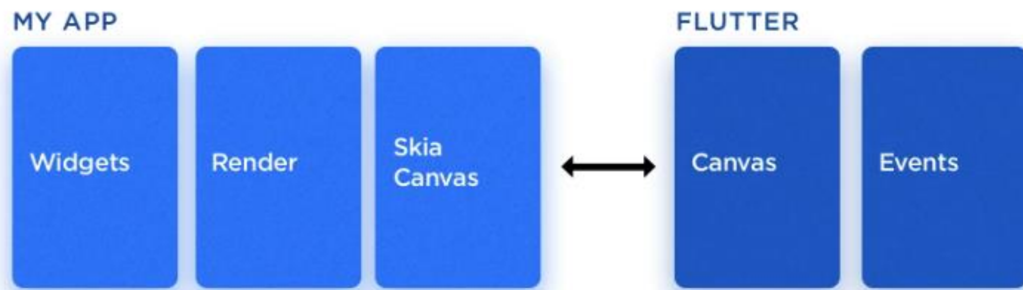
nentti pitää kartoittaa vastaamaan alustan vastaavanlaista komponenttia. Esimerkiksi jos ohjelmistokehyksen komponentin data muuttuu, pitää kutsua alustan omaa komponenttia piirtämään itsensä uudestaan. (4; 5.)

Kuva 1. React Nativen kuvantamistapa. (4)

Flutter ei tarvitse alustakohtaisia käyttöliittymä komponentteja käyttöliittymän kuvantamiseen. Flutter tarvitsee alustalta ainoastaan kankaan, jolle kuvantaa komponentteja. Komponentit näkyvät ruudulla, ja ne pääsevät käsiksi tapahtumiin kuten sormen kosketuksiin tai ruudun vieritysyrityksiin. Kuvassa 2 nähdään, miten Flutter kuvantaa käyttöliittymät. (4; 5.)

Kuvassa 2 ”Widgets”-niminen laatikko viittaa sovelluksessa käytettäviin komponentteihin. ”Render” viittaa prosessiin, jossa komponentin kuvannetaan Skia Canvasin avulla. Skia Canvasia näytetään Flutter-ohjelmistokehyksen paljastamalle kankaalle eli kuvassa nähtävälle Canvasille. Kuvassa 2 näkyvä ”Events” viittaa esimerkiksi ruudun koskettamiseen ja näppäimistöllä kirjoittamiseen. Kun sovellus on piirretty Flutterin tarjoamalle kankaalle, sovelluksen komponentit pääsevät ”Events”-laatikon sisältämiin toimintoihin käsiksi. Tämänkaltainen kuvantamistapa takaa nopeamman ruudunpäivityksen kuin esimerkiksi React Na-

tivessa sekä takaa sen, että käyttöliittymä on samannäköinen jokaisella alustalla. Flutterin kuvantamistapa takaa siis nopeamman ruudunpäivityksen, koska komponentit yksinkertaisesti piirrettään ruudulle sen sijasta, että komponentti pitää eka kartoittaa vastaamaan alustakohtaista komponenttia ennen kuin se piirretään. (4; 5.)



Kuva 2. Flutterin kuvantamistapa. (4)

Android-sovelluksen koonti testauslaitteelle kestää yleensä noin 40 sekuntia tai enemmän. Joskus voi kestää todella kauan muokata pientä yksityiskohtaa käyttöliittymässä, kun se on koko ajan koottava uudestaan, jotta muutokset näkyvät ruudulla. Flutterissa on toiminallisuus nimeltä kuuma uudelleenlataus, jonka avulla tehdyt muutokset voi nähdä lähestulkoon välittömästi ilman, että sovelluksen tila menetetään. Edellä mainitun toiminnallisuuden takia Flutter-kehittäminen on todella paljon nopeampaa ja kehittäjille miellyttävämpää kuin esimerkiksi Android-sovelluksen kehittäminen Java-ohjelmointikielellä. (4.)

Monet sovellukset ovat riippuvaisia käyttöliittymätason toiminnallisuuksista kuten GPS-koordinaattien hakemisesta, Bluetooth-kommunikaatioista, oikeuksien hallitsemista ja niin edelleen. Google tarjoaa monet näistä toiminnallisuuksista kirjastoina, jotka ovat heti valmiina käyttöön. On kuitenkin tapauksia, joissa sovellus saattaa tarvita jotakin käyttöliittymätason toiminnallisuuksia, joita ei ole vielä saatavilla valmiina kirjastoina. Flutter kuitenkin tarjoaa helpon tavan luoda alustakanavia, joiden avulla Flutter-koodi ja alustakohtainen natiivikoodi voivat

keskustella. Alustakanavien avulla päästään käyttämään mitä tahansa käyttöliittymä tason toiminnallisuuksia kirjoittamalla natiivikoodia Flutter-sovellukseen. Pub.dev-sivustolta löytyy kuitenkin monia kirjastoja, joiden avulla päästään käyttöliittymätason toiminnallisuuksiin käsiksi, joten natiivikoodin kirjoittaminen ei yleensä ole tarpeellista. (4.)

2.2 Flutterin huonot puolet

Flutter-ohjelmistokehyksen ohjelmointi kielenä toimii Dart. Dart-ohjelmointikielen paradigma on tuttu monille ohjelmoijille, koska se on olio-ohjelmointikieli. Monet uudet ohjelmoijat eivät kuitenkaan opettele Dartia, ja tämän takia voi olla vaikea löytää Dart-osaajia kehitystiimiin. Dart on myös suhteellisen uusi ja vähän käytetty ohjelmointikieli, joten voi olla vaikeaa löytää internetistä vastauksia kysymyksiin, joita kehittäjälle tulee sitä opeteltaessa Dartia. (6.)

Käyttäjillä on rajallinen määrä muistia laitteissaan, joten on toivottavaa julkaista sovellus, joka vie mahdollisimman vähän muistia, jotta käyttäjien ei tarvitsisi poistaa muuta sisältöä heidän laitteiltaan. Koska Flutter käyttää sisäänrakennettuja komponentteja alustakohtaisten komponenttien sijaan, Flutter-sovellusten minimikoko on yli 4 MB, mikä on paljon verrattuna Java-sovelluksiin, jotka vievät vähintään 539 KB ja Kotlin-sovelluksiin, jotka vievät vähintään 550 KB. Tämä pätee kuitenkin muihinkin alustariippumattomiin ohjelmistokehyksiin kuten esimeriksi React Nativeen, jonka pienimmän mahdollisen sovelluksen koko on 7 MB. (6.)

Kolmannen osapuolen kirjastot ovat iso osa ohjelmistokehitystä, koska niiden avulla vältytään kirjoittamasta kaikkea koodia alusta asti. Nämä kirjastot ovat yleensä avoimen lähdekoodin projekteja, jotka ovat helppokäyttöisiä ja testattuja. Vanhoille ja suosituille ohjelmistokehyksille kuten Androidille kirjastojen etsiminen ei ole vaikeaa toisin kuin taas uusille ohjelmistokehyksille kirjastojen etsiminen on usein hankalaa. Flutteria käyttäessä tähän edellä mainittuun ongelmaan voidaan törmätä juuri siksi, että Flutter on uusi ohjelmistokehys. Pub.dev osoitteesta löytyy kuitenkin jo yli 23 tuhatta kirjastoa, ja määrä kasvaa koko

ajan. Tämä on kuitenkin vähemmän kuin Flutterin kilpailijalla React Nativella, mutta suurin osa tärkeistä kirjastoista on jo kirjoitettu, joten kehittäjät pärjäävät niillä. (6.)

React Native on vastaavanlainen ohjelmistokehys, joka käyttää JavaScriptiä. Flutter käyttää Darttia JavaScriptin sijasta, joka antaa sille etulyöntiaseman suoritustehoissa. JavaScript on ollut olemassa jo vuodesta 1995. Dart on tullut vasta 2011. Tämä tarkoittaa sitä, että JavaScriptistä on huomattavasti helpompi löytää tietoa ja ohjeita kuin Dartista. JavaScriptillä on vakiintuneet parhaat käytännöt, joita Dart-osaajat vasta etsivät. (7.)

Flutterin heikkous on myös siinä, että kokeneita kehittäjiä on vähän. React Nativea käytetään monissa sovelluksissa, joten se on paljon tutumpi mobiilikehittäjille kuin Flutter, joka on uudempi ja vieraampi. Näin ollen ohjelmistokehitysyrityksillä voi olla vaikeuksia löytää kokeneita Flutter-kehittäjiä. Yritysten täytyy siis olla valmiita ottamaan riski, jos ne haluavat aloittaa Flutter-sovellusten tekemisen. Vaikka Flutterin suosio kasvaakin, on silti vaikea löytää kokeneita kehittäjiä, parhaita käytäntöjä ja tietolähteitä. (7.)

2.3 Dart-ohjelmointikieli

Dart on käyttöliittymäsovellusten tekemiseen optimisoitu ohjelmointikieli, jolla voi kehittää nopeasti suoritettavia sovelluksia monille mahdollisille alustoille. Dartin tavoitteena on tarjota produktiivisin ohjelmointikieli alustariippumattomien sovellusten kehittämiseen. Dart tarjoaa myös vaihtoehdon kääntää koodia inkrementaalisesti kehityksen aikana, mikä tarkoittaa sitä, että vain juuri lisätyt muutokset koodiin käännetään uudelleen, jolloin sovellus voidaan koota uudelleen alle sekunnissa. (8.)

Ohjelmistokielen määrittää sen tekninen kuori eli valinnat, jotka on tehty kehityksen aikana, jotka muovaavat kielen vahvuudet ja kykeneväisyydet. Dart on suunniteltu tekniselle kuorelle, joka erityisesti soveltuu käyttöliittymäsovellusten

kehitykseen priorisoiden kehitysnopeutta mahdollisille kohdealustoille kuten työpöydälle, selaimelle ja mobiilille. Dart muodostaa myös Flutterin perustuksen. Dart tarjoaa kielen ja ajoympäristöt, joissa Flutter-sovelluksia ajetaan, mutta Dart tukee myös muita keskeisiä kehittäjän tehtäviä kuten formatointia, jolla selkeytetään koodin ulkoasua, analysointia ja koodin testausta. (8.)

Dart-ohjelmointikieli on tyyppiturvallinen eli se käyttää staattista tyypin tarkastusta varmistaakseen, että muuttujan arvo vastaa aina muuttujan staattista tyyppiä. Joskus edellä mainittuun viitataan eheänä tyypityksenä. Vaikka muuttujalla on pakko olla tyyppi, muuttujan tyypin ilmoittaminen on vapaaehtoista sen takia, että Dart osaa päätellä muuttujan tyypin. (8.)

Dartin tyypitysjärjestelmä on myös joustava, koska se sallii dynaamisten tyyppien käyttämisen. Dynaamisesti tyypitetylle muuttujalle voidaan antaa esimerkiksi String-tyyppinen arvo ja muuttujaa voidaan käyttää niin kuin se olisi String-tyyppinen. Tämän dynaamisesti tyypitetyn muuttujan arvo voi kuitenkin olla myöhemmin ohjelman ajonaikana jotakin muuta tyyppiä kuten esimerkiksi tyyppiä int. Dart siis tekee sovelluksen ajonaikana tarkistuksia, joka mahdollistaa dynaamiset tyypit. (8.)

Dart tarjoaa eheää null-turvallisuutta, joka tarkoittaa sitä, että muuttujan arvo ei voi olla null-tyyppinen, ellei erikseen ilmoita sen olevan mahdollista. Eheällä null-turvallisuudella Dart voi suojella kehittäjää ajonaikaisilta null-poikkeuksilta staattisen koodianalyysin avulla. Kun Dart määrittää, että jokin muuttuja ei voi olla tyyppiä null, muuttuja ei voi ikinä olla null-tyyppinen toisin kuin monissa muissa null-turvallisissa kielissä. Jos tarkastellaan ajettavan ohjelman koodia, voidaan huomata, että jos muuttujan määritellyssä ei ole ilmoitettu, että se voi olla null-tyyppinen, ei se voi olla null-tyyppinen myöskään ajonaikana, ja juuri tästä syystä Dartin null-turvallisuutta kutsutaan eheäksi. (8.)

Dartin kääntäjä antaa kehittäjän ajaa koodia eri tavoilla. Dart sisältää Dart virtuaalikoneen juuri ajoissa kääntämistä (engl. just in time compilation) varten ja en-

nenaikaisen kääntäjän, joka tuottaa konekieltä. Selaimella käytettäviä sovelluksia varten Dart sisältää kehitys- ja tuotantokääntäjät, jotka kummatkin kääntävät Dartia JavaScriptiksi. Kehityksen aikana kehitysnopeus on tärkeää. Dart-virtuaalikoneen tarjoama kääntötapa mahdollistaa inkrementaalisen uudelleenkäännön, jonka avulla ohjelmakoodiin tehdyt muutokset näkyvät alle sekunnissa. Tämä luonnollisesti nopeuttaa kehitystä, kun uudelleen käännöt kestävät noin sekunnin verrattuna esimerkiksi Android-ohjelmiin, joiden kääntäminen voi kestää minuutteja. (4; 8.)

Esimerkkikoodi 1 on esimerkki yksinkertaisesta Dart-ohjelmasta. Rivillä 1 on yhdelle riville kirjoitettava kommentti. Dart tukee myös monen riville kirjoitettavia kommentteja. Rivillä 2 esiintyy erikoistyyppi void, jolla merkitään arvo, jota ei ikinä aiota käyttää. Esimerkkikoodin 1 funktiot kuten printInteger ja main eivät palauta mitään arvoa, joten ne on merkitty void-tyyppisiksi. Funktio printInteger hyväksyy aNumber-nimisen parametrin, joka on tyyppiä int. Funktion sisällä on funktiokutsu print, joka tulostaa printInteger-funktion saaman parametrin String-interpoloinnin avulla. Main on pakollinen funktio jokaisessa Dart-ohjelmassa ja main on se paikka, josta sovelluksen suoritus aloitetaan. Rivillä 9 nähtävä funktiokutsu toimii, vaikka number-nimistä muuttujaa ei ole erikseen määritetty int-tyyppiseksi, koska kääntäjä osaa päätellä muuttujan tyypin sille määrätystä arvosta, joka on 42. (9.)

```
1 // Define a function.
2 void printInteger(int aNumber) {
3   print('The number is $aNumber.');
4 }
5
6
7 void main() {
8   var number = 42;
9   printInteger(number);
10 }
```

Esimerkkikoodi 1. Yksinkertainen Dart-ohjelma. (9)

2.4 Flutterin perusteet

Flutter on yksinkertainen ja suoritusnopeudeltaan tehokas ohjelmistokehys, jonka perustana toimii Dart-ohjelmointikieli. Flutterin nopeus perustuu siihen, että se kuvantaa käyttöliittymän suoraan käyttöjärjestelmän tarjoamaan kankaaseen sen sijasta, että kuvantaminen tehtäisiin natiivin ohjelmistokehyksen kautta. (10.)

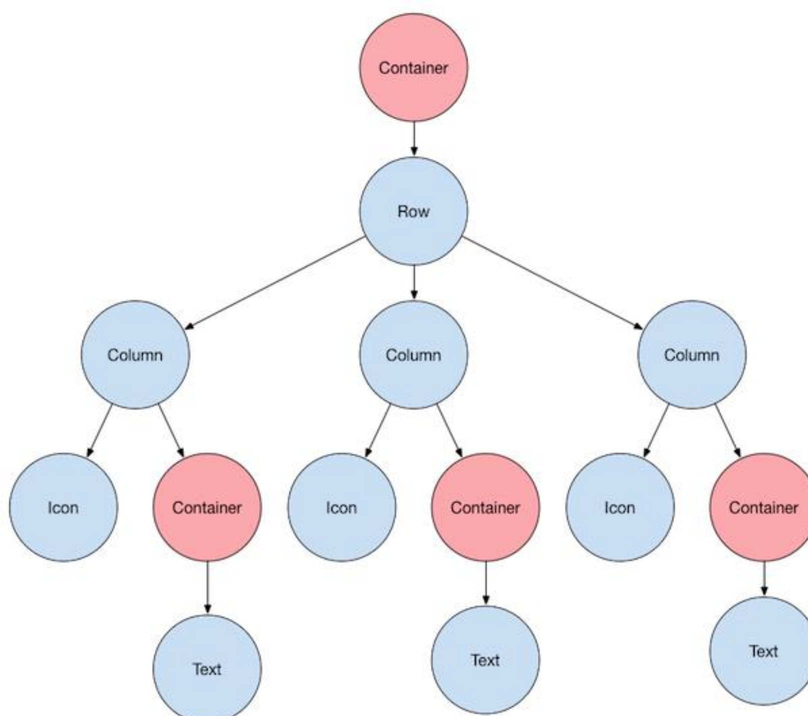
Flutterissa lähestulkoon kaikki asiat ovat komponentteja (11). Flutter-komponentit tukevat animaatioita ja eleitä. Flutter-sovelluksen toiminta perustuu reaktiiviseen ohjelmointiin. Reaktiivinen ohjelmointi on ohjelmointiparadigma, jossa esimerkiksi näytöllä näkyvän komponentin ulkonäkö automaattisesti muuttuu, kun jokin arvo, jota komponentti kuuntelee, muuttuu. Komponentilla voi olla tila, jos kehittäjä niin toivoo. Kun komponentin tilaa muutetaan, Flutter automaattisesti vertaa komponentin vanhaa tilaa uuteen tilaan ja kuvantaa komponentin niin, että vain tarpeelliset muutokset uudelleen kuvannetaan sen sijasta, että koko komponentti uudelleen kuvannettaisiin. (10.)

Flutterissa komponentteja sijoitetaan muiden komponenttien sisään. Komponentteja, joiden sisään laitetaan muita komponentteja, kutsutaan Layout-komponenteiksi. Layout-komponentin avulla voidaan luoda komponentti, joka koostuu useista kehittäjän itse luomista komponenteista ja Flutterin tarjoamista komponenteista. Flutter-sovellus on yksi iso komponenttipuu, jonka juuri on runApp-funktiolle parametrina annettu komponentti. Funktio runApp esitellään myöhemmin tässä aliluvussa. (11.)

Kuvassa 3 on käytetty kolmea erilaista Layout-komponenttia, jotka ovat Container, Column ja Row. Kuvassa käytettäville Layout-komponenteille on annettu ääriviivat sen takia, että ne olisi helpompi hahmottaa, koska Layout-komponentit ovat yleisesti ottaen näkymättömiä. Kuvassa nähtävä Container-komponentti koostuu rivistä eli Row-komponentista, jossa on kolme pystyriviä eli Column-komponenttia. Jokaisen pystyrivin sisällä on tekstiä ja kuvake. (11.)

Kuva 3. Kuva Container-komponentista. (11)

Kuvasta 4 nähdään, miltä edellisessä luvussa esitetyn komponentin komponenttipuu näyttää. Pinkillä merkittyjä Container-komponentteja käytetään tässä esimerkissä siksi, että niiden avulla saadaan Text-komponentti tietyn etäisyyden päähän Icon-komponentista. Tämä onnistuu käyttämällä Containerin margin-attribuuttia. Row on myös laitettu Containerin sisään, koska käyttämällä Containerin padding-attribuuttia, Rowin ja Containerin väliin saadaan luotua tyhjää tilaa. Columnilla ja Rowilla on attribuutteja, joiden avulla voidaan määrittää, miten niiden lapsikomponentit asettuvat horisontaalisesti tai vertikaalisesti. Lapsikomponentilla tarkoitetaan komponenttia, joka laitetaan toisen komponentin sisään käyttämällä komponentin child- tai children-attribuuttia. Tekstin fontti, väri ja muut sen ominaisuudet määritetään käyttämällä Text-komponentin attribuutteja. (11.)



Kuva 4. Komponenttipuu, jossa on rivi, joka koostuu kolmesta pystyrivistä. (11)

Kun uusi Flutter-sovellus luodaan, Flutter-ohjelmistokehys automaattisesti generoi joitakin tiedostoja ja kansioita. Olennaisimmat näistä kansioista ovat android, ios, lib ja build. Tiedostoista olennaisin on konfiguraatiotiedosto pubspec.yaml. Kaikkiin kansioihin ei saa tehdä muutoksia. Build-kansio on esimerkki kansiosta, johon kehittäjä ei saa tehdä muutoksia. (12.)

Android-kansio pitää sisällään kokonaisen natiivin Android-sovellus projektin, ja sitä käytetään, kun Flutter-sovellus kootaan Android-käyttöjärjestelmälle. Flutter-koodi siis käännettään, jonka jälkeen se injektoidaan edellä mainittuun Android-sovellukseen ja lopputuloksena on natiivi Android-sovellus. Android-kansion sisältä löytyy AndroidManifest.xml-tiedosto, jonka muokkaaminen on tarpeellista esimerkiksi silloin, kun sovellus tarvitsee oikeudet internetin käyttöön, jotta se voi tehdä HTTP-pyyntöjä (Hypertext Transfer Protocol). (13; 14.)

iOS-kansio pitää sisällään kokonaisen natiivin iOS-sovellusprojektin, jota käytetään, kun Flutter-sovellus kootaan iOS-käyttöjärjestelmälle. Tätä iOS-projektia käytetään samalla tavalla kuin Android-projektia, joka on Android-kansiossa. Flutter-sovellusten koonti iOS-käyttöjärjestelmälle on mahdollista vain, jos työskennellään tietokoneella, jonka käyttöjärjestelmä on macOS. Tämän takia iOS-kansio on saatavilla vain, jos projekti on luotu macOS-käyttöjärjestelmällä. (13.)

Build-kansio pitää sisällään käännetyn Flutter-sovelluksen. Tämän kansion sisältö generoidaan automaattisesti osana Flutter-sovelluksen koontiprosessia, joten kehittäjän ei tarvitse muokata mitään tiedostoja tämän kansion sisällä. (13.)

Lib-kansiosta löytyvät Dart-tiedostot, jotka sisältävät Flutter-sovelluksen koodin. Tämän kansion parissa kehittäjät viettävät suurimman osan ajastaan kehittäessään Flutter-sovellusta, koska tähän kansioon kuuluu kirjoittaa kaikki sovelluksessa käytettävä Dart-koodi. Tämän kansion sisästä löytyy myös vakiona main.dart-tiedosto. Jokainen Dart-ohjelma eli jokainen Flutter-sovellus tarvitsee main-funktion, josta sovelluksen suoritus aloitetaan. Tämän funktion on oltava

tiedostossa, joka on määritetty ohjelman aloituspisteeksi. Hyvänä käytäntönä pidetään, että main-funktio on main.dart-tiedostossa. (13; 15.)

Tiedosto pubspec.yaml on projektin konfiguraatiotiedosto, jota kehittäjän tarvitsee muokata useasti. Tämä tiedosto sisältää yleiset projektin asetukset kuten projektin nimen, kuvauksen ja version. Tiedostosta löytyvät myös kirjastot, joista projekti on riippuvainen. Resurssit eli esimerkiksi kuvat ja projektissa käytettävät fontit löytyvät myös pubspec.yaml-tiedostosta. Kun uusia riippuvuuksia lisätään tämän tiedoston riippuvuusosioon, Flutter SDK hoitaa riippuvuuden lataamisen ja lisäämisen. (13.)

Esimerkkikoodi 2 on yksinkertainen Flutter-sovellus, joka näyttää tekstin ”Hello, world” keskellä näyttöä. Funktio runApp ottaa sille annetun Center-komponentin ja tekee siitä komponenttipuun juuren. Tässä esimerkissä komponenttipuu koostuu kahdesta komponentista, Center-komponentista ja sen lapsesta Text-komponentista. Ohjelmistokehys pakottaa komponenttipuun juuren peittämään koko sivun, mikä tarkoittaa sitä, että Text-komponentti näkyy keskitettynä sivulla, koska Center-komponentti keskittää lapsensa itsensä sisällä ja Center-komponentti peittää koko sivun. Rivillä 8 pitää määritellä tekstin suunta käyttämällä Text-komponentin textDirection-attribuuttia. Myöhemmin esiteltävässä esimerkissä tekstin suuntaa ei tarvitse määrittää, koska komponenttipuun juurena käytetään MaterialApp-komponenttia. (16.)

```
1 import 'package:flutter/material.dart';
2
3 void main() {
4   runApp(
5     const Center(
6       child: Text(
7         'Hello, world!',
8         textDirection: TextDirection.ltr,
9       ),
10     ),
11   );
12 }
```

Esimerkkikoodi 2. Hello world -sovellus Flutterilla. (16)

Kehitettäessä sovellusta usein luodaan uusia komponentteja, jotka ovat joko StatelessWidget- tai StatefulWidget-komponentin aliluokkia riippuen siitä, tarvitseeko komponentilla olla tila. Jos halutaan, että komponentilla on tila, niin silloin komponentin pitää olla StatefulWidget-komponentin aliluokka. Komponentin pääsääntöinen tehtävä on implementoida build-metodi, jonka tehtävänä on rakentaa komponentti muiden alhaisen tason komponenttien avulla. (16.)

Monimutkaisten toiminnallisuuksien rakentamiseen vaaditaan yleensä komponentti, jolla on tila. Otetaan esimerkiksi tilanne, jossa jollakin sovelluksen sivulla halutaan näyttää jokin laatikko, jonka väriä voidaan vaihtaa painamalla painiketta. Tämänkaltaisen toiminnallisuuden toteuttamiseen tarvitaan komponenttia, jolla on tila. Flutterissa StatefulWidget-komponentit osaavat generoida State-objekteja, joita käytetään tilan säilyttämiseen. (16.)

Liitteen 1 esimerkkikoodi toimii esimerkkinä StatefulWidget-komponentin käytöstä. Esimerkissä nähdään StatefulWidget-komponentti nimeltä Counter ja State-objekti. Flutterissa näillä kahdella objektilla on eri elinkaaret. Komponentit ovat väliaikaisia objekteja, joiden avulla rakennettaan sovelluksen ulkoasu niin, että se vastaa sovelluksen tämänhetkistä tilaa. State-objektit ja niiden data säilyvät build-metodikutsusta toiseen, mikä mahdollistaa datan säilymisen. (16.)

Liitteen 1 esimerkkikoodista nähdään, että kun käyttäjä painaa ElevatedButton-komponenttia niin metodia _increment kutsutaan. Edellä mainittu metodi puolestaan kutsuu setState-metodia, jonka sisällä State-objektin _counter-nimisen ja kokonaislukutyyppisen luokkamuuttujan arvoa kasvatetaan yhdellä. Metodikutsu setState kertoo Flutterille, että jotain tässä kyseisessä tilassa on muuttunut, mikä saa Flutterin ajamaan build-metodin uudestaan, jotta näytöllä voidaan näyttää uusi _counter-nimisen muuttujan arvo. (16.)

Opinnäytetyön osana kehitettävässä säästösovelluksessa käytetään Provider-kirjastoa, jonka avulla voidaan pitää business-logiikka erillään komponenteista. Kehittäjä voi kirjoittaa aliluokkia, jotka periytyvät luokasta ChangeNotifier. Muut objektit voivat kuunnella ChangeNotifier-objektia. Kun ChangeNotifier-luokan

luokkamuuttujiin tehdään muutoksia, voidaan luokassa kutsua metodia notifyListeners, jolloin kaikki sen kuuntelijat saavat tiedon muuttuneesta tilasta. ChangeNotifierin kuunteleminen tapahtuu rekisteröimällä callback-funktio, jota kutsutaan, kun ChangeNotifier on kutsunut notifyListeners-metodia. Jokaisesta sovelluksessa käytettävästä ChangeNotifier-objektista luodaan ChangeNotifier-Provider-objekti sovelluksen käynnistyksen yhteydessä. (17.)

Consumer on Provider-kirjaston tarjoama komponentti, jonka avulla komponentit voivat kuunnella muutoksia ChangeNotifierProvider-objekteissa, joihin jatkossa viitataan nimellä Provider. Jos halutaan, että jokin komponentti kuuntelee Provideria, pitää komponentti asettaa Consumerin builder-metodin paluuarvoksi. Builder-metodissa päästään käsiksi Provideriin ja sen dataan. Consumer-komponentti on hyödyllinen siksi, että sen builder niminen callback-metodi rakentaa ainoastaan builder-metodin palauttaman komponentin uudestaan. Tämä tarkoittaa sitä, että Provideria kuunteleva komponentti rakentaa ainoastaan itsensä ja lapsensa uudelleen. (18.)

Otetaan esimerkiksi jokin sivukomponentti, jossa on lista erilaisia komponentteja. Jos jonkin sivulla näkyvän komponentin dataa pitää muuttaa, niin setState-metodin kutsuminen sivulla pakottaa Flutterin kutsumaan sivun build-metodia, jolloin kaikki sivulla listattavat komponentit on rakennettava uudelleen. Paljon hyödyllisempää on säilyttää sivulla tarvittavaa dataa Provider-objekteissa ja antaa listattavien komponenttien kuunnella haluamiansa Providereita. Kun listattavan komponentin kuuntelemassa Providerissa kutsutaan notifyListeners-metodia, vain listattava komponentti ja muut Provideria kuuntelevat komponentit rakennetaan uudelleen, jolloin vältetään esimerkiksi sovelluksen yläpalkin turha uudelleenrakennus. Säästösovelluksen toteutusta kuvaavissa luvuissa havainnollistetaan Provider-objektin ja Consumer-komponentin hyödyllisyyttä käyttämällä esimerkkinä yhtä bitcoin-säästösovelluksen sivuista.

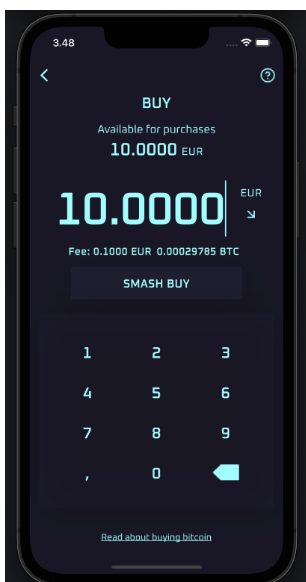
3 Bitcoin-säästösovellus

Tässä luvussa esitellään Bitcoin-säästösovellusdemoa käyttäjän näkökulmasta sekä kerrotaan, miten sovelluksen tuotantoversiolla voidaan tehdä rahaa. Flutter valittiin ohjelmistokehykseksi, jotta sovellus voitaisiin integroida jo olemassa oleviin sovelluksiin ja siksi, että ei tarvitse kehittää kahta eri mobiilisovellusta. Tekniset ratkaisut esitellään seuraavassa luvussa.

Liikeidea on myydä sovellusta asiakkaille, jotka voivat integroida sovelluksen jo olemassa oleviin sovelluksiin ja näin ollen tarjota asiakkaillensa bitcoin-säästösovellusta. Asiakkaat voivat myös tietenkin käyttää sovellusta itseään eikä integroida sitä mihinkään toiseen sovellukseen. Flutterin yksi hienoista ominaisuuksista on se, että Flutter-koodia voi integroida suoraan olemassa oleviin projekteihin, joten asiakkaan ei tarvitse luoda uusia projekteja alusta asti.

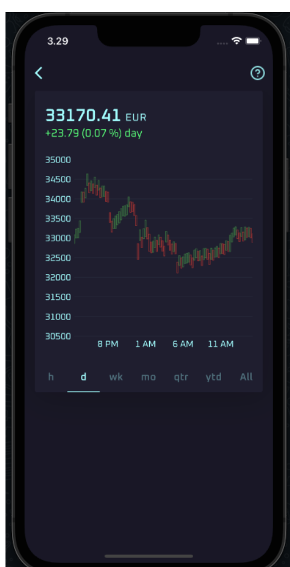
Sovelluksen omistaja, eli se taho kenelle säästösovellus on myyty, tekee rahaa sovelluksella niin, että jokaisesta osto- ja myynti tapahtumasta otetaan provisio, joka on pieni prosenttiosuus myytävästä tai ostettavasta summasta. Kuvassa 5 nähdään, miten provisio ilmoitetaan käyttäjälle sovelluksen ostosivulla. Rahaa voidaan tehdä myös ostamalla bitcoineja käyttäjiltä alle markkinahinnan, jolloin käyttäjä pääsee välittömästi eroon bitcoineistaan ja sovelluksen omistaja tekee rahaa, kun myy bitcoinin markkina-arvolla.

Kuvassa 5 näkyy myös, kuinka paljon bitcoineja käyttäjä saa sillä määrällä euroja, jonka hän on syöttänyt tekstikenttään. Käyttäjä näkee sivulta myös, kuinka paljon euroja hän on tallettanut sovellukseen. Käyttäjä voi vaihtoehtoisesti syöttää, kuinka monta bitcoinia hän haluaa ostaa. Tämä tapahtuu painamalla tekstikentän vieressä olevaa nuoli-kuvaketta tai painamalla kohtaa, jossa ilmoitetaan, kuinka paljon bitcoineja syötetyllä euromäärällä saa. Kun käyttäjä esimerkiksi kirjoittaa, kuinka monella eurolla hän haluaa ostaa bitcoineja, euroilla saatavien bitcoinien määrä automaattisesti päivittyy. Tämä pätee myös tilanteeseen, jossa käyttäjä syöttää, kuinka monta bitcoinia halutaan ostaa.



Kuva 5. Sovelluksen ostosivu

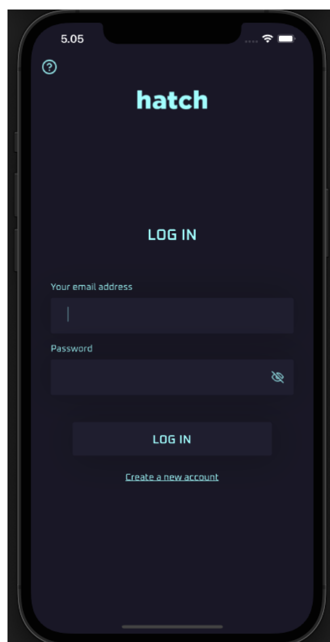
Sovelluksen käyttötarkoitus on, että käyttäjät voivat ostaa, myydä ja säilöä bitcoineja käyttäjätilillänsä. Käyttäjä voi myös säilyttää rahaa käyttäjätilillänsä ja nostaa siellä olevat rahat halutessaan pois. Kuva 6 havainnollistaa, kuinka käyttäjät näkevät sovelluksen kynttiläkaaviosta bitcoinin arvon vaihtelun aikavälillä, jonka käyttäjä voi itse valita.



Kuva 6. Sovelluksen kynttiläkaavio.

Kynttiläkaavio on kaavio, joka sisältää pystypalkkeja, joista näkee avaushinnan, korkeimman hinnan kyseisen ajanjakson aikana, matalimman hinnan kyseisen ajanjakson aikana ja sulkuhinnan eli mikä oli esimerkiksi osakkeen viimeisin hinta kyseisenä ajanjaksona. Kynttiläkaavioita voidaan käyttää osakkeen hinnan ennustamiseen. Lukemalla kynttilän eli pystypalkin arvoja voidaan päättää, onko järkevää ostaa vai myydä osake. (19.)

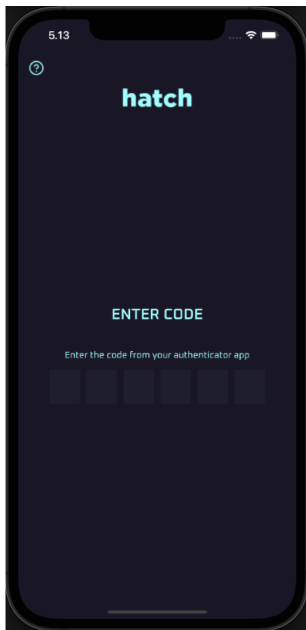
Esimerkki tyypillisestä sovelluksen käyttökerrasta on, että käyttäjä kirjautuu sovellukseen käyttäen kaksivaiheista tunnistautumista. Tunnistauduttuaan käyttäjä näkee sovelluksen kotisivulta, kuinka monta euroa ja bitcoinia hänen käyttäjätilillänsä on. Jos käyttäjällä ei ole euroja hänen käyttäjätilillänsä tulee hänen tallettaa sinne rahaa ennen kuin hän voi ostaa bitcoineja. Jos käyttäjän tilillä on rahaa, hän voi ostaa bitcoineja ja jättää ne käyttäjätilillensä säästöön tai nostaa bitcoinit hänen henkilökohtaiseen bitcoin-lompakkoonsa. Kuvassa 7 nähdään sivu, jota käyttäjä käyttää sisäänkirjautumiseen.



Kuva 7. Sivu, jossa käyttäjä voi kirjautua tai rekisteröityä.

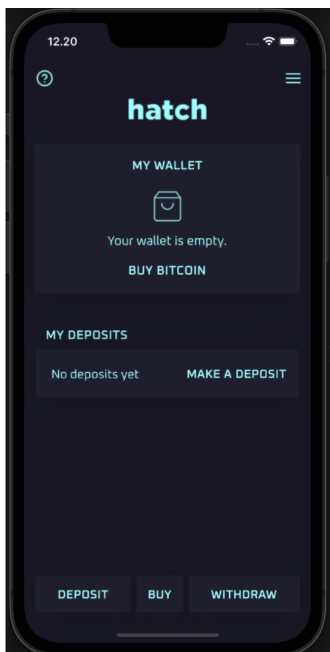
Kun käyttäjä on syöttänyt sähköpostin ja salasanan oikein, hänet ohjataan sivulle, jossa syötetään kertakäyttöinen salasana, joka generoidaan esimerkiksi

Google Authenticator -sovellusta käyttäen. Rekisteröitymisen yhteydessä käyttäjälle annetaan koodi, jonka hän voi syöttää esimerkiksi edellä mainittuun sovellukseen, jolloin sovellus voi generoida käyttäjälle kertakäyttöisiä salasanoja. Käyttäjä tarvitsee kertakäyttöisiä salasanoja kirjautuakseen sisään, koska sisäänkirjautuminen vaatii kaksivaiheisen tunnistautumisen. Kuvassa 8 nähdään sivu, jossa käyttäjä syöttää kertakäyttöisen salasanan. Jos käyttäjä syöttää oikean salasanan tällä sivulla, hänet ohjataan sovelluksen pääsivulle.



Kuva 8. Sivu, jossa käyttäjä syöttää kertakäyttöisen salasanan.

Sovelluksen pääsivulla käyttäjä näkee varallisuutensa sekä voi navigoida osto-, nosto- ja talletusosioihin. Käyttäjä näkee, paljon hänellä on rahaa euroina ja bitcoineina. Näkymän ulkoasu riippuu siitä, onko käyttäjällä talletuksia vai ei. Kuvassa 9 nähdään, miltä pääsivu näyttää, kun käyttäjällä ei ole Bitcoineja eikä hän ole tallettanut euroja sovellukseen.



Kuva 9. Sovelluksen pääsivu silloin, kun käyttäjän lompakko on tyhjä.

Kuvassa 10 nähdään sovelluksen pääsivu, kun käyttäjällä on talletuksia ja bitcoineja. Sovelluksen lompakkokomponentissa näkyy vaihtoehdot btc, sats ja eur, joita klikkaamalla käyttäjä näkee, kuinka paljon hänellä on mitäkin valuuttaa. Sats tarkoittaa satoshia, joka on yksi sadasmiljoonasosa bitcoinia. Sininen viiva valuutan alapuolella on animoitu ja siirtyy liukuen seuraavan valinnan alapuolelle. Painamalla lompakkokomponentin oikeassa yläkulmassa sijaitsevaa kuvaketta käyttäjä voi avata sivun, jossa näytetään aikaisemmin mainittu kynttiläkaavio.

Kuva 10. Sovelluksen pääsivu silloin, kun käyttäjällä on lompakossa valuuttaa.

4 Bitcoin-säästösovelluksen toteutus

Säästösovellus on toteutettu Flutter-ohjelmistokehyksellä ja sovelluksessa käytetään joitakin avoimen lähdekoodin kirjastoja kuten esimerkiksi syncfusion_flutter_charts-kirjastoa, jonka avulla piirretään sovelluksessa esiintyvä kynttiläkaavio. Tilanhallinta ratkaisuna käytetään Provider-kirjastoa ja HTTP-pyynnöt tehdään http-nimisen kirjaston avulla.

Sovellus toteutettiin iteratiivisesti eli valittiin aina noin kahden viikon mittaiseen jaksoon tietty määrä toimintoja, jotka haluttiin rakentaa. Sovelluksen kehitys aloitettiin kotisivusta, ostotoiminnosta ja talletustoiminnosta. Tämän jälkeen rakennettiin kirjautuminen ja rekisteröityminen sekä niihin liittyvä kaksivaiheinen tunnistautuminen ja toiminto, joka pyytää käyttäjää varmentamaan muuttuneen IP-osoitteen, jos käyttäjä käyttää sovellusta IP-osoitteesta, josta hän ei ole aiemmin sitä käyttänyt. Lopuksi toteutettiin toiminallisuus, joka kirjaa käyttäjän automaattisesti ulos sovelluksesta, jos hänen käyttäjävarmenteensa on vanhentunut.

Iteraatioiden aikana tehtiin myös useita optimointeja sovellukseen kuten esimerkiksi poistettiin sivukomponenttien turhat uudelleenrakennukset, jos niissä käytettävät tilamuuttujat eivät muuttuneet. Edellä mainittu esimerkki toteutettiin niin, että vain sivuilla esiintyvät komponentit rakennetaan uudelleen sen sijasta, että koko sivu rakennettaisiin uudelleen. Ylimääräisten sivujen poistaminen navigaatiopinosta on myös esimerkki optimoinnista, ja tästä puhutaan lisää aliluvussa 4.2. Sovelluksessa käytettään myös jokaisessa mahdollisessa kohdassa const-avainsanaa, joka kertoo Flutter-ohjelmistokehykselle, että lapsikomponenttia ei tarvitse uudelleen rakentaa, kun sen vanhempi rakennetaan uudelleen.

4.1 Sovelluksen sivut

Suurin osa sovelluksen sivuista on StatefulWidget-komponentteja, joilla on oma tilansa. Sivut ovat yleisesti ottaen StatefulWidgettejä siitä syystä, että niissä

käytetään TextEditingController-objektia, jonka avulla voidaan kuunnella muutoksia tekstikentässä, johon käyttäjä kirjoittaa. On suositeltavaa käyttää TextEditingControlleria ainoastaan StatefulWidgeteissä, jotta sivu ei rakennu koko ajan uudestaan, kun käyttäjä avaa esimerkiksi näppäimistön, koska tällöin TextEditingController luotaisiin uudestaan, kun sivu luodaan uudestaan näppäimistön auetessa. Tällöin TextEditingController-muuttuja alustettaisiin uudelleen ja data menetettäisiin. StatefulWidget muistaa tilansa, joten se myös säilyttää TextEditingController-muuttujan arvon eikä alusta sitä uudestaan. TextEditingControlleria on myös suositeltavaa käyttää vain StatefulWidget-komponenteissa siitä syystä, että TextEditingControllerin voi hävittää kutsumalla sen dispose-metodia samalla, kun komponentti kutsuu omaa dispose-metodiansa. Edellä mainittu estää muistivuotoja.

Useat sivut käyttävät tilamuuttujana Future-tyyppistä muuttujaa, johon voidaan asynkronisesti hakea dataa, esimerkiksi HTTP-pyynnöllä ja sivulla näytettävä data sekä mahdollisesti sen ulkonäkö rakennetaan HTTP-pyynnöstä palautuneen datan perustella. Future-tyyppiset muuttujat toimivat niin, että ne nimensä mukaisesti sisältävät jotain dataa tulevaisuudessa eli kestää hetki ennen kuin muuttuja saa jonkun arvon kuten esimerkiksi käyttäjän lompakon sisällön.

Useiden sovelluksen sivujen rakentamiseen käytetään FutureBuilder-komponenttia, jolle annetaan Future-tyyppinen muuttuja. FutureBuilder on Flutter-ohjelmistokehyksen tarjoama komponentti. FutureBuilder piirtää sivun sen mukaan, onko Future-tyyppinen muuttuja saanut jo arvon vai odottaako se vielä arvon saantia. Esimerkiksi tilanteessa, jossa odotetaan arvon saantia, voidaan sivulle piirtää esimerkiksi latausindikaattori. Virallisen Flutter-dokumentaation mukaan FutureBuilderia tulee käyttää ainoastaan StatefulWidget-komponenteissa, koska FutureBuilderille annettavaa Future-tyyppistä parametria ei saa luoda StatelessWidget-komponenteissa (20).

Esimerkkikoodi 3 havainnollistaa FutureBuilderin ja Futuren käyttöä. Metodi getFuture palauttaa Future<http.Response>-tyyppisen objektin. Future-tyyppinen muuttuja voi olla kolmessa eri tilassa. Future voi olla keskeneräisessä tilassa,

valmiissa tilassa sisältäen dataa ja valmiissa tilassa sisältäen virheen. Kun FutureBuilder rakennetaan, sille parametrina annettua getFuture-metodia kutsutaan. FutureBuilder-komponentin builder-metodin dataSnapshot-parametria käytetään, kun yritetään selvittää, missä tilassa FutureBuilderille annettu Future-tyyppinen muuttuja on.

Esimerkkikoodista 3 nähdään, että FutureBuilder-komponentin builder-metodin paluuarvona on oltava komponentti. Jos Future-tyyppinen response-muuttuja on keskeneräisessä tilassa, niin builder-metodi palauttaa CircularProgressIndicator-komponentin, joka toimii latausindikaattorina. Jos response-muuttuja on valmiissa tilassa ja sisältää virheen, niin ruudun keskellä näytetään teksti ”An error occured”. Jos virhettä ei ole, niin voidaan olla varmoja siitä, että response muuttuja on valmiissa tilassa ja sisältää dataa. Response muuttujan dataan päästään käsiksi dataSnapshot-objektin avulla.

```
Widget buildFutureBuilder() {
    return FutureBuilder(
      future: getFuture(),
      builder: (context, dataSnapshot) {
        if (dataSnapshot.connectionState == ConnectionState.waiting) {
          return const Center(
            child: CircularProgressIndicator(),
          );
        } else {
          if (dataSnapshot.error != null) {
            return const Center(
              child: Text('An error occured'),
            );
          } else {
            return Text('data from server ${dataSnapshot.data}');
          }
        }
      },
    );
  }

  Future<http.Response> getFuture() async {
    var response = await http.get('https://exampleurl.com/exam-
pledata');
    return response;
  }
```

Esimerkkikoodi 3. Esimerkki FutureBuilder-komponentin ja Future-tyyppisen muuttujan käytöstä.

Sivuille voidaan navigoida nimettyjen reittien avulla. Jokaisessa sovelluksen sivussa määritellään sivun reitin nimi. Main.dart-tiedostossa kaikki nämä reittien nimet annetaan Map-tyyppisenä parametrina GetMaterialApp-komponentille. Kaikki nämä reittien nimet on ilmoitettava siksi, että reittien nimiä käytetään navigoidessa Navigator-komponentilla, jonka on tiedettävä, mikä sivu vastaa mitäkin reitin nimeä, jotta sivut voidaan rakentaa ja lisätä navigaatiopinoon. Esimerkkikoodi 4 näyttää, miten Map-tyyppinen objekti, joka sisältää reittien nimet, annetaan parametrina GetMaterialApp-komponentin konstruktorille.

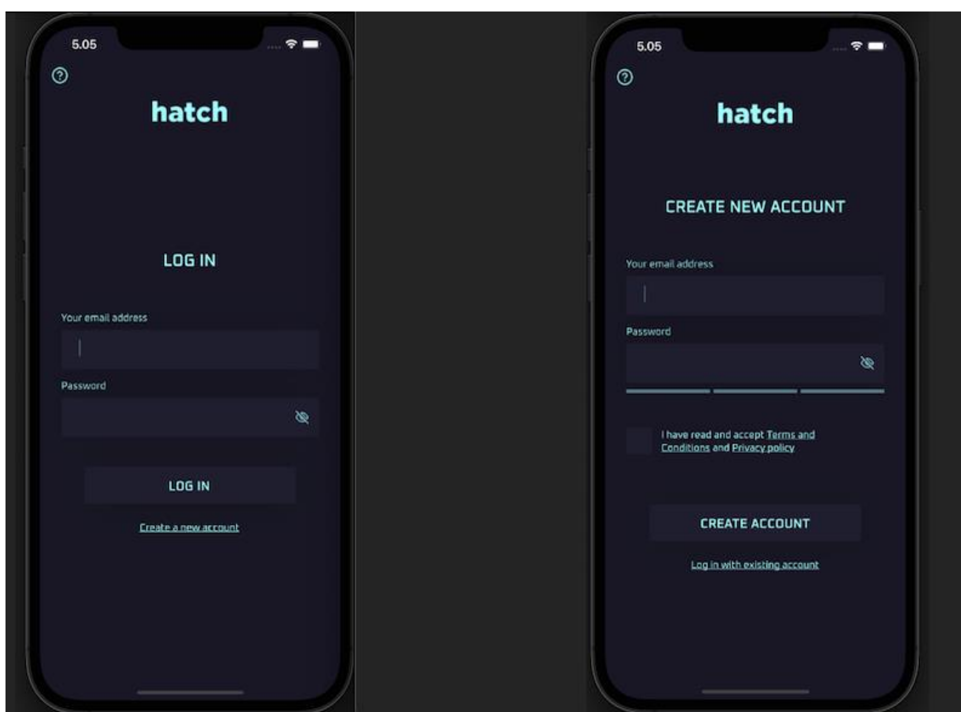
```
      child: GetMaterialApp(
        debugShowCheckedModeBanner: false,
        title: 'Hatch',
        initialRoute: '/',
        routes: {
          '/': (ctx) => const AuthScreen(),
          RegisterCodeScreen.routeName: (ctx) => const Register-
CodeScreen(),
          OtpCodeScreen.routeName: (ctx) => const OtpCodeScreen(),
          AccountCreatedScreen.routeName: (ctx) => const AccountCreat-
edScreen(),
          ConfirmIPAdrressScreen.routeName: (ctx) =>
              const ConfirmIPAdrressScreen(),
          WalletScreen.routeName: (ctx) => const WalletScreen(),
          BuyScreen.routeName: (ctx) => const BuyScreen(),
          BtcRatesChartScreen.routeName: (ctx) => const
BtcRatesChartScreen(),
          BuySuccessfulScreen.routeName: (ctx) => const BuySuccess-
fulScreen(),
          DepositScreen.routeName: (ctx) => const DepositScreen(),
          DepositSuccessfulScreen.routeName: (ctx) =>
              const DepositSuccessfulScreen(),
        },
      ),
```

Esimerkkikoodi 4. Reittien määritys GetMaterialApp-komponentin konstruktorissa.

GetMaterialApp on eräänlainen kääre MaterialApp-komponentille, joka toimii komponenttipuun juurena. GetMaterialApp on get-kirjaston tarjoama komponentti ja sitä käytetään projektissa, koska sen avulla voidaan navigoida ilman BuildContext-objektia, jota voidaan käyttää vain komponentin build-metodissa. Edellä mainittu mahdollistaa navigaation sovelluksen ErrorHelper-luokassa, jonka handleError-funktiota kutsutaan, kun mikä tahansa HTTP-pyyntö, joka vaatii käyttäjävarmenteen, palauttaa virheviestin. Jos virheviesti kertoo, että

käyttäjän varmentaminen epäonnistui, navigoidaan sivulle, jossa käyttäjä voi kirjautua sovellukseen sisään.

Sivu, jolla käyttäjä voi kirjautua sisään ja rekisteröityä, ovat yksi ja sama sivu. Tämä on mahdollista käyttämällä animaatioita. Kun käyttäjä painaa kuvan 11 vasemmalla olevalla sivulla näkyvää ”Create a new account” -nappia, sivu animoidaan näyttämään kuvan 11 vasemmalla puolella olevalta sivulta. Animointi tapahtuu käyttämällä Flutter-ohjelmistokehyksen tarjoamaa komponenttia AnimatedContainer, joka saa parametreinaan animaation tyypin, animaation keston ja komponentin korkeuden. Kaikki sivulla näkyvät komponentit ovat AnimatedContainerin lapsikomponentteja lukuun ottamatta sivun yläosassa näkyvää ”hatch” logoa. Kun AnimatedContainer rakennettaan uudelleen, sen korkeutta animoidaan, joka saa sen näyttämään siltä kuin se venyisi.



Kuva 11. Käyttäjän todentamiseen käytetty sivu kahdessa eri tilassa.

Animoiminen viime luvussa esitetyssä esimerkissä tapahtuu niin, että kun painiketta painetaan, kutsutaan sivun setState-metodia ja sivukomponentin luokka-

muuttujan, joka pitää sisällään tiedon siitä, ollaanko rekisteröitymässä vai kirjautumassa sisään, arvoa muutetaan. Tämän luokkamuuttujan nimi on _authMode. Sivua rakennettaessa uudelleen myös AnimatedContainer-komponentti rakennetaan uudelleen ja sen korkeus animoidaan sen mukaan, mikä on muuttujan _authMode arvo.

4.2 Navigoiminen

Navigoiminen on olennainen osa jokaista sovellusta. Tässä aliluvussa käsitellään yksi esimerkkitapaus, jossa navigoiminen toteutettiin aluksi väärin ja kerrotaan, mitä ongelmia se aiheutti ja miten ongelma korjattiin. Sovelluksessa käytetään Flutter-ohjelmistokehyksen tarjoamaa Navigator-komponenttia. Aliluvussa kerrotaan myös esimerkin avulla, miten sivulta toiselle voidaan välittää dataa Navigator-komponentin avulla.

Navigoiminen tapahtuu nimettyjen reittien avulla niin kuin aiemmin mainittiin. Sivulta toiselle navigoidaan käyttämällä navigoinnin kohteena olevan sivun reitti nimeä, sillä sivulla, josta halutaan navigoida pois. Kun sovellus käynnistetään, ensimmäisenä näytetään sivu, jossa käyttäjä voi kirjautua sisään. Tämä sivu lisätään navigointipinoon. Ei ole kuitenkaan tarkoituksen mukaista, että käyttäjä voi palata tälle sivulle painamalla takaisin-navigointipainiketta, joka löytyy joistakin laitteista, joiden käyttöjärjestelmä on Android. Jos ei haluta, että käyttäjä voi navigoida takaisinpäin edelliselle sivulle, on se sivu poistettava navigaatiopinosta. Tämä voidaan hoitaa yksinkertaisesti kutsumalla Navigator-komponentin pushReplacementNamed-metodia, joka poistaa nykyisen sivun navigaatiopinosta samalla, kun se navigoi uudelle sivulle.

Tehtäessä säästösovellusta ilmeni ongelma. Kun oltiin sovelluksen pääsivulla ja painettiin takaisinpäin navigointinappia Android-laitteella, ruudulle ilmestyi uusi pääsivu sen sijasta, että sovellus suljettaisiin. Ongelma johtui siitä, että navigaatiopinossa oli useita pääsivuja, koska niitä ei poistettu navigaatiopinosta navigoitaessa sivulta toiselle. Esimerkiksi pääsivulta navigoitiin talletussivulle, josta navigoitiin onnistunut talletussivulle, josta navigoitiin takaisin pääsivulle. Tämä

aiheutti sen, että navigaatiopinossa oli kaksi pääsivua. Ongelma korjattiin käyttämällä Navigator-komponentin pushNamedAndRemoveUntil-metodia onnistunut talletussivulla. Tämän metodin avulla kaikki muut sivut paitsi sivu, jolle navigoidaan eli pääsivu, poistettiin navigaatiopinosta, jolloin navigaatiopinossa ei ollut ylimääräistä ja mahdollisesti epätoivottuja sivuvaikutuksia aiheuttavaa pääsivua.

Sovelluksessa käytetään myös Navigator-komponentin tarjoamien metodien arguments-nimistä parametria. Arguments-parametria käyttämällä voidaan välittää dataa sivulta toiselle. Esimerkiksi sivulla, jolla käyttäjä voi ostaa bitcoineja, toimitaan niin, että ostettujen bitcoinien määrä välitetään sivulle, jossa ilmoitetaan onnistuneesta ostotapahtumasta. Tällä sivulla, jossa ilmoitetaan onnistunut ostotapahtuma, voidaan hakea sille arguments-parametrin avulla välitettyä dataa. Data haetaan sivun build-metodissa ja sitten sitä käytetään näyttämällä juuri ostettujen bitcoinien määrä sivun keskellä.

4.3 Tilanhallinta

Tässä aliluvussa esitellään, miten Provider-objektia ja Consumer-komponenttia on hyödynnetty bitcoin-säästösovelluksen toteutuksessa. Toimintoja, joissa käytetään tilanhallintaratkaisuja, on paljon, joten alilukuun valittiin yksi esimerkki, joka on yritetty selittää mahdollisimman tarkasti. Aliluvun lopussa on kappale, jossa selitetään käsiteltävä esimerkki abstraktilla tasolla.

Kuvassa 12 sininen neliö on BuyScreen-komponentti. Punainen neliö on NumberKeyboard, joka on BuyScreenin lapsikomponentti. Keltainen neliö on BuyScreenTextField-komponentti, joka on myös BuyScreenin lapsikomponentti. BuyScreenTextField on Consumer-komponentin builder metodin palauttama komponentti. Edellä mainittu Consumer kuuntelee Trading-objektissa tapahtuvia muutoksia. Trading on ChangeNotifier-luokan aliluokka eli siitä tehdään ChangeNotifierProvider-tyyppinen objekti, kun sovellus käynnistetään. Kun näppäimistökomponentilla kirjoitetaan, niin teksti ilmestyy BuyScreenTextFieldin

tekstikenttään eli kuvassa 12 nähtävään tekstikenttään, joka on ”Fee”-tekstin yläpuolella.

Kuvassa 12 nähtävän näppäimistön tuottamassa tekstissä tapahtuvia muutoksia kuunnellaan kuuntelijassa, joka on rekisteröity BuyScreen-komponentissa. Kuuntelijalle on annettu funktio, jota kutsutaan joka kerta, kun näppäimistöllä kirjoitetaan. Kuuntelijalle annettu funktio kutsuu Trading-objektin fiatToCrypto-Conversion-metodia, joka saa parametrikseen näppäimistöllä tuotetun tekstin.

Metodi fiatToCryptoConversion tekee muunnoksen euroista bitcoineihin, eli se kertoo, kuinka monta bitcoinia syötetyllä euro määrällä saa ja paljonko joudutaan maksamaan provisiota. Kun fiatToCryptoConversion-metodi saa palvelimelta muunnoksen tuloksen, se ottaa tuloksen talteen Trading-objektin muuttujiin ja kutsuu notifyListeners-metodia. NotifyListeners-metodi saa Consumerin kutsumaan sen builder-metodia, joka palauttaa BuyScreenTextField-komponentin. Builder-metodin sisässä päästään käsiksi muuttuneeseen dataan ja sen avulla kuvassa 12 näkyvä BuyScreenTextField voidaan rakentaa uudelleen niin, että siinä käytetään muuttunutta dataa.

Kuva 12. Ostosivu ja sen komponentit.

Kolmessa aikaisemmassa kappaleessa käsitellyn tapauksen voi tiivistää lyhyesti. Kun näppäimistöllä kirjoitetaan, niin teksti ilmestyy tekstikenttään, ja aina kun teksti muuttuu, se lähetetään palvelimelle, jossa sille tehdään muunnos. Kun palvelin on palauttanut muunnoksen tuloksen, niin tekstikenttä piirrettään uudelleen ruudulle niin, että siinä käytetään juuri palvelimelta saatua muunnoksen tulosta. Tämä tarkoittaa siis sitä, että kohta, jossa provisio näytetään, päivittyy vastaamaan siitä summasta otettavaa provisioita, joka tekstikenttään on syötetty. Myös kohta, jossa näytetään, kuinka monta bitcoinia syötetyllä sumalla saadaan, päivittyy. Consumer on todella hyödyllinen tässä tilanteessa, koska vain tekstikenttä tarvitsee piirtää uudelleen ruudulle sen sijasta, että koko sivu piirrettäisiin uudelleen, joka olisi turhaa.

5 Yhteenveto

Työssä pyrittiin antamaan yleiskuva Flutter-ohjelmistokehyksen valitsemisen hyödyistä ja huonoista puolista. Pyrittiin myös kuvaamaan Flutteria teknisestä näkökulmasta eli esimerkiksi annettiin esimerkkikoodeja, joiden avulla havainnollistettiin, miten Flutter-sovellusten koodia kirjoitetaan. Työn tavoitteena oli myös rakentaa bitcoin-säästösovellusdemo ja dokumentoida osa sen toiminnallisuuksista. Kaikki työn tavoitteet saavutettiin.

Demosovellus saatiin rakennettua ja dokumentoitua. Havaittiin, että Flutter oli hyvä valinta demosovelluksen toteuttamiseen, koska sillä on nopea kehittää sovelluksia. Saatiin demo, joka toimii iOS- ja Android-puhelimilla sekä verkkoselaimella. Ilman demosovellusta tuotetta ei voitaisi esitellä mahdollisille asiakkaille yhtä vakuuttavalla tavalla.

Kaikki demosovellukseen halutut toiminnot ja sivut saatiin toteutettua ja ne toimivat halutulla tavalla. Sovelluksen uusimmasta versioista ei ole löydetty vikoja. Kehityksen aikana ilmeni joitain vikoja kuten se, että navigaatiopinossa oli sama sivu monta kertaa, mutta kaikki ilmenneet viat korjattiin kehityksen aikana tai kehitysprosessin lopussa. Demoa voi siis käyttää juuri niin kuin haluttiin eli sitä voi esitellä mahdollisille asiakkaille, jotka ovat kiinnostuneita bitcoin-säästösovelluksesta.

Demosovelluksen toteutuksesta on taloudellista hyötyä, koska esittelemällä sitä voidaan saada asiakkaita, jotka haluavat ostaa bitcoin-säästösovelluksen. Taloudellinen hyöty tulee siis siitä, että jos asiakas haluaa ostaa bitcoin-säästösovelluksen, he maksavat sen kehityksestä Oivanille eli tämän insinöörityön tilaajalle. Oivan siis kehittää tuotantokelpoisen, kustomoidun ja tietoturvallisen sovelluksen, jonka kehittämisen asiakas maksaa.

Sovelluksessa voisi olla parempi arkkitehtuuri eli esimerkiksi koodi olisi voitu jakaa useampiin luokkiin. Arkkitehtuuri on kuitenkin sopiva demosovellukselle, koska tarkoituksena oli luoda demo, jota päästäisiin nopeasti esittelemään mahdollisille asiakkaille. Arkkitehtuurin ei siis tarvitse olla niin edistynyt, kun se tulee

olemaan versioissa, joissa on enemmän toimintoja ja joita myydään asiakkaille. Useita sovelluksen komponentteja voidaan kuitenkin käyttää asiakkaille tehtävissä tuotantoversioissa. Esimerkiksi ostosivun näppäimistökomponentista voi tehdä kirjaston, jolloin sitä voi helposti käyttää muissakin sovelluksissa, joissa tarvitaan näppäimistöä.

Työssä ei tarvinnut tehdä kompromisseja esimerkiksi sen suhteen, ehtiikö demoon lisätä jonkin toiminnallisuuden. Kaikki toiminnollisuudet, joita pidettiin olennaisina, rakennettiin demosovellukseen, jotta demo on esiteltävän arvoinen. Se antaa tarpeeksi hyvän kuvan siitä, millainen sovelluksesta tehtävä tuotantoversio olisi. Jatkokehityksen varaan ei siis jäänyt mitään.

Lähteet

1 Wikipedia contributors. Cross-platform software. 2022. Verkkoaineisto <https://en.wikipedia.org/wiki/Cross-platform_software>. Luettu 16.2.2022.

2 Kononenko, Vitaliy. 2020. FLUTTER: 7 MAJOR BENEFITRS FOR BUSINESSES. Verkkoaineisto <https://computools.com/flutter-benefits-for-businesses/>. Luettu 16.2.2022.

3 Agrawal, Mitisha. 2021. Major Advantages For Choosing Flutter For New App Development. Verkkoaineisto <https://www.zehntech.com/major-advantages-for-choosing-flutter-for-new-app-development/>. Luettu 12.4.2022.

4 Dziuba, Anna. Top 8 Flutter Advantages and Why You Should Try Flutter on Your Next Project. Verkkoaineisto <https://relevant.software/blog/top-8-flutter-advantages-and-why-you-should-try-flutter-on-your-next-project/#2_Reduced_Code_Development_Time>. Luettu 12.4.2022.

5 Goncharenko, Oleg. 2022. Flutter vs. React Native in 2022 – Detailed Framework Comparison [Update January]. Verkkoaineisto <https://brocoders.com/blog/flutter-vs-react-native/>. Luettu 18.4.2022.

6 The Good and the Bad of Flutter App Development. 2021. Verkkoaineisto <https://www.altexsoft.com/blog/engineering/pros-and-cons-of-flutter-app-development/>. Luettu 18.4.2022.

7 Raheja, Sandeep. 2022. Should You Use Flutter For Mobile App Development in 2022. Verkkoaineisto <https://taazaa.com/use-flutter-for-mobile-app-development/>. Luettu 21.4.2022.

8 Dart overview. Verkkoaineisto <https://dart.dev/overview/>. Luettu 25.4.2022.

9 A tour of the Dart Language. Verkkoaineisto <https://dart.dev/guides/language/language-tour/>. Luettu 25.4.2022.

10 Flutter - Introduction. Verkkoaineisto <https://www.tutorialspoint.com/flutter/flutter_introduction.htm/>. Luettu 26.5.2022.

11 Layouts in Flutter. Verkkoaineisto <https://docs.flutter.dev/development/ui/layout/>. Luettu 26.5.2022.

12 Bansal, Ujjwal. 2021. An Overview of the generated files and folders in the new flutter project. Verkkoaineisto <https://towardsdev.com/an-overview-of-the-generated-files-and-folders-in-the-new-flutter-project-61c596ca81c6/>. Luettu 26.5.2022.

13 Eschweiler, Sebastian. 2019. Getting Started With Flutter – (2) Project Structure. Verkkoaineisto <https://medium.com/codingthesmartway-com-blog/getting-started-with-flutter-2-project-structure-8066bde05270/>. Luettu 26.5.2022.

14 Networking. Verkkoaineisto. <https://docs.flutter.dev/development/data-and-backend/networking/>. Luettu 26.5.2022.

15 The main function. 2020. Verkkoaineisto <https://flutter-byexample.com/lesson/the-main-function/>. Luettu 26.5.2022.

16 Introduction to widgets. Verkkoaineisto <https://docs.flutter.dev/development/ui/widgets-intro/>. Luettu 26.5.2022.

17 ChangeNotifierProvider. 2020. Verkkoaineisto <https://flutter-byexample.com/lesson/change-notifier-provider/>. Luettu 27.5.2022.

18 Rebuilding widhets with Consumer. 2020. Verkkoaineisto <https://flutter-byexample.com/lesson/rebuilding-widgets-with-consumer/>. Luettu 27.5.2022.

19 Kynttiläkaavion lukeminen. Verkkoaineisto <https://fiksusijoittaminen.fi/kynttilat/>. Luettu 28.5.2022.

20 FutureBuilder<T> class. Verkkoaineisto <https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html/>. Luettu 29.5.2022.

Esimerkki StatefulWidget-komponentin käytöstä

```
import 'package:flutter/material.dart';

class Counter extends StatefulWidget {
  // This class is the configuration for the state.
  // It holds the values (in this case nothing) provided
  // by the parent and used by the build  method of the
  // State. Fields in a Widget subclass are always marked
  // "final".
  const Counter({Key? key}) : super(key: key);

  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _counter = 0;

  void _increment() {

    setState(() {
  // This class is the configuration for the state.
  // It holds the values (in this case nothing) provided
  // by the parent and used by the build  method of the
  // State. Fields in a Widget subclass are always marked
  // "final".
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    // This method is rerun every time setState is called,
    // for instance, as done by the _increment method above.
    // The Flutter framework has been optimized to make
    // rerunning build methods fast, so that you can just
    // rebuild anything that needs updating rather than
    // having to individually changes instances of widgets.
    return Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        ElevatedButton(
          onPressed: _increment,
          child: const Text('Increment'),
        ),
        const SizedBox(width: 16),
        Text('Count: $_counter'),
      ],
    );
  }
}

void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
```

```
          child: Counter(),
        ),
      ),
    ),
  );
}
```

Esimerkkikoodi 1. Esimerkki StatefulWidget komponentin käytöstä. (16)