Matti Närhi

Mobiiliverkkokauppasovelluksen toteuttaminen monialustaisella teknologialla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto -ja viestintätekniikka

Insinöörityö

26.11.2021

Tiivistelmä

Tekijä:	Matti Närhi
Otsikko:	Mobiiliverkkokauppasovelluksen toteuttaminen monialustaisella teknologialla
Sivumäärä:	55 sivua
Aika:	26.11.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto -ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Juha Kämäri

Sovellusten kehittämiseen mobiililaitteille käytetään yleensä laitteiden käyttöjärjestelmien natiiveja teknologioita. Androidilla natiivina teknologiana käytetään yleensä Java-ohjelmointikieltä, kun taas iOS:llä teknologiana on yleensä Objective-C-ohjelmointikieli. Vaihtoehtona sovelluksen kehittämiselle natiivin teknologian avulla toimivat monialustaiset teknologiat, joita ovat muun muassa React Native ja Dart Flutter. React Native on Facebookin vuonna 2015 julkaisema sovelluskehys, joka perustuu Facebookin React-nimiseen JavaScript-kirjastoon. Dart Flutter puolestaan on Googlen vuonna 2018 julkaisema sovelluskehys, jonka taustalla toimii oliopohjainen Dart-ohjelmointikieli. React Nativea ja Dart Flutteria yhdistää se, että yhden lähdekoodin pohjalta saadaan toteutettua sovellus sekä Android- että iOS-alustalle.

Tässä insinöörityössä tutkittiin React Nativen ja Dart Flutterin toimintaperiaatteita sekä verrattiin näiden teknologioiden etuja ja rajoitteita natiiveihin teknologioihin nähden. React Nativen ja Dart Flutterin eduksi natiiveihin teknologioihin nähden osoittautui yhden lähdekoodin käyttö sovelluksen toteuttamiseen Android- ja iOS-alustoille. Tällöin ohjelmakoodin ylläpito on helpompaa, sillä ohjelmakoodista ei tarvitse alustojen vuoksi olla useampaa eri versiota. Lisäksi sovelluksen saa todennäköisesti julkaistua kummallekin alustalle nopeammalla aikataululla. React Nativen ja Dart Flutterin rajoitteena natiiveihin teknologioihin nähden on rajoitettu pääsy alustojen natiiveihin ominaisuuksiin ja palveluihin. Tällöin sovellus ei välttämättä ole yhteensopiva jonkin tietyn alustan natiivin ominaisuuden tai palvelun kanssa. Insinöörityössä verrattiin myös React Nativen ja Dart Flutterin eroavaisuuksia sekä näiden etuja ja rajoitteita. Dart Flutterin eduksi osoittautui parempi suorituskyky Material- ja Cupertino-kirjastojen käytön ansiosta. Dart Flutterin rajoitteena on puolestaan se, että se ei tällä hetkellä tue 3D-grafiikkaa käyttöliittymässä.

Insinöörityön aikana kehitettiin Dart Flutterilla verkkokauppasovellus, jolla pyrittiin havainnollistamaan Dart Flutterin ominaisuuksia sekä tarvittavan alustariippuvaisen määrittelyn määrä. Kaikki sovelluksen ominaisuudet saatiin toteutettua ilman alustariippuvaista koodia. Alustariippuvainen määrittely rajoittui tarvittavien käyttöoikeuksien määrittämiseen kameran ja tiedostojärjestelmän käyttöä varten.

Avainsanat: React Native, Dart Flutter, Android, iOS

Abstract

Author:	Matti Närhi
Title:	Implementing Mobile Online Store Application with Cross-Platform Technology
Number of Pages:	55 pages
Date:	26 November 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Software Engineering
Supervisors:	Juha Kämäri, Senior Lecturer

In general, applications for mobile devices are developed with the native technology of the operating system of the device. The native technology used for Android is usually Java and for iOS it is Objective-C. Cross-platform technologies such as React Native and Dart Flutter are an alternative to native technologies when developing mobile applications. React Native is a framework published by Facebook in 2015 based on Facebook's JavaScript library, React. Dart Flutter is a framework published by Google in 2018 and its underlying programming language is an object-oriented language called Dart. Both React Native and Dart Flutter can implement an application for Android and iOS platforms based on one source code.

In this bachelor's thesis mechanisms of React Native and Dart Flutter and their benefits and limitations are compared to native technologies. The benefit of using React Native and Dart Flutter is the ability to use one source code for both platforms. With less versions of the code, it is easier to maintain it. It is also likely that the application is published faster. Limited access to platform-specific features and services that can cause compatibility issues, on the other hand, is a limitation of React Native and Dart Flutter compared to native technologies. Differences between React Native and Dart Flutter and benefits and limitations between them are also discussed. The benefit of Dart Flutter is better performance by using Material and Cupertino libraries. Dart Flutter does not currently support 3D graphics in the user interface and therefore it is a limitation of this technology.

For demonstrating the features of Dart Flutter and platform-specific code and configuration, a mobile online store application was developed. All the features of the application were implemented without platform-specific code. Platform-specific configuration was limited to defining required rights for using the camera and the file system of the device.

Keywords: React Native, Dart Flutter, Android, iOS

Sisällys

Lyhenteet

AOT: *Ahead-of-time. Ohjelman käännöstapa, jossa ohjelma käännetään korkeamman tason ohjelmointikielestä matalamman tason kielelle, kuten esimerkiksi konekielelle. Tämä vähentää ajonaikaisen työn määrää ja siten parantaa suorityskykyä.*

API: Application Programming Interface eli ohjelmointirajapinta. Mahdollistaa sovelluksen kommunikoinnin muiden sovellusten tai käyttöjärjestelmän kanssa.

DOM: Document Object Model. Tapa kuvata rakenteisen dokumentin, kuten HTML:n rakenne puuna, jonka olioihin voi kohdistaa muutoksia esimerkiksi JavaScriptin avulla.

1 Johdanto

Mobiilisovellusten kehittämisessä natiivien teknologioiden lisäksi sovellusten kehittämiseen käytetään monialustaisia teknologioita. Mobiilisovelluksista puhuttaessa yleensä kohdealustoina ovat Googlen kehittämä Android-käyttöjärjestelmä ja Applen kehittämä iOS-käyttöjärjestelmä. Android-käyttöjärjestelmälle kehitetään sovelluksia tyypillisesti Java-ohjelmointikielen avulla. Tämän lisäksi Androidille voi toteuttaa sovelluksen Kotlin-ohjelmointikielen avulla. Applen iOS-käyttöjärjestelmälle sovellukset toteutetaan Objective-C-ohjelmointikielellä. Edellä mainitut ohjelmointikielet ovat natiiveja teknologioita mobiilisovellusten kehittämisessä.

React Native on yksi yleisimmistä ohjelmointikielistä, jota käytetään teknologiana monialustaisten mobiilisovellusten toteuttamisessa. React Native on Facebookin kehittämä JavaScript-pohjainen ohjelmointikieli, jonka avulla sama lähdekoodi toimii sekä Android- että iOS-käyttöjärjestelmille. Tällöin ei siis tarvitse erikseen kehittää kummallekin alustalle omaa versiota. React Native julkaistiin maaliskuussa vuonna 2015.

Dart Flutter puolestaan on Googlen joulukuussa vuonna 2018 julkaisema monialustainen teknologia. Kyseinen teknologia käyttää ohjelmointikielenään oliopohjaista Dart-ohjelmointikieltä. Dart on syntaksiltaan samankaltainen kuin esimerkiksi Java- tai C#-ohjelmointikielet. Dart Flutterin Flutter-osio käsittää teknologian käyttöliittymäkomponentit. Flutterilla toteutetut käyttöliittymät eivät yleensä eroa natiivin toteutustavan käyttöliittymästä. Vaikka Dart Flutter on teknologiana melko uusi, sen käyttö on yleistynyt nopeasti.

Insinöörityön tavoitteena on tutustua monialustaisten teknologioiden toimintaan mobiilisovellusten kehittämisessä. Insinöörityössä tutustutaan aluksi React Nativen ja Dart Flutterin ominaisuuksiin sekä näiden etuihin ja rajoituksiin natiivien teknologioiden toimintaan verrattuna. Tämän jälkeen tutustutaan tarkemmin

Dart Flutter -teknologian toimintamekanismiin ja sen eroavaisuuksiin React Nativeen nähden. Dart Flutterin ominaisuuksien havainnollistamista varten insinöörityön aikana kehitettiin verkkokauppasovellus. Sovelluksen avulla pyritään tuomaan esiin Dart Flutterin hyödylliset ominaisuudet sekä teknologian haasteet sovelluskehityksessä. Lopuksi insinöörityössä käydään läpi Dart Flutterin antamia mahdollisuuksia sovelluksen jatkokehittämiselle.

2 Monialustaiset teknologiat mobiilisovelluskehityksessä

Mobiilisovelluksia voi kehittää joko natiivisti tai monialustaisesti. Natiivilla mobiilisovelluksen kehittämisellä tarkoitetaan sovelluksen toteuttamista jollekin tietylle alustalle käyttäen kyseiselle alustalle ominaista ohjelmointikieltä. Android-käyttöjärjestelmälle natiivit sovellukset ohjelmoidaan käyttäen joko Java- tai Kotlin-ohjelmointikieltä. IOS-käyttöjärjestelmälle käytettävä ohjelmointikieli on puolestaan Objective-C. Natiivien sovelluksien suorituskyky on yleensä hyvä, sekä niiden luotettavuus on korkealla tasolla. Natiivit sovellukset ovat myös yhteensopivia laitteen muiden toimintojen, kuten kameran tai osoitekirjan kanssa. Yrityksille sovelluksien natiivi toteutustapa voi olla kallista, sillä sovelluksesta joudutaan tekemään kaksi versiota, mikäli sovellus julkaistaan sekä Androidille että iOS:lle.

Monialustainen sovelluskehitys puolestaan tarkoittaa sitä, että sovellusta ei toteuta kohdealustojen natiiveilla ohjelmointikielillä vaan teknologialla, joka on suunniteltu monialustaisiin toteutuksiin. React Native ja Dart Flutter ovat tällaisia teknologioita. Kuten natiivit teknologiat, myös monialustaiset teknologiat ovat yleensä yhteensopivia laitteen muiden toimintojen kanssa. Monialustaisten teknologioiden yhtenä etuna on se, että sama versio käy sekä Androidille että iOS:lle, joten sovellus saadaan todennäköisesti nopeammin julkaisuun. Monialustaisten sovellusten suorituskyky yleensä yltää hyvälle tasolle, mutta joissain tapauksissa monialustainen toteutus saattaa toimia natiivia toteutusta hitaammin. Näiden edellä mainittujen seikkojen lisäksi monialustaisilla teknologioilla on myös muita etuja ja rajoitteita. Seuraavaksi käydään läpi React Nativen

ja Dart Flutterin ominaisuuksia yleisellä tasolla sekä tutustutaan tarkemmin näiden teknologioiden etuihin ja rajoitteisiin. [1.]

2.1 React Native

React Native on mobiilisovellusten kehittämiseen käytetty JavaScript-pohjainen sovelluskehys, joka käyttää Android- ja iOS-alustojen natiiveja komponentteja käyttöliittymän renderöintiin. React Nativen toiminta perustuu Facebookin kehittämään JavaScript-kirjastoon, Reactiin. Erona Reactiin on kohdealusta, joka on web-selaimen sijasta mobiililaitteiden käyttöjärjestelmät. React Nativea oppii siis todennäköisesti melko nopeasti, jos on aikaisempaa kokemusta Reactista. Kuten React, myös React Native käyttää JSX-ohjelmointikieltä, joka on JavaScriptin laajennus. JSX-kieli on JavaScriptin ja XML-kielen yhdistelmä [3]. React Native ei käytä käyttöliittymän piirtämiseen WebView-komponentteja, vaan React Native toteuttaa tämän käyttämällä Android- ja iOS-alustojen natiiveja ohjelmointikieliä. Tästä seuraa se, että sovelluksen käyttämät käyttöliittymäkomponentit ovat kummallakin alustalla natiiveja komponentteja. React Nativen JavaScript-rajapinnoilla on pääsy kummallakin alustalla alustan omaan APIin, jonka ansiosta sovellukset ovat yhteensopivia laitteen muiden toimintojen, kuten kameran ja mikrofonin kanssa. [2, luku 1.]

React Native käyttää Reactin tavoin Virtual DOMia käyttöliittymän renderöintiin. Web-sovelluksissa Virtual DOM toimii kerroksena kehittäjän koodin ja renderöintimekanismin välillä. Selaimen DOMiin muutosten tekeminen suoraan on työläs operaatio, joten muutokset tehdään ensin Virtual DOMiin. Tämän jälkeen Virtual DOM päivittää varsinaisen DOMin tekemällä mahdollisimman vähän muutoksia DOMiin. React Nativen tapauksessa Virtual DOMin renderöinnin kohteena ei ole selain, vaan Objective-C:n API iOS-komponenttien renderöintiä varten tai Javan API Android -komponenttien renderöintiä varten. Koska Virtual DOM pääsee käsiksi natiiveihin käyttöliittymäkomponentteihin, React-komponenttien render-funktiot palauttavat alustan natiivin toteutuksen lähdekoodissa

käytetystä komponentista. Esimerkiksi alustan ollessa iOS lähdekoodissa oleva View-komponentti renderöidään sovellukseen iOS:n UIView-komponenttina.

React Native käyttää renderöintiin HTML-elementtien sijasta alustakohtaisia React-komponentteja. Osalla tyypillisistä HTML-elementeistä on React Nativessa monialustainen toteutus, ja alusta määrittää oikean komponentin, esimerkiksi View on iOS:llä UIView-komponentti. Taulukossa 1 käydään HTML-elementtien vastineet React Nativessa. [2, luku 2.]

Taulukko 1. HTML-elementtien vastineet React Nativessa.

HTML	**React Native**
	<View>
	<Text>
,	<FlatList>
<img>	<Image>

Yllä mainitut komponentit taulukossa 1 ovat siis monialustaisia käyttöliittymäkomponentteja. Muut React Nativen komponentit ovat alustakohtaisia, joten lähdekoodissa tulee käyttää oikeaa versiota komponentista alustasta riippuen. Tyypillisesti alustariippuvaisten komponenttien nimi päättyy alustan nimeen. Esimerkkejä tällaisista komponenteista ovat <DatePickerIOS>, <ToolBarAndroid> ja <TabBarIOS>. Ohjelmakoodin uudelleenkäytettävyyden kannalta on tärkeää, miten alustariippuvaisia komponentteja on käytetty ohjelmakoodissa. Käytetyt komponentit ja ohjelmakoodin logiikka komponentin käyttöä varten tulisi pitää toisistaan erillään, sillä alustariippuvaista komponenttia ei voi käyttää toisella alustalla uudelleen. Ohjelmakoodin logiikkaa voi kuitenkin käyttää uudelleen kummallakin alustalla. Yksi toteutustapa logiikan ja komponenttien erottamiselle

toisistaan on luoda Android- ja iOS-versioille omat tiedostonsa, joissa kummassakin on oman alustansa toteutus samasta komponentista. Tällöin ohjelmakoodissa käytetään sen alustan versiota toteutuksesta, jolla sovellus on käynnissä.

React Native käyttää alustojen natiivien komponenttien ulkoasun muokkaamiseen yksinkertaistettua versiota CSS-teknologiasta. Komponentin tyyli määritetään komponentin yhteydessä style-attribuutin avulla. Koodiesimerkissä 1 havainnollistetaan, miten tyyli määritetään komponentille.

```
const style = {
backgroundColor: 'white',
fontSize: '16px'

};

const txt = (
<Text style={style}>
A styled text
</Text>);
```

Esimerkkikoodi 1. Tyylin määrittely komponentille.

Yllä olevassa esimerkkikoodissa 1 määritetään tyyli tekstikomponentille. Ensin tyyli määritetään JavaScript-oliona, ja tämän jälkeen se annetaan parametrina komponentin style-attribuutille. Lopputuloksena on teksti valkoisella taustavärillä, jonka fonttikoko on 16 pikseliä. React Nativessa komponenttien tyylimäärittelyt ovat siis samassa tiedostossa muun koodin kanssa. [2, luku 2.]

2.2 Dart Flutter

Dart Flutter on Googlen vuonna 2018 julkaisema avoimen lähdekoodin sovelluskehys, jota käytetään mobiilisovellusten kehittämiseen Android- ja iOS-alustoille. Dart Flutter käyttää ohjelmointikielenään Dart-ohjelmointikieltä, joka esitettiin ensimmäisen kerran vuonna 2011. Versio 1.0 Dart-ohjelmointikielestä julkaistiin marraskuussa vuonna 2013. Tällä hetkellä viimeisin vakaa versio Dartista on 2.13.4, joka julkaistiin 28. kesäkuuta 2021. Dart on käännettävä ohjelmointikieli, joten koodia ei voida suorittaa sellaisenaan, vaan kääntäjän tulee muuttaa se ensin konekielelle. Dart on oliopohjainen ja vahvasti tyypitetty kieli.

Tämä tarkoittaa sitä, että esimerkiksi kaikille muuttujille pitää olla määritetty niille annettavan datan tyyppi ohjelmakoodissa. Dartin tukemat datatyypit ovat String, num, int, double, boolean ja object eli oliot. Dart ei suoraan tue taulukoita, vaan niitä käytetään Collection-toteutuksen avulla. Lisäksi Dart tukee operaattoreiden kuormittamista. Dartilla on maine helposti opittavana ohjelmointikielenä, sillä sen syntaksi muistuttaa muiden oliopohjaisten kielten syntaksia. [4; 5.] Koodiesimerkissä 2 havainnollistetaan Dartin syntaksia yksinkertaisen luokan avulla.

```
import 'dart:math' as math

class Point {

final num x, y;

Point(this.x, this.y);

Point.origin()
:x = 0,
 y = 0;

num distanceTo(Point other) {

 var dx = x - other.x;
 var dy = y - other.y;

 return math.sqrt(dx*dx + dy*dy);

}

num get magnitude => math.sqrt(x*x + y*y);

Point operator +(Point other) => Point(x + other.x, y + other.y);

}

void main(){

var p1 = Point(10, 10);
print(p1.magnitude);
var p2 = Point.origin();
var distance = p1.distanceTo(p2);
print(distance);
```

Esimerkkikoodi 2. Syntaksin havainnollistaminen. Koodissa on toteutettu luokka kahden pisteen välisen etäisyyden laskemiseen.

Dart Flutterin Flutter-kieli puolestaan on ohjelmistokehityspaketti, joka sisältää tarvittavat työkalut mobiilisovelluksen käyttöliittymän rakentamiseen. Flutter tarjoaa tällä hetkellä pluginit käyttöä varten Android Studio-, IntelliJ IDEA- ja VS Code -editoreille. Vaihtoehtoisesti Flutteria voi käyttää komentorivipohjaisesti editorilla, joka tukee Dart-ohjelmointikieltä. Flutterin avulla on tähän mennessä kehitetty yli 100 000 sovellusta. Flutter käyttää grafiikan renderöintiin Googlen Skia-renderöintimoottoria, jonka avulla voidaan renderöidä 2D-grafiikkaa. Tällä hetkellä Flutter ei tue 3D-grafiikkaa sovelluksissa, mutta suunnitelmissa on toteuttaa Flutterille optimoitu API 3D-grafiikkaa varten. Flutterin ohjelmistomoottori on toteutettu C- ja C++-ohjelmointikielien avulla. Paremman suorituskyvyn saavuttamiseksi Flutterilla toteutetut sovellukset käännetään AOT-käännöksellä.

Flutter ei käytä käyttöliittymän rakentamiseen Android- ja iOS-alustojen natiiveja käyttöliittymäkomponentteja, vaan Material- ja Cupertino-kirjastojen komponentteja. Cupertino noudattaa iOS:n komponenttien tyyliä. Kirjastot eivät kuitenkaan tästä huolimatta ole alustakohtaisia, vaan kumpaakin kirjastoa voi käyttää sovelluksen käyttöliittymässä alustasta riippumatta. Valmiiden komponenttien lisäksi Flutterilla on mahdollista luoda omia komponentteja yhdistelemällä näitä valmiita komponentteja.

Vaikka Flutter ei käytä alustojen natiiveja käyttöliittymäkomponentteja, on sillä kuitenkin kummallakin alustalla pääsy osaan niiden natiivista APIsta. Niiden avulla myös Dart Flutterilla koodattu sovellus pääsee käsiksi muun muassa laitteen kameraan ja tallennustilaan. Alustakohtaisia ohjelmarajapintoja varten on valmiit paketit pub.dev-sivustolla. Halutun paketin saa lisättyä sovellusta varten suorittamalla editorin komentorivillä komennon: pub get paketin_nimi. [6; 7.]

2.3 Edut natiiveihin teknologioihin nähden

Natiivit teknologiat tarjoavat paljon etuja mobiilisovelluksen kehittämisessä, mutta tietyissä asioissa monialustaiset teknologiat, kuten React Native ja Dart Flutter, toimivat paremmin. React Nativella sekä Dart Flutterilla on yhteisenä

etuna natiiveihin teknologioihin nähden se, että yhden lähdekoodin avulla saadaan toteutettua sovellus sekä Android että iOS-alustoille. Tällä tavalla sovelluksen kehittäminen on yleensä halvempaa sekä lähdekoodia on helpompi ylläpitää, sillä kahden version sijaan versioita on vain yksi. Tämän edun lisäksi React Nativella ja Dart Flutterilla on myös muita etuja.

Yksi React Nativen eduista on se, että se on avoimen lähdekoodin sovelluskehys ja se perustuu Facebookin JavaScript-kirjastoon, Reactiin. Avoimen lähdekoodin ansiosta React Nativella voi vapaasti kehittää sovelluksia ja jakaa koodia. Tämän vuoksi React Nativen kehittäjäyhteisö kasvaa ja kehittyy aktiivisesti, jonka ansiosta ongelmatilanteisiin sovelluksen kehittämisessä on yleensä löytyvissä nopeasti ratkaisu. Lisäksi React Nativea on helppo oppia, jos on aikaisempaa kokemusta web-sovelluskehityksestä React-kirjaston avulla. React Nativen etuna on myös se, että sovelluksen käyttöliittymäkomponentit ovat Android -ja iOS-alustojen natiiveja komponentteja. Tämän vuoksi React Nativella kehitetty sovellus näyttää samanlaiselta kuin natiivilla teknologialla kehitettynä. React Native tukee Hot Reload -ominaisuutta, joka helpottaa ohjelmakoodin debuggaamista. Ominaisuuden avulla ohjelmakoodia ei aina tarvitse koota uudestaan koodin muutosten testaamista varten, vaan pelkästään muutokset ladataan sovellukseen ilman sovelluksen uudelleenkäynnistämistä. [8.]

React Nativen tavoin Dart Flutterin etuna on avoin lähdekoodi. Dart Flutter on uudempi teknologia kuin React Native, joten kehittäjäyhteisö ei ole yhtä laaja kuin React Nativella. Tästä huolimatta sovelluksia on kehitetty Dart Flutterilla jo yli 100 000 kappaletta. Dart Flutterin etuna on myös Dart-ohjelmointikielen helppous. Dartin syntaksi muistuttaa Java -ja C-ohjelmointikielien syntaksia, ja Dart Flutter ei vaadi aikaisempaa kokemusta muista teknologioista. React Nativen tavoin Dart Flutter käyttää myös Hot Reload -ominaisuutta. Dart Flutter ei tarvitse siltakomponenttia alustojen natiivien moduulien kanssa kommunikointiin, joten käyttöliittymän kuvataajuus on 60-120 kuvaa sekunnissa. Näin ollen Dart Flutterin suorituskyky on samaa luokkaa kuin natiivien teknologioiden. Lisäksi Dart Flutter mahdollistaa omien komponenttien koodaamisen, joten se antaa hyvät mahdollisuudet olemassa olevien komponenttien muokkaamiselle. [9.]

2.4 Monialustaisten teknologioiden rajoitteet

Natiivien teknologioiden avulla kehitetyllä sovelluksella on todennäköisesti hyvä suorituskyky. Ne pystyvät hyödyntämään alustan kaikkia ominaisuuksia. React Nativen ja Dart Flutterin yhtenä rajoitteena natiiveihin teknologioihin nähden on rajallinen pääsy Andoid- ja iOS-alustojen APIin. Mikäli jotain tiettyä alustan ominaisuutta tai palvelua ei ole tuettu, ainoa vaihtoehto on yrittää toteuttaa yhteensopiva moduuli natiivilla teknologialla. Tämä puolestaan vie aikaa eikä lopputulos välttämättä vastaa suorituskyvyltään natiivia teknologiaa. Tämän rajoitteen lisäksi React Nativella ja Dart Flutterilla on myös muita rajoitteita.

React Nativen yksi rajoitteista on se, että sen viimeisin versio on tällä hetkellä 0.65.1 eli versiota 1.0 ei ole vielä edes julkaistu. Tämä tarkoittaa sitä, että React Native ei välttämättä käytä kaikkea potentiaalia Android- ja iOS-alustoista, joka tällä hetkellä näkyy rajallisena pääsynä alustojen natiiviin APIin ja kolmannen osapuolen pakettien puutteena. React Nativella on myös käyttöliittymään liittyviä rajoitteita. Yksi rajoitteista on rajallinen ulkoasun muokkausmahdollisuus, sillä yhden lähdekoodin vuoksi käyttöliittymän ulkoasu on samanlainen kummallakin alustalla. Lisäksi monimutkaiset käyttöliittymäkomponentit ja animaatiot eivät ole React Nativen vahva puoli. Kaikkia käyttöliittymäkomponenteista ei pysty koodaamaan JavaScriptin avulla, joten kehittäjällä olisi hyvä olla kokemusta natiiveista teknologioista. Lisäksi laitteeseen liittyvien ongelmien korjaamiseen saattaa kulua enemmän aikaa, koska JavaScript ei ole kummankaan alustan natiivi ohjelmointikieli. React Native kommunikoi sovelluksen alustan kanssa siltakomponentin välityksellä, joka voi aiheuttaa käyttöliittymässä suorituskykyongelmia tietyissä tilanteissa. Esimerkiksi pitkissä listanäkymissä ongelmana on se, että käyttöliittymän React-komponentit joutuvat aina välittämään viestin alustalle, kun käyttäjä on selannut listaa. Tästä seuraa se, että alustalle joudutaan lähettämään useita viestejä, ja jokaisen viestin käsittely hidastaa käyttöliittymän toimintaa. React Nativen debuggaustyökaluissa saattaa myös esiintyä bugeja. [8.] Taulukossa 2 ovat React Nativen edut ja rajoitteet koottuna.

Taulukko 2. React Nativen edut ja rajoitteet.

Edut	**Rajoitteet**
Kustannustehokas ratkaisu	React Native vielä versiossa 0.65.1
Ei eroa Android -ja iOS-versioiden välillä	Rajoitetut ulkoasun muokkausmahdollisuudet alustakohtaisesti
Natiivit käyttöliittymäkomponentit ja alustan natiivi renderöinti	Rajallinen pääsy natiiviin APIin ja kolmannen osapuolen pakettien puute
Avoin lähdekoodi ja perustuu Reactiin	Laiteperäisten ongelmien ratkominen voi viedä aikaa
Hot Reload -ominaisuus nopeuttaa debuggausta	Debuggaustyökaluissa esiintyy bugeja
Kehittyvä ja kasvava kehittäjäyhteisö	Kaikkia komponentteja ei voi toteuttaa JavaScriptillä. Vaatii iOS -ja Android-osaamista
Helppo oppia, jos kokemusta Reactista	JavaScriptin yksisäikeisessä ympäristössä voi ilmetä suorituskykyongelmia

Dart Flutterin rajoitteena on puolestaan se, että se ei käytä alustojen natiiveja käyttöliittymäkomponentteja, vaan Material- ja Cupertino-kirjastojen komponentteja. Tästä johtuen käyttöliittymä saattaa poiketa iOS:llä natiiveista käyttöliittymäkomponenteista, jos sovelluksen käyttöliittymän toteuttaa käyttäen Material-kirjastoa. Lisäksi Dart Flutter tukee tällä hetkellä vain 2D-grafiikkaa, joka puolestaan rajoittaa sovelluksen kehittämistä. Dart Flutter on uusi teknologia, joten sille ei ole paljoa kolmannen osapuolen paketteja. React Nativen tavoin laiteperäisiä ongelmia voi olla työlästä ratkoa, sillä Dart Flutter ei käytä alustojen natiiveja ohjelmointikieliä. Flutterin ohjelmistomoottorin vuoksi sovellukset vievät noin 40 % enemmän tallennustilaa kuin natiivi sovellus. [9.]

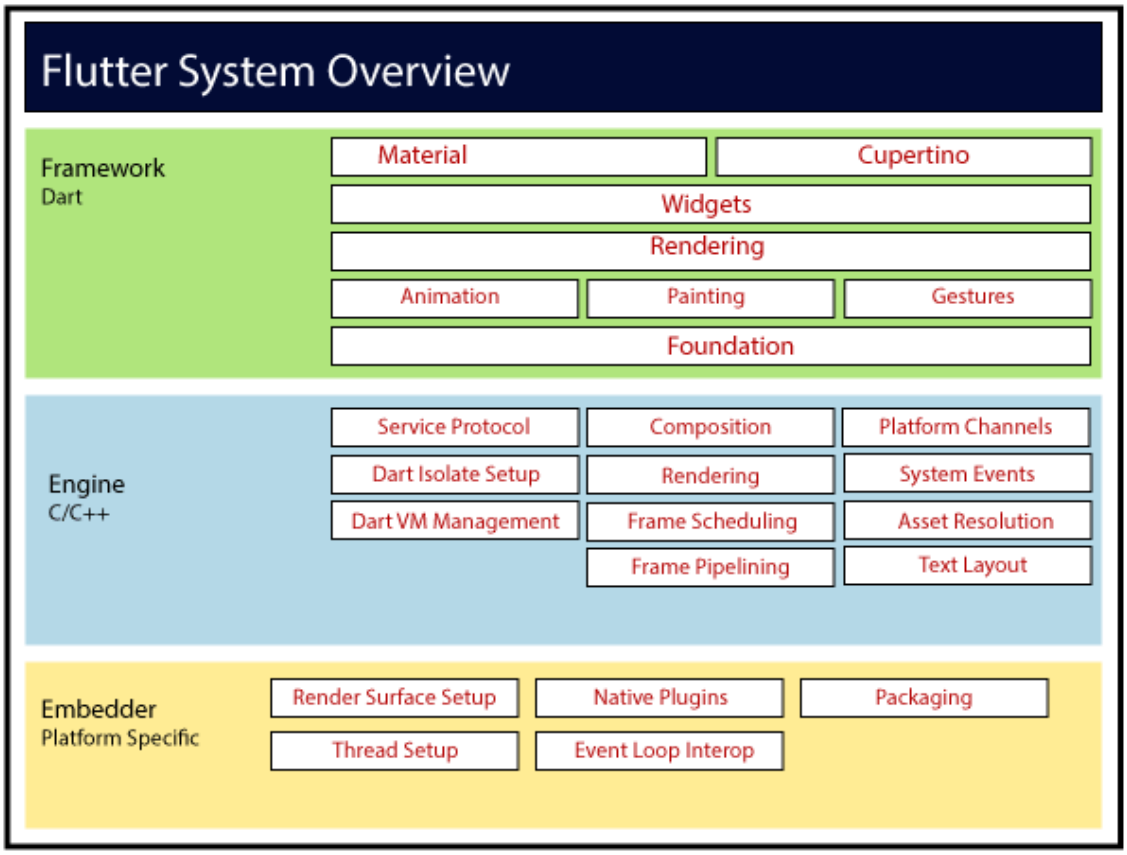
3 Sovelluskehitys yleisesti Dart Flutterin avulla

3.1 Teknologian toimintaperiaate

Dart Flutterin arkkitehtuuri koostuu pääosin neljästä osa-alueesta.

- Flutterin ohjelmistomoottorista
- Foundation-kirjastosta
- Widget-käyttöliittymäkomponenteista
- suunnittelukohtaisista widget-komponenteista.

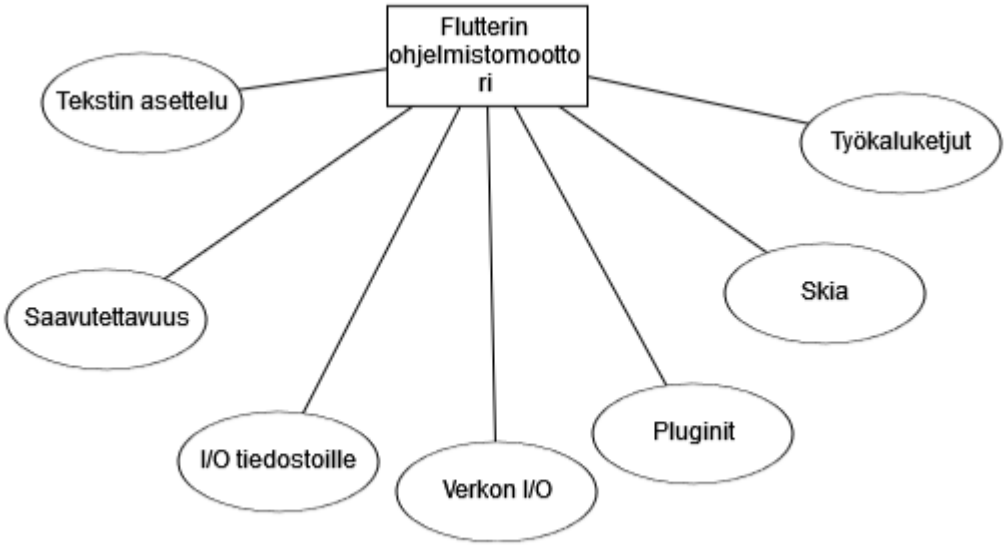
Tämän lisäksi Dart Flutter voidaan jakaa kolmeen eri kerrokseen: framework-, engine- ja embedder-kerrokseen. Jokainen kerros koostuu itsenäisistä kirjastoista, jotka kuitenkin ovat riippuvaisia tästä kerroksesta. Millään näistä kerroksista ei ole etuoikeutettua pääsyä alempaan kerrokseen. Kaikki framework-kerroksen kirjastot ovat tarvittaessa vaihdettavissa. Kuvassa 1 havainnollistetaan Flutterin kerrosarkkitehtuurin rakenne. [10.]



Kuva 1. Dart Flutterin kerrokset. [5.]

Embedder-kerros sisältää muun muassa sovelluksen paketoinnin taustalla olevalle käyttöjärjestelmälle. Dart Flutter toteuttaa sovelluksen paketoinnin samalla tavalla kuin natiivit teknologiat. Embedder-kerros on alustakohtainen, ja se toimii sisäänpääsynä alustaan. Embedder kommunikoi alustan käyttöjärjestelmän kanssa päästäkseen käsiksi alustan palveluihin, kuten renderöintipinnoille, saavutettavuuteen sekä syötteeseen. Lisäksi embedder hallitsee viestitapahtumasilmukkaa. Toisin sanoen embedder on natiivi sovellus, joka toimii isäntänä Dart Flutterilla toteutetulle sisällölle. IOS-alustalle Flutter ladataan embedderiin UIViewController- tai NSViewController-komponenttina. Tämän jälkeen embedder alustaa Flutterin ohjelmistomoottorin, joka toimii isäntänä Dart VM -virtuaalikoneelle sekä Flutterin runtimelle. Flutterin ohjelmistomoottori yhdistetään FlutterViewController-komponenttiin, jonka avulla ohjelmistomoottori välittää syötetapahtumat Flutterille sekä renderöi kuvat käyttäen Metal- tai OpenGL-APIa. Androidilla Flutter ladataan embedderiin aktiviteettina. Käyttöliittymän näkymää

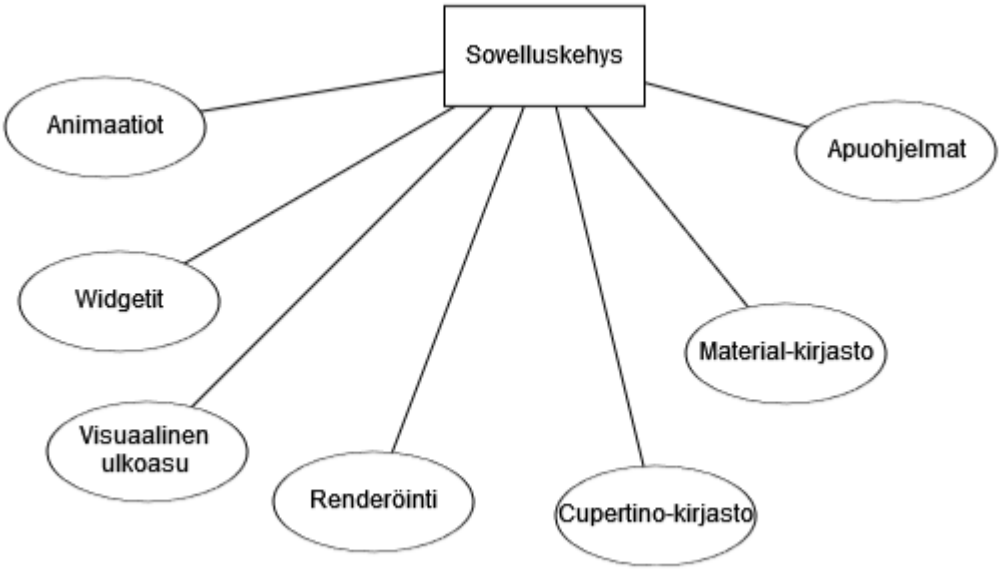
hallitsee FlutterView-komponentti, joka renderöi Flutterin sisällön ikkunanäkymänä tai tekstuurina sisällöstä riippuen. Embedder on toteutettu kullekin alustalle niille ominaisella ohjelmointikielellä. Androidille embedder on toteutettu Javalla ja C++-kielellä. IOS:lle toteutus on puolestaan tehty Objective-C / Objective-C++-kielellä. Embedderin avulla Dart Flutterilla toteutettu ohjelmakoodi voidaan tarvittaessa myös integroida moduulina olemassa olevaan sovellukseen.



Kuva 2. Ohjelmistomoottorikerroksen sisältö pääpiirteittäin.

Embedder-kerroksen jälkeen tulee Dart Flutterin arkkitehtuurin keskimmäinen kerros, joka käsittää Flutterin ohjelmistomoottorin. Ohjelmistomoottori on pääasiassa toteutettu C++-ohjelmointikielellä, ja se tukee tarvittavia primitiivisiä datatyyppejä, jotta se olisi yhteensopiva kaikkien Dart Flutterilla toteutettujen sovellusten kanssa. Ohjelmistomoottori välittää matalan tason toteutuksen Flutterin APIsta, johon kuuluu muun muassa grafiikka Skia-kirjastoa käyttäen, tekstin asettelut, tiedostojen sekä verkon I/O, saavutettavuuden tuki, pluginien arkkitehtuuri sekä Dartin ajonaikainen ja käännöstyökaluketju. Ohjelmistomoottori on yhteydessä Flutterin sovelluskehykseen eli arkkitehtuurin ylimpään kerrokseen Dartin ui-kirjaston välityksellä, joka puolestaan pakkaa taustalla olevan C++-koodin Dart-luokiksi. [10.]

Flutterin arkkitehtuurin ylin kerros on framework eli sovelluskehys. Sovelluskehys on se kerros, jonka kautta Flutterin ominaisuuksiin pääsee käsiksi. Sovelluskehyskerros sisältää sarjan alusta-, asettelu- ja foundation-kirjastoja, jotka koostuvat sarjasta kerroksia. Sovelluskehyksen alimpana kerroksena on Foundation-kirjasto, joka sisältää matalan tason apuohjelmaluokkia- ja funktioita, joita sovelluskehyksen muut kirjastot käyttävät. Seuraavassa kerroksessa on puolestaan animation-, painting- ja gestures-kirjastot. Nämä tarjoavat yleisesti käytettyjä abstraktioita foundation-kirjastosta. Tämän kerroksen jälkeen tulee renderöintikerros, joka tarjoaa abstraktion käyttöliittymän ulkoasua varten. Tämän kerroksen avulla voi rakentaa puurakenteen, joka koostuu renderöitävistä olioista. Olioihin voi kohdistaa puussa muutoksia dynaamisesti, ja puu päivittää käyttöliittymän ulkoasun automaattisesti muutosten mukaisesti. Renderöintikerroksen jälkeen tulee widgets-kerros, joka on koosteabstraktio. Jokaisella renderöintikerroksen oliolla on oma luokkansa widgets-kerroksessa. Widgets-kerros antaa myös mahdollisuuden määritellä yhdistelmiä näistä luokista, joita voi käyttää koodissa uudelleen. Widgets-kerros myös toteuttaa reaktiivista ohjelmointimallia. Widgets-kerroksen jälkeen tulee ylin kerros, joka koostuu Material -ja Cupertino-kirjastoista. Nämä kirjastot tarjoavat kattavan sarjan hallintatyökaluja, jotka käyttävät widgets-kerroksen koosteen primitiivejä toteuttaakseen Material - ja iOS-suunnittelukieliä. Kaiken kaikkiaan Flutterin sovelluskehys ei ole kovin laaja kooltaan. Monet korkeamman tason ominaisuuksista, kuten alustakohtaiset pluginit kameraa ja webview-komponentteja varten, on toteutettu pakkauksina. Lisäksi pakkauksina tulevat Dartin ja Flutterin ydinkirjastot, jotka sisältävät alustariippumattomia ominaisuuksia, kuten merkit, http:n sekä animaatiot. Jotkin sovelluskehyksen pakkauksista ovat osa isompia järjestelmiä, ja pakkausten mukana tulee sellaisia palveluita kuin esimerkiksi Applen autentikointipalvelu. [10.]

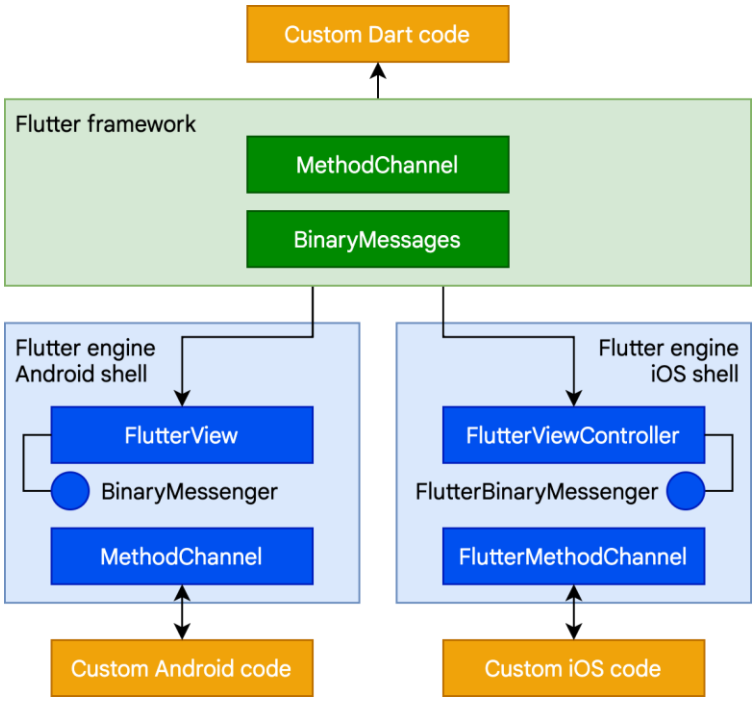


Kuva 3. Sovelluskehyskerroksen sisältö pääpiirteittäin.

Dart Flutterin sovelluskehys on reaktiivinen ja pseudo-deklaratiivinen, joka tarkoittaa sitä, että sovelluksen kehittäjä välittää koodissa kartoituksen sovelluksen tilasta käyttöliittymän tilalle. Tämän jälkeen sovelluskehys huolehtii käyttöliittymän päivityksestä sovelluksen ajon aikana sovelluksen tilan muuttuessa. Flutterin toimintamalli ottaa siis vaikutteita Facebookin React-kirjaston toimintatavasta, joka poikkeaa perinteisistä sovelluksen suunnitteluperiaatteista. Perinteisesti sovelluskehyksissä käyttöliittymän alkutila on kuvattu kerran ja tämän jälkeen tilan muutokset käsitellään erikseen ajonaikana asiakaskoodin kautta, kun sovelluksessa tehdään jotain. Tällaisen lähestymistavan ongelmana on se, että sovelluksen monimutkaistuessa voi olla vaikeampaa hahmottaa, miten muutos tilassa vaikuttaa käyttöliittymän toimintaan. Sovelluksen käyttäjän vuorovaikuttaessa käyttöliittymän kanssa tehdyt muutokset käyttöliittymään tulee näkyä kaikkialla. Mikäli muutoksia ei käsitellä ohjelmakoodissa asianmukaisesti, pienikin muutos käyttöliittymässä voi aiheuttaa sen, että jotkin ohjelmakoodin haarat eivät enää toimi oikein. Tähän ongelmaan ratkaisuna on esimerkiksi MVC-sovellusarkkitehtuuri, jossa datan muutokset viedään model-komponentteihin controller-komponentin kautta, ja model-komponentit puolestaan vievät uuden tilan

view-komponenttiin eli käyttöliittymään controller-komponentin kautta. Tässä lähestymistavassa on kuitenkin haasteena se, että käyttöliittymäelementtien luominen ja päivittäminen ovat kaksi erillistä vaihetta, joten niiden pitäminen synkronoituneena keskenään voi olla haasteellista. Dart Flutter tuo tähän ongelmaan reaktiivisena sovelluskehyksenä ratkaisun, jossa käyttöliittymä erotetaan sen taustalla olevasta tilasta. Sovellusta varten luodaan vain käyttöliittymän kuvaus ja sen avulla luodaan käyttöliittymä sekä kohdistetaan siihen muutoksia. Flutter käyttää widget-komponentteja olioista koostuvan puurakenteen määrittämiseen. Widget-komponentit ovat muuttumattomia luokkia. Widget-komponentit puolestaan hallitsevat erillistä puurakennetta, joka koostuu asetteluun liittyvistä olioista. Tämä puurakenne puolestaan hallitsee koosteolioista koostuvaa puurakennetta. Flutter siis pohjimmiltaan sisältää sarjan mekanismeja, joiden avulla voi tarkastella puutietorakenteisiin tehtyjä muutoksia, muuntaa olioista koostuvia puurakenteita matalamman tason puurakenteiksi sekä lisää muutoksia näiden puiden välille. Flutterin widget-komponentit määrittelevät käyttöliittymän käyttämällä build-metodia, joka muuntaa sovelluksen tilan käyttöliittymään. Build-metodin suoritusaika on nopea, ja tämän vuoksi sovelluskehys antaa vapaasti suorittaa kyseisen metodin aina tarpeen vaatiessa, jopa kerran per renderöity kuva. [10.]

Dart Flutter tarjoaa valikoiman yhteensopivuusmekanismeja Android- ja iOS-alustojen ohjelmakoodeille sekä ohjelmointirajapinnoille, jotka on toteutettu alustojen natiiveilla ohjelmointikielillä. Dart Flutter mahdollistaa kustomoidun ohjelmakoodin kutsumisen alustakanavan kautta. Alustakanava on mekanismi, joka mahdollistaa Dartilla kirjoitetun ohjelmakoodin ja isäntäsovelluksen alustakohtaisen koodin kommunikoinnin keskenään. Kapseloimalla kanavan nimi ja koodekki saadaan muodostettua kummallekin alustalle yhteinen kanava, jonka avulla voi lähettää ja vastaanottaa viestejä Dart-ohjelmakoodin ja alustakohtaisen komponentin välillä. Dartin puolelta tuleva data ensin serialisoidaan Dartin datatyypistä, kuten Map-rakenteesta, standardoituun formaattiin. Tämän jälkeen data deserialisoidaan vastaavaan muotoon kullekin alustalle, esimerkiksi Kotlinilla HashMap-rakenteeksi ja Swiftillä Dictionary-rakenteeksi. [10.] Kuvassa 4 havainnollistetaan kaavion avulla alustakanavan toimintaa.



Kuva 4. Alustakanavan toimintaperiaate. [10.] Alustakanavana toimii MethodChannel-luokka, joka on yhteydessä alustasta riippuen joko alustan FlutterView- tai FlutterViewController-komponenttiin.

Koodiesimerkissä 3 esitetään alustakanavan toimintaperiaate ohjelmakoodissa. Tässä koodiesimerkissä Dart-koodi kutsuu tapahtumakäsittelijää Android-alustan Kotlin-koodin puolella.

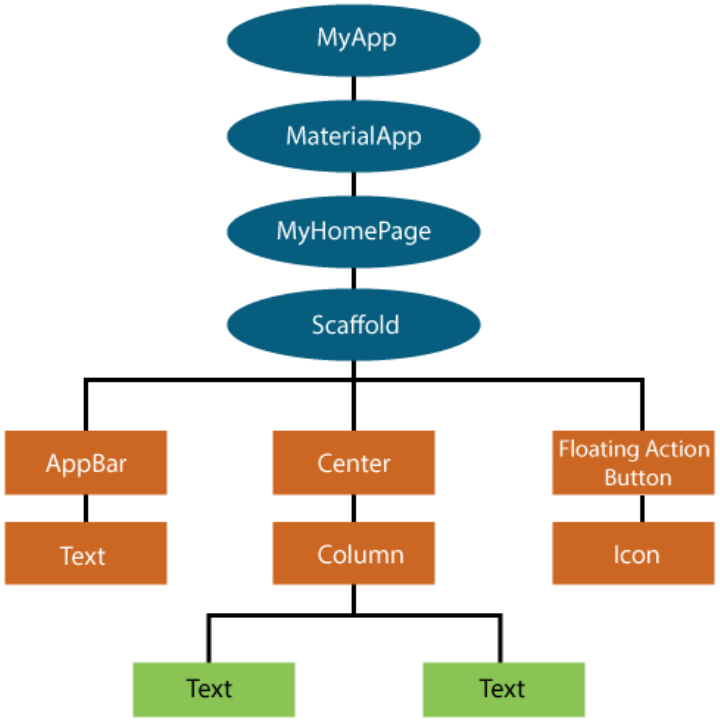
```
//Dart-koodi
const channel = MethodChannel("foo");
final String text = await channel.invokeMethod("bar", "world");
print(text);
//Android(Kotlin)
val channel = MethodChannel(flutterView, "foo)
channel.setMethodCallHandler { call, result ->
     when(call.method) {
     "bar" -> result.success("Hello, ${call.arguments}")
     else -> result.notImplemented()
```

Esimerkkikoodi 3. Kotlin-koodin metodin kutsu Dart-koodissa.

3.2 Käyttöliittymäkomponentit

3.2.1 Widgetit

Kuten luvussa 3.1 on mainittu, widgetit ovat osa Dart Flutterin arkkitehtuuria. Kuvassa 5 on havainnollistettu, minkälaisia widgetejä voi sovelluksessa esimerkiksi olla.



Kuva 5. Kuvassa olevat elementit ovat kaikki widgetejä.

Dart Flutter käyttää käyttöliittymän rakentamiseen widget-komponentteja. Widget-komponentteja käytetään koostamisen yksikkönä, ja jokainen widget on muuttumaton käyttöliittymän määrittelyn osa. Widget-komponenttien välillä on koostamiseen perustuva hierarkia. Jokaisen widgetin sisään voi laittaa toisen widgetin, ja näin ollen parent-widgetin lapsikomponentti voi vastaanottaa parent-widgetin kontekstiolion. Tämäntyyppistä rakennetta käyttöliittymän koostamisessa noudatetaan aina käyttöliittymän juurielementtiin asti, joka pitää sisällään koko Flutter-sovelluksen. Toisin sanoen Flutterin näkökulmasta katsoen myös itse sovellus on widget-komponentti. Sovelluksen juurielementtinä on

yleensä joko MaterialApp -tai CupertinoApp-widget riippuen siitä, käytetäänkö koodissa Material- vai Cupertino-kirjaston käyttöliittymäkomponentteja. [10.] Koodiesimerkki 4 havainnollistaa käyttöliittymän rakentamista widget-komponenttien avulla.

```
@override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('My Home Page'),
        ),
        body: Center(
          child: Builder(
            builder: (BuildContext context) {
              return Column(
                children: [
                  Text('Hello World'),
                  SizedBox(height: 20),
                  ElevatedButton(
                    onPressed: () {
                      print('Click!');
                    },
                    child: Text('A button'),
```

Esimerkkikoodi 4. Käyttöliittymän rakentaminen widget-komponenttien avulla.

Dart Flutterin sovelluskehys huolehtii käyttöliittymän päivityksestä. Kun sovelluksen käyttöliittymässä tapahtuu muutoksia, sovelluskehys korvaa vanhan widget-komponentin uudella widget-komponentilla. Vanhaa sekä uutta widget-komponenttia vertaillaan keskenään ja muutokset käyttöliittymään toteutetaan mahdollisimman tehokkaasti. Dart Flutter ei käytä käyttöliittymissä alustojen omia käyttöliittymän säätimiä, vaan jokaiselle käyttöliittymän säätimelle on oma versionsa Dart-kielellä toteutettuna. Tällä tavoin Dart Flutter tarjoaa kattavat mahdollisuudet toteuttaa variantteja käyttöliittymän eri säätimistä. Alustojen omat komponenttien laajennusmahdollisuudet eivät rajoita varianttien toteutusta. Lisäksi sovelluksen suorituskyky pysyy hyvänä, koska sovelluskehys pystyy koostamaan kerralla kokonaisen näkymän ilman monimutkaista viestiketjua Dart Flutterin ohjelmakoodin ja alustakohtaisen koodin välillä. Tämän lisäksi Dart Flutterin omat toteutukset käyttöliittymän säätimistä erottaa sovelluksen käyttäy-

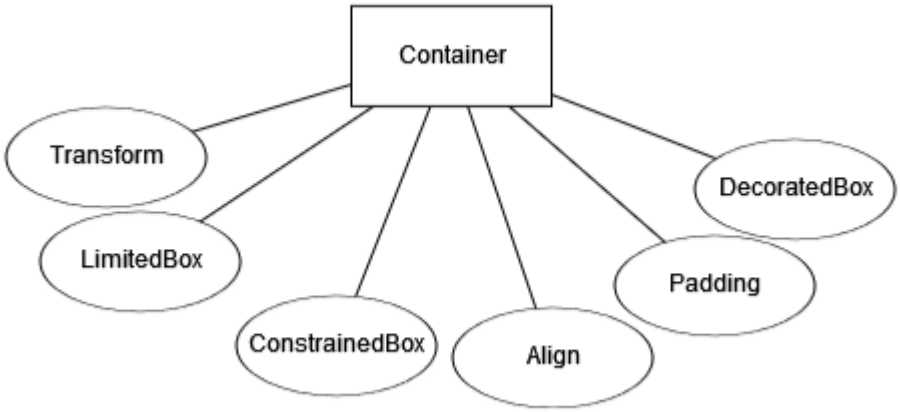
tymisen alustan riippuvaisuuksista. Tämä tarkoittaa sitä, että sovelluksen käyttöliittymän tuntuma pysyy samanlaisena, vaikka käyttöjärjestelmät muuttaisivat omien käyttöliittymäsäätimien toteutusta. [10.]

3.2.2 Widgetien rakenne

Dart Flutterin widget-komponentit yleensä koostuvat pienemmistä yhteen tarkoitukseen suunnatuista widgeteistä, joita yhdistelemällä pyritään mahdollisimman tehokkaasti toteuttaa erilaisia ominaisuuksia sovelluksen käyttöliittymään. Tällä tavoin käyttöliittymän suunnittelukonseptien määrä pidetään pienenä, mutta toteutettavien ominaisuuksien määrä pidetään puolestaan kattavana. Flutterin sovelluskehyksen widgets-kerroksessa widget-komponenentit ovat Flutterin käyttämä suunnittelukonsepti eri ominaisuuksien toteuttamiseen. Toisin sanoen widget-komponentteja käytetään sovelluksessa grafiikan piirtämiseen näytölle, asetteluun, vuorovaikutustoimintoihin, sovelluksen tilan hallintaan, teemoihin, animaatioihin sekä sovelluksen sisäiseen navigointiin. Animaatiokerroksessa puolestaan käytetään Animation- ja Tween-komponentteja suunnittelukonsepteina. Renderöintikerroksessa sen sijaan käytetään RenderObject-komponentteja asettelun, osumatestauksen sekä saavutettavuuden kuvaukseen. [10.]

Widget-komponenttien välinen hierarkia on tarkoituksella matala ja laaja. Keskittymällä pienikokoisiin widget-komponentteihin, joilla on yksi tarkoitus, saadaan maksimoitua mahdollisten widget-yhdistelmien määrä. Jopa perustavanlaatuiset ominaisuudet käyttöliittymän asettelussa, kuten padding ja alignment, on toteutettu omina komponentteinaan. Esimerkiksi jos tekstin haluaa asettaa keskelle ruutua, se täytyy asettaa Center-nimisen widgetin sisälle. Tästä voidaan havaita, että kaikilla widget-komponenteilla ei ole visuaalista toteutusta käyttöliittymässä, vaan tällaisten widgetien tehtävänä on muokata muiden komponenttien asettelua ja ulkoasua. Esimerkkejä tällaisista komponenteista ovat Padding-, Align-, Row- ja Column-komponentit. Yksi yleisimmistä asetteluun käytetyistä widgeteistä on Container-komponentti. Container hyödyntää muiden komponenttien tapaan rakenteessaan koostamista, sillä se koostuu pienemmistä Limi-

tedBox-, ConstrainedBox-, Align-, Padding-, DecoratedBox- ja Transform-komponenteista. Koska Container koostuu useammasta pienemmästä komponentista, voi Container-komponentista tarvittaessa tehdä oman muokatun version esimerkiksi lisäämällä jonkin uuden asetteluun vaikuttavan komponentin omaan toteutukseen Containerista. Toisin sanoen widgetien koostaminen antaa kehittäjälle paljon erilaisia muokkausmahdollisuuksia käyttöliittymään.



Kuva 6.Container-komponentin rakenne.

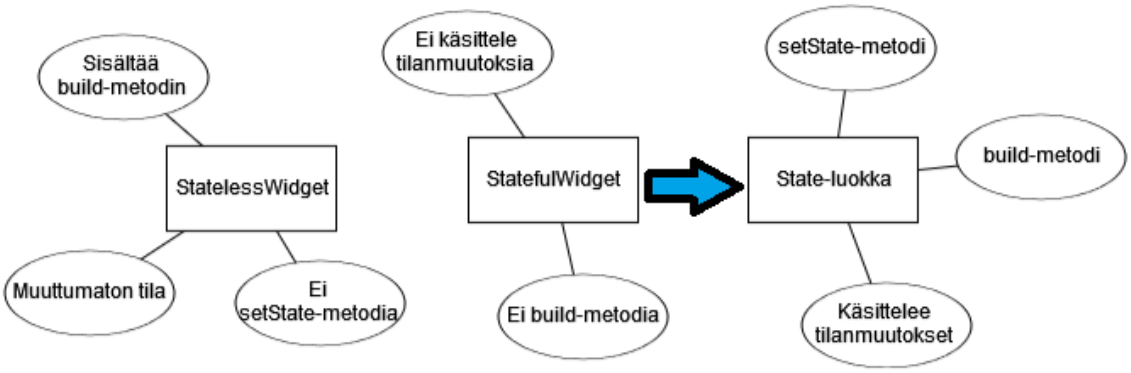
3.2.3 Build-metodi

Widget-komponentin visuaalinen ulkoasu määritellään ylikirjoittamalla build-metodi, jonka paluuarvona on elementtipuu. Kyseinen puurakenne on konkreettinen kuvaus widgetin käyttöliittymästä. Esimerkiksi build-metodi voi palauttaa näkymän, joka koostuu tekstistä sekä napeista. Build-metodi toimii siten, että sovelluskehys rekursiivisesti kutsuu jokaisen build-metodissa olevan widgetin omaa build-metodia, kunnes koko puurakenne on kuvattu renderöitävillä olioilla. Tämän jälkeen sovelluskehys vielä kasaa kaikki renderöitävät oliot renderöitäväksi puurakenteeksi. Build-metodi palauttaa aina kutsuttaessa uuden widgeteistä koostuvan puurakenteen. Näin ollen metodin paluuarvo ei riipu siitä, minkä paluuarvon metodi sai edellisellä kutsukerralla. Sovelluskehys päättelee renderöitävistä olioista koostuvan puurakenteen perusteella, minkä widget-komponenttien build-metodia tulee kutsua. Sovelluksen jokaisessa renderöidyssä

kuvassa Flutterin tarvitsee luoda vain ne käyttöliittymän osat uudelleen, joissa on tila muuttunut. Tämä tapahtuu kutsumalla tilaa muuttaneiden widgetien build-metodia. Flutter tekee tarvittavat laskutoimenpiteet muutosten laskemiseen asynkronisesti build-metodin ulkopuolella ja laskennan tulos liitetään tilaan, jota puolestaan build-metodi voi käyttää hyödykseen käyttöliittymän rakentamisessa. Käyttämällä automaattista muutosten laskemista build-metodin ulkopuolella itse metodin suoritusaika on nopea, ja siten Flutterin avulla pystyy kehittämään suorituskyvyltään hyviä sovelluksia. Lisäksi build-metodi yksinkertaistaa koodia, sillä se keskittyy pelkästään widget-komponentteihin eikä käyttöliittymän tilan manuaalisen hallintaan, joka voi olla haastavaa. [10.]

3.2.4 StatelessWidget ja StatefulWidget

Dart Flutter jakaa widget-komponentit kahteen eri luokkaan: Stateless- ja Stateful-widgeteihin. Kuvassa 7 on havainnollistettu näiden väliset eroavaisuudet.



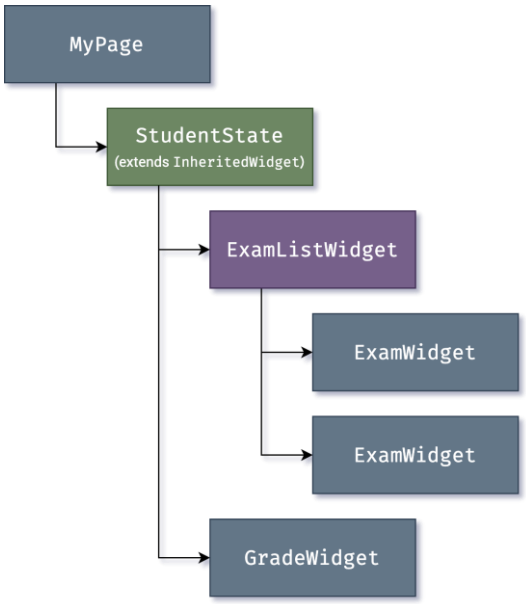
Kuva 7. Stateless- ja StatefulWidgetin eroavaisuudet.

Widget-komponentti perii StatelessWidget-luokan, mikäli kyseisen widgetin tila ei muutu eli se ei sisällä ominaisuuksia, jotka voisivat muuttua esimerkiksi käyttäjän vuorovaikutuksen toimesta. Widget puolestaan perii StatefulWidget-luo-

kan, kun widget sisältää muuttuvia ominaisuuksia. Esimerkki tällaisesta tilanteesta on laskuri, jonka arvo kasvaa käyttäjän painaessa nappia. Koska itse widget on muuttumaton, muuttunut tila talletetaan erilliseen luokkaan, joka perii State-luokan. StatefulWidget ei sisällä build-metodia, vaan se on tämän tilaolion sisällä, ja tämän kautta widget rakentaa käyttöliittymänsä. Tilaolion tilaa muuttaessa ohjelmakoodissa tulee kutsua setState-metodia, jonka avulla sovelluskehys päivittää käyttöliittymän kutsumalla tilaolion build-metodia. Erillisten tila- ja widget-olioiden avulla muut widgetit pystyvät käsittelemään Stateless- ja Stateful-widgetejä samalla tavalla ilman vaaraa tilan menettämisestä. Parent-komponentit voivat aina luoda uuden instanssin lapsikomponentistaan menettämättä lapsikomponentin tilaa. Sovelluskehys automaattisesti huolehtii olemassa olevan tilan uudelleenkäyttämisestä uuteen lapsikomponenttiin, jos se on tarpeellista.

3.2.5 InheritedWidget

Widget-komponentin datan pystyy alustamaan konstruktorin avulla, jonka avulla build-metodi pystyy alustamaan myös kyseisen widgetin lapsiwidgetit. Sovelluksen monimutkaistuessa myös widgeteistä koostuva puurakenne syvenee, jolloin tiedon välittäminen widgetien ja lapsiwidgetien välillä käy työlääksi. Tähän haasteeseen ratkaisuna on InheritedWidget-luokka, jonka avulla tiedon välittäminen on helppoa käyttämällä yhteistä kantaluokkaa. InheritedWidgetiä käytetään niin, että widget-puurakenteeseen lisätään kantaluokka, joka perii InheritedWidget-luokan. Kuvassa 8 havainnollistetaan sovelluksen rakennetta, jossa hyödynnetään InheritedWidget-luokkaa. Kuvan mukaisesti StudentState-luokka perii InheritedWidget-luokan, ja StudentState-luokka toimii kantaluokkana ExamWidget- ja GradeWidget-luokille. ExamWidget- ja GradeWidget-oliot pääsevät käsiksi StudentState-olion sisältämään dataan tarvittaessa koodirivillä: StudentState.of(context). Kyseinen of-metodi palauttaa widget-puurakenteesta kutsuvaa widgetiä lähinnä olevan kantaluokan, joka on StudentState-tyyppinen. InheritedWidget-luokka sisältää myös updateShouldNotify-metodin, joka laskee, tarvitseeko kantaluokan lapsiwidgetejä koota uudelleen tilamuutoksen jälkeen. [10.]



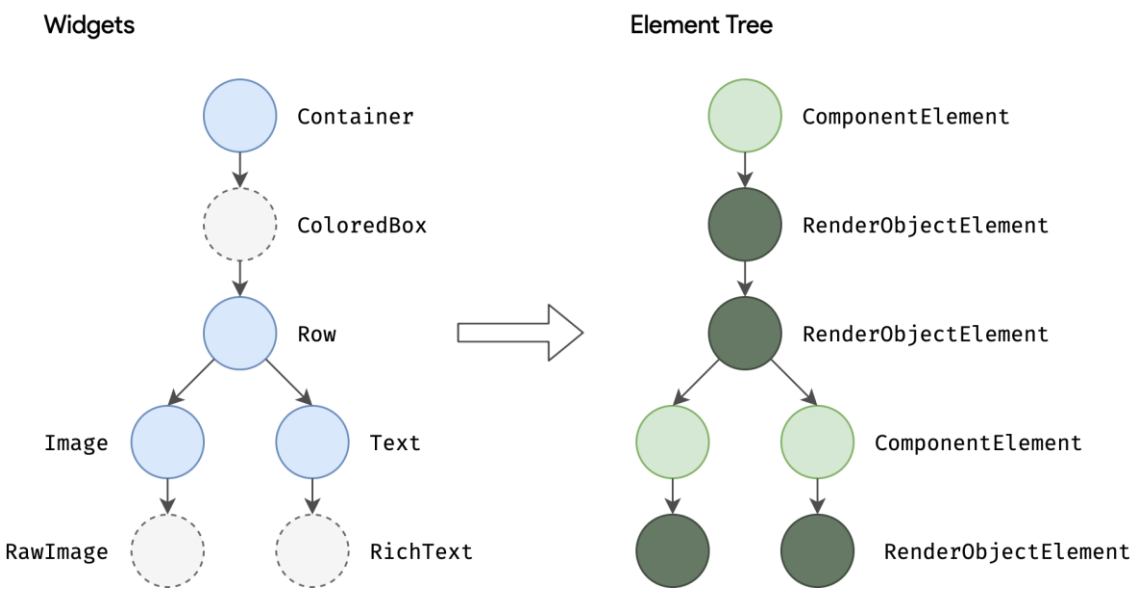
Kuva 8. InheritedWidgetin käyttö sovelluksessa.

Dart Flutterin sovelluskehyskerros käyttää myös InheritedWidgetin toimintaa hyödykseen esimerkiksi jakamalla sovelluksen teeman kaikkialle sovelluksessa. Tarvittaessa teemaan pääsee käsiksi kutsumalla Theme-luokan of-metodia. Tämän lisäksi sovelluksen sisäinen navigaatio sekä MediaQuery-luokka, jolla päästään käsiksi muun muassa tietoon näytön leveydestä ja korkeudesta, hyödyntävät InheritedWidgetin toimintaa. InheritedWidget on vain yksi monista Flutterin ratkaisuista sovelluksen tilan hallintaan. Joissain ratkaisuissa InheritedWidget toimii osana ratkaisua, esimerkki tällaisesta on pakkaus nimeltä provider.

3.2.6 Renderöinti

Dart Flutterin käyttöliittymäkomponentit renderöidään käyttäen Skia-grafiikkamoottoria, joka on ohjelmoitu C- ja C++-ohjelmointikielillä. Kyseistä grafiikkamoottoria käyttää alustan natiivi koodi, joka on käännetty Dart-koodin niistä osista, joita käytetään käyttöliittymän visuaalisen ulkoasun piirtämiseen. Grafiikkamoottorin avulla kutsutaan laitteen CPU:ta tai GPU:ta piirtämään sovelluksen käyttöliittymä näytölle. Skia sisältyy Flutterin ohjelmistomoottoriin, jonka ansiosta itse sovellukseen voi julkaista päivityksiä, vaikka laitteen käyttöjärjestelmää

ei olisi päivitetty viimeisimpään versioon. Flutterin sovelluskehyskerros käyttää ohjelmakoodin osien renderöintiin build-metodia. Metodin paluuarvona on widgeteistä koostuva puurakenne, jonka avulla käyttöliittymä renderöidään sovelluksen senhetkisen tilan perusteella. Tämän prosessin aikana build-metodi voi lisätä puurakenteeseen uusia widget-komponentteja. Esimerkiksi Container-widgetiin voi lisätä ColoredBox-widgetin sillä ehdolla, että Containerin väriparametri ei ole null arvoltaan. Samaan tapaan Image-widgetiin voi lisätä lapsiwidgetiksi RawImage-widgetin ja Text-widgetiin RichText-widgetin build-metodin aikana. Tällaisissa tapauksissa widgetien hierarkia on syvempi kuin alkuperäisessä koodissa, sillä ylimääräiset widgetit lisätään vasta build-metodin aikana. Build-metodin aikana Flutter muuttaa widgeteistä koostuvan puurakenteen elementtipuuksi.

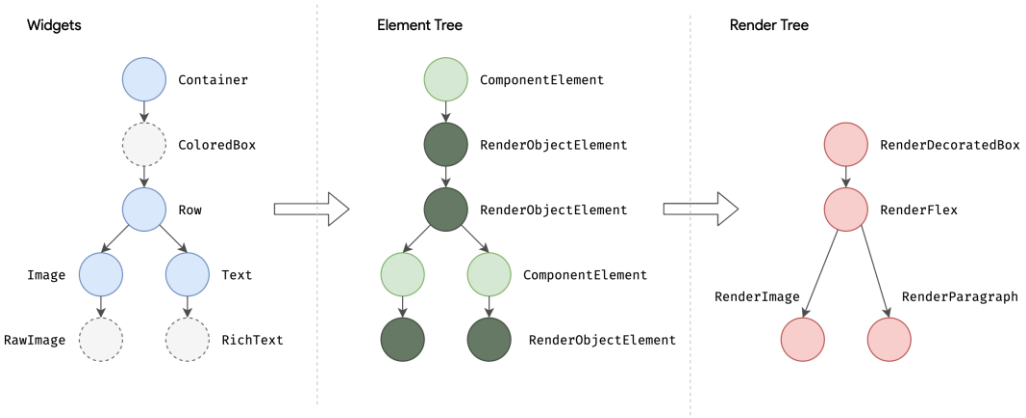


Kuva 9. Widget-puusta elementtipuuksi.

Jokaiselle widgetille on oma elementtinsä, ja jokainen elementti edustaa tiettyä widgetin instanssia tietyssä sijainnissa puurakenteessa. Elementit jaetaan kahteen eri tyyppiin: ComponentElement- ja RenderObjectElement-elementtiin. ComponentElement toimii isäntänä muille elementeille, kun taas RenderObjectElement-elementtiä käytetään lähinnä käyttöliittymän asetteluun liittyvissä vaiheissa. Lisäksi RenderObjectElement toimii widgetien ja ComponentElement-

elementtien välikappaleena. Minkä tahansa widgetin elementtiin voi viitata BuildContext-olion kautta, jonka avulla pääsee käsiksi widgetin sijaintiin elementtipuussa. BuildContext-olio tulee build-metodin mukana parametrina. Elementtipuuta ei tarvitse myöskään aina koota uudestaan, kun widgeteihin kohdistuu muutoksia. Tällainen tilanne on esimerkiksi käyttöliittymän tekstin muuttuminen käyttäjän vuorovaikutuksen seurauksena. Koska jokaisella widgetillä on oma elementtinsä, sovelluskehys käy läpi muuttuneiden widgetien muutokset, ja tämän jälkeen widgetejä vastaaviin elementteihin tehdään tarvittavat muutokset. Tällainen lähestymistapa parantaa sovelluksen suorituskykyä, sillä elementtipuun uudelleenrakentaminen aina, kun muutoksia tapahtuu, olisi työlästä.

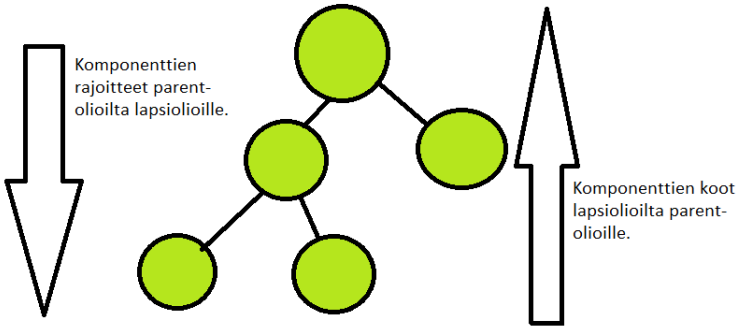
Build-metodin aikana muodostettu elementtipuu muutetaan vielä renderöitäväksi puurakenteeksi. Kyseisen puurakenteen avulla kaikille elementeille voidaan määrittää koko ja sijainti näytöllä ennen varsinaista laitteen näytölle renderöintiä. Puun jokaisen solmun kantaluokkana toimii RenderObject-luokka, joka määrittelee abstraktin mallin asettelulle ja visuaaliselle ulkoasulle. Jokainen RenderObject-olio tuntee parent-komponenttinsa, mutta lapsikomponentistaan se tietää vain lapsikomponentin rajat sekä minkälaisia operaatioita siihen tulee kohdistaa. Tämän vuoksi RenderObject-luokka soveltuu useaan eri tilanteeseen renderöinnissä. Kuvan 10 mukaisesti build-metodin aikana jokaisesta elementtipuun RenderObjectElement-oliosta luodaan renderöitävään puurakenteeseen RenderObject-olio. Mikäli olio on jo olemassa, samaa oliota päivitetään uuden datan mukaisesti tarvittaessa. RenderObject-oliot ovat primitiivisiä: RenderParagraph-luokka renderöi tekstiä ja RenderImage puolestaan kuvia. [10.]



Kuva 10. Elementtipuu muutetaan renderöitäväksi puuksi.

3.2.7 Box Constraint -malli

Widgetien renderöintiin käytetään usein oliota, joka perii RenderBox-luokan. RenderBox-luokka edustaa RenderObject-luokkaa määrätyn kokoisessa 2-ulotteisessa karteesisessa koordinaatistossa. RenderBox-luokan toiminta perustuu box constraint -malliin, jossa jokaiselle renderöitävälle widgetille määritellään korkeudelle sekä leveydelle minimi- ja maksimiarvot. Renderöitävä puurakenne käydään läpi ensisijaisesti syvyyssuunnassa, ja parent-olioiden korkeuden ja leveyden rajoitteet välitetään lapsiolioille. Lapsiolioiden tulee noudattaa parent-olioiden antamia koon rajoitteita, ja lapsioliot välittävät oman korkeutensa ja leveytensä takaisin parent-olioille. Renderöitävän puurakenteen läpikäynnin jälkeen puun jokaiselle oliolle on määritelty koko parent-olioiden koon rajoitteiden perusteella, joten käyttöliittymä on valmis piirrettäväksi paint-metodin avulla.



Kuva 11. Box constraint -mallin idea.

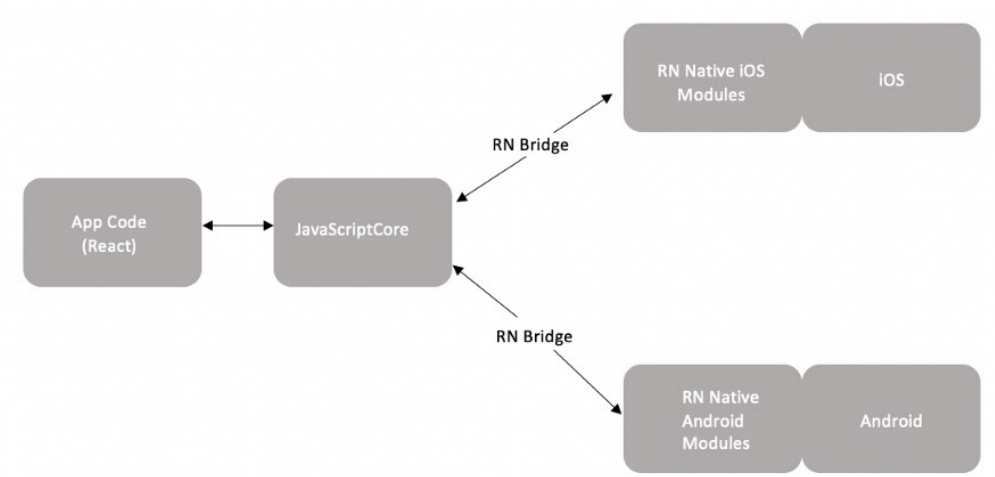
Renderöinnissä käytettävän box constraint -mallin avulla olioiden asettelun aikavaatimus on O(n) eli kyseisen mallin avulla puurakenteen oliot saadaan renderöityä tehokkaasti. Parent-oliot pystyvät määrittelemään lapsiolioiden koon asettamalla oman korkeutensa ja leveytensä minimi- ja maksimiarvot samaan arvoon. Esimerkiksi sovelluksen ylimmäisin renderöitävä olio voi määrittää lapsiolionsa olemaan samankokoinen kuin näyttö. Lapsiolio pystyy itse määrittelemään, miten tila käytetään. Lisäksi parent-olio voi määrittää lapsioliolle pelkästään joko korkeuden tai leveyden, ja antaa lapsiolion määrittää vapaammin jommankumman näistä kahdesta. Esimerkki tällaisesta on flow text -elementti, jolle täytyy määrittää koko vaakasuunnassa eli leveys, mutta elementin korkeus riippuu tekstin määrästä. Mikäli puurakenteen lapsiolion täytyy tutkia, kuinka paljon sille on vielä tilaa jäljellä renderöintiä varten, kyseisen widgetin voi laittaa LayoutBuilder-widgetin sisälle build-metodissa. Tällöin lapsiolio pääsee tutkimaan koon rajoituksia ja määrittelemään, miten se käyttää niitä.

Renderöitävän puurakenteen kaikkien RenderObject-olioiden taustalla on RenderView-luokka, joka edustaa koko puurakenteen lopputulosta. Alustan pyytäessä uutta kuvaa renderöitäväksi RenderView-olion compositeFrame-metodia kutsutaan. Kyseinen metodi luo instanssin SceneBuilder-luokasta, joka puolestaan laukaisee näkymän päivittämisen. Näkymän valmistuttua se välitetään Window.render-metodille RenderView-olion avulla. Render-metodi puolestaan välittää saamansa näkymän GPU:lle, joka suorittaa renderöinnin. Kaiken kaikkiaan voidaan siis todeta, että käyttöliittymän näyttämiseen laitteen näytöllä liittyy useita eri vaiheita. [10.]

3.3 Eroavaisuudet React Nativeen

Dart Flutter sekä React Native molemmat monialustaisina teknologioina mahdollistavat sovelluksen toteuttamisen Android- ja iOS-alustoille vain yhden lähdekoodin avulla. Lisäksi molemmat teknologiat reaktiivisina sovelluskehyksinä toteuttavat sovelluskehitysmallia, jossa kehittäjän täytyy vain kirjoittaa kuvaus sovelluksen käyttöliittymästä, ja sovelluskehys huolehtii käyttöliittymän tilan päivittämisestä. React Native ja Dart Flutter ovat kuitenkin kilpailevia teknologioita, joten niiden välillä on myös eroavaisuuksia.

Dart Flutterin yksi merkittävimmistä eroavaisuuksista React Nativeen on sen toimintaperiaate. React Nativen toiminta perustuu siltakomponentin käyttöön ohjelmakoodin ja alustan välillä. Dart Flutterin toiminta perustuu kerrosarkkitehtuurin eri kerroksiin, kuten luvussa 3.1 on kerrottu. Dart Flutter pääsee käsiksi sovelluksen senhetkiseen alustaan embedder-kerroksen avulla, joka alustakohtaisen embedder-sovelluksen avulla lataa Flutterin alustalle ja alustaa Flutterin ohjelmistomoottorin. Kuvassa 12 on havainnollistettu React Nativen siltakomponentin toimintaperiaatetta.



Kuva 12. React Nativen siltakomponentti.

Kuten aiemmissa luvuissa käy ilmi, React Native käyttää kuvan 12 mukaista siltakomponenttia käyttöliittymän renderöintiin alustan natiivien käyttöliittymäkomponenttien avulla. Siltakomponentti välittää viestin React-komponenteilta sovelluksen alustan grafiikkamoottorille siitä, mitä näytölle tulee renderöidä ja minkälaisia muutoksia käyttöliittymään on tullut. Monimutkaisissa käyttöliittymärakenteissa tämä on ongelmallista sovelluksen suorituskyvyn kannalta, sillä alustan ja React-komponenttien välillä kulkee tällöin useita viestejä, jotka kaikki täytyy käsitellä asianmukaisesti. Tämä puolestaan hidastaa käyttöliittymän toimintaa. Dart Flutter sen sijaan käyttää omaa Skia-grafiikkamoottoria käyttöliittymäkomponenttien renderöintiin. Lisäksi Dart Flutter käyttää käyttöliittymäkomponentteina Material- ja Cupertino-kirjastojen komponentteja. Näiden edellä mainittujen seikkojen vuoksi Dart Flutterilla ei ole samanlaisia suorituskykyongelmia kuin React Nativella.

Toinen React Nativen ja Dart Flutterin välisistä merkittävistä eroavaisuuksista on niiden käyttämä ohjelmointikieli. React Native käyttää JSX-kieltä, joka on JavaScriptin ja XML-kielen yhdistelmä. Dart Flutter puolestaan käyttää oliopohjaista Dart-ohjelmointikieltä, joka muistuttaa syntaksiltaan muun muassa C-ohjelmointikieliä. Taustalla olevien ohjelmointikielien eroavaisuus on huomattavissa selkeästi esimerkiksi kohdissa, joissa rakennetaan käyttöliittymän ku-

vausta käyttöliittymäkomponenttien avulla. React Nativen ohjelmakoodissa korostuu tässä tilanteessa XML-kielen käyttö käyttöliittymäkomponenttien asettelua varten. Esimerkki tällaisesta on komponentti tekstiä varten, joka on <Text></Text> XML-kielellä ilmaistuna. Dart Flutterin ohjelmakoodissa sen sijaan käyttöliittymäkomponenteista luodaan instansseja olioiden avulla, ja instanssi alustetaan halutulla datalla syöttämällä data instanssin luonnin yhteydessä konstruktoriin. Esimerkki instanssin luominen tekstikomponentista: Text (”Hello World”). Näiden eroavaisuuksien lisäksi React Nativen ja Dart Flutterin ohjelmakoodissa esiintyy myös muita eroavaisuuksia. Yksi näistä on sovelluksen tilan hallinta. Dart Flutter käyttää käyttöliittymäkomponentteina widgetejä, jotka ovat muuttumattomia kuvauksia käyttöliittymästä. Widgetien tilaa muutetaan State-luokan build -ja setState-metodien avulla. React Nativessa käyttöliittymäkomponenttien dataa puolestaan hallitaan state -ja props-nimisten JavaScript-olioiden avulla [11]. React Nativen props-oliota käytetään datan hakemiseen käyttöliittymäkomponentteihin, mutta kyseisen olion sisältämään dataan ei kohdisteta muutoksia. Muutokset kohdistetaan sen sijaan state-olioon, ja props-olion avulla haetaan state-olion sisältämä data. State-olion voi alustaa olion luonnin yhteydessä syöttämällä olion sisään haluamansa datan. Tilan muuttuessa state-oliota muutetaan samalla tavalla antamalle sille uusi arvo. Dart Flutterissa widgetin tilan voi alustaa ylikirjoittamalla State-luokan initState-metodi. Tilan muuttuessa kutsutaan setState-metodia, ja metodin sisällä yleensä annetaan jollekin muutettavalle muuttujan arvolle uusi arvo.

Kolmas merkittävä eroavaisuus React Nativen ja Dart Flutterin välillä on vaaditun taustatiedon määrä. React Native perustuu Facebookin JavaScript-kirjastoon, Reactiin. Toisin sanoen React Nativea varten olisi hyvä olla aikaisempaa kokemusta web-sovelluksiin käytettävästä Reactista. Tämän lisäksi olisi hyvä myös olla kokemusta web-teknologioista yleisesti, sillä React Nativen taustalla olevaa JavaScript-ohjelmointikieltä käytetään web-ohjelmoinnissa. Sovelluksen kehittäjällä tulisi myös olla kokemusta Android -ja iOS-alustojen natiiveista teknologioista. Vain osa React Nativen käyttöliittymäkomponenteista ovat monialustaisia, ja alustakohtaisia komponentteja varten on kummallekin alustalle omat komponenttinsa. Vaikka React Nativen kaikki käyttöliittymäkomponentit

löytyvät virallisesta dokumentaatiosta, yleinen tuntemus alustojen käyttöliittymäkomponenteista helpottaa käyttöliittymän rakentamista. Lisäksi React Native ei välttämättä tue jotain tiettyä alustan käyttöliittymäkomponenttia, jolloin React Nativen kanssa yhteensopiva moduuli tulee toteuttaa alustan natiivilla ohjelmointikielellä. Dart Flutterin taustalla on sen sijaan Dart-ohjelmointikieli, joten aikaisempaa kokemusta web-teknologioista ei tarvitse olla. Dart Flutterin ohjelmakoodi on syntaksiltaan yksinkertaista, joten myös aloitteleva kehittäjä pystyy todennäköisesti oppimaan teknologian nopeasti. Dart Flutter myös tarjoaa omat kirjastot käyttöliittymäkomponentteja varten, joten kokemusta natiiveista teknologioista ei välttämättä tarvitse olla. Teknologian virallinen dokumentaatio lisäksi sisältää kattavat kuvaukset, mihin eri tilanteisiin kirjastojen komponentit soveltuvat. Kaiken kaikkiaan Dart Flutter ei vaadi siis yhtä paljon aikaisempaa kokemusta sovellusten kehittämisestä kuin React Native, vaikka kokemuksesta on aina etua myös Dart Flutterinkin tapauksessa.

3.4 Dart Flutterin edut React Nativeen nähden

Dart Flutterin ja React Nativen välillä on monialustaisina teknologioina sekä yhtäläisyyksiä sekä eroavaisuuksia. Osa näiden teknologioiden eroavaisuuksista ei aina välttämättä osoitu Dart Flutterin eduksi. Vaikka Dart Flutter on teknologiana uudempi kuin React Native, on sillä kuitenkin React Nativeen nähden joitain selkeitä etuja. Yksi näistä Dart Flutterin eduista on Material- ja Cupertino-kirjastojen käyttö käyttöliittymäkomponentteihin. Tämän ansiosta Dart Flutter ei tarvitse siltakomponenttia alustojen omia käyttöliittymäkomponentteja varten, sillä se ei käytä niitä. Lisäksi Dart Flutter käyttää omaa versiota Skia-grafiikkamoottorista, joten ohjelmakoodin ei tarvitse pyytää alustojen omia grafiikkamoottoreita renderöimään käyttöliittymän osia. Näiden edellä mainittujen seikkojen perusteella ohjelmakoodin ja alustan välillä ei tarvitse välittää yhtä paljon viestejä kuin React Nativen tapauksessa. Tämä puolestaan takaa sen, että sovelluksen kuvataajuus on aina 60-120 kuvaa sekunnissa, joten monimutkaisissa käyttöliittymärakenteissa Dart Flutter yltää parempaan suorituskykyyn kuin React Native. Material- ja Cupertino-kirjastojen käyttämisessä natiivien kompo-

nenttien sijasta etuna on myös käyttöliittymän sama tuntuma kummallakin alustalla. Lisäksi käyttöliittymän tuntuma pysyy samana, vaikka alustoilla natiivit käyttöliittymäkomponenttien toteutukset muuttuisivat. React Nativen tapauksessa muutos vaikuttaisi sovelluksen käyttöliittymään. [12.]

Toinen Dart Flutterin eduista React Nativeen on se, että teknologian pystyy aloitteleva kehittäjäkin oppimaan nopeasti. Dart Flutter ei vaadi aikaisempaa kokemusta web-teknologioista, sillä Dart Flutter ei perustu JavaScriptiin, vaan Googlen kehittämään oliopohjaiseen Dart-ohjelmointikieleen. Tämä tarkoittaa sitä, että taustalla olevan Dart-ohjelmointikielen oppiminen on helppoa, jos on tutustunut aiemmin oliopohjaisiin ohjelmointikieliin. Aikaisempaa kokemusta ei kuitenkaan tarvitse olla paljon, sillä Dart Flutterin virallinen sivusto sisältää kattavat dokumentaatiot teknologian ominaisuuksista sekä selkeät esimerkit niiden toteuttamisesta omassa ohjelmakoodissa. Dart Flutterin helposti opittavuutta lisää myös se, että natiiveista teknologioista ei välttämättä tarvitse olla yhtä paljon kokemusta kuin React Nativen tapauksessa. Dart Flutter käyttää omia käyttöliittymäkomponentteja natiivien komponenttien sijasta. Kaikki komponentit on dokumentoitu ja niiden käytöstä on selkeät esimerkit dokumentaation yhteydessä. Kaikki komponentit ovat monialustaisia, joten samasta komponentista ei tarvitse kirjoittaa kummallekin alustalle omaa toteutusta, kuten React Nativen tapauksessa. [12.]

3.5 Dart Flutterin heikkoudet

Dart Flutterin toiminnassa on siis etuja React Nativeen verrattuna. Eduista huolimatta Dart Flutterin toimintaperiaatteessa on kuitenkin myös heikkouksia, joita React Nativen toimintaperiaatteessa ei ole. Yksi heikkouksista liittyy siihen, että Dart Flutter ei käytä alustojen natiiveja käyttöliittymäkomponentteja. Tällä hetkellä nimittäin Dart Flutterin käyttöliittymäkomponentit eivät tue 3D-grafiikkaa, vaan ainoastaan 2D-grafiikkaa. React Native puolestaan pystyy renderöimään 3D-grafiikkaa, sillä se käyttää renderöintiin alustojen omia grafiikkamoottoreita sekä käyttöliittymäkomponentteja. Tämä siis tarkoittaa sitä, että Dart Flutterilla

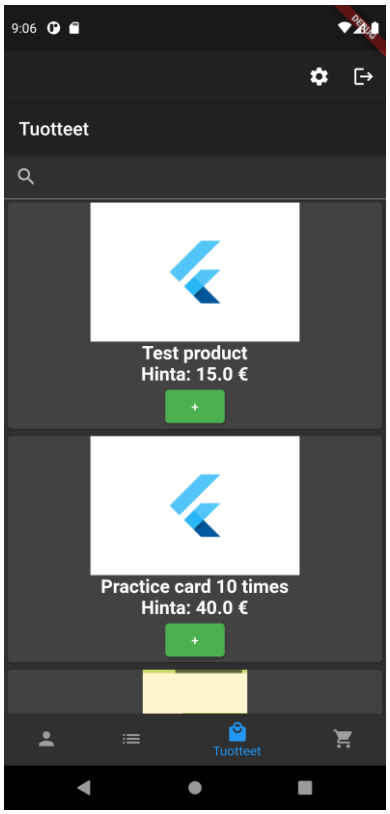
toteutettavien mahdollisten sovellusten määrä ei ole yhtä suuri kuin React Nativella. Mikäli sovellus halutaan siis toteuttaa niin, että siinä on 3D-grafiikkaa, sovellus täytyy tällöin toteuttaa joko React Nativella tai natiiveilla teknologioilla. Material- ja Cupertino-kirjastojen käyttämisessä sovelluksen käyttöliittymäkomponentteihin on myös se heikkous, että ne vain jäljittelevät alustojen omien käyttöliittymäkomponenttien ulkoasua ja toimintatapaa. Tämä tarkoittaa sitä, että etenkin iOS:llä saattaa käyttöliittymäkomponenttien ulkoasussa olla pieniä eroavaisuuksia. Tämän lisäksi sovelluksen käyttöliittymä todennäköisesti poikkeaa natiiveilla teknologioilla toteutetuista sovelluksista, mikäli alustojen natiivien käyttöliittymäkomponenttien toteutusta on muutettu. Material- ja Cupertino-kirjastojen käytöllä käyttöliittymässä voi siis olla joitain heikentäviä vaikutuksia sovelluksen käytettävyyteen. [12.]

Toinen Dart Flutterin heikkouksista liittyy siihen, että se on teknologiana vielä melko uusi. React Native julkaistiin vuonna 2015, kun taas Dart Flutter on julkaistu vasta vuonna 2018. Tämä tarkoittaa sitä, että React Nativella on ollut huomattavasti enemmän aikaa kehittää ominaisuuksiaan, jotta sovellukset toimisivat mahdollisimman vakaasti laitteilla ja bugeja toiminnallisuuksissa esiintyisi mahdollisimman vähän. Dart Flutterilla toteutetut sovellukset toimivat pääosin moitteettomasti alustoilla, mutta build-metodissa saattaa esiintyä joitain bugeja. Yksi esimerkki tällaisesta on Image.file-widgetin käyttö build-metodissa, jonka avulla voidaan lisätä kuva käyttöliittymään laitteen tiedostoista. Mikäli käyttäjälle annetaan mahdollisuus vaihtaa kuva esimerkiksi gallerian kautta, uusi kuva ei välttämättä aina päivity käyttöliittymään, vaikka build-metodin pitäisi päivittää Image.file-widgetin data uuden kuvan mukaiseksi. Tämä johtuu siitä, että aikaisempi kuva on tallennettu välimuistiin, ja Image.file saattaa käyttää välimuistissa olevaa dataa build-metodin yhteydessä. Kyseiseen bugiin ratkaisuna on se, että ennen uuden kuvan päivittämistä käyttöliittymään kuvia sisältävä välimuisti tyhjennetään ohjelmallisesti. Dart Flutterin uutuus teknologiana näkyy myös siten, että se ei välttämättä tue kaikkia laitteen palveluita ja ominaisuuksia. Laitteen natiivin ohjelmakoodin ja Dart Flutterin ohjelmakoodin välille voi tietysti aina kehittää yhteensopivan moduulin, mutta se voi olla haastavaa sekä aikaa vievää. Dart Flutterilla ei ole myöskään yhtä paljon kolmannen osapuolen

paketteja kuin React Nativella, ja osa paketeista on vielä kehitysvaiheessa. Tämä tarkoittaa sitä, että paketin ominaisuudet eivät välttämättä ole samalla tasolla kuin React Nativessa. [12.]

4 Verkkokauppasovelluksen toteutus Dart Flutterilla

Insinöörityön aikana toteutettiin Dart Flutterilla verkkokauppasovellus Android- ja iOS-alustoille kuvitteelliselle yritykselle. Verkkokaupassa myydään erään urheilulajin harjoittelupaikan treenimaksuja sekä kyseiseen lajiin kuuluvia harjoitteluvarusteita. Sovellusta käyttääkseen sovelluksen käyttäjän tulee luoda ensin profiili, johon tulee käyttäjän yhteystiedot sekä syntymäaika. Sovelluksessa käyttäjät on jaettu kahteen eri tyyppiin: perus- ja pääkäyttäjiin. Profiilin luonnin jälkeen käyttäjällä on peruskäyttäjän oikeudet. Tällä käyttäjätyypillä pystyy tarkastelemaan verkkokaupan tuotteita sekä lisäämään niitä ostoskoriin, tekemään tilauksia, tarkastelemaan omien tilausten tilaa sekä muokkaamaan omaa profiilia ja vaihtamaan salasanan. Sovelluksessa on oletuksena jo yksi pääkäyttäjä, jonka käyttäjänimi on admin. Pääkäyttäjä pystyy tarvittaessa antamaan peruskäyttäjälle pääkäyttäjän oikeudet. Sovelluksen pääkäyttäjä pystyy tarkastelemaan ja muokkaamaan kaikkien profiilien tietoja sekä resetoimaan käyttäjien salasanoja. Pääkäyttäjä pystyy myös tarvittaessa poistamaan olemassa olevia profiileja. Lisäksi pääkäyttäjä pystyy lisäämään verkkokauppaan uusia tuotteita sekä poistamaan tai muokkaamaan olemassa olevia tuotteita. Pääkäyttäjä näkee myös kaikki verkkokaupassa tehdyt tilaukset sekä pystyy tarvittaessa vahvistamaan tai peruuttamaan tilauksen. Näiden edellä kuvattujen ominaisuuksien lisäksi pääkäyttäjällä on sovelluksessa samat toiminnallisuudet kuin peruskäyttäjällä. Sovellus on lokalisoitu suomen kielelle, ja käyttäjä voi asetusnäkymän kautta valita, näytetäänkö sisältö englanniksi vai suomeksi. Samaisesta asetusnäkymästä voi myös valita, onko sovelluksen teemana vaalea vai tumma tila. Sovelluksen käyttöliittymä on toteutettu Material-kirjaston käyttöliittymäkomponenttien avulla. Kuvassa 13 on peruskäyttäjän näkymä verkkokaupan tuotteet -välilehdeltä.



Kuva 13. Peruskäyttäjän näkymä verkkokaupassa.

Kuvan 13 mukaisessa tilanteessa on näkyvissä listanäkymä, jossa on verkkokaupan tuotteita. Tuotteen hinnan alla olevasta vihreästä plusnapista saa lisättyä kyseisen tuotteen ostoskoriin. Välilehden yläosassa olevasta hakukentästä voi hakea tuotteita tuotteen nimellä. Tällöin välilehden listanäkymä päivittyy automaattisesti hakutulosten mukaiseksi.

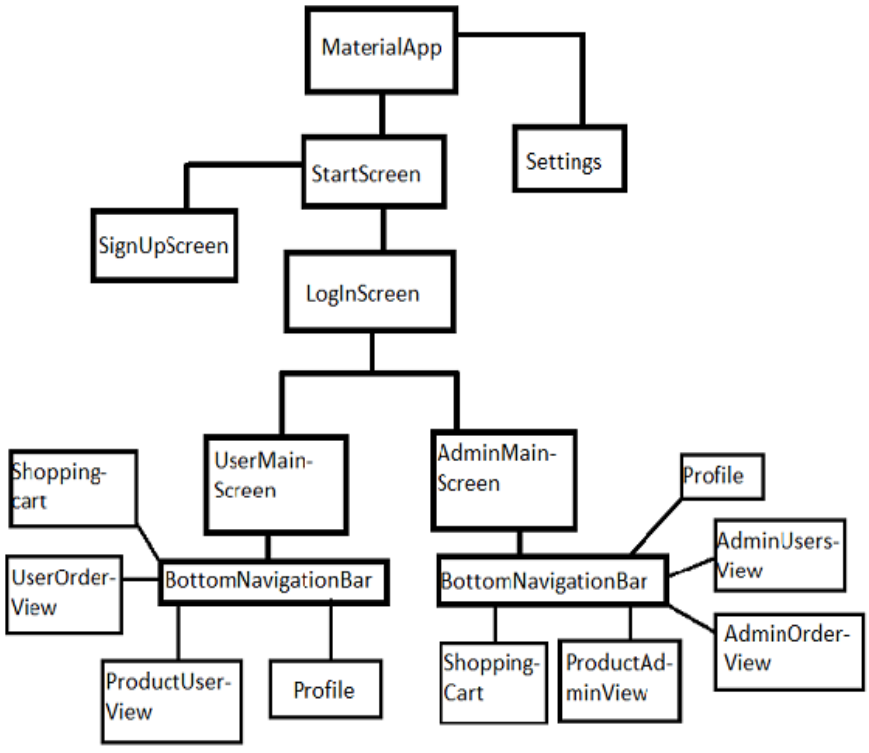
4.1 Sovelluksen tarkoitus

Sovelluksen tarkoituksena on havainnollistaa, miten helposti Dart Flutter -teknologian avulla saa toteutettua monialustaisen sovelluksen puhelimille. Tavoitteena on myös tutkia, minkälaisia alustakohtaisia asioita sovelluksen kehittämisessä on otettava huomioon ja kuinka paljon ominaisuuksia itse sovelluskehys tarjoaa kehittäjälle ilman ulkoisia pakkauksia. Alustakohtaiset ominaisuudet ja palvelut, kuten laitteen kamera ja tallennustilan käyttö, on toteutettu pakkauk-

sina, joten sovelluksen avulla saadaan myös selville, kuinka hyvin kyseiset pakkaukset toimivat. Sovelluksessa tietojen tallettamiseen käytetään SQLite-tietokantaa, joten samalla saadaan havainnollistettua, miten käyttöliittymä kommunikoi taustalla olevan tietokannan kanssa. Sovellukseen toteutettiin myös näkymä asetuksia varten, jonka kautta voidaan vaihtaa koko sovelluksen teemaa tai kieltä. Tämän avulla pystytään havainnollistamaan, miten sovelluksen tilaa voidaan hallita.

4.2 Sovelluksen rakenne

Seuraavaksi käydään tarkemmin läpi, minkälaisista osista sovelluksen käyttöliittymä koostuu.



Kuva 14. Sovelluksen rakenne pääpiirteittäin.

Kuvassa 14 kuvataan pääpiirteittäin sovelluksen widgetit eli luokat. Kullakin kuvassa olevalla luokalla on oma näkymänsä käyttöliittymässä. Kuvassa olevaan kaavioon ei ole kuitenkaan merkitty kaikkia luokkia. Osaan näkymistä pääsee

vain tietyn näkymän kautta, joten tällaiset näkymät on jätetty kaaviosta selkeyden vuoksi pois. Esimerkki tällaisesta näkymästä on ProductUserView-näkymässä, jossa on listattu kaikki tuotteet ja tuotteen napauttaminen avaa ikkunan, jossa on tuotteen tarkemmat tiedot. Tämän lisäksi kaikilla luokilla ei ole visuaalista ulkoasua, sillä niitä käytetään niin sanottuina model-luokkina, joiden avulla olioiden tietoja voidaan tallettaa tietokantaan. Sovelluksessa on myös yksi luokka tietokantaa varten, joka sisältää kaikki operaatiot tietokannassa olevan datan käsittelyyn. Kaiken kaikkiaan sovelluksessa on 34 luokkaa.

Käyttöliittymän rakentamiseen käytetään sovelluksessa Material-kirjaston käyttöliittymäkomponentteja. Tämän vuoksi kuvan 14 mukaisesti sovelluksen ylin widget on MaterialApp-luokka, joka pitää sisällään koko sovelluksen. Sovelluksen asetuksia pääsee tarkastelemaan kaikkialla sovelluksessa, joten Settings-luokka on kytköksissä suoraan MaterialApp-luokkaan. MaterialApp-luokkaan on kytköksissä myös sovelluksen aloitusnäkymä eli StartScreen-luokka. Tästä näkymästä puolestaan pääsee joko kirjautumaan eli LogInScreen-luokkaan tai rekisteröitymään eli SignUpScreen-luokkaan. Kirjautumisen jälkeen edetään joko UserMainScreen- tai AdminMainScreen-luokkaan riippuen siitä, onko käyttäjällä perus- vai pääkäyttäjän oikeudet. Kumpikin edellä mainituista luokista sisältää näkymissään alapalkin eli BottomNavigationBar-widgetin, jonka välilehtinä ovat kuvan 7 mukaiset BottomNavigationBar-widgetien alla olevat luokat. BottomNavigationBar-widgetit on laitettu IndexedStack-widgetin sisään, jonka avulla välilehtien tila pystytään säilyttämään. Esimerkiksi jos tuotevälilehden hakukenttään kirjoittaa tekstiä, teksti säilyy hakukentässä, vaikka sovelluksen käyttäjä vaihtaisi johonkin toiseen välilehteen. Taulukossa 3 on lueteltu kuvan 14 mukaiset BottomNavigationBar-widgetien välilehdet sovelluksessa.

Taulukko 3. Sovelluksen välilehdet perus- ja pääkäyttäjille.

Luokan nimi	**Luokan näkymän kuvaus**
Profile	Sisältää käyttäjän syöttämät tiedot, kuten nimen ja yhteystiedot.

AdminUsersView	Sisältää listanäkymän sovelluksen kaikista käyttäjistä. Näkyy vain pääkäyttäjillä.
AdminOrderView	Pääkäyttäjille näytettävä listanäkymä kaikista käyttäjien tekemistä tilauksista.
ProductAdminView	Listanäkymä kaikista verkkokaupan tuotteista ja tuotteita on mahdollisuus muokata. Näkyy pääkäyttäjillä.
ProductUserView	Peruskäyttäjien näkymä tuotelistauksesta. Käyttäjällä oikeudet pelkästään tuotteiden lisäämiseen ostoskoriin.
UserOrderView	Peruskäyttäjien näkymä, jossa käyttäjä näkee omat tilauksensa.
ShoppingCart	Ostoskori, jossa näkyy listattuna kaikki ostoskoriin lisätyt tuotteet.

Sovelluksen kehittämisessä painotettiin Dart Flutterin ominaisuuksiin tutustumista. Tämän vuoksi sovelluksen rakenne ei noudata mitään yleisiä sovellusarkkitehtuureja, kuten MVP:tä. Sovelluksen tietokantaluokka on toteutettu hyödyntäen Singleton-suunnittelumallia, sillä useamman kuin yhden instanssin luominen tietokannasta ajon aikana vaikuttaisi sovelluksen suorituskykyyn.

4.2.1 Alustariippuvainen määrittely

Sovelluksen kaikki käyttöliittymän osat saatiin toteutettua Material-kirjaston käyttöliittymäkomponenttien avulla. Näin ollen ohjelmakoodiin ei tarvinnut lisätä kummallekaan alustalle natiiveilla ohjelmointikielillä toteutettuja osia. Lisäksi ohjelmakoodissa ei tarvitse ollenkaan tarkistaa sovelluksen ajonaikaista alustaa. Toisin sanoen alustariippuvaista määrittelyä sovelluksen kehittämiseen ei juuri tarvittu. Alustariippuvainen määrittely kohdistui sovelluksen ohjelmakoodissa tarvittavien käyttöoikeuksien lisäämiseen, joiden avulla sovellus saa oikeudet käyttää määritettyä laitteen ominaisuutta tai palvelua, kuten laitteen tiedostojärjestelmää. IOS-alustaa varten käyttöoikeudet määritetään info.plist-tiedostoon ja Androidia varten puolestaan AndroidManifest.xml-tiedostoon. Sovellukselle piti

lisätä iOS-alustaa varten oikeudet kameran ja kuvagallerian käyttöä varten. Koodiesimerkissä 5 havainnollistetaan, miten oikeudet lisätään info.plist-tiedostoon kameran ja kuvagallerian käyttöä varten.

```
<key>NSPhotoLibraryUsageDescription</key>
<string>Allow access to photo library</string>
<key>NSCameraUsageDescription</key>
<string>Allow access to camera to capture photos</string>
<key>NSMicrophoneUsageDescription</key>
<string>Allow access to microphone</string>
```

Esimerkkikoodi 5. Käyttöoikeudet iOS:lle kameraa ja kuvagalleriaa varten.

Androidia varten ohjelmakoodi puolestaan tarvitsi oikeudet päästä käsiksi laitteen tiedostojärjestelmään ja tallentaa sinne uusia tiedostoja. Näitä oikeuksia tarvitaan sovelluksessa, kun olemassa olevaa tuotetta päivitetään, sillä tuotteen kuva täytyy väliaikaisesti tallentaa tietokannasta väliaikaistiedostoihin. Koodiesimerkissä 6 havainnollistetaan, miten edellä mainitut oikeudet tiedostoihin annetaan AndroidManifest.xml-tiedostossa.

```
<uses-permission android:name=”android.permission.WRITE_EXTERNAL_STOR-
AGE/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STOR-
AGE"/>
```

Esimerkkikoodi 6. Oikeudet Androidin tiedostojärjestelmään.

4.2.2 Käytetyt ulkoiset pakkaukset

Dart Flutter käyttää ulkoisia pakkauksia laitteen ominaisuuksien ja palveluiden käyttämistä varten. Sovelluksen toteutuksessa on käytetty seuraavia pakkauksia:

- sqflite
- path_provider
- image_picker
- flutter_localizations
- provider
- shared_preferences.

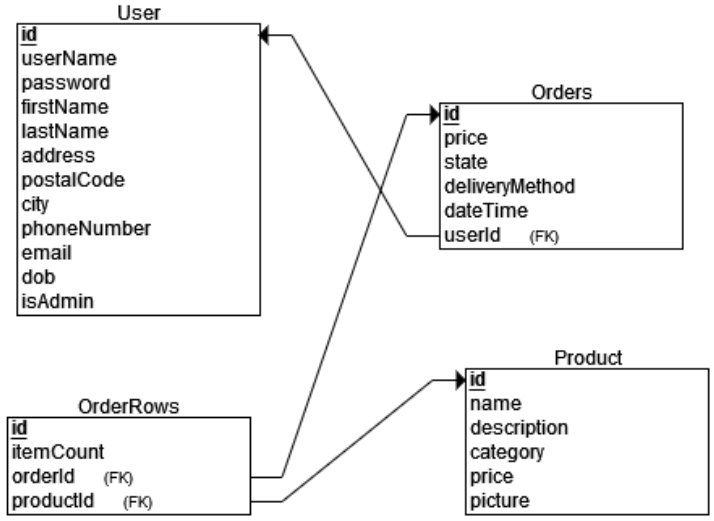
Sqflite-paketin avulla laitteeseen saa luotua SQL-tietokannan, johon talletetaan säilytettävää dataa. Paketti sisältää kaikki tarvittavat funktiot tietokannan alustamiseen sekä ylläpitoon, ja paketti toimii samalla tavalla kuin natiiveilla teknologioilla. Path_provider-pakkausta puolestaan käytetään laitteen tiedostojärjestelmää varten, ja pakkauksen avulla pääsee käsiksi hakemistojen polkuun. Tämän avulla sovellus pystyy tallentamaan hakemistoihin tiedostoja tarvittaessa. Sovelluksessa käytetään myös kameraa ja kuvagalleriaa kuvan lisäämiseen tuotteita varten. Tätä varten on käytetty pakkausta nimeltä image_picker, jonka pickImage-metodin avulla kuvan voi lisätä joko laitteen kameran tai gallerian kautta. Sovellus on lokalisoitu suomen kielelle, ja tähän käytetään flutter_localizations-pakkausta. Kyseisen pakkauksen avulla ohjelmakoodissa voidaan tutkia, mikä on laitteen kieliasetus ja sen perusteella voidaan ladata käyttöliittymän teksti oikeasta resurssitiedostosta. Sovelluksen tilanhallintaan voi käyttää provider-pakkauksen toimintoja, jotka perustuvat luvussa 3.2 mainittuun InheritedWidgetin käyttöön. Ohjelmakoodissa provider-pakkausta käytetään esimerkiksi ostoskorin toteuttamisessa, sillä ostoskoriin lisättyjen tuotteiden pitää päivittyä automaattisesti ostoskorin listanäkymään. Shared_preferences-pakkausta käytetään puolestaan yksinkertaisen datan tallettamiseen iOS:n NSUserDefaults-tietokantaan ja Androidin SharedPreferences-tietokantaan. [13.] Esimerkiksi sovelluksessa tallennetaan tällaiseen tietokantaan totuusarvomuuttujan tieto siitä, onko sovelluksessa päällä tumma tila vai ei.

Kaikki Dart Flutterin pakkaukset ovat löydettävissä pub.dev-sivustolta, jossa on myös kattava dokumentaatio pakkausten ominaisuuksista. Pakkauksen voi asentaa projektiin joko komennolla pub get paketin_nimi tai lisäämällä pakettien nimet pubspec.yaml-tiedostoon ja tämän jälkeen tallentamalla tiedoston, jolloin editori suorittaa pub get -komennon automaattisesti.

4.3 Tietokannan toteutus

Kaikki sovelluksen tietokantaan liittyvät operaatiot on toteutettu yhdessä luokassa. Tämä kyseinen luokka on toteutettu Singleton-suunnittelumallia hyödyn-

täen, sillä useamman kuin yhden instanssin luominen tietokantayhteydestä huonontaisi sovelluksen suorituskykyä. Lisäksi suunnittelumallia käyttämällä varmistetaan myös se, että tietokantaan ei ole samanaikaisesti kirjoittamassa useampi instanssi dataa, jolloin tietokantaan ei pitäisi tulla virheellistä dataa. Sovelluksen tietokannassa on kaiken kaikkiaan neljä taulukkoa: User-taulukko käyttäjien profiileja varten, Orders-taulukko tilausten tallettamiseen, OrderRows-taulukko tilausten ja tuotteiden yhdistämiseen keskenään ja Product-taulukko tuotteiden tallettamista varten. Kuvassa 15 on havainnollistettu tietokannan rakenne.



Kuva 15. Sovelluksen tietokannan rakenne.

Tietokanta ja sen taulukot luodaan laitteeseen sovelluksen ensimmäisellä ajokerralla. Sqflite-pakkaus sisältää initDB-metodin, jonka sisällä ensin luodaan instanssi tietokannasta ja tämän jälkeen instanssin avulla voidaan kutsua sqflite-pakkauksen execute-metodia taulujen luomista varten. Execute-metodi ottaa parametrikseen sql-lauseen, ja näin ollen kyseisen metodin avulla taulukot saadaan luotua kirjoittamalla tavanomainen sql-muotoinen CREATE TABLE -lauseke. Sqliten tukemat datatyypit ovat null, integer, real, text ja blob [14]. Blob-datatyyppiä käytetään datan tallettamiseen, mikäli muut datatyypit eivät ole yh-

teensopivia talletettavan datan kanssa. Esimerkiksi sovelluksessa kuvat talletetaan blob-muotoisena tietokantaan. Koodiesimerkissä 7 havainnollistetaan Product-taulukon avulla, miten taulukko luodaan sqlite-tietokantaan.

```
await db.execute(

  "CREATE TABLE Product ("
  "id INTEGER PRIMARY KEY,"
  "name TEXT,"
  "description TEXT,"
  "category TEXT,"
  "price REAL,"
  "picture BLOB"")"

);
```

Esimerkkikoodi 7. Taulukon luonti Sqlite-tietokantaan.

Koodiesimerkin 7 mukaisen execute-metodin lisäksi sqflite-pakkaus tarjoaa myös muita metodeja, kuten update- ja query metodit. Update-metodin avulla voidaan päivittää olioiden sisältämää dataa tietokantaan. Toisin kuin execute-metodi, update-metodi ei ota parametrikseen sql-lausetta, vaan olion sisältämän datan map-tietorakenteessa. Tämän vuoksi tietokannan tauluista täytyy luoda ohjelmakoodiin model-luokat eli esimerkiksi Product-taulukkoa varten luodaan ohjelmakoodiin Product-luokka. Jokaiseen model-luokkaan tulee toteuttaa fromMap- ja toMap-metodit, sillä Sqlite-tietokanta käyttää JSON-muotoista dataa tiedon välittämiseen. Koodiesimerkissä 8 on Product-luokan toMap- ja fromMap-metodit.

```
factory Product.fromMap(Map<String, dynamic> data) {
 return Product(
      data['id'],
       data['name'],
       data['description'],
       data['category'],
       data['price'],
       data['picture']
    );
  }
Map<String, dynamic> toMap() => {

    "id": id,

    "name": name,

    "description": description,

    "category": category,

    "price": price,

    "picture": picture
};
```

Esimerkkikoodi 8. Product-luokan toMap- ja fromMap-metodit.

Sovelluksen kaikkien model-luokkien fromMap- ja toMap-metodit ovat siis toteutukseltaan samankaltaisia koodiesimerkin 8 kanssa. Kun metodit on toteutettu, olion tiedot voi päivittää tietokantaan update-metodilla, ja käyttämällä olion toMap-metodia metodin parametrina. Koodiesimerkissä 9 päivitetään olemassa olevan tuotteen tietoja.

```
updateProduct(Product product) async {

   final db = await database;

   await db.update("Product", product.toMap(), where: "id = ?", where-
Args: [product.id]);
```

Esimerkkikoodi 9. Tuoteolion tietojen päivittäminen.

Koodiesimerkin 9 mukaisessa sqliten update-metodissa parametrina käytettävä tuoteolion toMap-metodi muuttaa olion sisältämän datan ensin JSON-muotoiseksi, ja tämän jälkeen update-metodi päivittää tuoteolion id:n määrittelemän rivin Product-taulukosta tämän JSON-muotoisen datan avulla. Tuoteolion fromMap-metodia puolestaan käytetään tilanteissa, joissa tietokannasta haetaan dataa. Datan hakeminen tietokannasta voidaan toteuttaa sqliten query-metodin

avulla, joka vastaa toiminnallisuudeltaan SQL-kielen SELECT-lausetta. Koodiesimerkissä 10 käytetään sqliten query-metodia kaikkien tuotteiden tietojen hakemiseen.

```
  Future<List<Product>> getAllProducts() async {

   final db = await database;
   List<Map> results = await db.query("Product");
   List<Product> products = results.isNotEmpty ? results.map((e) =>
Product.fromMap(e)).toList() : [];

    return products;
```

Esimerkkikoodi 10. Kaikkien tuotteiden haku tietokannasta.

Koodiesimerkin 10 mukaisesti query-metodin tulos tallennetaan ensin Map-tietorakenteista koostuvaan listaan ja tämän jälkeen fromMap-metodin avulla luodaan Product-luokan instanssit listassa olevan datan perusteella. Product-taulukon päivittämiseen ja tarkasteluun käytetään apuna siis Product-luokan toMap- ja fromMap-metodeja. Taulukkoon uuden rivin lisäämisessä sen sijaan ei hyödynnetä näitä metodeja, vaan siihen käytetään rawInsert-metodia, joka ottaa parametrikseen SQL-kielen INSERT-lauseen. Koodiesimerkissä 11 havainnollistetaan, miten Product-taulukkoon lisätään uusi rivi ohjelmakoodissa.

```
  insertProduct(Product product) async{

    final db = await database;
var maxId = await db.rawQuery("SELECT MAX(id)+1 as last_inserted_id
FROM Product");
var id = maxId.first["last_inserted_id"];
     var result = db.rawInsert(
     "INSERT INTO Product"
       "('id', 'name', 'description', 'category', 'price', 'picture')"
       "values (?, ?, ?, ?, ?, ?)",
       [id, product.name, product.description, product.category, prod-
uct.price, product.picture]

    );

    return result;
```

Esimerkkikoodi 11. Uuden rivin lisääminen Product-taulukkoon.

Kaikki tietokantaluokassa olevat metodit ovat asynkronisia, mikä tarkoittaa sitä, että ohjelmakoodi ei jää erikseen odottamaan vastausta metodilta. Ohjelmakoodin toteutuksen näkökulmasta tämä tarkoittaa sitä, että mikäli asynkronisen metodin paluuarvo halutaan tallettaa muuttujaan, asynkronisen metodin kutsun yhteydessä tulee käyttää await-komentoa. Asynkronisia metodeja voi kutsua vain asynkronisista funktioista. Käyttöliittymän rakentamiseen käytettävä build-metodi ei ole asynkroninen, joten sen sisältä ei voi suoraan kutsua tietokantaluokan metodeja. Asynkronisten metodien paluuarvot ovat Future-tyyppisiä. Esimerkiksi paluuarvona voi olla Future<List<Product>>. Dart Flutter sisältää FutureBuilder-nimisen käyttöliittymäkomponentin, jonka avulla build-metodissa voi kutsua asynkronista metodia, ja metodin paluuarvon data talletetaan snapshot-olioon. Koodiesimerkissä 12 esitetään FutureBuilder-komponentin toimintaa ja snapshot-olion datan käyttämistä käyttöliittymän sisällössä.

```
@override
Widget build(BuildContext context) {

return Scaffold(

body: FutureBuilder(future: SQLiteDbProvider.db.getAllProducts(),
builder: (context,AsyncSnapshot<List<Product>> snapshot) {

if(snapshot.hasData) {
return ListView.builder(itemCount: snapshot.length,
builder: (context, index) {
return(Text(snapshot.data[index].name));

}
else {

return Text(“Something went wrong”);
```

Esimerkkikoodi 12. FutureBuilder-komponentin käyttö build-metodissa.

Esimerkkikoodin 12 mukaisesti Futurebuilder sisältää kaksi parametria: future- ja builder-parametrit. Future-parametriin tulee laittaa arvoksi asynkroninen metodi, jota Futurebuilder-komponentin tulee kutsua. Builder-parametrin arvona on puolestaan anonyymi funktio, johon laitetaan parametriksi build-metodin kontekstiolio sekä snapshot-olio. Koodiesimerkissä 12 snapshot-olio pitää sisällään listan Product-olioita, ja tämän avulla snapshot-olio voi viitata Product-olioiden kenttiin. Koodiesimerkissä rivi snapshot.data[index].name viittaa siis listassa

olevan Product-olion name-kenttään. Toisin sanoen FutureBuilder-komponentin avulla tietokannassa olevan datan pystyy näyttämään käyttöliittymässä yksinkertaisesti ja tehokkaasti.

4.4 Hyödylliset ominaisuudet sovelluksen kehittämisessä

Sovelluksen kehittämisessä Dart Flutterin avulla asetettiin painoarvo teknologian ominaisuuksien havainnollistamiseen. Dart Flutter on teknologiana melko uusi, joten uuteen teknologiaan tutustuessa sovelluksen kehittämisessä saattaa esiintyä haasteita. Dart Flutter kuitenkin sisältää joitain hyödyllisiä ominaisuuksia, joiden ansiosta sovelluksen kehittäminen on helpompaa.

Yksi merkittävimmistä sovelluksen kehittämistä helpottavista ominaisuuksista on ohjelmakoodin debuggauksessa käytettävä Hot Reload -ominaisuus. Hot Reload -ominaisuuden avulla ohjelmakoodia ajettaessa debug-tilassa ohjelmakoodia ei aina tarvitse kääntää kokonaan uudelleen, vaan tämän ominaisuuden avulla sovelluskehys vain käy läpi ohjelmakoodiin tehdyt muutokset, ja muutokset ovat heti näkyvissä sovelluksessa. Tämä nopeuttaa virheiden etsimistä ja niiden korjaamista ohjelmakoodista. Esimerkiksi tekstin asetteluun tehdyt muutokset pystyy nopeasti tällöin havaitsemaan käyttöliittymästä. Kuvassa 16 on näkyvissä editorissa ohjelman ajon aikana näkyvä nappi Hot Reload -ominaisuuden käyttöä varten.

Kuva 16. Editorin palkki, jossa nappi Hot Reload -ominaisuudelle.

Kuvan 16 mukaisessa editorin palkissa salamaikoni on Hot Reload -ominaisuuteen käytettävä nappi. Salamaikonin oikealla puolella olevaa vihreää päivitysnappia käytetään Hot Restart -ominaisuuden käyttämiseen. Erona Hot Reload -toimintoon on se, että tämä toiminto käynnistää koko sovelluksen uudestaan.

Tästä huolimatta Hot Restart -toiminnon käyttö on nopeampi vaihtoehto kuin ohjelmakoodin pysäyttäminen kokonaan ja tämän jälkeen käynnistää ohjelmakoodi manuaalisesti uudelleen.

Toinen merkittävä sovelluksen kehittämistä helpottava ominaisuus on StatefulWidget-luokkien käyttäminen käyttöliittymän rakentamisessa. StatefulWidget-luokat ovat kytköksissä tilan muutoksia käsitteleviin State-luokkiin, jotka tarjoavat build-metodin lisäksi setState-metodin. Tätä kyseistä metodia käyttämällä, kun käyttöliittymään kohdistuu muutoksia, build-metodi laskee, miten sovelluksen tila on muuttunut ja automaattisesti tekee tarvittavat muutokset käyttöliittymään. Tämä helpottaa kehittäjän työtä, sillä tällöin ei tarvitse ottaa huomioon pienen tilan muutoksen aiheuttamia vaikutuksia käyttöliittymään, koska sovelluskehys huolehtii siitä automaattisesti. Lisäksi tilan automaattinen hallinta ennaltaehkäisee käyttöliittymän virheellistä toimintaa.

Näiden edellä mainittujen ominaisuuksien lisäksi yhtenä hyödyllisenä ominaisuutena on se, että sovelluksen tilan hallintaan on tarjolla monia eri vaihtoehtoja. Sovelluksessa käytetään State-luokan setState-metodin lisäksi provider-pakkauksen ominaisuuksia sovelluksen tilan hallintaan. Kyseistä pakkausta voidaan käyttää tilanteissa, joissa useamman luokan tila riippuu tietyn muuttujan arvosta. Esimerkki tällaisesta on sovelluksen käyttöliittymän osien teeman riippuvuus siitä, onko totuusarvomuuttujan arvo true vai false. Tätä varten tulee toteuttaa luokka, joka perii ChangeNotifier-luokan. Tähän luokkaan toteutetaan tarvittavat metodit, joiden avulla esimerkiksi muokataan tietyn muuttujan arvoa halutulla tavalla. Kunkin metodin sisällä tulee kutsua notifyListeners-metodia, jonka avulla muuttujan arvoa seuraavat luokat saavat tiedon muutoksesta. Luokan toteuttamisen jälkeen muuttujan muutoksia voi aloittaa kuuntelemaan käyttämällä build-metodissa ChangeNotifierProvider-komponenttia. Muuttujan arvoja pääsee lukemaan jossain toisessa luokassa käyttämällä Consumer-komponenttia, jonka sisällä voi kutsua provider-luokan metodeja.

Neljäntenä sovelluksen kehittämistä helpottavana ominaisuutena on luvussa 4.3 mainittu FutureBuilder-widget asynkronisten metodien käyttämisessä. Ilman FutureBuilder-komponenttia ainoa mahdollinen paikka kutsua asynkronista metodia build-metodin sisältä olisi käyttämällä nappia anonyymin funktion kutsumiseen. Lisäksi FutureBuilder automaattisesti suorittaa asynkronisen metodin uudelleen, kun build-metodia kutsutaan uudelleen setState-metodin kautta. Tämä puolestaan takaa sen, että käyttöliittymässä näytettävä data on ajan tasalla tietokannassa olevan datan kanssa. FutureBuilder-komponentin avulla voi kutsua samaan aikaan myös useampaa kuin yhtä asynkronista metodia käyttämällä Future.wait-metodia future-parametrissa. Tämä tarkoittaa siis sitä, että kyseisen komponentin avulla tietokannasta saadaan tehokkaasti kaikki tarvittava tieto käyttöliittymän toimintaa varten.

4.5 Haasteet

Dart Flutterilla on siis ominaisuuksia, jotka helpottavat sovelluksen kehittämistä iOS- ja Android-alustoille. Vaikka sovelluksen ominaisuuksien toteuttaminen Dart Flutterilla on yleisesti ottaen melko yksinkertaista, voi sovelluksen kehittämisessä esiintyä joitain haasteita. Osa verkkokauppasovelluksen kehittämisessä ilmenneistä haasteista saatiin ratkaistua kokonaan ja osaan saatiin kehitettyä väliaikainen ratkaisu.

Yksi huomattavimmista haasteista sovelluksen kehittämisen aikana oli se, että Dart Flutter ei tarjoa varsinaisesti käyttöliittymän esikatselua ohjelmakoodin perusteella. Ainoa tapa tutkia, miltä sovelluksen käyttöliittymä näyttää, on käyttää esimerkiksi Android Studion emulaattoria. Tämän vuoksi käyttöliittymän rakentaminen saattaa olla työlästä, sillä mistään ei pysty suoraan katsomaan, kuinka paljon jokin komponentti vie tilaa. Tähän ongelmaan käytettiin ratkaisuna sitä, että ohjelmakoodin kehittämisen yhteydessä ohjelmakoodia ajettiin taustalla, jolloin muutoksia voi testata Hot Reload -ominaisuuden avulla. Lisäksi muutoksia tehtiin pienissä erissä, jotta ohjelmakoodin kaatavat virheet pystyttiin paikantamaan helpommin. Käyttöliittymäkomponenttien asetteluun liittyvät virheilmoituk-

set eivät aina myöskään suoraan kertoneet, mikä käyttöliittymäkomponentti aiheutti virhetilanteen ohjelmakoodissa. Myös tämän vuoksi muutoksia testattiin tiheään tahtiin emulaattorissa.

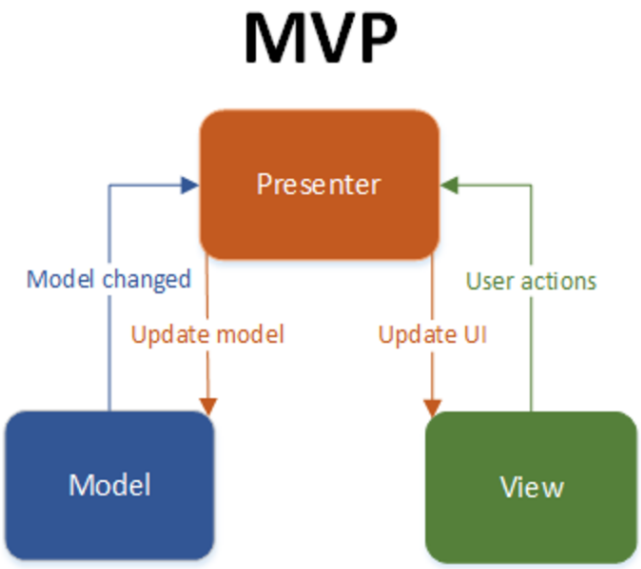
Toinen haaste sovelluksen kehittämisessä oli tilausvälilehden päivittäminen sen jälkeen, kun erillisestä ikkunasta käsin on päivitetty tietokannan dataa. Tavoitteena oli saada listanäkymä tilauksista päivittymään dynaamisesti käyttäjälle, kun hän on tehnyt sovelluksessa uuden tilauksen. Luokkien setState-metodia ei voi kutsua toisesta luokasta, joten tilauslistan päivittäminen setState-metodin avulla ei onnistu. Näkymä, josta tilaus tehdään, avataan ostoskorivälilehdestä, joten näkymät eivät ole suoraan kytköksissä toisiinsa. Toisin sanoen tilauksen vahvistus -näkymä ei pysty suoraan välittämään dataa tilauslistalle. Tätä ongelmaa varten väliaikaisena ratkaisuna toteutettiin päivitysnappi tilauslistan välilehteen. Lopullisena ratkaisuna voisi toimia yhteisen kantaluokan luominen edellä mainittujen kahden näkymän välille, jolloin dataa pystyy välittämään näiden välillä.

Kolmantena haasteena oli Dart Flutterin bugi tiedostosta ladattavien kuvien näyttämiseen. Näkymässä, jossa voi päivittää tuotteen tietoja, voi myös vaihtaa tuotteen kuvaa. Tuotteen olemassa oleva kuva ladataan tietokannasta väliaikaistiedostoon ja tämän jälkeen kuva ladataan tiedostosta Image.file-komponentin avulla. Kuvan vaihtamisen jälkeen ohjelmakoodissa kutsutaan setState-metodia, jonka pitäisi vaihtaa Image.file-komponenttiin uusi kuva. Kun näkymä tuotteen päivittämiseksi avattiin uudelleen, kuvana oli kuitenkin tuotteen vanha kuva. Kävi ilmi, että Image.file-komponentissa on bugi, joka aiheuttaa sen, että setState-metodi ei aina päivitä kyseisen komponentin dataa. Tähän ongelmaan käytettiin ratkaisuna välimuistin tyhjentämistä kuvan päivittämisen yhteydessä komennolla: imageCache.clear().

5 Jatkokehitysmahdollisuudet

Dart Flutter soveltui teknologiana siis hyvin sovelluksen ominaisuuksien toteuttamiseen. Teknologia on ominaisuuksiltaan monipuolinen, ja tämä saatiin havainnollistettua myös sovelluksessa. Sovellus on tyyliltään demotyyppinen sovellus. Tämän vuoksi sovellus sisältää ominaisuuksia, joita voisi selkeästi vielä kehittää eteenpäin.

Sovelluksen tavoitteiden vuoksi sovelluksen ohjelmakoodin rakenteessa ei sovellettu mitään yleisesti tunnettua arkkitehtuuria, joten tämä on yksi sovelluksen kehityskohdista. Yhtenä arkkitehtuurina sovellukseen voisi käyttää MVP-arkkitehtuuria, jonka nimi tulee sanoista: model, view ja presenter. Tässä arkkitehtuurissa presenter-komponentti toimii välikappaleena model- ja view-komponenttien datan päivittämiseen. Kuvassa 17 on havainnollistettu MVP-arkkitehtuurin toimintaperiaate.



Kuva 17. MVP-arkkitehtuurin toimintaperiaate.

Kuvan 17 mukaisesti MVP koostuu kolmesta eri komponentista: model-, view - ja presenter-komponenteista. Model-komponentti on rajapinta käyttöliittymässä

näytettävän datan määrittämiselle. View-komponentti on puolestaan passiivinen rajapinta model-komponentin datan näyttämiselle ja käyttäjän tekemien toimintojen presenter-komponentille välittämiselle. Presenter-komponentti puolestaan välittää dataa model- ja view-komponenttien välillä siten, että view välittää käyttäjän toimet presenterille, joka puolestaan päivittää model-komponenttia saamansa datan perusteella. Model-komponentti välittää päivitetyn datan presenter-komponentille, jotta data voitaisiin näyttää view-komponentissa. [15.] MVP-arkkitehtuuri tekee ohjelmakoodista modulaarisempaa ja toteuttaa rajapintaa vasten ohjelmointia, jolloin koodista tulee uudelleenkäytettävämpää. Lisäksi ohjelmakoodi on yleensä tällöin siistimpää.

Toinen merkittävistä kehityskohteista on sovelluksen tietokannan toteutus. Tällä hetkellä sovellus käyttää paikallista SQLite-tietokantaa laitteella tietojen tallettamiseen. Sovelluksen käytännöllisyyden kannalta olisi kuitenkin mielekkäämpää, että sovellus käyttäisi verkossa olevaa tietokantaa, jotta talletetut tiedot pysyisivät tallessa esimerkiksi sovelluksen uudelleenasennuksen yhteydessä. Tämän toteuttamiseen yhtenä vaihtoehtona on Googlen Firestore-tietokanta, joka on pilvipohjainen NoSQL-tietokanta. Etuna tässä olisi se, että Dart Flutterille on kehitetty pakkaukset, joiden avulla sovelluksen voi yhdistää Firestore-tietokantaan. Firestoren käyttöönotto vaatii firebase_core -ja cloud_firestore -pakettien lisäämisen ohjelmakoodiin sekä käyttäjätilin luonnin Googlen Firebase-alustalle. Lisäksi tietokanta tulee luoda Firebase-alustan kautta. Kaiken kaikkiaan tämä tarkoittaa siis sitä, että sovelluksen tietokantaluokan toteutus tulisi muuttumaan huomattavasti. Vaihtoehtona Firestore-tietokannan käyttämiselle olisi käyttää esimerkiksi MySQL-tietokantaa omalla palvelimella, johon sovellus yhdistettäisiin. Dart Flutterilla on pakkaus myös MySQL-tietokantoja varten, joten tietokannan toteuttaminen tällä tavalla pitäisi myös onnistua. Tämä vaihtoehto kuitenkin vaatii myös sen, että PHP:llä ohjelmoidaan palvelimen logiikka sovelluksen pyyntöjen käsittelyyn.

6 Yhteenveto

React Native ja Dart Flutter sisältävät molemmat sellaisia ominaisuuksia, joiden ansiosta Android -ja iOS-alustoille pystyy kehittämään vakaasti toimivan sovelluksen. Kilpailevina teknologioina React Nativen ja Dart Flutterin välillä on omat eroavaisuutensa sekä näiden tuomat edut ja rajoitteet. React Nativen etuna sovelluskehityksessä Dart Flutteriin nähden on se, että kyseinen teknologia käyttää alustojen natiiveja käyttöliittymäkomponentteja käyttöliittymän renderöinnissä. React Nativen heikkoutena on puolestaan siltakomponentin käyttö natiiveja komponentteja varten, jonka vuoksi sovelluksen suorituskyky heikkenee monimutkaisissa käyttöliittymärakenteissa. Dart Flutterin etuna on juuri tämä parempi suorituskyky React Nativeen nähden, sillä se käyttää Material- ja Cupertino-kirjastojen komponentteja käyttöliittymässä natiivien komponenttien sijasta. Dart Flutterin rajoitteena on puolestaan se, että 3D-grafiikkaa ei tueta tällä hetkellä, joka rajoittaa sovelluksenkehittämismahdollisuuksia. React Nativen ja Dart Flutterin yhteisenä etuna natiiveihin teknologioihin nähden on se, että niiden avulla on mahdollista toteuttaa sovellus kummallekin alustalle pelkästään yhden lähdekoodin pohjalta. Lisäksi ohjelmakoodin ylläpito on helpompaa, sillä ohjelmakoodista tarvitaan vain yksi versio molempia alustoja varten. Näiden teknologioiden rajoitteena natiiveihin teknologioihin nähden on se, että niillä on rajoitettu pääsy alustojen omiin ominaisuuksiin sekä palveluihin. Toisin sanoen kehittämällä sovelluksen natiivilla teknologialla, sovellus pääsee käsiksi alustojen uusimpiin toimintoihin sekä sovelluksen suorituskyky on todennäköisesti parempi kuin monialustaisella teknologialla.

Insinöörityön aikana kehitetyn sovelluksen avulla pyrittiin tutustumaan Dart Flutterin tarjoamiin ominaisuuksiin sekä selvittämään, kuinka paljon alustariippuvaista määrittelyä ja ohjelmakoodia sovellus tarvitsee. Dart Flutterin avulla saatiin toteutettua kaikki sovellukseen suunnitellut ominaisuudet. Sovelluksen laitekohtaisia ominaisuuksia ovat kameran käyttö kuvia varten sekä tiedostojärjestelmän käsittely tiedostojen tallentamista ja lukemista varten. Sovelluksen avulla saatiin selkeästi havainnollistettua se, että tuetut laite- ja alustakohtaiset

ominaisuudet toimivat hyvin Dart Flutterilla kehitetyssä sovelluksessa. Sovelluksen avulla saatiin myös havainnollistettua tarvittavan alustakohtaisen määrittelyn määrä. Sovelluksen ohjelmakoodissa ei tarvinnut kertaakaan ohjelmallisesti tarkistaa ajonaikaista alustaa, ja alustariippuvainen määrittely rajoittui vain käyttöoikeuksien määrittämiseen kameran ja tiedostojärjestelmän käyttöä varten. Toisin sanoen suurin osa Dart Flutterin ominaisuuksista ovat yhteensopivia kummankin alustan kanssa, joka puolestaan helpottaa sovelluksen kehittämistä entistä enemmän.

React Native ja Dart Flutter ovat teknologioina jatkuvasti kehittyviä, ja tulevat varmasti jatkossa haastamaan natiiveja teknologioita yhä enemmän. Tällä hetkellä kuitenkin valinnan natiivin teknologian ja monialustaisen teknologian välillä määrittää seuraavat kysymykset: Kuinka nopeasti sovellus täytyy kehittää kummallekin alustalle? Kuinka iso rooli suorituskyvyllä sekä laite- ja alustakohtaisilla ominaisuuksilla on sovelluksessa? Mikäli aikamääre sovelluksen kehittämiselle molemmille alustoille on tiukka, on kannattavampaa valita teknologiaksi joko React Native tai Dart Flutter. Mikäli puolestaan kaikista suurin painoarvo sovelluksessa on suorituskyvyllä tai laite- ja alustakohtaisilla ominaisuuksilla, kannattaa valita natiivi teknologia sovelluksen toteuttamiseen. Voidaankin siis todeta, että oikea teknologia sovelluksen kehittämiseen Android - ja iOS-alustoille riippuu itse ohjelmistokehitysprojektin ominaisuuksista.

Lähteet

1 Techopedia.com. Verkkoaineisto. https://www.techopedia.com/definition/27568/native-mobile-app. Luettu 3.9.2021.

2 Learning React Native, 2nd Edition, Eisenman Bonnie, Metropolia Finna E-kirja.

3 ReactJS.org. Verkkoaineisto. https://reactjs.org/docs/introducing-jsx.html. Luettu 5.9.2021.

4 Wikipedia. Verkkoaineisto. https://en.wikipedia.org/wiki/Dart_(programming_language). Luettu 7.9.2021.

5 Javatpoint.com. Verkkoaineisto. https://www.javatpoint.com/flutter-dart-programming. Luettu 7.9.2021.

6 Flutter.dev. Verkkoaineisto. https://flutter.dev/docs/resources/faq. Luettu 7.9.2021.

7 Wikipedia. Verkkoaineisto. https://en.wikipedia.org/wiki/Flutter_(software). Luettu 7.9.2021.

8 Medium.com. Verkkoaineisto. https://medium.com/walmartglobaltech/native-vs-cross-platform-322e9896e745. Luettu 9.9.2021.

9 Simform.com. Verkkoaineisto. https://www.simform.com/blog/flutter-vs-native-technology/. Luettu 9.9.2021.

10 Flutter.dev. Verkkoaineisto. https://flutter.dev/docs/resources/architectural-overview. Luettu 19.9.2021.

11 Tutorialspoint.com. Verkkoaineisto. https://www.tutorialspoint.com/react_native/react_native_state.htm. Luettu 25.9.2021.

12 TheDroidsOnDroids.com. Verkkoaineisto. https://www.thedroidsonroids.com/blog/flutter-vs-react-native-what-to-choose-in-2021. Luettu 27.9.2021.

13 Pub.dev. Verkkoaineisto. https://pub.dev/. Luettu 5.10.2021.

14 Sqlite.org. Verkkoaineisto. https://www.sqlite.org/datatype3.html. Luettu 8.10.2021.

15 Medium.com. Verkkoaineisto. https://medium.com/codechai/the-mvp-architecture-pattern-in-flutter-with-simple-demo-65ab3282c54b. Luettu 8.10.2021.