jamk

Cross-Platform-sovelluskehitys

Jonne Okkonen

Opinnäytetyö, AMK
Toukokuu 2023
Insinööri (AMK), Tieto- ja viestintätekniikan tutkinto-ohjelma

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Okkonen, Jonne

Cross-Platform-sovelluskehitys

Jyväskylä: Jyväskylän ammattikorkeakoulu. Toukokuu 2023, 30 sivua.

Insinööri (AMK). Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Verkkojulkaisulupa myönnetty: kyllä

Tiivistelmä

Tarkoituksena oli tutkia Cross-Platform-sovelluskehitystä, sekä siihen liittyviä alan kehityssuuntauksia. Tutkimuksen taustalla oli kasvava trendi Cross-Platform-sovelluskehityksen käytöstä ja tarve selvittää, onko se todella tehokkaampi ratkaisu verrattuna perinteiseen alustakohtaiseen kehitykseen teollisessa kehityksessä tulevaisuuden markkinoilla. Aiheen valintaan vaikutti myös oma mielenkiinto aiheeseen ja halu oppia uutta. Tavoitteena oli löytää vastaus siihen, mitkä ovat Cross-Platform-sovelluskehityksen hyödyt ja haitat, rajoitteet, mahdollisuudet ja minkälaiselle yritykselle tai projektille se soveltuu parhaiten.

Tutkimus toteutettiin kartoittamalla ja ristiinvertailemalla eri teknologioiden sovelluskehitysprosessia alusta loppuun saakka yleisellä tasolla hyödyntämällä Flutter- ja Electron.js-teknologioita erilaisille alustoille, kuten Android, iOS, Web, Windows, MacOS ja Linux. Jokaisessa teknologiassa syvennyttiin myös hieman enemmän johonkin tiettyyn aiheeseen. Flutterissa tämä oli esimerkiksi alustakohtaiset komponentit ja Electronissa arkkitehtuuri. Tuotteena kehitettiin muutama esimerkkiprojekti, joita käytettiin alustoihin tutustumiseen ja käytännön esimerkkeinä teorian vastapainona.

Johtopäätöksenä voidaan todeta, että Cross-Platform-sovelluskehitys on lupaava menetelmä, jolla on useita etuja, kuten yhden koodipohjan käyttö kaikille alustoille, nopeampi kehitysprosessi ja alhaisemmat kehityskustannukset. Pienet ja keskisuuret yritykset hyötyvät niistä eniten kehityskustannusten ja -ajan säästön vuoksi. Electron.js soveltuu erityisesti työpöytäsovellusten kehittämiseen ja Flutter on loistava alusta sekä työpöytä- että mobiilisovellusten kehittämiseen. Haittoja ovat alustojen väliset erot, suorituskyvyn heikkeneminen, rajalliset mahdollisuudet mukauttamiseen ja puutteet laitetuen suhteen. On kuitenkin tärkeää harkita tarkkaan, minkä kehitysmenetelmän valitsee ja varmistaa, että sen ominaisuudet vastaavat projektin vaatimuksia.

Avainsanat (asiasanat)

Cross-Platform-sovelluskehitys, Flutter, Electron.js, Android, iOS, Windows, MacOS, Linux, .NET MAUI, React Native

jamk

Description

Okkonen Jonne

Cross-Platform App Development

Jyväskylä: JAMK University of Applied Sciences, May 2023, 30 pages.

Degree Programme in Information and Communication Technology. Bachelor´s thesis.

Permission for web publication: Yes

Language of publication: Finnish

Abstract

The purpose of the study was to investigate Cross-Platform application development and related industry trends. The background for the research was the growing trend of using Cross-Platform application development and the need to determine if it is truly a more effective solution compared to traditional platform-specific development in industrial development for future markets. The topic was also chosen due to personal interest in the subject and a desire to learn something new. The goal was to find out what the benefits and drawbacks, limitations, opportunities, and which type of company or project is best suited for Cross-Platform application development.

The study was conducted by mapping and cross-comparing the application development process of various technologies from start to finish on a general level, utilizing Flutter and Electron.js technologies for different platforms such as Android, iOS, Web, Windows, MacOS, and Linux. In each technology, a specific topic was also explored in more depth. For example, platform aware widgets in Flutter and architecture in Electron.js. Several sample projects were developed and used to familiarize oneself with the platforms and as practical examples to balance the theory.

In conclusion, Cross-Platform application development is a promising method with several advantages, such as using a single codebase for all platforms, a faster development process, and lower development costs. Small and medium-sized enterprises benefit the most from these cost and time-saving benefits. Electron.js is particularly suitable for developing desktop applications, while Flutter is an excellent platform for both desktop and mobile application development. The drawbacks include platform differences, decreased performance, limited customization options, and shortcomings in hardware support. However, it is essential to carefully consider which development method to choose and ensure that its features meet the project's requirements.

Keywords/tags (subjects)

Cross-Platform App Development, Flutter, Electron.js, Android, iOS, Windows, MacOS, Linux, .NET MAUI, React Native

Sisältö

1 Johdanto ... 4
1.1 Taustaa ... 4
1.2 Tavoitteet ... 4
1.3 Tutkimusasetelma ... 4
2 Cross-Platform-sovelluskehitys ... 5
2.1 Yleistä ... 5
2.2 Hyödyt ja haitat ... 5
2.3 Business näkökulma ... 6
3 Electron.js ... 6
3.1 Yleistä ... 6
3.2 Uuden projektin aloitus ... 7
3.3 Web sovelluksen muuntaminen työpöytäsovellukseksi ... 9
3.4 Arkkitehtuuri ... 9
3.4.1 Prosessit ... 10
3.4.2 Rajapinnat ... 13
3.5 Sovelluksen julkaiseminen ... 13
3.6 Yhteenveto ... 14
4 Flutter ... 15
4.1 Yleistä ... 15
4.2 Uuden projektin aloitus ... 16
4.3 Kirjastojen käyttö ... 16
4.4 Käyttöliittymä rakenne ... 16
4.5 Alustatietoiset widgetit ... 19
4.5.1 Johdanto ... 19
4.5.2 Ulkoasu ... 20
4.5.3 Koodi ... 22
4.5.4 Yhteenveto ... 24
4.6 Tiedon tallentaminen ... 24
4.7 Sovelluksen julkaiseminen ... 26
4.8 Yhteenveto ... 27
5 Muita sovelluskehyksiä ... 27
5.1 .NET MAUI ... 27
5.2 React Native ... 28

6 Johtopäätökset 28

Lähteet 30

Kuviot

Kuvio 1. Electron.js index.js-tiedosto 8

Kuvio 2. Electron.js:n prosessikuva 10

Kuvio 3. Context Bridgen käyttö main-prosessissa 12

Kuvio 4. Rajapintojen käyttö Context Bridgellä renderer-prosessista 12

Kuvio 5. Electron-sovellus ja Context Bridgen välittämä konsoli viesti main-prosessilta 12

Kuvio 6. Electron Builder build-parametrit package.json-tiedostossa 13

Kuvio 7. Muistio esikatselunäkymän käyttöliittymän puurakenne 17

Kuvio 8. Käytettävän FormFactorin tunnistaminen koodissa 18

Kuvio 9. Simple notes työpöytänäkymä 18

Kuvio 10. Simple notes mobiilinäkymät 19

Kuvio 11. Simple notes -sovelluksen muistiolistaus Android- ja iOS-alustoilla 20

Kuvio 12. PlatformPageRoute alustakohtainen toteutus 21

Kuvio 13. Simple notes -sovelluksen muistion esikatselunäkymä Android- ja iOS-alustoilla 22

Kuvio 14. Abstrakti PlatformWidget-luokka esimerkki 22

Kuvio 15. Alustakohtainen AppBar-luokka 23

Kuvio 16. Notes-taulun luominen SQLitellä 25

Kuvio 17. SQLite-tietokannan avausfunktio 25

Kuvio 18. Muistiinpanon lisäys SQLite-tietokantaan 26

Kuvio 19. Kaikkien muistiinpanojen hakeminen tietokannasta 26

Taulukot

Taulukko 1. Context Bridgen tukemat tietotyypit ja niihin liittyvät rajoitukset 11

Lyhenteet

.NET	Microsoftin kehittämä kehitysalusta (luetaan “dot net”)
API	Application Programming Interface
ASAR	Electronin käyttämä yksinkertainen arkistointimuoto
CSS	Cascading Style Sheets
FAB	Floating Action Button (Android)
HTML	Hypertext Markup Language
ID	Identifier
IPC	Inter Process Communication
JS	JavaScript
JSON	JavaScript Object Notation
NPM	Node Package Manager
REST	Representational state transfer
SDK	Software Development Kit
SPA	Single Page Application
SQL	Structured Query Language
UI	User Interface
XAML	eXtensible Application Markup Language

1 Johdanto

1.1 Taustaa

Cross-Platform-sovellukset ovat nousseet viime vuosina yhä suositummaksi ratkaisuksi mobiili- ja desktop-sovellusten kehittämiseen, kun yhä useammat yritykset ja organisaatiot pyrkivät tavoittamaan käyttäjäkuntaansa useilla eri alustoilla, kuten mobiililaitteilla, tietokoneilla ja tableteilla. Tämä on johtanut teknologioiden, kuten Flutterin ja Electron.js:n kasvavaan käyttöön, sillä ne mahdollistavat useiden käyttäjien tavoittamisen eri alustoilla ilman tarvetta kehittää erillisiä sovelluksia jokaiselle alustalle erikseen. Cross-Platform teknologiaratkaisujen ajatellaan säästävän merkittävästi aikaa ja resursseja ja tavoitteena on tutkia ja selvittää, pitääkö se paikkaansa. Tämä säästää merkittävästi aikaa ja resursseja. Cross-Platform-sovellukset tarjoavat myös yhtenäisen käyttökokemuksen kaikilla alustoilla, mutta se voi myös aiheuttaa haasteita, kuten suorituskyvyn, tehokkuuden sekä käyttöliittymän ja toiminnallisuuksien rajoituksia. Tämä voi johtaa erilaisiin ongelmiin, joten ne on hyvä ottaa huomioon jo suunnitteluvaiheessa.

1.2 Tavoitteet

Tämän opinnäytetyön tavoitteena on tutkia Cross-Platform-sovellusten kehittämistä teorian ja käytännön tasolla sekä tuoda mukaan ohjelmistokehityksen Business-näkökulmaa. Esimerkkiprojektien avulla vastataan seuraaviin kysymyksiin: Mitä hyötyjä ja haittoja, ohjelmistorajapintoja ja rajoitteita Cross-Platform-sovelluksilla on, miksi Cross-Platform-sovellukset yleistyvät, millaisiin projekteihin ja minkä kokoisten yritysten tarpeisiin sovellukset soveltuvat, millainen on Cross-Platform-sovelluksien suorituskyky, kehitysaika ja elinkaari verrattuna natiivi sovelluksiin. Opinnäytetyön käytännön osiossa tutkitaan teknologian vaatimuksia, projektin luontiprosessia, sisäänrakennettujen ja kolmannen osapuolten kirjastojen käyttämistä, käyttöliittymän luontityyliä ja sen eroja, ohjelmistorajapinta esimerkkejä, sovelluksen julkaisua ja paketointia. Lopuksi kootaan yhteen teknologioiden eroavaisuudet ja koostetaan yhteenveto aiemmin käsitellyistä vaiheista.

1.3 Tutkimusasetelma

Opinnäytetyössä käytetään Flutteria ja Electron.js:ää teknologioiden vertailtavuuden ja käytännön sovellusprojektien toteutuksen vuoksi. Nämä valittiin myös aiemman kokemuksen ja mielenkiinnon pohjalta. Tutkimus toteutetaan esimerkkiprojektien avulla, joita on käytetty vastaamaan yllä

mainittuihin tutkimuskysymyksiin. Esimerkkiprojektien toteutuksessa huomioidaan sovellusten suorituskyky, kehitysaika ja elinkaari sekä käyttöliittymän suunnittelu. Tutkimusaineisto kerätään esimerkkiprojektien kehityksen aikana ja analyysit suoritetaan ristiinvertailemalla Flutteria ja Electron.js:ää keskenään. Lopullisessa raportissa esitetään yhteenveto aineiston perusteella saaduista tuloksista ja vastataan tutkimuskysymyksiin.

2 Cross-Platform-sovelluskehitys

2.1 Yleistä

Cross-Platform-sovellusten määrä on kasvanut mobiilisovellusten suosion mukana ja nykyisin moni yritys haluaakin tehdä oman mobiilisovelluksen ja harkitsee eri vaihtoehtoja sen toteuttamiseksi. Cross-Platform-sovellus on yksi näistä vaihtoehdoista ja sillä on paljon etuja, mutta myös rajoitteita. Yksi tämän teknologian suurimmista hyödyistä on varmasti yhteinen koodipohja, joka mahdollistaa sovelluksen kirjoittamisen vain kerran. Se mahdollistaa sen, että pystytään luomaan nopeasti ja kustannustehokkaasti sovellus, joka toimii kaikilla suosituilla ympäristöillä. Tästä hyötyvät etenkin startup-henkiset yritykset, joille on tärkeää toimia kustannustehokkaasti ja päästä markkinoille mahdollisimman nopeasti. Myös ylläpidon helppous on yksi merkittävä etu, kun sovellus on kirjoitettu kerran ja julkaistu eri alustoille, sen ylläpito on huomattavasti helpompaa kuin, jos yritys joutuisi ylläpitämään useita erilaisia natiiveja sovelluksia.

2.2 Hyödyt ja haitat

Asioita, joita pitää kuitenkin ottaa huomioon valitessa natiivi ja Cross-Platform-sovellusten välillä on esimerkiksi: suorituskyky, kehitysaika, kustannukset, sovelluksen elinkaaren pituus, hardware resurssien tärkeys ja taustaprosessien tarve. Native teknologioilla päästään parempaan suorituskykyyn, kaikkiin hardware resursseihin ja käyttämään taustaprosesseja, mutta myös kehitysaika ja kustannukset ovat suuremmat kuin Cross-Platform-sovelluksella. Sovellusta rajoittaa myös se, että päästään käsiksi vain tiettyihin hardware-komponentteihin ja taustaprosesseja ei pystytä tekemään ollenkaan, mutta native sovelluksiin verrattuna saadaan aikaan huomattavia säästöjä kehitysajan ja kustannuksien suhteen. Sovelluksen elinkaaren pituus kannattaa myös ottaa huomioon. Mikäli tekeillä on hyvin pitkän aikavälin projekti, kannattaa harkita native teknologioiden käyttöä, mutta jos kyseessä on lyhyemmän aikavälin projekti, niin silloin kannattaa harkita Cross-Platformin hyödyntämistä.

2.3 Business näkökulma

Cross-Platform-sovellukset ovat hyödyllisiä silloin, kun tavoitteena on nopea ja kustannustehokas kehitysprosessi, joka mahdollistaa sovelluksen nopean lanseerauksen markkinoille. Tämän mahdollistaa se, että Cross-Platform sovelluskehitys vähentää kehittäjien tarvetta keskittyä teknisiin yksityiskohtiin ja mahdollistaa sen sijaan keskittymisen sovelluksen ominaisuuksien kehittämiseen. Yritysten on kuitenkin tärkeää ottaa huomioon sovelluksen tarpeet ja vaatimukset, kun valitaan Native- ja Cross-Platform teknologioiden välillä. Jos sovelluksen on tarkoitus olla pitkäaikainen projekti tai se vaatii paljon hardware resursseja ja taustaprosesseja, kannattaa harkita Native-teknologioiden käyttöä. Toisaalta, jos tavoitteena on lyhytaikainen projekti, jossa halutaan säästää aikaa ja rahaa, Cross-Platform teknologioiden käyttö voi olla erittäin kannattavaa. Lopullinen valinta riippuu projektin tavoitteista ja vaatimuksista.

3 Electron.js

3.1 Yleistä

Electron.js on avoimeen lähdekoodiin perustuva sovelluskehys, jota käytetään Cross-Platform-sovellusten kehitykseen Windows-, MacOS- ja Linux-ympäristöissä. Electron.js käyttää hyödykseen Chromium pohjaista selainta ja Node.js toimiakseen näissä eri ympäristöissä. Juuri näiden teknologioiden ansiosta sovelluskehitystä voidaankin tehdä millä tahansa web-teknologialla. Yksinkertaisimmillaan sovelluksia voidaan kuitenkin tehdä HTML:llä, CSS:llä ja puhtaalla JavaScriptillä tai niiden sijasta voidaan hyödyntää suosittuja SPA teknologioita, kuten Reactia, Angularia tai Vue.js. Myös nykyisten web-sovellusten muuntaminen työpöytäsovellukseksi on tehty erittäin helpoksi, koska mikä tahansa web-teknologioilla tehty sovellus voi toimia Electronissa. Web-sovellusten lisäksi myös Node.js pohjaisia sovelluksia voidaan hyödyntää suoraan Electronissa. (What is Electron.js 2020).

Electron.js on alun perin kehittänyt GitHubille työskentelevä insinööri Cheng Zhao. Hän päätyi luomaan nykyisen Electron.js kun hän työskenteli projektissa, jonka tarkoituksena oli kehittää Cross-Platform versiota Atom-tekstieditorista, joka toimi vain MacOS-käyttöjärjestelmällä. Aluksi tekstieditoria lähdettiin kehittämään node-webkitin avulla, mutta arkkitehtuuri ongelmien ja teknologian alkuperäisen kehittäjän vastustuksen takia, hän päätyi kirjoittamaan node-webkitin uudestaan. Lopulta syntyi nykyinen Electron.js. Atom olikin ensimmäinen Electron-sovellus ja pian

lähdekoodin julkistamisen jälkeen sen suosio alkoi kasvamaan. (Zhao 2016). Nykyisin Electron.js hyödyntää esimerkiksi Microsoftin Visual Studio Code ja Teams, Twitch, Facebook Messenger, Slack sekä Discord (Perkaz 2020).

Electron sovelluksilla on käytössään selaimissa käytettyjen rajapintojen lisäksi myös normaalisti selaimelta estettyjä rajapintoja. Näiden rajapintojen käyttäminen tapahtuu Node.js:n kautta, josta myöhemmin lisää arkkitehtuuri osiossa. Electron.js:n ansiosta sovelluksella on esimerkiksi täysi pääsy käyttöjärjestelmän tiedostojärjestelmään ja monisäikeistykseen, ikkunan hallintaan, käyttöjärjestelmä kohtaisiin native ominaisuuksiin kuten (Windows Taskbar, MacOS Dock, valikot, dialogit, jne), Bluetooth:iin, näytönkaappaukseen ja moniin muihin. (What is Electron.js 2020; Electron.js docs N.d.). Kattavien rajapintojen lisäksi Electron.js mahdollistaa myös web-teknologioiden ansiosta esimerkiksi: helpon videon ja kuvan näyttämisen, hyvien UI-suunnittelu työkalujen käytön ja monia muita hyvin monipuolisia ominaisuuksia natiivi teknologioihin verrattuna, kuten koneoppimisen Tensorflow.js-kirjaston avulla (Perkaz 2020).

Sovellus koostuu kahdesta prosessista, jotka molemmat käyttävät Chromiumin multiprocess arkkitehtuuria omissa säikeissään. Molemmilla prosesseilla on omat tehtävänsä, näistä ensimmäinen pääprosessi vastaa sovelluksen käynnistämisestä ja elinkaaresta, kuten käynnistys, sammutus, sammutukseen valmistuminen, jne. Näiden lisäksi prosessi mahdollistaa pääsyn native rajapintoihin. Pääprosessin tehtävät tulisi pitää hyvin kevyinä. Toinen prosessi on pääprosessin käynnistämä renderöinti prosessi, jolla on käytössään web-rajapinnat ja tämän prosessin vastuulla on ajaa itse sovellusta. Renderöinti prosesseja voi olla myös useampi, kun taas pääprosesseja on vain yksi. (Process Model N.d.). Prosessien välinen kommunikointi tapahtuu kaksisuuntaisen Context Bridgen avulla, joka on mahdollistaa tietoturvallisen pääsyn native rajapintoihin. Lisää tästä Arkkitehtuuri osiossa.

3.2 Uuden projektin aloitus

Electron.js on rakennettu Node.js:n päälle, joten se täytyy ensin asentaa, jotta uuden projektin voi aloittaa. Viimeisimmän version saa ladattua heidän virallisilta verkkosivustoiltaan. Node.js:n mukana tulee myös npm-paketinhallintatyökalu, jota käytetään projektin pakettienhallintaan. Kun

nämä vaatimukset täyttyvät, voidaan käyttää `npm init` komentoa, joka luo package.json-tiedoston, joka sisältää tietoa projektista ja sen riippuvuuksista. Electron.js-paketin asentamiseksi ajetaan `npm install electron --save-dev ` komento.

Paketin asennuksen jälkeen voidaan luoda entry-point-tiedosto applikaation main-prosessille kuvion 1 mukaisesti. Ensimmäisellä rivillä importoidaan Electron-moduuli, jonka jälkeen määritellään createWindow-funktio, joka nimensä mukaisesti luo sovelluksen ikkunan ja lataa vakiona index.html-tiedoston näkymään. Funktion BrowserWindow-luokan parametreja muokkaamalla voidaan vaikuttaa ikkunaan. Tässä tapauksessa ikkunalle asetetaan alustava korkeus (800) ja leveys (600), sekä lisäksi otetaan käyttöön node integraatio, joka mahdollistaa renderer-prosessin pääsyn Node.js:n rajapintoihin.

```
JS index.js > ...
1   const { app, BrowserWindow } = require('electron')
2
3   function createWindow () {
4     const win = new BrowserWindow({
5       width: 800,
6       height: 600,
7       webPreferences: {
8         nodeIntegration: true
9       }
10    })
11
12    win.loadFile('index.html')
13  }
14
15  app.whenReady().then(() => {
16    createWindow()
17
18    app.on('activate', () => {
19      if (BrowserWindow.getAllWindows().length === 0) {
20        createWindow()
21      }
22    })
23  })
24
25  app.on('window-all-closed', () => {
26    if (process.platform !== 'darwin') {
27      app.quit()
28    }
29  })
```

Kuvio 1. Electron.js index.js-tiedosto

Seuraavaksi kuunnellaan applikaation lifecycle-eventtejä app-objektia käyttämällä. Kun applikaatio on valmis, luodaan ikkuna kutsumalla aiemmin luotua createWindow-funktiota. Ready-eventin lisäksi luodaan myös ikkuna activate-eventin yhteydessä, mikäli ikkunaa ei ole jo avoinna. Activate-event vastaanotetaan, kun käyttäjä klikkaa sovelluksen kuvaketta dockissa (macOS) tai taskbarissa

(Windows). Lopuksi vielä kuunnellaan `window-all-closed`-eventtiä, joka vastaanotetaan sovelluksen sammuessa. Tällöin sovellus voidaan sulkea, pois lukien macOS, jossa tyypillisesti sovellus jätetään pyörimään taustalle, vaikka ikkunoita ei olekaan avattuna.

3.3 Web sovelluksen muuntaminen työpöytäsovellukseksi

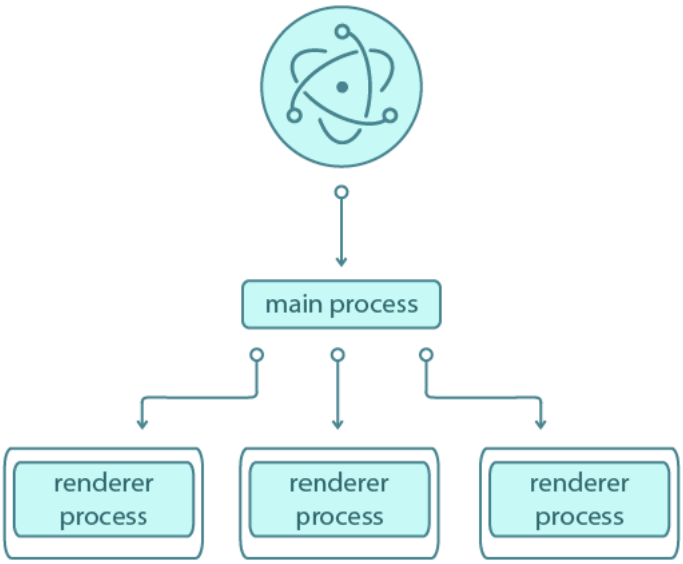
Olemassa olevan web-sovelluksen pystyy muuntamaan työpöytäsovellukseksi hyvinkin helposti Electron.js:n avulla, koska se käyttää Chromium pohjaista selainta sovelluksen ajamiseen. Web-sovelluksen tuominen aloitetaan luomalla uusi Electron.js-projekti Uuden projektin luonti luvussa kuvatulla tavalla. Kun uusi projekti on luotu, voidaan kopioida olemassa olevan web-sovelluksen lähdekoodi projektihakemistoon. Tärkeää on muistaa tuoda kaikki HTML, CSS, JavaScript ja muut sovelluksen vaatimat materiaalit projektiin. Tiedostojen tuomisen jälkeen varmistetaan, että Electronin pääprosessi osaa ladata oikean HTML-tiedoston, jotta sovellus latautuu oikein käynnistyksen jälkeen. Tämän jälkeen voidaan tehdä ensimmäinen testaus ja alkaa suunnittelemaan ja tekemään mahdollisia muutoksia tai optimointeja Electron.js-ympäristöön. Mikäli sovelluksessa halutaan hyödyntää Electronin tarjoamia rajapintoja selaimen tarjoamien suppeampien rajapintojen lisäksi, täytyy renderer-prosessin kommunikoida pääprosessin kanssa. Tästä vielä tarkemmin seuraavassa Arkkitehtuuri osiossa, joka kuvaa tarkemmin, kuinka tämä käytännössä tapahtuu.

3.4 Arkkitehtuuri

Electron.js:n arkkitehtuuri perustuu pääasiassa Chromium selaimeen ja Node.js:ään. Nämä mahdollistavat web-teknologioiden hyödyntämisen sovellusten kehityksessä, kuten HTML, CSS ja JavaScript. Näiden lisäksi voidaan ottaa käyttöön myös muita suosittuja web-kehitys teknologioita, kuten React tai Angular. Esimerkiksi Microsoftin Visual Studio Code on toteutettu Electron.js ja Angularilla.

3.4.1 Prosessit

Prosessit jakautuvat kahteen osaan main-prosessiin ja renderer-prosesseihin kuvion 2 mukaisesti. Main-prosesseja on vain yksi ja sen vastuulla on sovelluksen ikkunoiden luominen ja hallinnointi sekä järjestelmän tapahtumien, käyttäjien vuorovaikutuksen ja kommunikaation hallinnointi. Pääprosessilla on täysi pääsy Node.js- ja Electron.js-rajapintoihin, joiden avulla hallitaan ikkunoita ja ollaan vuorovaikutuksessa käyttöjärjestelmän kanssa. Pääprosessin lisäksi on renderer-prosesseja, joita voi olla useita. Yksi prosessi on aina vastuussa yhdestä ikkunasta ja sen käyttöliittymän renderöinnistä. Moderneissa SPA-sovelluksissa ikkunoita on yleensä vain yksi, mutta on myös mahdollista käyttää useampaa ikkunaa samanaikaisesti. Renderer-prosessin käyttöliittymä on pohjimmiltaan verkkosivu, joka on ladattu sovelluksen ikkunaan käyttäen Chromium-selainta.



Kuvio 2. Electron.js:n prosessikuva (Cabotin artikkeli 2017)

Prosessien välinen kommunikaatio toimi ennen Electronissa IPC:n (Inter Process Communication) avulla. Electron 12 jälkeen käyttöön tuli Context Bridge, joka on uusi turvallisempi tapa hoitaa prosessien välinen kommunikaatio. Context Bridge on turvaominaisuus, joka mahdollistaa kaksisuuntaisen kommunikaation main- ja renderer-prosessien välillä. Sitä tarvitaan, jos halutaan päästä käsiksi Node.js:n tai Electronin tarjoamiin rajapintoihin, koska nämä ovat saatavilla ainoastaan main-prosessista. Siihen liittyy kuitenkin muutamia rajoituksia, koska välitettävät funktion arvot kopioidaan, jotta konteksti pysyy eristyksissä. Kopioimisen takia vain tiettyjä tyyppejä voidaan käyttää. Tarkemmin näistä alla olevassa taulukossa (ks. taulukko 1).

Taulukko 1. Context Bridgen tukemat tietotyypit ja niihin liittyvät rajoitukset (Context Bridge N.d, muokattu)

Tyyppi	Monimutkaisuus	Tuettu	Rajoitukset
string	Yksinkertainen	☑	-
number	Yksinkertainen	☑	-
boolean	Yksinkertainen	☑	-
Object	Monimutkainen	☑	Avaimet ovat tuettuja vain tämän taulukon yksinkertaisilla tietotyypeillä, ja arvojen tietotyyppien täytyy löytyä tästä taulukosta. Prototyyppi muutokset menetetään ja Custom luokkien arvot kopioidaan, mutta ei prototyyppejä.
Array	Monimutkainen	☑	Samat rajoitukset kuin Object tyypillä
Error	Monimutkainen	☑	Heitetyt virheet kopioidaan, mutta virheiden viesti ja stack trace voivat muuttua hieman, koska ne heitetään eri kontekstiin ja kaikki Error-objektin custom propertyt menetetään.
Promise	Monimutkainen	☑	-
Function	Monimutkainen	☑	Prototype muutokset menetetään. Luokkien tai constructoreiden lähetys ei toimi.
Cloneable Types	Yksinkertainen	☑	Tarkista linkattu dokumentti cloneable tyypeistä. The structured clone algorithm - Web APIs | MDN (mozilla.org)
Element	Monimutkainen	☑	Prototype muutokset menetetään. Custom elementtien lähetys ei toimi.
Blob	Monimutkainen	☑	-
Symbol	-	☒	Symboleita ei voida kopioida kontekstien välillä, joten ne jätetään pois

Rajapintojen julkaisu tehdään preload.js-skriptin avulla. Skripti ladataan uuden renderer-prosessin luonnin yhteydessä ennen kuin sovelluksen koodi ajetaan. Skripti määritellään BrowserWindow-objektin webPreferences-parametrissa main.js-tiedostossa (ks. kuvio 1). Preload-skriptissä määritellään kuvion 3 mukaisesti esimerkki myAPI-rajapinta, joka sisältää metodin. Metodi ottaa message-attribuutin ja lisää sen palautusviestiin mukaan. API:n määrityksen jälkeen se voidaan antaa ContextBridgen exposeInMainWorld-funktiolle, joka julkaisee sen renderer-prosessin käytettäväksi.

```
JS preload.js > ...
1   const { contextBridge } = require('electron')
2
3   const myAPI = {
4       foo: (message) => {
5         return "Hello from main process!\nYou passed following message: " + message
6       }
7     };
8
9   contextBridge.exposeInMainWorld('myAPI', myAPI);
10
```

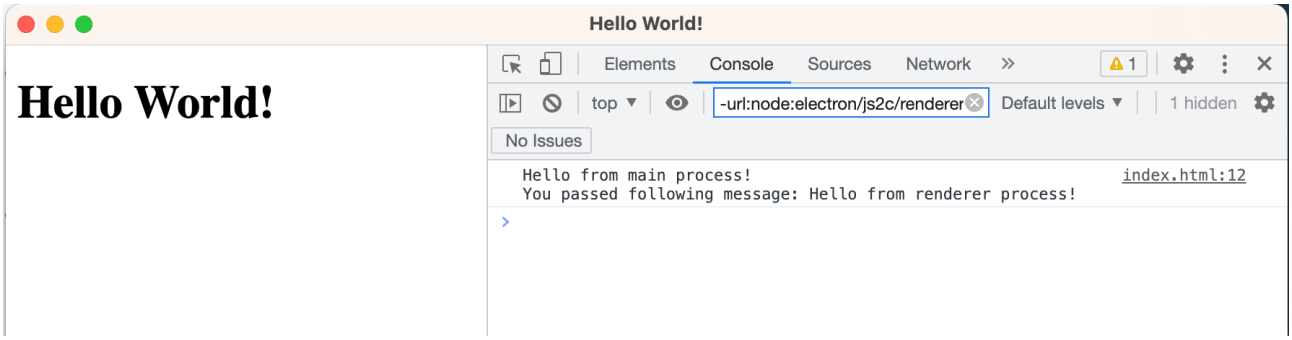
Kuvio 3. Context Bridgen käyttö main-prosessissa

Rajapinnan julkaisemisen jälkeen se on käytettävissä JavaScriptin globaalissa window-objektissa, joka edustaa selaimen ikkunaa. Kuviossa 4 nähdään kutsu aiemmin luotuun myAPI objektin foo-

```
10  <script>
11      let message = window.myAPI.foo("Hello from renderer process!")
12      console.log(message)
13  </script>
```

Kuvio 4. Rajapintojen käyttö Context Bridgellä renderer-prosessista

metodiin ja sen palauttama viesti on tulostettu konsoliin (ks. kuvio 5). Kuten kuviossa 5 nähdään Electron sovelluksessa päästään käsiksi myös Chromiumin kehittäjätyökaluihin. Nämä voidaan kuitenkin halutessaan piilottaa käyttäjältä.



Kuvio 5. Electron-sovellus ja Context Bridgen välittämä konsoli viesti main-prosessilta

3.4.2 Rajapinnat

Electron.js:n sovellukset toimivat samalla tavalla kuin web-sovellukset, mutta Electron-sovellusten etuna on pääsy järjestelmätason rajapintoihin, jotka eivät ole selaimelle normaalisti saatavilla. Web-sovellusta rajoittaa selaimelle saatavilla olevat rajapinnat, joita ovat esimerkiksi: Sijainti, Canvas-grafiikka, localStorage, sessionStorage, ääni ja video. Näiden rajapintojen lisäksi Electron-sovelluksella on pääsy useisiin järjestelmätason rajapintoihin. Esimerkiksi Desktop Capturer -rajapinnan avulla sovellus voi kaapata kuvia ja videota tietokoneen näytöstä. Dialog-rajapinta mahdollistaa erilaisten ikkunoiden avaamisen, joilla käyttäjä voi suorittaa erilaisia toimintoja. Global keyboard shortcuts mahdollistaa globaalien pikanäppäinten asettamisen. Menu-rajapinnan avulla sovellus voi näyttää omia valikoita käyttöjärjestelmän valikkorivillä. InAppPurchase ja macOS Touch Bar rajapinnat mahdollistavat sovellusten sisäisten ostot ja mukautettujen painikkeiden lisäämisen Macbook Pro:n TouchBariin macOS-käyttöjärjestelmällä. Lisäksi on saatavilla monia muitakin rajapintoja, jotka mahdollistavat esimerkiksi asetuksien ja käyttöjärjestelmän tilan lukemisen sekä monia muitakin toimintoja.

3.5 Sovelluksen julkaiseminen

Electron.js sovelluksen julkaisemiseen on saatavilla useita eri työkaluja, kuten Electron Builder, Electron Forge ja Electron Packager. Tässä esimerkissä käytetään Electron builderia. Electron builderin asentamiseksi ajetaan seuraava npm komento `npm install electron-builder –save-dev` projekti hakemistossa. Seuraavaksi asennetaan myös globaali versio Electron builderista `npm install -g electron-builder` komennolla.

Electron builderin asennuksen jälkeen määritellään package.json-tiedostossa tarvittavat build-parametrit (ks. kuvio 6). Näitä parametreja on application id sekä alustakohtaiset parametrit. MacOS-

```
6   "build": {
7     "appId": "com.thesis.electron.test",
8     "mac": {
9       "category": "public.app-category.education",
10      "asar": true,
11      "target": "dir",
12      "icon": "dist/appIcon.ico"
13    }
14  },
```

Kuvio 6. Electron Builder build-parametrit package.json-tiedostossa

käyttöjärjestelmälle voidaan esimerkiksi määritellä sovelluksen kategoria, kohde, sovelluksenkuvake sekä asar-paketointimuoto, joka paketoi sovelluksen lähdekoodin yhteen arkistotiedostoon. Sen tarkoitus on parantaa sovelluksen suorituskykyä pienentämällä ladattavien staattisten tiedostojen määrää sovelluksen käynnistyksen yhteydessä. Näiden parametrien lisäksi on myös saatavilla monia muita build-parametrejä, joilla voidaan vaikuttaa sovelluksen paketointiin.

Build-parametrien määrityksen jälkeen voidaan käyttää `electron-builder` komentoa release-pakettien generoimiseksi. Tämä komento luo vakiona dist-hakemiston luoduille release-tiedostoille. Tämän jälkeen sovellus on valmis julkaistavaksi valitulla alustalla tai ottaa suoraan käyttöön sellaisenaan.

3.6 Yhteenveto

Electron.js on avoimen lähdekoodin alusta, joka mahdollistaa työpöytäsovellusten luomisen käyttäen web-tekniikoita. Se on monipuolinen ja tehokas alusta työpöytäsovelluksien kehitykseen. Electron.js tarjoaa laajan valikoiman rajapintoja, jotka eivät ole selaimelle saatavilla, kuten tiedostojärjestelmän, käyttöjärjestelmän tiedot, ilmoitukset ja näytön hallinta. Electron.js mahdollistaa myös helposti nykyisen web-sovelluksen kääntämisen työpöytäsovellukseksi. Web-kehittäjien on helppo siirtyä kehittämään Electron.js-sovelluksia, koska niiden kehitykseen käytetään samoja teknologioita kuin web-sovelluksiin. Electron.js arkkitehtuuri koostuu main-prosessista ja renderer-prosessista, joiden välillä kommunikoidaan Context Bridgen avulla. Main-prosessin päätehtävänä on hallinnoida kaikkia sovelluksen toimintoja, kuten ikkunoita, tiedostojärjestelmää, tapahtumia ja pääsyä käyttöjärjestelmän rajapintoihin. Renderer-prosessit ovat vastuussa ikkunoiden piirtämisestä ja käyttäjän toimintojen käsittelystä. Jokaisella ikkunalla on oma renderer-prosessi, joka hallitsee sen sisältöä. Electron.js sovellusten julkaisemiseen voidaan hyödyntää erilaisia työkaluja, kuten electron-builder tai electron-packager.

4 Flutter

4.1 Yleistä

Flutter on Googlen tekemä avoimen lähdekoodin UI-toolkit, joka mahdollistaa Cross-Platform-sovelluksien tekemisen Windows-, Mac-, Linux- ja Web-ympäristöjen lisäksi myös Android- ja iOS-mobiiliympäristöille. Flutter on hyvä vaihtoehto kaikenkokoisille yrityksille, mutta erityisesti pienille tiimeille/yrityksille, jotka pystyvät Flutterin avulla kerralla luomaan sovelluksen, joka toimii kaikilla relevanteilla ympäristöillä. Flutterin tunnetuimpia käyttäjiä ovat esimerkiksi: BMW, Alibaba, eBay, Stadia, Toyota ja Tencent (Flutter homepage 2021).

Flutter sovelluksia kehitetään Dart-kielellä, joka muistuttaa hyvin pitkälti TypeScriptiä. Sovellukset koostuvat erilaisista Widgeteistä, jotka muistuttavat hieman Reactin komponentteja. Widgettejä on saatavilla valmiina hyvin paljon ja näitä hyödyntämällä sovelluksen rakentaminen nopeutuu huomattavasti. Myös sovelluksen suorituskyky on hyvin lähellä native-sovelluksia, koska valmis sovellus käännetään suoraan konekoodiksi, joka on optimoitu eri ympäristöille. Flutter ei myöskään hyödynnä kontekstin vaihtoa, käytä WebViewiä tai yhteisellä kielellä tehtyä siltaa, jonka ansiosta sovellus toimii nopeasti. (Aguinis 2019.)

Flutterissa tulee vakiona kaksi eri tyylistä widget-kirjastoa, jotka ovat Material ja Cupertino. Material toteuttaa Googlen kehittämää muotokieltä, kun taas Cupertino seuraa Applen Human Interface Guidelinea. Cupertinon avulla saadaan luotua natiivi iOS sovelluksen näköisiä UI-elementtejä ja Material kirjastolla Android-sovelluksen. Näitä molempia kirjastoja voidaan kuitenkin hyödyntää molemmilla alustoilla ja ainoa eroavaisuus on vain se, että Cupertino käyttää Android alustalla eri fonttia lisensointi syistä. On myös mahdollista luoda alustatietoisia widgettejä, jotka käyttävät iOS-laitteilla Cupertinon tyylejä ja Androidilla Material-tyylejä. (Kulinski 2018.) Tästä myöhemmin lisää Alustatietoiset widgetit osiossa.

4.2 Uuden projektin aloitus

Ennen uuden projektin aloittamista täytyy varmistaa, että Flutter SDK on asennettuna koneella. Mikäli se vielä puuttuu, voidaan se ladata Flutterin virallisilta kotisivuilta. Asennuksen jälkeen asennuksen onnistuminen varmistetaan ajamalla komentorivillä `flutter doctor` komento. Komento varmistaa, että järjestelmä täyttää tarvittavat vaatimukset ja ympäristö on luotu oikein. Kun Flutter SDK on asennettu ja sen toimivuus on varmistettu, voidaan aloittaa uuden projektin luominen. Tämä on helppoa, sillä Flutterin komentorivityökaluissa on valmis komento uuden projektin luomiseen. Uusi projekti luodaan komennolla `flutter create project_name`. Komento luo valmiin projektirakenteen, joka sisältää kaiken tarvittavan koodin, esimerkki aloitusnäkymän ja muita vakio widgettejä. Tämä säästää aikaa ja vaivaa projektin perusrakenteen luomisessa.

4.3 Kirjastojen käyttö

Kirjastojen ja rajapintojen suhteen Flutter tarjoaa peruslaite ominaisuudet ja rajapinnat, kuten kameran, sijaintipalvelut, Bluetooth, tiedostojärjestelmään pääsyn, sensoreita sekä audio ja video ominaisuudet. Näiden lisäksi Flutter-yhteisöt ovat kehittäneet ja julkaisseet laajan valikoiman lisäkirjastoja, joiden avulla kehittäjät voivat nopeuttaa sovelluskehitystä. Näitä kirjastoja on saatavana Flutterin oman Pub-paketinhallintajärjestelmästä, joka mahdollistaa kirjastojen lataamisen, jakamisen ja hallinnoin. Kirjastoja on saatavilla monilta eri alueilta, kuten animaatiot, käyttöliittymäkomponentit, http-kirjastot, tilanhallinta, reititys ja käyttäjäkokemus. Esimerkkejä Flutter-kirjastoista ovat Firebase, RxDart, shared_preferences ja http.

4.4 Käyttöliittymä rakenne

Flutterissa käyttöliittymä rakennetaan käyttäen widgeteistä koostuvaa puurakennetta (ks. kuvio 7). Widget on Flutterin käyttämä rakennuspalikka, jota käytetään sovelluksen käyttöliittymän rakentamiseen. Pääpiirteittäin on olemassa kahdenlaisia widgettejä stateless ja stateful. Stateless widget on muuttumaton, joten se ei muutu ajan kuluessa. Niitä käytetään tyypillisesti UI elementeissä, jotka pysyvät muuttumattomina, kuten ikonit, napit ja tekstit. Muuttumattoman luonteen ansiosta ne ovat myös tehokkaampia käyttää kuin stateful widgetit, jotka puolestaan taas ovat muuttuvia. Näitä hyödynnetään, kun tarvitaan UI-elementtejä, jotka reagoivat käyttäjien toimintaan tai muutoksiin datassa. Hyvä esimerkki tästä on lista elementti, joka päivittyy käyttäjän tehdessä muutoksia dataan. Sisäänrakennettujen widgettien lisäksi kehittäjä voi luoda omia custom

widgettejä, jotka voivat koostua olemassa olevista tai uusista widgeteistä. Näiden avulla voidaan luoda tehokkaasti monimutkaisia ja dynaamisia käyttöliittymiä.

```
10  @override
11  Widget build(BuildContext context) {
12    return StreamBuilder(stream: globals.noteStream,
13      builder: (context, snapshot) {
14        if (snapshot.data == null) {
15          return Container(color: Colors.white);
16        }
17        Note? note = snapshot.data as Note;
18        return SafeArea(
19          child: Container(
20            constraints: const BoxConstraints.expand(),
21            color: Colors.white,
22            child: Column(
23              crossAxisAlignment: CrossAxisAlignment.start,
24              children: [
25                Container(
26                  padding: const EdgeInsets.symmetric(horizontal: 15, vertical: 20),
27                  child: RichText(
28                    text: TextSpan(
29                      text: note.content,
30                      style: const TextStyle(
31                        fontSize: 14.0,
32                        color: Colors.black,
33                        fontWeight: FontWeight.bold
34                      ) // TextStyle
35                    ) // TextSpan
36                  ), // RichText
37                ) // Container
38              ]
39            ) // Column
40          ) // Container
41        ); // SafeArea
42      }
43    ); // StreamBuilder
44  }
45 }
```

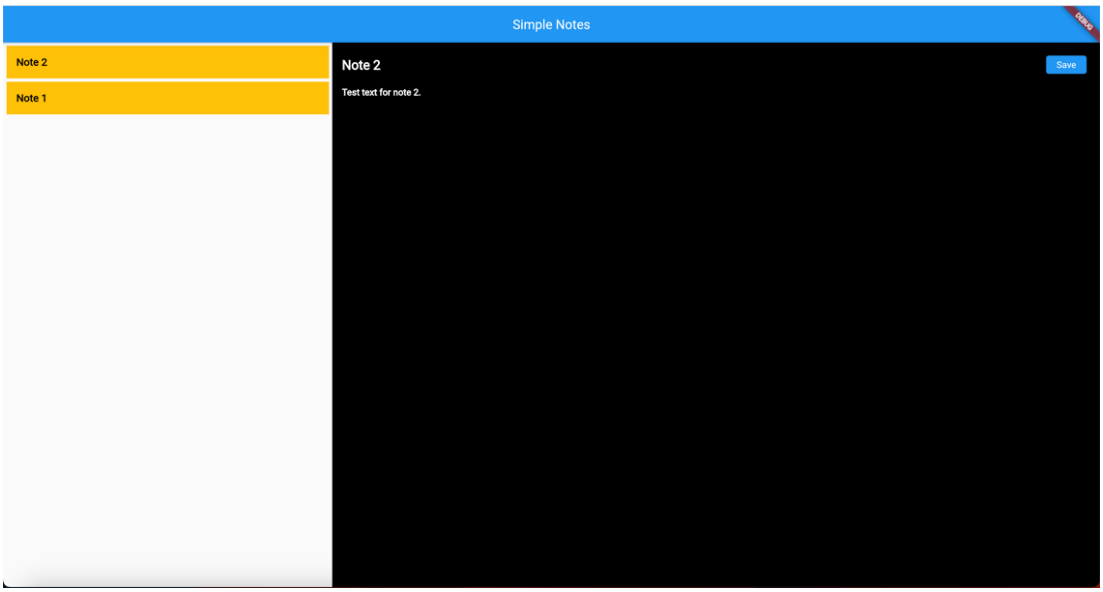
Kuvio 7. Muistio esikatselunäkymän käyttöliittymän puurakenne

Käyttöliittymän rakenne voidaan rakentaa siten, että se mukautuu mahdollisimman hyvin eri alustoilla tietokoneiden suurilta näytöiltä älypuhelimien pienille näytöille (ks. kuvio 8). Navigointityylit ja toiminnallisuus voidaan sopeuttaa näppäimistöltä ja hiireltä kosketusnäytölle sopivaksi. Sovelluksen toiminnallisuutta voidaan myös rajoittaa pienemmillä näytöillä ja tarjota suppeammat muokkausmahdollisuudet kuin isoilla tietokoneen näytöillä. Isommalla näytöllä voidaan myös näyttää useampaa näkymää samanaikaisesti, kun taas pienellä puhelimen näytöllä näytetään pääasiallisesti vain yhtä näkymää kerrallaan. Mikäli käyttöliittymä on taas hyvin erilainen jokaisella alustalla ja vaikea toteuttaa siten, että samaa näkymää hyödynnetään kaikkialla, voidaan käyttöliittymän rakenne räätälöidä myös erikseen jokaiselle alustalle.

```
3   import 'package:flutter/material.dart';
4
5   enum FormFactor { Mobile, Desktop }
6
7   class ViewType {
8     static bool isDesktop(BuildContext context) {
9       return MediaQuery.of(context).size.shortestSide > 800;
10    }
11
12    static bool isMobile(BuildContext context) {
13      return MediaQuery.of(context).size.shortestSide < 800;
14    }
15
16    static FormFactor getFormFactor(BuildContext context) {
17      double deviceWidth = MediaQuery.of(context).size.shortestSide;
18      if (deviceWidth > 800) {
19        return FormFactor.Desktop;
20      } else {
21        return FormFactor.Mobile;
22      }
23    }
24  }
```

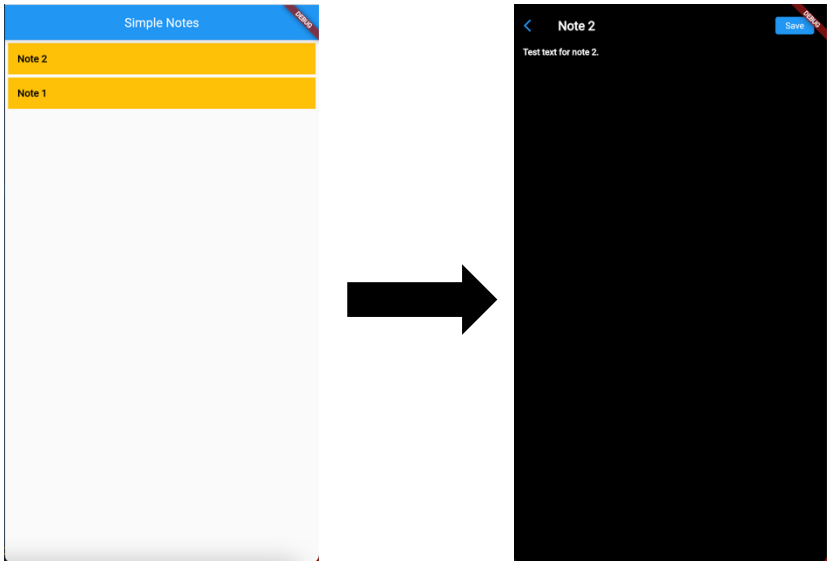
Kuvio 8. Käytettävän FormFactorin tunnistaminen koodissa

Simple Notes -muistiosovellus havainnollistaa käyttöliittymä rakenteen optimoimista eri alustoille ja näyttää, kuinka käyttöliittymän navigointi sopeutuu työpöytä- ja mobiilikäyttöön. Sovellus käyttää samoja komponentteja kaikissa näkymissä, ainoastaan komponenttien näyttäminen on optimoitu erikseen työpöytä- ja mobiilinäkymiin. Kun käytössä on iso näyttö, sovellus näyttää muistilistan ja muistion esikatselu näkymän samanaikaisesti vierekkäin (ks. kuvio 9). Mikäli näkymän kokoa pienennetään liian kapeaksi, sovellus siirtyy käyttämään mobiilinäkymää.



Kuvio 9. Simple notes työpöytänäkymä

Mobiililaitteilla sovellus näyttää vain yhden näkymän kerrallaan (ks. kuvio 10), koska pienellä näytöllä ei mahduta näyttämään molempia näkymiä samanaikaisesti. Ensimmäisessä näkymässä näytetään muistiolistaus, josta päästään navigoimaan muistion esikatseluun painamalla jotakin muistioista. Tämä taas avaa uuden näkymän, joka näyttää esikatselun muistion sisällöstä.



Kuvio 10. Simple notes mobiilinäkymät

4.5 Alustatietoiset widgetit

4.5.1 Johdanto

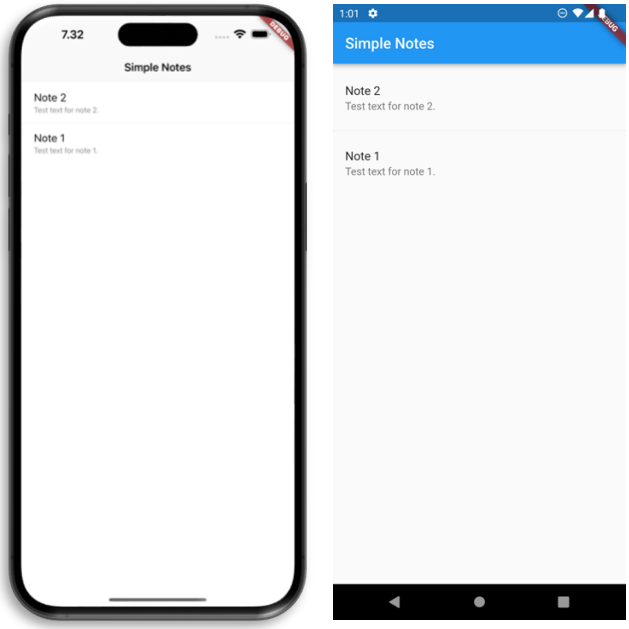
Flutterin avulla on mahdollista luoda samalla lähdekoodilla useille alustoille sovellus, joka näyttää tismalleen samalta kaikkialla, mutta mitä jos haluttaisiinkin pitää sovelluksessa alustakohtainen ulkoasu ja tuntuma sekä seurata Androidin käyttämää Material Designia tai Applen käyttämää iOS Human Interface languagea. Flutter mahdollistaa tämän, koska se sisältää Cupertino ja Material kirjastot, joissa on valmiina useita eri widgettejä iOS:ille ja Androidille. Näiden avulla on mahdollista luoda käyttöliittymä, joka tuntuu ja näyttää samalta kuin natiivisti luotu käyttöliittymä iOS tai Androidin puolella.

Sitten vain tarvittaisiin keino luoda natiivi käyttöliittymä molemmille alustoille samalla lähdekoodilla. Yksi mahdollinen keino voisi olla oman UI-layoutin luominen molemmille alustoille, jota voitaisiin vaihtaa käytettävän alustan mukaan. Tämä kuitenkin lisää duplikaatti koodin määrää ja hankaloittaa ylläpitoa, koska sama muutos pitää tehdä useampaan paikkaan. Parempi vaihtoehto tälle

on hyödyntää samaa ideaa widgetin tasolla. Otetaan esimerkiksi AppBar widget. Molemmilla alustoilla kyseinen widgetti on hyvin samantyylinen toteutukseltaan. Pääasiassa vain ulkoasu on erilainen ja toiminnallisuus lähes identtinen. Tämä mahdollistaa alustakohtaisen widgetin luomisen, jota voitaisiin kutsua PlatformAppBariksi. Kun käytämme alustakohtaista widgettiä iOS alustalla, se muuttuu iOS:in käyttämäksi CupertinoNavigationBariksi ja Android alustalla taas Materialin AppBariksi. Käsitellään tätä vielä hieman myöhemmin käytännön tasolla.

4.5.2 Ulkoasu

Luomalla tarvittavat alustakohtaiset widgetit ja ottamalla ne käyttöön jo aiemmin näytetyssä Simple Notes -sovelluksessa, pystymme tarkastelemaan alustakohtaisten widgettien toimintaa käytännössä. Kuviossa 11 nähdään Simple Notes -sovelluksen muistionäkymä, joka on toteutettu hyödyntäen alustakohtaisia widgettejä. Kuviossa on näkyvissä miltä samalla lähdekoodilla tuotettu näkymä näyttää Android- ja iOS-ympäristöissä. Ensimmäisenä huomio kiinnittyy sovelluksen yläreunassa näkyvään AppBar:iin, joka näyttää erilaiselta kummallakin alustalla. Molemmilla alustoilla AppBar näyttää samalta kuin natiivisti luodun sovelluksen AppBarin tulisikin näyttää. Myös muistiolista ja listan rivit näyttävät molemmilla alustoilla alustojen omilta UI komponenteilta.



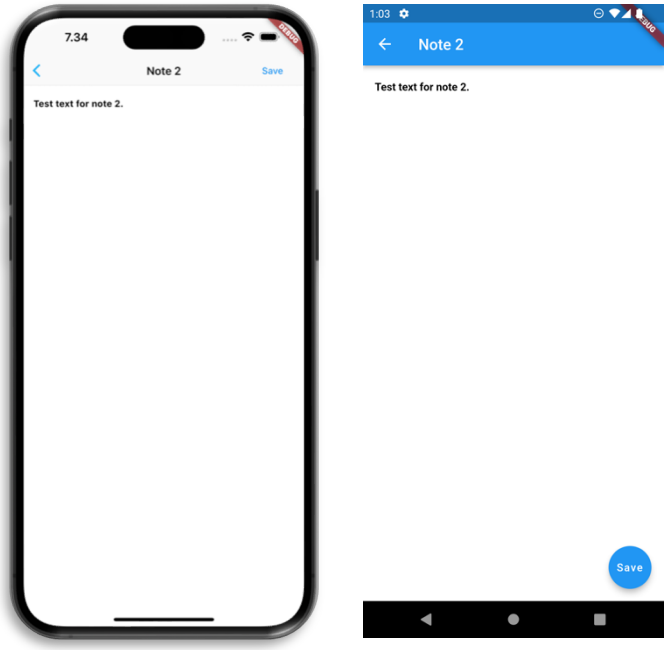
Kuvio 11. Simple notes -sovelluksen muistiolistaus Android- ja iOS-alustoilla

Kuviossa 11 näkyvältä muistiolistaukselta muistiota painettaessa siirrytään muistion esikatselunäkymään kunkin alustan omalla navigointityylillä. Android alustalla tämä tapahtuu käyttämällä Material kirjaston PageRoutea ja iOS alustalla Cupertino kirjaston PageRoutea. Myös näille luokille voidaan luoda yhteinen PlatformPageRoute, joka vaihtaa käytettyä luokkaa alustan mukaisesti (ks. kuvio 12). Yhteisiä parametrejä PageRoute-luokilla on: builder, settings, maintainState, fullscreenDialog ja allowSnapshotting. Tässä tapauksessa molemmilla luokilla oli samat parametrit, joten näiden avulla pystyttiin luomaan täydellinen alustakohtainen luokka.

```
1   import 'package:flutter/cupertino.dart';
2   import 'package:flutter/foundation.dart';
3   import 'package:flutter/material.dart';
4
5   class PlatformPageRoute {
6
7     final Widget Function(BuildContext) builder;
8     final RouteSettings? settings;
9     final bool maintainState;
10    final bool fullscreenDialog;
11    final bool allowSnapshotting;
12
13    PlatformPageRoute({
14      required this.builder,
15      this.settings,
16      this.maintainState = true,
17      this.fullscreenDialog = false,
18      this.allowSnapshotting = true,
19    });
20
21    PageRoute getRoute() {
22      if (defaultTargetPlatform == TargetPlatform.iOS) {
23        return CupertinoPageRoute(
24          builder: builder,
25          settings: settings,
26          maintainState: maintainState,
27          fullscreenDialog: fullscreenDialog,
28          allowSnapshotting: allowSnapshotting
29        );
30      } else {
31        return MaterialPageRoute(
32          builder: builder,
33          settings: settings,
34          maintainState: maintainState,
35          fullscreenDialog: fullscreenDialog,
36          allowSnapshotting: allowSnapshotting
37        );
38      }
39    }
40
41  }
```

Kuvio 12. PlatformPageRoute alustakohtainen toteutus

Kuviossa 13 nähdään miltä avattu esikatselunäkymä näyttää kullakin alustalla. Molempien alustojen yläpalkin ulkoasu vastaa alustalta odotettua omaa tyyliään, sisältää otsikon ja siirtymäpainikeen takaisin edelliseen näkymään. Kuviosta 13 nähdään myös, kuinka tallentamiseen käytetty painike sijaitsee iOS alustalla sille tyypillisellä paikalla yläpalkissa, ja Android hyödyntää sille tyypillistä FAB:iä (Floating Action Button) samaan tarkoitukseen. Alustakohtaisten widgettien avulla voidaan myös käsitellä tämän tyylisiä design eroja alustojen välillä ja seurata alustan omaa suunnittelutyyliä.



Kuvio 13. Simple notes -sovelluksen muistion esikatselunäkymä Android- ja iOS-alustoilla

4.5.3 Koodi

Yksi tapa luoda alustakohtaisia widgettejä on käyttää apuna abstraktia-luokkaa, joka luo alustakohtaisia widgettejä iOS- ja Android-alustoille. Kuviossa 14 on määritelty abstrakti PlatformWidget-luokka, joka laajentaa StatelessWidget-luokkaa ja jolle määritellään tyyppiparametrien avulla iOS- ja Default-widgetit. Näiden avulla tyypitetään kaksi abstraktia metodia: createIosWidget ja createDefaultWidget. Nämä metodit palauttavat alustakohtaisen widgetin iOS:lle ja muille alustoille. Build metodissa nykyinen alusta havaitaan käyttäen defaultTargetPlatformia ja palautetaan

```
1   import 'package:flutter/foundation.dart';
2   import 'package:flutter/widgets.dart';
3
4   abstract class PlatformWidget<IOS extends Widget, Default extends Widget>
5     extends StatelessWidget {
6
7     const PlatformWidget({super.key});
8
9     @override
10    Widget build(BuildContext context) {
11      if (defaultTargetPlatform == TargetPlatform.iOS) {
12        return createIosWidget(context);
13      } else {
14        return createDefaultWidget(context);
15      }
16    }
17
18    IOS createIosWidget(BuildContext context);
19
20    Default createDefaultWidget(BuildContext context);
21  }
22
```

Kuvio 14. Abstrakti PlatformWidget-luokka esimerkki

sopiva alustakohtainen widgetti. Jos alusta ei ole iOS niin käytetään default widgettiä. Tämä logiikka voitaisiin myös tehdä niin, että alustat olisivat määritelty vielä tarkemmin ja tukemattoman alustan tapauksessa palautettaisiin virhe tai tyhjä widgetti.

Seuraavaksi tarkastellaan kuviota 15, jossa luodaan PlatformAppBar-luokka. Se laajentaa abstraktia PlatformWidget-luokkaa, joka on määritelty CupertinoNavigationBar- ja AppBar-tyyppiparametreilla. CupertinoNavigationBar edustaa iOS-alustan navigaatiopalkkia, kun taas AppBar edustaa Android-alustan navigaatiopalkkia. PlatformWidget-luokka määrittelee implementoitavaksi abstraktit createDefaultWidget- ja createIosWidget-metodit, jotka palauttavat kullekin alustalle sopivat navigaatiopalkit. Metodeja määritellessä tulee ottaa huomioon molempien alustojen widgettien tarvitsemat parametrit. Nämä parametrit ovat molemmilla luokilla hyvin samantyylisiä, mutta eroja on kuitenkin.

Kuten kuviosta 15 huomataan molempien alustojen AppBar:illa yhteisiä parametreja on ainakin key, leading, title ja background color parametrit. Tehdessä alustakohtaista widgettiä täytyy molempien alustojen widgetin parametrit käydä läpi ja etsiä yhteisiä parametreja, jotka ovat samoja molemmilla alustoilla. Alusta, joka tukee pienempää määrää parametreja rajoittaa toisen alustan widgetin toteutusta. Tässä tapauksessa Androidin AppBar:illa on paljon enemmän parametrejä,

```
class PlatformAppBar extends PlatformWidget<CupertinoNavigationBar, AppBar> {

  final Widget? leading;
  final Widget? title;
  final Color? backgroundColor;

  const PlatformAppBar({
    super.key,
    this.leading,
    this.title,
    this.backgroundColor
  });

  @override
  AppBar createDefaultWidget(BuildContext context) => AppBar(
    key: key,
    leading: leading,
    title: title,
    backgroundColor: backgroundColor
  );

  @override
  CupertinoNavigationBar createIosWidget(BuildContext context) =>
    CupertinoNavigationBar(
      key: key,
      leading: leading,
      middle: title,
      backgroundColor: backgroundColor
    );
}
```

Kuvio 15. Alustakohtainen AppBar-luokka

jotka pääasiassa liittyvät AppBarin-tyylittelyyn. iOS-puolella tyylittelyyn taas ei ole saatavilla samanlaisia parametrejä, joten toteutus rajoittuu iOS-widgetin mukaan. iOS-widgetit ovat yleensä paljon yksinkertaisempia verrattuna Android puolen vastaaviin komponentteihin. Pääasiassa molemmilla alustoilla on kuitenkin tarpeeksi samanlaiset parametrit, jotta platform widgettejä voidaan luoda ja joissakin tapauksissa näitä rajoitteita voidaan kiertää tarpeeksi, jotta tarpeellinen widgetin toiminnallisuus saadaan luotua.

4.5.4 Yhteenveto

Alustakohtaiset widgetit ovat hyvä keino rakentaa yhtenäinen käyttökokemus useammalle alustalle, joissa on samoja ominaisuuksia. Tämä helpottaa käyttöliittymän ylläpitoa, kun yhdellä koodipohjalla voidaan saavuttaa sama lopputulos molemmilla alustoilla. Natiivin käyttökokemuksen mahdollistaminen tuo etua iOS- ja Android-käyttäjille. Myös sovelluksen ylläpito helpottuu, kun käytössä on vain yksi käyttöliittymä ja koodipohja molemmille alustoille.

Vaikka alustakohtaiset widgetit ovatkin hyödyllisiä, niiden käyttöön liittyy myös rajoituksia. Esimerkiksi iOS:n käyttämät widgetit ovat yleensä paljon yksinkertaisempia kuin Androidilla, mikä voi rajoittaa käyttöliittymän suunnittelumahdollisuuksia, koska yhteisiä parametrejä widgettien välillä on tällöin vähemmän. Widgettien luontia hankaloittaa myös, jos alustojen väliset käyttötarkoitukset UI-komponenteilla poikkeavat suuresti toisistaan. Tällöin voidaan joutua tekemään kompromisseja ja custom komponentteja natiivi komponenttien tilalle. Pääasiassa kuitenkin alustakohtaisista komponenteista on enemmän hyötyä kuin haittaa ja rajoitteet kannattaa ottaa huomioon jo suunnitteluvaiheessa. Varsinkin yksinkertaisemmat sovellukset voidaan helposti toteuttaa käyttämällä alustakohtaisia widgettejä.

4.6 Tiedon tallentaminen

Flutter tukee natiivisti muutamia erilaisia datan tallennuskeinoja. Ensimmäinen keino on shared preferences, joka on helppo tapa tallentaa pieniä määriä dataa, kuten käyttäjän asetukset, valinnat tai muita tila tietoja. Shared preferences tallentaa tietoja avainarvo-pareina, joita on helppo käyttää ja muokata. Flutter tarjoaa tämän shared_preferences paketin avulla, minkä vuoksi se on helppo ottaa käyttöön.

Toinen tapa tallentaa tietoa on SQLite-relaatiotietokanta. Sen avulla voidaan käsitellä suurtakin määrää dataa kätevästi ja jakaa tieto selkeisiin rakenteisiin. Flutterissa SQLite saadaan käyttöön sqflite paketilla. Simple Notes -projektissa käytettiin SQLiteä muistiinpanojen tallennukseen. Tätä varten luotiin NotesDatabase-luokka, joka hallinnoi tietokannan käyttöä. Taulun luominen tapahtuu hyvin samaan tyyliin kuin SQL:ssä, CREATE TABLE -komennon avulla (ks. kuvio 16).

```
Future<void> _createNotesTable(Database db) => db.execute(
  "CREATE TABLE $_notes(id INTEGER PRIMARY KEY, title TEXT, content TEXT)"
);
```

Kuvio 16. Notes-taulun luominen SQLitellä

SQLite-tietokannan avaus tapahtuu open-funktiolla, joka avaa tietokannan tiedostosta ja alustaa sen, mikäli kyseessä on ensimmäinen kerta (ks. kuvio 17). Avattu tietokanta tallennetaan db-muuttujaan ja isOpen-muuttujaan asetetaan arvoksi true.

```
22  Future<void> open() async {
23    if (_isOpen) return;
24
25    // Avoid errors caused by flutter upgrade.
26    WidgetsFlutterBinding.ensureInitialized();
27
28    // Open the database and store the reference.
29    _database = openDatabase(
30      join(await getDatabasesPath(), _databaseName),
31      // First time initialization
32      onCreate: (db, version) {
33        _createNotesTable(db);
34      },
35      version: _version
36    );
37    final db = await _database;
38    _isOpen = db.isOpen;
39  }
```

Kuvio 17. SQLite-tietokannan avausfunktio

Muistiinpanojen lisäys SQLite-tietokantaan tapahtuu insert-funktion avulla (ks. kuvio 18). Insert-funktiolle annetaan parametreina taulun nimi, muistiinpano-objekti ja konfliktitilanteessa kutsu määritellään epäonnistumaan. Mikäli tietokanta yhteyttä ei ole vielä avattu heitetään DatabaseNotOpenException niminen poikkeus.

```
47  Future<void> insert(Note note) async {
48    if (!_isOpen) throw DatabaseNotOpenException();
49    final db = await _database;
50    db.insert(_notes, note.toMap(), conflictAlgorithm: ConflictAlgorithm.fail);
51    _notesChangeRelay.add(Void);
52  }
```

Kuvio 18. Muistiinpanon lisäys SQLite-tietokantaan

Kaikkien muistiinpanojen hakeminen listaukseen tapahtuu getAllNotes-funktiolla (ks. kuvio 19). Funktio hakee tietokannasta kaikki rivit ja mappaa ne Note-objekteiksi. Muistiinpanolista järjestellään uusimmasta vanhimpaan. Mikäli tietokanta yhteyttä ei ole avattu heitetään aiemmin mainittu poikkeus.

```
62  Future<List<Note>> getAllNotes() async {
63    if (!_isOpen) throw DatabaseNotOpenException();
64    final db = await _database;
65    final List<Map<String, dynamic>> maps = await db.query(_notes);
66
67    return List.generate(maps.length, (i) {
68      return Note(
69        id: maps[i]['id'],
70        title: maps[i]['title'],
71        content: maps[i]['content'],
72      ); // Note
73    }).reversed.toList(); // List.generate
74  }
```

Kuvio 19. Kaikkien muistiinpanojen hakeminen tietokannasta

Näiden tallennustapojen lisäksi Flutter tarjoaa myös rajapinnat tiedostojen tallentamiseen ja laitteelta lukemiseen. Tätä voidaan hyödyntää kuvien, videoiden ja muiden tiedostotyyppien tallentamiseen. Paikallisten tallennus vaihtoehtojen lisäksi voidaan myös tietysti hyödyntää pilvitallennus ratkaisuja, kuten Googlen Firebase tietokantaa tai muita REST API rajapintoja.

4.7 Sovelluksen julkaiseminen

Flutterin avulla sovellus voidaan helposti julkaista usealle eri alustalle, kuten Android, iOS ja Web. Paketointi vaiheessa samalla lähdekoodilla generoidaan koodista tarvittavat binäärit ja resurssit, joita käytetään sovelluksen asentamiseen ja ajamiseen kohdealustalla. Jokaiselle alustalle täytyy generoida oma asennuspaketti Flutterin build-komennon avulla. Komennon avulla voidaan luoda testaamiseen tarkoitettu debug-paketti tai julkaisuun tarkoitettu release-paketti. Kun release-paketti on luotu, voidaan se laittaa jakeluun halutulla jakelukanavalla. Tyypillisimpiä jakelu kanavia on Androidin käyttämä Google Play -sovelluskauppa ja iOS:in käyttämä AppStore.

4.8 Yhteenveto

Flutter on Googlen kehittämä mobiilikehitysalusta, joka perustuu Dart-ohjelmointikieleen. Flutterilla on mahdollista luoda sovelluksia usealle alustalle, kuten Android, iOS, Windows, Mac ja Linux käyttöjärjestelmille yhdellä lähdekoodilla. Alustatietoiset widgetit mahdollistavat alustakohtaisen ulkoasun ja tuntuman luomisen samalla lähdekoodilla, mikä tekee sovellusten kehittämisestä helppoa. Flutter tarjoaa peruslaite ominaisuuksia ja rajapintoja, kuten kameran, sijaintipalvelut, Bluetooth, tiedostojärjestelmään pääsyn, eri sensoreita sekä audio- ja video-ominaisuuksia. Flutter-yhteisö tarjoaa myös monia muita lisäkirjastoja. Flutterin käyttöliittymä rakennetaan käyttämällä widgeteistä koostuvaa puurakennetta. Widgetit ovat joko stateless tai stateful, ja ne mahdollistavat käyttöliittymän dynaamisen päivittämisen ja sisällön näyttämisen käyttäjille. Tiedon tallentamiseen on useita keinoja, kuten shared_preferences, SQLite-relaatiotietokanta ja pilvitallennusratkaisut. Sovelluksen julkaisu voidaan tehdä usealle eri alustalle generoimalla samalla lähdekoodilla tarvittavat binäärit ja resurssit, joita käytetään sovelluksen asentamiseen ja ajamiseen kohdealustalla.

5 Muita sovelluskehyksiä

5.1 .NET MAUI

.NET 6 MAUI on Microsoftin .NET-frameworkiä hyödyntävä uusi Cross-Platform-sovelluskehitys alusta, joka on kehitetty jatkoksi Xamarin Forms -teknologialle. Uutena ominaisuutena on lisätty mahdollisuus tehdä iOS ja Android mobiilisovellusten lisäksi MacOS- ja Windows-työpöytäsovelluksia samalla koodipohjalla. .NET 6 MAUI yhdistää Android-, iOS-, MacOS- ja Windows-rajapinnat yhdeksi yhteiseksi rajapinnaksi, minkä lisäksi pystytään vielä hyödyntämään alustakohtaisia rajapintoja, joille on myös tehty .NET-pohjaiset versiot. Pääasiassa kuitenkin käytetään yhteistä rajapintaa ja tarvittaessa hyödynnetään alustakohtaisia rajapintoja. Yhteinen rajapinta tarjoaa pääsyn eri sensoreihin kuten kiihtyvyys, gyroskooppi ja kompassi sensoreihin, verkkorajapintoihin, laitetietoihin, leikepöydälle, tiedostojärjestelmään, key/value parien tallennukseen ja sisäänrakennettuihin text-to-speech järjestelmiin. Käyttöliittymien suunnitteluun käytetään XAML:ia ja ohjelmointikielenä toimii C#. (Britch, D., Conray, J 2021.)

5.2 React Native

React Native on avoimen lähdekoodiin perustuva JavaScript pohjainen ohjelmistokehys, jota käytetään mobiilisovellusten kehittämiseen. Sen on kehittänyt alun perin Facebook ja se perustuu Reactiin, joka on suosittu web-kehitykseen käytetty JavaScript-kirjasto. React Native mahdollistaa mobiilisovellusten kehityksen, jotka toimivat iOS- ja Android-alustoilla käyttäen samaa koodipohjaa. Yksi sen tärkeimmistä eduista on, että se mahdollistaa sovellusten kehittämisen, joilla on noin sama suorituskyky ja käyttökokemus kuin natiivisovelluksilla, mutta silti käyttämällä tuttua JavaScript-syntaksia. Tämän mahdollistavat esirakennetut komponentit, jotka ovat samanlaisia kuin iOS- ja Android-käyttöjärjestelmissä käytetyt natiivikomponentit. Kaiken kaikkiaan React Native on tehokas ja joustava alusta mobiilisovellusten kehitykseen ja se on myös suosittu valinta Cross-Platform-sovellusta aloittaessa. (Budziński, M. 2022.)

6 Johtopäätökset

Yhteenvetona voidaan todeta, että Cross-Platform-sovellukset ovat osoittautuneet erittäin tehokkaiksi ja nopeiksi tavoiksi rakentaa mobiilisovelluksia, joita voidaan käyttää eri alustoilla. Vaikka Cross-Platform-sovellukset soveltuvat hyvin monenlaisiin projekteihin, pienet ja keskisuuret yritykset hyötyvät niistä eniten, koska ne voivat säästää kehityskustannuksissa ja nopeuttaa kehitysprosessia.

Electron.js soveltuu erityisesti työpöytäsovellusten kehittämiseen ja koska se hyödyntää web-teknologioita, pystyvät myös web-kehittäjät kehittämään työpöytäsovelluksia hyödyntämällä samoja teknologioita. Tämä myös mahdollistaa olemassa olevien web-sovellusten helpon kääntämisen työpöytäsovelluksiksi. Toisaalta Flutter on loistava alusta sekä työpöytä- että mobiilisovellusten kehittämiseen yhdeltä koodipohjalta. Se mahdollistaa työpöytäsovellukselle monipuolisemman UI:n luomisen ja lisätoimintojen määrittelyn, jotka voidaan taas karsia tai muokata paremmin sopivaksi mobiiliversioon. Tämä mahdollistaa vielä useamman alustan saavuttamisen samalla lähdekoodilla. Joskin se voi myös aiheuttaa lisää rajoitteita, mikäli mobiili- ja työpöytäversiot eroavat toisistaan suuresti.

On kuitenkin tärkeää valita huolellisesti oikea kehitystyökalu ja varmistaa, että sen ominaisuudet vastaavat projektin vaatimuksia. .NET MAUI ja React Native ovat myös lupaavia vaihtoehtoja, joita nyt käytiin vain pintapuolisesti läpi ja pääpaino suunnattiin enemmän Flutterin ja Electron.js käyttöön. Cross-Platform-työkalujen käyttö lisääntyy jatkuvasti ja ne tulevat todennäköisesti olemaan yksi tärkeimmistä kehitystyökaluista tulevaisuudessa.

Lähteet

Aguinis, M. 2019. How is Flutter different for app development. Youtube-videopalvelu. Julkaistu 17.9.2019. Viitattu 26.11.2021. https://www.youtube.com/watch?v=l-YO9CmaSUM.

Britch, D., Conray, J. 2021.What is .NET MAUI?. Microsoft 24.12.2021. Viitattu 21.11.2021. https://docs.microsoft.com/en-us/dotnet/maui/what-is-maui.

Budziński, M. 2022. What is React Native? Complex Guide for 2022. Netgurun artikkeli. Viitattu 26.4.2023. https://www.netguru.com/glossary/react-native.

Context Bridge. N.d. Electron.js dokumentaatio. Viitattu 4.1.2022. https://www.electronjs.org/docs/latest/api/context-bridge.

Electron.js docs. N.d. Electron.js dokumentaatio. Viitattu 4.1.2022. https://www.electronjs.org/docs/latest.

Flutter architectural overview. 2021. Virallinen dokumentaatio Flutterin verkkosivuilla 20.11.2021. Viitattu 25.11.2021. https://docs.flutter.dev/resources/architectural-overview.

Flutter homepage. 2021. Flutterin kotisivu. Viitattu 25.11.2021. https://flutter.dev/.

Kulinski, S. 2018. Do flutter apps dream of platform aware widgets?. Artikkeli 1.3.2018. https://medium.com/flutter/do-flutter-apps-dream-of-platform-aware-widgets-7d7ed7b4624d.

Perkaz, A. 2020. Things I Wish I Knew Before Working with Electron.js. FreeCodeCamp 28.5.2020. Viitattu 26.11.2021. https://www.freecodecamp.org/news/lessons-learned-from-electronjs/.

Process Model. N.d. Electron.js dokumentaatio. Viitattu 4.1.2022. https://www.electronjs.org/docs/latest/tutorial/process-model.

What is Electron.js. 2020. Artikkeli. Julkaistu 15.12.2020. Viitattu 25.11.2021. https://brain-hub.eu/library/what-is-electron-js/.

Zhao, C. 2016. Evolution of Electron. Blogi. Julkaistu 13.11.2016. Viitattu 25.11.2021. https://web.archive.org/web/20211119182636/http://cheng.guru/blog/2016/11/13/evolution-of-electron.html