Arttu Ojala

MOBIILISOVELLUKSEN TOTEUTUS ANDROID- JA IOS-ALUSTOILLE

Xamarin vastaan natiivi

Opinnäytetyö
Tieto- ja viestintätekniikan koulutus

2019

Kaakkois-Suomen
ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Arttu Ojala	Insinööri (AMK)	Joulukuu 2019

Opinnäytetyön nimi

Mobiilisovelluksen toteutus Android- ja iOS-alustoille : Xamarin vastaan natiivi

32 sivua
0 liitesivua

Toimeksiantaja

GoodLife Technology Oy

Ohjaaja

Lehtori Niina Mässeli

Tiivistelmä

Opinnäytetyön toimeksiantaja, GoodLife Technology Oy, käyttää Android- ja iOS-mobiilisovellusten toteutukseen Xamarin-kehitystyökaluja. Xamarin mahdollistaa ohjelmakoodin jakamisen Android- ja iOS-alustojen välillä, mikä nopeuttaa mobiilisovellusten kehitystä. Opinnäytetyön tavoitteena oli selvittää, voisiko toimeksiantaja hyötyä mobiilisovellusten toteutustavan vaihtamisesta natiiveihin työkaluihin.

Opinnäytetyön aikana suunniteltiin ja toteutettiin kaksi Android-sovellusta, kaksi iOS-sovellusta sekä Azure-pilvirajapinta. Android- ja iOS-mobiilisovellukset toteutettiin sekä Xamarin- että natiivityökaluja käyttäen. Mobiilisovelluksissa voi lukea käyttäjien kommentteja sekä lähettää uusia kommentteja. Mobiilisovelluksista tehtiin ulkonäöltään ja toiminnallisuudeltaan samanlaisia, jotta toteutusprosessien erot saataisiin selville. Azure-rajapintaan toteutettiin toiminnallisuudet kommenttien hakemiseen ja tallentamiseen. Kaikki mobiilisovellukset integroitiin käyttämään toteutettua Azure-rajapintaa.

Mobiilisovellusten toteutusprosessien tuloksena todettiin, että käyttöliittymien toteutus natiivityökalujen visuaalisilla editoreilla on helpompaa kuin Xamarin-sovellusten käyttöliittymien toteutus. Xamarin-sovelluksista saa kuitenkin tehtyä lähes identtisiä natiivisovellusten kanssa, vaikka se onkin hieman vaikeampaa. Natiivisovellukset pitää kuitenkin toteuttaa molemmille alustoille erikseen, eikä ohjelmakoodia voi jakaa alustojen kesken kuten Xamarin-sovelluksissa, minkä takia toimeksiantaja ei päätynyt vaihtamaan mobiilisovellusten toteutustapaa.

Asiasanat

mobiilisovellus, Xamarin, Android, iOS, Azure

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Arttu Ojala	Bachelor of Engineering	December 2019

Thesis title

Mobile app development for Android and iOS platforms : Xamarin versus native

32 pages
0 pages of appendices

Commissioned by

GoodLife Technology Oy

Supervisor

Niina Mässeli, Senior Lecturer

Abstract

The goal of this thesis was to research if the commissioner of the thesis, GoodLife Technology Oy, could benefit from switching from Xamarin mobile app development to native mobile app development. The thesis was conducted as a comparison between Xamarin, Android and iOS development processes.

The implementation process began with designing a mobile app which would be created in multiple different development environments. Microsoft Azure API was also designed, which all four mobile apps would be integrated with. The philosophy of the design process was to guarantee that there would be a parity between the mobile apps to ensure that the comparison between the implementation processes would be as fair as possible.

Two Android apps, two iOS apps and an Azure API were created during the implementation process. The mobile apps can send and receive comments via the API. The result of the research was that the native tools offer better visual UI-tools, slightly better performance and less bugs than the Xamarin framework, but in the end the native tools do not bring enough added value for the commissioner to switch their current mobile development methods.

Keywords

mobile app, Xamarin, Android, iOS, Azure

SISÄLLYS

KUVALUETTELO

1 JOHDANTO

Tämän opinnäytetyön toimeksiantaja, GoodLife Technology Oy, on 2013 perustettu teknologiayritys, jolla on toimipisteet Kotkassa ja Helsingissä. GoodLife kehittää toimivia ja helppokäyttöisiä ratkaisuja fysioterapian ja kuntoutuksen avuksi hyödyntäen uusia teknologioita, kuten liiketunnistusta. Sen lisäksi GoodLife toteuttaa tilaustyönä ohjelmisto- ja teknologiaratkaisuja, kuten mobiilisovelluksia ja internetsivustoja. GoodLife työllistää noin kymmenen työtekijää.

Toimeksiantaja käyttää Android- ja iOS-mobiilisovellusten toteutukseen Xamarin-kehitystyökaluja. Tämän opinnäytetyön tarkoituksena on selvittää, olisiko toimeksiantajalle kannattavaa siirtyä toteuttamaan mobiilisovelluksia Android- ja iOS-alustojen natiiveilla työkaluilla Xamarin-työkalujen sijaan. Monialustaisten mobiilisovellusten kehitykseen on olemassa muitakin kehitystyökaluja kuin Xamarin, kuten Ionic, React Native ja Flutter, mutta niitä ei käsitellä tässä opinnäytetyössä.

Opinnäytetyön aikana toteutettiin sama mobiilisovellus neljällä eri toteutustavalla: Xamarin Androidilla, Xamarin iOSilla, natiivi Androidilla ja natiivi iOSilla. Mobiilisovellusten lisäksi opinnäytetyön aikana toteutettiin rajapinta Azure-pilvipalveluun, johon kaikki neljä sovellusta integroitiin. Toimeksiantaja käyttää mobiilisovelluksissa paljon Azure-pilvipalveluja, joten mobiilisovellusten helppo integroitavuus Azure-palveluihin on toimeksiantajalle tärkeää.

Opinnäytetyön alussa käydään läpi mobiilisovellusten sekä Azure-rajapinnan suunnitteluprosessia. Sen jälkeen tutustutaan tarkemmin opinnäytetyön toteutuksessa käytettyihin kehitystyökaluihin ja menetelmiin. Seuraavaksi tarkastellaan mobiilisovellusten sekä pilvirajapinnan toteutusprosesseja. Opinnäytetyön lopussa tarkastellaan opinnäytetyön tuloksia ja niistä saatuja johtopäätelmiä, sekä vertaillaan eri kehitystyökalujen hyviä ja huonoja puolia sekä eroavaisuuksia.

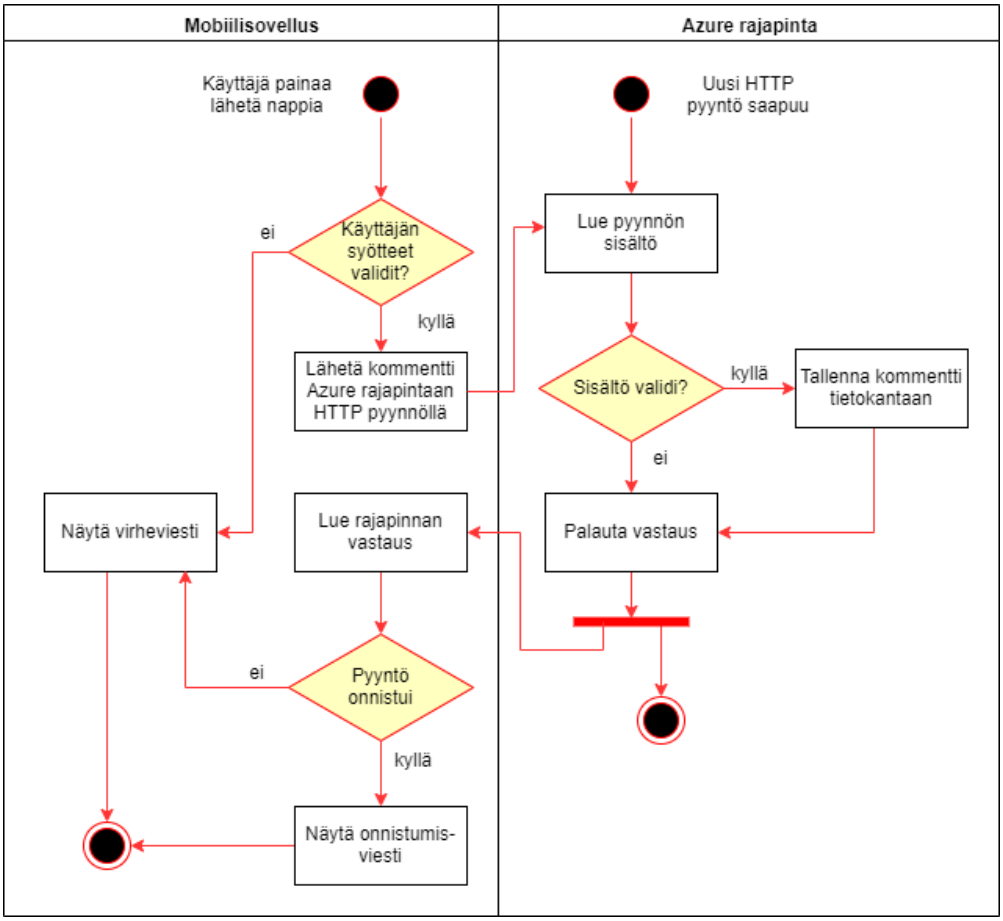
2 SUUNNITTELU

Kehittämisprosessi aloitettiin sovelluksen suunnittelulla. Sovelluksen tulisi olla yksinkertainen, koska muuten neljän sovelluksen kehittämiseen menisi liian paljon aikaa, mutta samalla sovelluksen tulisi olla tarpeeksi kattava, jotta eri versioissa on tarpeeksi vertailtavaa.

Suunnittelun alussa sovellukselle valittiin seuraavat kriteerit:

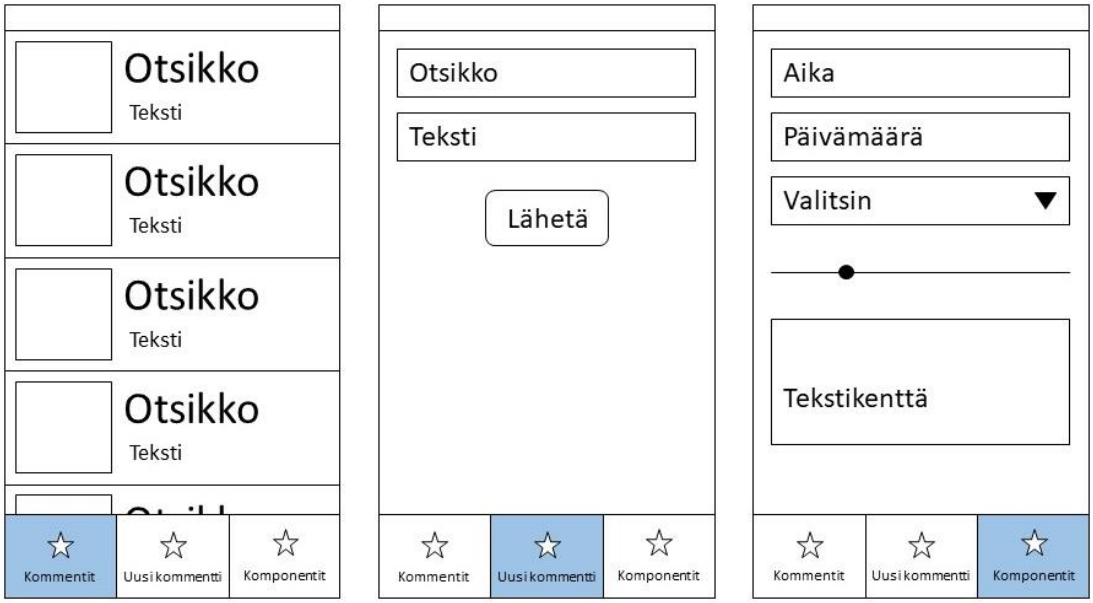
- Sovelluksen tulee hakea dataa Azure-rajapinnasta.
- Sovelluksen tulee lähettää dataa Azure-rajapintaan.
- Sovelluksen tulee sisältää paljon erilaisia käyttöliittymäkomponentteja.
- Sovelluksen tulee olla yksinkertainen toteuttaa.

Suunnittelussa päädyttiin sovellukseen, jolla voi lukea kommentteja sekä lähettää uusia kommentteja. Kuvassa 1 on suunniteltu toimintakaavio kommentin tallentamisesta, missä kommentti lähetetään Azure-rajapintaan, joka tallentaa kommentin tietokantaan. Kommenttien hakeminen ja lähettäminen toteutettaisiin http-kutsuja käyttäen.



Kuva 1. Toimintakaavio kommentin tallentamisesta

Kuvassa 2 on suunnitelma mobiilisovelluksen käyttöliittymistä. Sovelluksessa on kolme sivua, joiden välillä navigoidaan sovelluksen alapalkissa olevaa navigointipalkkia käyttäen. Ensimmäisellä sivulla on lista, missä näytetään Azure-rajapinnasta haetut kommentit. Toisella sivulla on kaksi tekstikenttää ja nappi; ensimmäiseen tekstikenttään voi kirjoittaa kommentin otsikon ja toiseen kenttään kommentin viestin. Nappia painamalla kommentti lähetetään Azure-rajapintaan, missä se tallennetaan tietokantaan.



Kuva 2. Mobiilisovelluksen käyttöliittymien hahmotelma

Kuvan 2 oikeassa reunassa on sovelluksen kolmas sivu, missä on kokoelma erilaisia käyttöliittymäelementtejä ja -komponentteja. Sivun tarkoituksena on vertailla elementtien ulkonäköä ja toiminnallisuutta eri alustojen välillä.

3 KEHITYSTYÖKALUT

Tässä luvussa tarkastellaan opinnäytetyön kehityksessä käytettyjä työkaluja. Ensimmäisenä tutustutaan Xamarin-kehitystyökaluihin, jonka jälkeen tarkastellaan natiiveja Android- ja iOS-työkaluja. Luvun lopussa tutustutaan lyhyesti opinnäytetyössä käytettyihin Microsoft Azure -työkaluihin ja -palveluihin.

3.1 Xamarin

Xamarin on avoimen lähdekoodin kehitystyökalu mobiilisovellusten kehittämiseen. Xamarin mahdollistaa koodin jakamisen useiden eri alustojen välillä, mutta säilyttää natiivisovellusten ulkonäön ja suorituskyvyn. Xamarin-sovellusten kehittämiseen käytetään C#-ohjelmointikieltä sekä Xaml-merkintäkieltä. Tuettuja alustoja ovat Android, iOS, Universal Windows Platform sekä osittain tuettuja alustoja ovat Tizen, macOS, GTK# ja WPF. Xamarin on saatavilla Windows-laitteilla Microsoft Visual Studio -ohjelmiston mukana ja macOS-laitteilla Visual Studio for Mac -ohjelmiston mukana. (Microsoft 2019e.)

Xamarin-asennus sisältää Android-emulaattorin, joten Android-projekteja voi kehittää suoraan Visual Studio -ohjelmistolla, vaikka ei omistaisi fyysistä Android-laitetta. Visual Studio -ohjelmistolla voi kehittää myös iOS-projekteja, mutta Visual Studio pitää olla liitettynä lähiverkossa olevaan macOS-laitteeseen missä on Xcode-ohjelmisto asennettuna, jotta iOS-projekteja voi kehittää Windowsilla. MacOS-laitteilla Xamarin-sovelluksia voi kehittää Visual Studio for Mac -ohjelmistolla. Siinä tulee mukana Android-emulaattori sekä iOS-simulaattori, joten sillä voi kehittää molempia Android- ja iOS-sovelluksia. (Microsoft 2019e.)

Xamarin.iOS ja Xamarin.Android on rakennettu Mono-sovelluskehyksen päälle. Mono on avoimen lähdekoodin toteutus Microsoft .NET-sovelluskehyksestä. Xamarin.iOS käyttää Ahead-Of-Time-kääntämistekniikkaa, missä projekti käännetään suoraan natiiviksi ARM-konekieleksi. Xamarin.Android hyödyntää Just-In-Time-kääntämistä, missä Xamarin kääntää projektin välittäjäkieleksi, ja välittäjäkieli käännetään natiiviksi konekieleksi vasta kun sovellus aukaistaan. Xamarin.iOS tuottaa ipa-tiedostoja ja Xamarin.Android apk-tiedostoja, mitkä ovat samat mitä natiivityökalut tuottavat. (Microsoft 2019d.)

Xamarin-sovellukset näyttävät erilaisilta eri alustoilla, vaikka ne kaikki käyttävät samaa koodia. Xamarin käyttää jokaisella alustalla sen omia käyttöliittymäkomponentteja, minkä takia eri alustojen sovellukset näyttävät hieman erilaisilta, vaikka ne käyttävät samaa koodia. Natiivikomponenttien käyttö takaa sen, että käyttöliittymä näyttää käyttäjille tutulta ja suorituskyky pysyy korkeana.

Sovelluksiin voi tehdä yksinkertaisia animaatioita animaatiorajapinnan avulla. Rajapinnan avulla visuaalisia elementtejä voi siirtää, kääntää, häivyttää ja skaalata. Animaatiossa tulee määrittää, mitä elementtiä animoidaan, animaation kesto, animaation tavoite, esimerkiksi x- ja y-koordinaatit sekä animaation nopeuskäyrä. Animaation nopeuskäyrällä voidaan määrittää animaatiolle lineaarinen tai epälineaarinen nopeus, esimerkiksi animaatio voi alkaa hitaasti ja nopeutua loppua kohti, mikä saa animaation näyttämään sulavammalta. Valmiina valittavista funktioista löytyy muun muassa sini- ja eksponentiaalinen nopeuskäyrä. (Microsoft 2019b.)

3.1.1 Xaml

Sovellusten käyttöliittymiä voi tehdä kahdella eri tavalla: Xaml-merkintäkielellä tai C#-ohjelmointikielellä. Kuvassa 3 on esimerkki XML-merkintäkieleen pohjautuvasta Xaml-merkintäkielestä. Xaml-merkintäkieltä suositellaan käytettäväksi käyttöliittymien määrittelyyn, koska se on luettavampaa kuin vastaava C#-koodilla määritelty käyttöliittymä, koska siinä on helpompi tulkita komponenttien välistä hierarkiaa, sekä sitä on helpompi generoida automaattisesti. Se myös soveltuu MVVM-malliin paremmin kuin C#-käyttöliittymä. (Microsoft 2017b.)

```xml
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Oppari.Views.ProductListView">
    <ContentPage.Content>
        <ListView ItemsSource="{Binding Items}"
                  RowHeight="120">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <Grid Padding="10"
                              BackgroundColor="White"
                              HeightRequest="120"
                              ColumnSpacing="10">

                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="100"/>
                                <ColumnDefinition Width="*"/>
                            </Grid.ColumnDefinitions>

                            <Image Source="ic_launcher.png"
                                   WidthRequest="80"
                                   HeightRequest="80"
                                   VerticalOptions="Center"
                                   HorizontalOptions="Center"/>

                            <StackLayout Grid.Column="1"
                                         VerticalOptions="Center"
                                         Spacing="0">

                                <Label Text="{Binding Title}"
                                       FontSize="Large"/>

                                <Label Text="{Binding Description}"
                                       FontSize="Medium"/>
                            </StackLayout>
                        </Grid>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </ContentPage.Content>
</ContentPage>
```

Kuva 3. Esimerkki Xaml-merkintäkielestä

Xaml-käyttöliittymissä on myös huonoja puolia C#-käyttöliittymiin verrattuna: Xaml ei voi sisältää ohjelmakoodia, joten kaikki logiikka täytyy eritellä toiseen C#-tiedostoon. Xaml ei myöskään sisällä ehto- tai toistorakenteita, joten monimutkaisten käyttöliittymien tekemiseen tarvitsee hyödyntää myös C#-koodia. Käyttöliittymiin ei ole visuaalista suunnittelutyökalua, vaan ohjelma pitää kääntää uudelleen jokaisen muutoksen jälkeen, mikä tekee käyttöliittymien hienosäädöstä hidasta. (Microsoft 2017b.)

Xaml-tiedostot voidaan kääntää välittäjäkieleksi sovelluksen käännösvaiheessa, tai ne voidaan kääntää sovelluksen suorituksen aikana. Xamarin suosittelee Xaml-tiedostojen kääntämistä sovelluksen käännösvaiheessa, koska se mahdollistaa virheiden löytämisen ennen sovelluksen suoritusta, nopeuttaa

Xaml-elementtien latausaikoja ja pienentää sovelluksen kokoa. (Microsoft 2018b.)

3.1.2 Alustakohtaiset ominaisuudet

Xamarin-kehityksessä joitakin ominaisuuksia pitää tehdä alustakohtaisesti. Resursseja, kuten kuvia ja fontteja, käsitellään joka alustalla eri tavalla ja ne pitää lisätä jokaiseen projektiin erikseen kyseisen alustan vaatimalla tavalla. Alustakohtaisesti pitää myös tehdä mm. latausnäyttö, sovelluksen ikoni ja muut sovelluskohtaiset asetukset, kuten käytetäänkö sovellusta pysty- ja vaaka-asennossa ja mitä käyttölupia sovellus tarvitsee toimiakseen.

Xamarin-projektit on jaettu useisiin eri aliprojekteihin. Jaetulle koodille ja muille resursseille on yksi projekti, sekä jokaiselle halutulle alustalle oma projektinsa. Jaettuun projektiin laitetaan kaikki koodi, mikä halutaan jakaa kaikkien eri versioiden välillä ja alustakohtaiset koodit laitetaan alustan omaan projektiin.

Interface

```
namespace GoodLifeMobileTrainer.Interfaces
{
    public interface IDevice
    {
        string GetDeviceId();
    }
}
```

Android

```
using Android.Provider;
using Xamarin.Forms;

[assembly: Dependency(typeof(GoodLifeMobileTrainer.Droid.Utilities.Device))]
namespace GoodLifeMobileTrainer.Droid.Utilities
{
    public class Device : Interfaces.IDevice
    {
        public string GetDeviceId()
        {
            return Settings.Secure.GetString(MainActivity.ContentResolverInstance, Settings.Secure.AndroidId);
        }
    }
}
```

iOS

```
using UIKit;
using Xamarin.Forms;

[assembly: Dependency (typeof(Device))]
namespace GoodLifeMobileTrainer.iOS.Utilities
{
    class Device : Interfaces.IDevice
    {
        public string GetDeviceId()
        {
            return UIDevice.CurrentDevice.IdentifierForVendor.AsString();
        }
    }
}
```

Kuva 4. Esimerkki DependencyService toteutuksesta

Joskus alustakohtaisia ominaisuuksia tarvitsee myös jaetussa koodissa. Xamarin sisältää DependencyService-ominaisuuden, jolla kehittäjä voi määrittää yhteisen rajapinnan, jolle tehdään toteutus erikseen jokaiselle alustalle, kuten kuvan 4 esimerkissä toteutettu laitetunnisteen hakeminen. Rajapinnan palautusarvoja voi käyttää jaetussa koodissa normaalisti, koska funktioiden palautusarvot on jo määritelty rajapinnassa etukäteen. Sen avulla voi käyttää alustakohtaisia ominaisuuksia ilman että tarvitsee kirjoittaa ehtolauseita jaettuun koodiin. Yleisiä käyttökohteita ovat muun muassa laitetunnus, video- ja äänisoitin, lokalisaatio ja ilmoitukset.

Suorituskykyero natiivi- ja Xamarin-sovellusten välillä on pieni, koska Xamarin sisältää sidokset eri alustojen natiiveihin funktioihin. Sen ansiosta C#-koodista voi suoraan kutsua Java- ja Objective-C-toteutuksia. Välillä haluttua koodia ei meinaa löytyä, koska C#-nimeämistyyli on erilainen muiden alustojen nimeämistyyliin verrattuna. Natiivisidokset ovat käytettävissä ainoastaan sen alustan projektissa, jaetusta koodista ei voi suoraan kutsua natiivikoodia. Natiivisidosten muuttujat on tyypitetty vahvasti, mikä auttaa välttämään suorituksenaikaisia virheitä, koska muuttujatyypit tarkistetaan jo kääntämisvaiheessa.

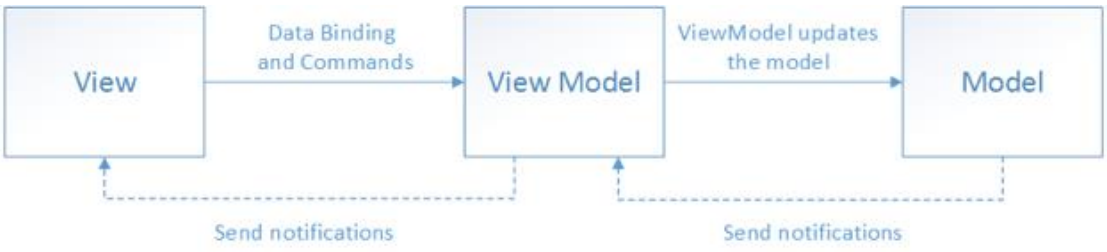
Xamarin tukee omia kustomoituja käyttöliittymäkomponentteja. Komponentit voivat perustua johonkin valmiiseen Xamarin-komponenttiin ja käyttää sen komponentin renderöijää, tai ne voivat olla täysin uusia komponentteja, joilla on oma renderöijä. Kustomoidut komponentit ovat hyödyllisiä, koska Xamarin ei sisällä kaikkia mahdollisia komponentteja, koska niillä ei ole vastaavaa natiivikomponenttia kaikilla tuetuilla alustoilla.

Käyttöliittymät renderöidään jokaisen alustan omia komponentteja käyttäen, jotta sovellukset näyttäisivät mahdollisimman paljon natiivisovelluksen näköisiltä. Valmiiden ja kustomoitujen komponenttien ulkonäköä voi muuttaa alustakohtaisesti Xamarin Custom Renderer -ominaisuutta käyttäen. Se on todella tehokas ominaisuus hienojen käyttöliittymien tekemiseen, mutta se pitää tehdä jokaiselle alustalle erikseen, joten sen käyttämistä kannattaa harkita tarkasti.

3.1.3 MVVM

MVVM eli Model-View-ViewModel on Xamarin-sovellusten kehittämiseen suositeltu arkkitehtuurimalli. MVVM-mallin ideana on erottaa käyttöliittymä ja sovelluksen logiikka kokonaan toisistaan. MVVM helpottaa logiikan testaamista, koska käyttöliittymää ei tarvitse ottaa ollenkaan huomioon testeissä. Mallin avulla koko käyttöliittymä voidaan suunnitella uudestaan, ilman että taustalla olevaa dataa tai logiikkaa muutetaan ollenkaan. Useita kehittäjiä voi työstää samaa sivua saman aikaa; yksi kehittäjä voi suunnitella käyttöliittymää samalla kun toinen tekee siihen liittyvää logiikkaa. (Microsoft 2017a.)

MVVM koostuu kolmesta komponentista: Model, View ja ViewModel, mitkä näkyvät kuvassa 5. Model sisältää sovellukseen liittyvää dataa sekä logiikkaa datan käsittelyyn. ViewModel on käyttöliittymän malli; se muokkaa Model-luokassa olevan datan sellaiseen muotoon, että sen voi näyttää käyttöliittymässä sekä tarvittaessa muokkaa Model-luokan dataa käyttöliittymän syötteiden perusteella. (Microsoft 2017a.)



Kuva 5. MVVM-malli (Microsoft 2017a)

View on käyttöliittymä, joka on yleensä määritelty Xaml-merkintäkielellä. Käyttöliittymän dataa voidaan yhdistää ViewModel-luokan muuttujiin Binding-ominaisuuden avulla. Esimerkiksi tekstikentän fontti, fonttikoko ja sijainti voi olla määritetty staattisesti Xaml-tiedostossa, mutta tekstikentän teksti haetaan ViewModel-luokasta. Kun tekstikentän tekstiin liitetyn muuttujan arvo muuttuu, päivitetty arvo haetaan ViewModel-luokasta ja käyttöliittymään päivitetään tekstikentän uusi arvo. ViewModel ei siis suoraan muokkaa käyttöliittymää, vaan se vain ilmoittaa milloin data on muuttunut, jotta View pystyy mukautumaan muutoksiin. (Microsoft 2017a.)

MVVM-mallin käyttämiseen on saatavilla useita kirjastoja, kuten MVVMLight ja FreshMVVM. Kirjastot helpottavat MVVM-mallin käyttöä, koska niissä on valmiina toteutettu Binding-ominaisuus View- ja ViewModel-komponenttien välillä. Kirjastoissa on yleensä valmiina myös sivunavigoinnin toteutus, jotta sivujen välillä pystyy navigoimaan helposti suoraan ViewModel-luokasta.

3.1.4 Asynkroninen ohjelmointi

Asynkronista ohjelmointia käytetään esimerkiksi http-kutsuihin, koska kutsut saattavat kestää useita sekunteja. Asynkronista ohjelmointia tarvitaan, koska muuten koko mobiilisovellus pysähtyisi http-kutsun suorituksen ajaksi. Asynkronisen ohjelmoinnin avulla http-kutsu voidaan suorittaa taustalla, ja sovellus pystyy jatkamaan toimintaansa normaalisti http-kutsun aikana.

C#-ohjelmointikielessä asynkroninen ohjelmointi tapahtuu TAP-mallin (Task-based asynchronous pattern) mukaisesti. Malli mahdollistaa helposti luettavan asynkronisen ohjelmakoodin kirjoittamisen, vaikka koodin suoritus onkin todellisuudessa paljon monimutkaisempaa (Microsoft 2019c).

TAP-mallin perustana on Task-luokka, millä luodaan asynkronisia tehtäviä, sekä async- ja await-avainsanat. Async-avainsana pitää lisätä funktioihin, missä kutsutaan asynkronista koodia. Await-avainsanalla voidaan odottaa asynkronisen tehtävän suoritusta, ennen kuin funktion suoritusta jatketaan seuraavalle koodiriville.

3.2 Android Studio

Android Studio on virallinen ohjelmointiympäristö Android-sovellusten kehittämiseen. Ohjelmointiympäristön pohjana on JetBrains-yrityksen kehittämä IntelliJ IDEA Community Edition -ohjelmointiympäristö, mihin Google on lisännyt työkaluja Android-sovellusten kehittämisen tueksi (Ducrohet ym. 2013). Android-sovelluksia voi kehittää Java-, Kotlin- ja C++-ohjelmointikielillä, mutta kehityksessä suositellaan käyttämään Kotlin-ohjelmointikieltä (Haase 2019).

Android-sovelluksissa ohjelman pääsäie vastaa käyttöliittymän päivittämisestä. Jos pääsäie kuormittuu likaa, ohjelma saattaa hidastua tai pysähtyä.

Kotlin-ohjelmointikielessä coroutines-ominaisuus mahdollistaa koodin suorittamisen ohjelman eri säikeillä, sekä suoritussäikeen vaihtamisen koodin suorituksen aikana.

3.3 Xcode

Xcode on kehitysympäristö Mac-, iPhone-, iPad-, Apple Watch-, ja Apple TV -sovellusten kehittämiseen. Xcode sisältää työkalut sovellusten kehittämiseen, testaamiseen, optimointiin sekä julkaisuun. Xcode tukee sovellusten kehittämistä Swift-, Objective-C-, C- ja C++-ohjelmointikielillä. (Apple s.a.)

Xcode sisältää graafisen suunnittelutyökalun käyttöliittymien tekemiseen. Navigoinnin sovelluksen sivujen välillä voi myös konfiguroida visuaalisesti storyborad-editorin avulla (Apple s.a.). Käyttöliittymäelementteihin voi tehdä helposti referenssejä kooditiedostoihin vetämällä elementin graafisesta suunnittelutyökalusta kooditiedostoon, mikä helpottaa käyttöliittymäelementtien hallintaa ohjelmallisesti.

3.4 Azure-pilvipalvelut

Azure tarjoaa paljon monipuolisia pilvipalveluita, mutta tässä oppinäytetyössä tarkastellaan ainoastaan opinnäytetyön toteutuksessa käytettyjä Azure Functions- ja Azure Table Storage -palveluita.

3.4.1 Azure Functions

Azure Functions -palvelun avulla on helppo kehittää skaalautuvia ja halpoja sovellusten rajapintoja ja taustajärjestelmiä. Funktioita voi kehittää monilla eri ohjelmilla ja ohjelmointikielillä, kuten C#, JavaScript, Java ja Python. (Microsoft 2018a.)

Azure Functions -palvelun hinnoittelu perustuu funktioiden suorituskertojen ja käytettyjen resurssien määrään. Käytetty resurssimäärä lasketaan gigatavusekunteina, missä funktion käyttämä muistimäärä kerrotaan funktion suoritukseen kuluneella ajalla. Funktio, mikä käyttää 500 Mt muistia ja kestää yhden sekunnin, käyttää resursseja 0,5 Mt-s verran. Funktion käyttämä muistimäärä

pyöristetään lähimpään 128 Mt:n, ja se voi olla maksimissaan 1 536 Mt. Funktion suoritusaika mitataan millisekunneissa. Palveluun kuuluu ilmaiseksi 1 000 000 funktion suorituskertaa ja 400 000 Gt-s resurssikäyttöä kuukaudessa. (Microsoft s.a.) Jokaiseen funktiosovellukseen kuuluu myös tallennustili, jota voi käyttää tiedon tallentamiseen esim. tietokantaan tai tiedostoihin. Tallennustilin käytöstä veloitetaan erikseen siirretyn sekä tallennetun datan ja tietokantapyyntöjen määrän mukaan.

Azure Functions -palvelu skaalautuu tarvittavan kapasiteetin mukaan automaattisesti. Funktioon voidaan määrittää, kuinka monta samanaikaista pyyntöä yhdellä instanssilla voi olla samaan aikaan. Kun sallittujen samanaikaisten pyyntöjen määrä tulee täyteen, Azure ottaa automaattisesti uuden instanssin käyttöön. Jos käynnissä olevien pyyntöjen määrä laskee, Azure sulkee automaattisesti ylimääräisiä instansseja, jotta käyttäjälle ei koidu ylimääräisiä kustannuksia turhista instansseista.

Automaattinen skaalautuvuus tekee Functions-palvelusta erittäin helppokäyttöisen, koska piikkeihin sovelluksen käytössä ei tarvitse erikseen varautua, palvelimia ei tarvitse varata, eikä tyhjäkäyvistä palvelimista tarvitse maksaa turhaan.

Funktioiden päivittäminen Functions-palvelussa on helppoa. Jokaiselle Functions-sovellukselle on monta tallennuspaikkaa, joihin voidaan laittaa useita eri versioita sovelluksesta. Uutta versiota voidaan ensin testata toisessa tallennuspaikassa, ja kun uusi versio on todettu toimivaksi, tuotannossa oleva sovellus voidaan vaihtaa yhdellä napinpainalluksella. Jos uudessa sovellusversiossa on jotain vialla, voidaan sovellus vaihtaa takaisin vanhaan versioon nopeasti.

Funktioiden suoritus voidaan aloittaa muutamien erilaisten tapahtumien seurauksena, minkä ansiosta Azure Functions sopii moniin erilaisiin käyttötapauksiin. Tässä raportissa käsitellään HttpTrigger- ja TimerTrigger-ominaisuuksia, mutta saatavilla on muitakin funktioiden suoritustyyppejä, kuten QueueTrigger ja BlobTrigger. (Microsoft 2019a.)

HttpTrigger-funktio suoritetaan aina, kun funktioon lähetetään http-pyyntö. Esimerkiksi mobiilisovelluksen sisäänkirjautuminen voi olla HttpTrigger-funktio. Kun funktioon tulee kirjautumispyyntö, funktio tarkistaa, että käyttäjän kirjautumistiedot ovat oikein ja palauttaa vastauksena esimerkiksi session tokenin taikka virheviestin virheellisestä kirjautumisesta.

TimerTrigger-funktio suoritetaan ajastimen mukaan. Ajastin voidaan asettaa suorittamaan funktio esimerkiksi viiden minuutin välein, joka päivä kello 12.00 tai joka kuukauden ensimmäisenä maanantaina. TimerTrigger-funktiota voi käyttää esimerkiksi vuorokautisen käyttökatsauksen luomiseksi; ajastin laukeaa kerran päivässä, jolloin funktio luo päivän aikana kertyneestä datasta päiväkatsauksen ja lähettää sen sähköpostilla asiakkaalle tai ylläpitäjälle.

3.4.2 Azure Table Storage

Azure Table Storage on avain-arvopareihin perustuva NoSQL-tietokanta. Se on vain osittain järjestelty tietokanta, joten samaan tietokanta tauluun voi tallentaa monia erityyppisiä entiteettejä.

Data tallennetaan PartitionKey- ja RowKey-arvojen mukaan. PartitionKey määrittää, mihin osioon tietokantarivi kuuluu. Tietokantojen jakaminen osioihin on tärkeää, koska tietokantataulun eri osiot voivat sijaita eri palvelimilla ja Azure saattaa automaattisesti siirtää osioita eri palvelimille niiden käyttöasteen mukaan. Osio, jota käytetään paljon, saatetaan siirtää palvelimelle, joka tukee suurempaa määrää liikennettä kuin peruspalvelin. RowKey on tietokantarivin uniikki avain. Samaa avainta ei voi käyttää saman osion sisällä useita kertoja, mutta sama RowKey voi olla käytössä useissa eri osioissa.

Nopein tapa hakea dataa tietokannasta on määrittää haussa molemmat Partition- ja RowKey-arvot. Dataa voi myös hakea ja suodattaa muidenkin rivin arvojen mukaan, mutta haut useiden eri osioiden välillä ovat hitaampia kuin yhden osion sisäiset haut, koska osiot eivät välttämättä sijaitse samalla palvelimella, tai edes samassa palvelinsalissa. Tietokantarakenteen suunnittelu on tärkeää, jotta tietokantahaut ovat mahdollisimman nopeita ja käyttäjille saadaan tarjottua mahdollisimman hyvä käyttökokemus. Rivit, joita tarvitaan usein

samalla kertaa, kannattaa sijoittaa samaan osioon. Kaikkea dataa ei kuitenkaan kannata tallentaa yhteen osioon, koska haut ovat sitä hitaampia, mitä enemmän rivejä yhdellä osiolla on tallennettuna. Azure suosittelee useiden eri osioiden välisten tietokantahakujen välttämistä, PartitionKey kannattaa määrittää haussa aina, jos se vain on mahdollista.

Azure Table Storage mahdollistaa myös datan redundanttisuuden sekä monistamisen useaan eri maanosaan, jotta kaikilla käyttäjillä maailmassa on yhtä nopea käyttökokemus. Tietokannan käyttö tapahtuu toisen Azure-palvelun kautta, kuten esimerkiksi funktion sisällä, tai suoraan http-pyynnöillä REST-rajapinnan kautta. Tietokantoja, tiedostoja ja muita Azure-tallennuspalveluja voi tarkastella ja käyttää Azure Storage Explorer -ohjelmalla.

4 TOTEUTUS

Tässä luvussa tutustutaan tarkemmin mobiilisovellusten sekä Azure-rajapinnan toteutuksiin. Ensimmäiseksi käydään läpi Xamarin-sovellusten toteutus, jonka jälkeen on vuorossa natiivi Android-toteutus ja sen jälkeen natiivi iOS-toteutus. Lopuksi tarkastellaan Azure-rajapinnan toteutusta.

Toteutus tehtiin suunnitteluvaiheen aikana toteutetun suunnitelman pohjalta, mutta mobiilisovellukseen suunniteltu kolmas sivu, käyttöliittymäkomponenttien kokoelma, jätettiin lopullisesta toteutuksesta pois, koska sen ei koettu tuovan lisäarvoa toteutusprossien vertailuun.

4.1 Xamarin, Android ja iOS

Xamarin-sovellusten kehittäminen aloitettiin luomalla uusi Xamarin.Forms-projekti Visual Studio -ohjelmalla. Projektin pohjaksi valittiin tyhjä .NET Standard Android- ja iOS-projekti. Ensimmäisenä projektiin lisättiin TabbedPage-komponentti, mikä näkyy kuvassa 6. TabbedPage sisältää kaikki ohjelman eri sivut, mille halutaan oma välilehti. TabbedPage-komponentin navigointipalkin paikka määritettiin Android-versiossa sivun alaosaan lisäämällä sivun Xaml-tiedostoon android:TabbedPage.ToolbarPlacement="Bottom", koska Xamarin.Android-navigointipalkki on oletuksena sivun yläreunassa ja se haluttiin pitää sivun alareunassa, jotta se näyttäisi yhtenäiseltä iOS-version kanssa.

```
<?xml version="1.0" encoding="utf-8" ?>
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            xmlns:views="clr-namespace:Oppari.Views"
            xmlns:android="clr-namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"
            xmlns:ios="clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
            android:TabbedPage.ToolbarPlacement="Bottom"
            ios:Page.UseSafeArea="true"
            x:Class="Oppari.Views.MainPage">

    <views:ProductListView Title="Comments"
                           Icon="ic_action_list.png"/>
    <views:AddProductView Title="Add comment"
                          Icon="ic_action_add.png"/>
</TabbedPage>
```

Kuva 6. TabbedPage XAML-määrittely

Projektiin lisättiin kaksi Content Page -komponenttia. Content Page on tavallinen sivu, millä voi olla mitä tahansa sisältöä. Molemmille sivuille lisättiin ViewModel-tiedosto, johon sivuun liittyvä C#-ohjelmakoodi lisätään. Ensimmäiselle Content Page -sivulle lisättiin ListView-komponentti, mikä tulee sisältämään Azure-rajapinnasta ladatut kommentit listana. ListView-komponenttiin määritettiin listan solujen malli, mikä määrittää, miltä listan solut näyttävät. Mallin määritys tehtiin Xaml-merkintäkieltä käyttäen.

Kuvassa 7 on kommenttilistan päivitysominaisuuden määrittely. Kommenttilistan päivittäminen alaspäin vetämällä aktivoitiin lisäämällä ListView-komponentin määrityksiin IsPullToRefreshEnabled = ”True” sekä päivittämiseen käytettävä funktio määritettiin lisäämällä RefreshCommand = ”{Binding RefreshComments}”, mikä tarkoittaa, että päivitykseen käytetään RefreshComments-funktiota ViewModel-tiedostosta. IsRefreshing = ”{Binding IsRefreshing}” määrittää, että listaa päivitetään, kun ViewModel-tiedoston IsRefreshing-muuttuja on arvossa tosi.

```
<ListView ItemsSource="{Binding Items}"
          RefreshCommand="{Binding RefreshComments}"
          IsRefreshing="{Binding IsRefreshing}"
          IsPullToRefreshEnabled="True"
          VerticalOptions="Fill"
          HorizontalOptions="Fill"
          HasUnevenRows="True">
```

Kuva 7. ListView-komponentin määrittely

Kun käyttäjä vierittää kommenttilistaa alaspäin, sivun ViewModel-tiedoston RefreshComments-funktiota kutsutaan, mikä asettaa IsRefreshing-muuttujan arvoon tosi ja hakee Azure-rajapinnasta uuden kommenttilistan. Jos kom-

menttilistan haku onnistuu, lista tyhjennetään ja siihen lisätään Azure-rajapinnasta saadut uudet kommentit. Käyttäjälle näytetään viesti, jos kommenttien haku rajapinnasta epäonnistuu sekä IsRefreshing-muuttuja asetetaan takaisin arvoon epätosi.

Toiselle sivulle lisättiin kaksi Entry-komponenttia, Button-komponentti sekä ActivityIndicator-komponentti. Ensimmäiseen Entry-komponenttiin käyttäjä antaa kommentin otsikon ja toiseen kommentin viestin. Button-komponenttia painamalla käyttäjän kommentti lähetetään Azure-rajapintaan. Kun kommentin lähetys on käynnissä, molemmat Entry-komponentit ja Button-komponentti kytketään pois päältä, jotta käyttäjä ei pysty lähettämään samaa kommenttia uudelleen, kun edellisen kommentin lähetys on kesken. Sen lisäksi ActivityIndicator-komponentti tuodaan näkyviin, jotta käyttäjä näkee, että lataus on kesken. Kun Azure-rajapinnasta saadaan vastaus, sivun komponentit aktivoidaan jälleen ja ActivityIndicator-komponentti piilotetaan. Käyttäjälle näytetään viesti, missä ilmoitetaan, onnistuiko kommentin tallentaminen Azure-rajapinnan tietokantaan. Jos kommentin tallennus onnistui, Entry-komponenttien sisältö tyhjennetään, jotta käyttäjä voi kirjoittaa uuden kommentin.

4.2 Natiivi Android

Natiivi Android-sovellus toteutettiin lähes samalla tavalla kuin Xamarin-sovellus. Sovelluksen sivut toteutettiin Fragment-komponentteina. Sovelluksen pääsivun alareunassa on navigointipalkki, ja muu sivu sisältää sillä hetkellä valitun sivun Fragment-komponentin.

Navigointipalkin toteutus erosi hieman Xamarin-versiosta, etusivun XML-tiedostoon määritettiin BottomNavigationView-komponentti. Komponentin sisältö määriteltiin erilliseen XML-tiedostoon, mihin jokaisen välilehden otsikko ja kuva määriteltiin.

Kommenttilistan toteutus oli lähempänä Xamarin-toteutusta kuin navigointipalkki. Kommenttisivun fragment-komponenttiin lisättiin ListView-komponentti ja listan malli määritettiin erilliseen XML-tiedostoon. Kommenttilistaa varten piti toteuttaa myös ListAdapter, mikä määrittelee, miten listaan lisättävä data asetellaan XML-tiedostossa määriteltyyn malliin.

Sovelluksen Azure-integraatioon käytettiin OkHttp3-kirjastoa http-kutsujen toteutusta varten. Kommentit haetaan GET-metodilla Azure-rajapinnasta. Rajapinnan palauttama JSON-vastaus parsitaan sellaiseen muotoon, että dataa voidaan käyttää kommenttilistassa. Kommenttien lähetyksessä käyttäjän antama data muutetaan JSON-muotoon ja lähetetään POST-metodilla Azure-rajapintaan.

4.3 Natiivi iOS

Natiivi iOS-sovelluksen kehitys aloitettiin lisäämällä projektiin Tab Bar Controller. Tab Bar Controller hallitsee sovelluksen navigointipalkkia ja sen avulla määritetään mistä napista pääsee millekin sivulle. Xcode mahdollistaa navigoinnin määrittämisen visuaalisen editorin avulla, mikä ei ole mahdollista muilla opinnäytetyössä käytetyillä alustoilla. Visuaalisen editorin avulla projektiin lisättiin kaksi uutta sivua, ja määriteltiin navigointipalkki.

Seuraavaksi projektiin toteutettiin kommentin datamalli, mikä näkyy kuvassa 8. Datamalli sisältää kommentin tiedot: otsikon, viestin ja aikaleiman, mitkä haetaan Azure-rajapinnasta. Datamalli periytyy Codable-luokasta, jotta se voidaan serialisoida ja deserialisoida objektin ja Azure-rajapinnannasta saadun JSON-muotoisen kommentin välillä käyttäen JSONEncoder- ja JSONDecoder-luokkia.

```
class CommentList: Codable {
    var items: Array<Comment> = []
}

class Comment: Codable {
    var title: String
    var message: String
    var timestamp: Date

    init(title: String, description: String, timestamp: Date) {
        self.title = title
        self.message = description
        self.timestamp = timestamp
    }
}
```

Kuva 8. Kommentin datamalli

Ensimmäiselle sivulle lisättiin Table View -elementti. Table View -elementtiä käytettiin kommenttilistan visualisointiin. Elementtiin lisättiin View Cell -elementti, mikä määrittää yksittäisen listaelementin mallin. Malliin lisättiin visuaalisella editorilla kolme tekstikenttää, otsikko, viesti ja aikaleima. Seuraavaksi View Cell -elementin tekstikentistä liitettiin referenssit kooditiedostoon, jotta elementin tekstikenttiä voitiin muokata ohjelmallisesti.

Datamallin data liitettiin View Cell -elementin tekstikenttiin käyttämällä TableViewController-luokkaa. TableViewController saa listan kommenttien datamalleja, ja se liittää datamallissa olevan datan View Cell -elementin kooditiedostossa referoituihin tekstikenttiin.

```
static func getComments(completionHandler: @escaping (Array<Comment>?) -> Void) {
    let url = URL(string: baseUrl + "comment")!
    var request = URLRequest(url: url)
    request.addValue(functionsKey, forHTTPHeaderField: "X-Functions-Key")

    let task = URLSession.shared.dataTask(with: request) {(data, response, error)
        in guard let data = data else {
            completionHandler(nil)
            return
        }

        do {
            let responseContent = String(data: data, encoding: .utf8)!
            print(responseContent)
            let comments = try jsonDecoder.decode(CommentList.self, from: data)
            completionHandler(comments.items)
        }
        catch {
            completionHandler(nil)
        }
    }

    task.resume()
}
```

Kuva 9. Kommenttien haku rajapinnasta

Projektin integrointi Azure-rajapintaan aloitettiin toteuttamalla kommenttien haku rajapinnasta, minkä toteutus näkyy kuvassa 9. Kommenttienhakufunktion parametrina on completionHandler, mikä on callback-funktio, jota kutsutaan, kun kommenttien lataus on suoritettu tai jos kommenttien lataus epäonnistuu. Jos kommenttien lataus onnistuu, rajapinnasta saatu vastaus yritetään muuntaa JSON-tiedostosta CommentList-objektiksi. Muunnoksen onnistuessa

kutsutan sisääntuloparametrina saatua callback-funktiota, mihin annetaan parametriksi lista saaduista kommenteista.

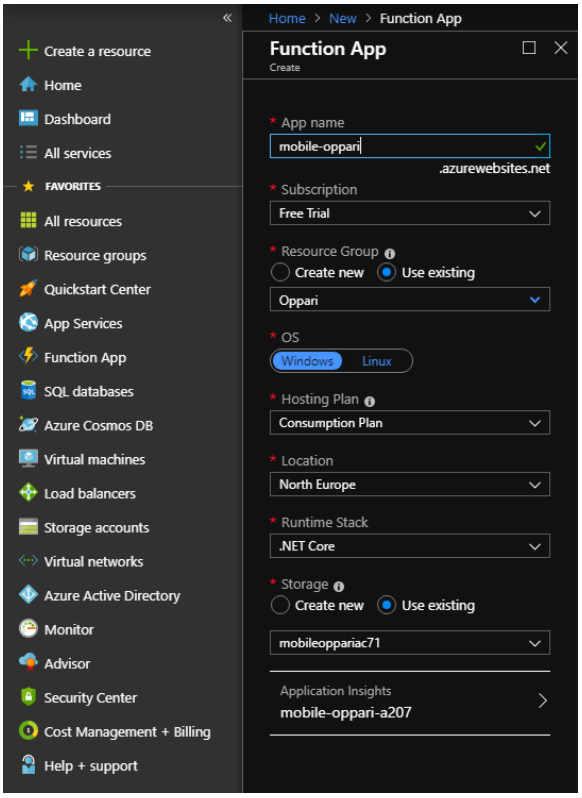
```
var comments = [Comment]()

private func loadComments() {
    Azure.getComments(completionHandler: onCommentsLoaded)
}

private func onCommentsLoaded(comments: Array<Comment>?) {
    if (comments != nil) {
        self.comments = comments!

        DispatchQueue.main.async {
            self.tableView.reloadData()
        }
    }
}
```

Kuva 10. Kommenttien päivitys listaan

Kuvassa 10 on ohjelmakoodi kommenttien päivittämiseen. Funktiossa loadComments kutsutaan Azure-rajapintatoteutuksen getComments-funktiota, jonka sisääntuloparametrina on callback-funktio, jota kutsutaan, kun kommentit on ladattu. Parametriksi annetaan onCommentsLoaded-funktio, joka päivittää luokan sisältämän kommenttilistan, jos kommenttien haku onnistuu. Listan sisällön päivitys kutsutaan DispatchQueue.main.async-blokin sisällä, mikä takaa sen, että kyseinen koodi kutsutaan ohjelman pääsäikeessä. Koodin kutsuminen pääsäikeessä on tärkeää, koska kommenttien lataus suoritetaan taustasäikeellä, jolla ei saa tehdä muutoksia käyttöliittymään.

4.4 Azure-rajapinta

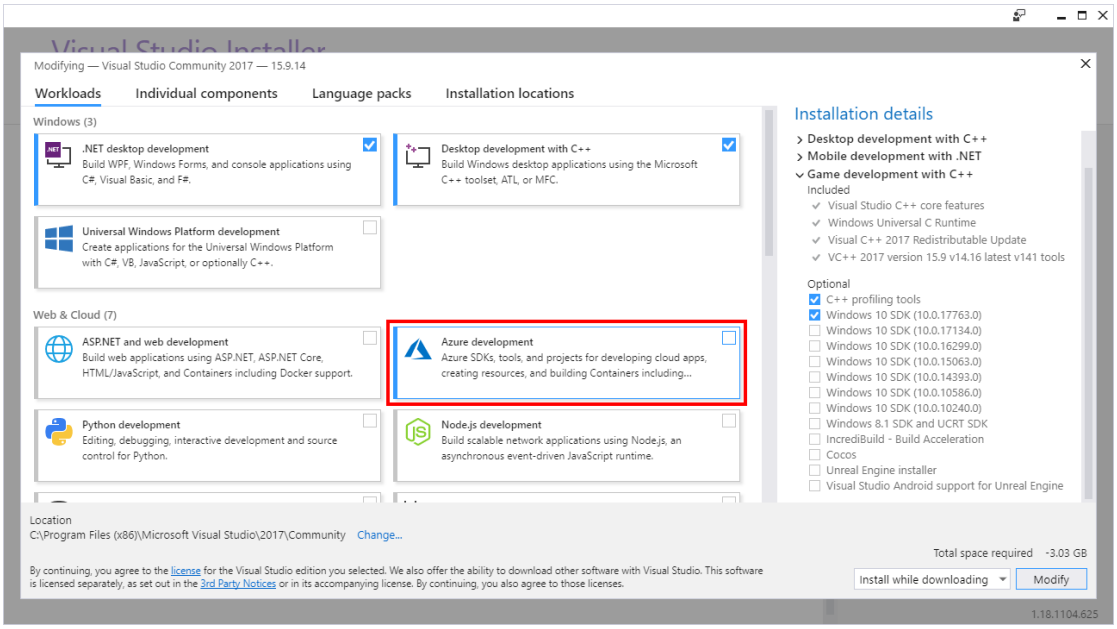
Azure Functions -rajapinnan luomista varten tarvittiin Microsoft tili sekä Visual Studio -ohjelmisto. Azure-portaaliin piti rekisteröityä Microsoft-tilillä. Portaalissa uuden funktio-sovelluksen voi luoda painamalla ensin Create a resource ja sen jälkeen valitsemalla listalta Function App. Kuvassa 11 on sovelluksen konfigurointi-ikkuna, missä sovellukselle annettiin nimi, resurssiryhmä, maksusuunnitelma, sijainti, runtime tyyppi ja tallennustili.



Kuva 11. Azure-resurssin luonti

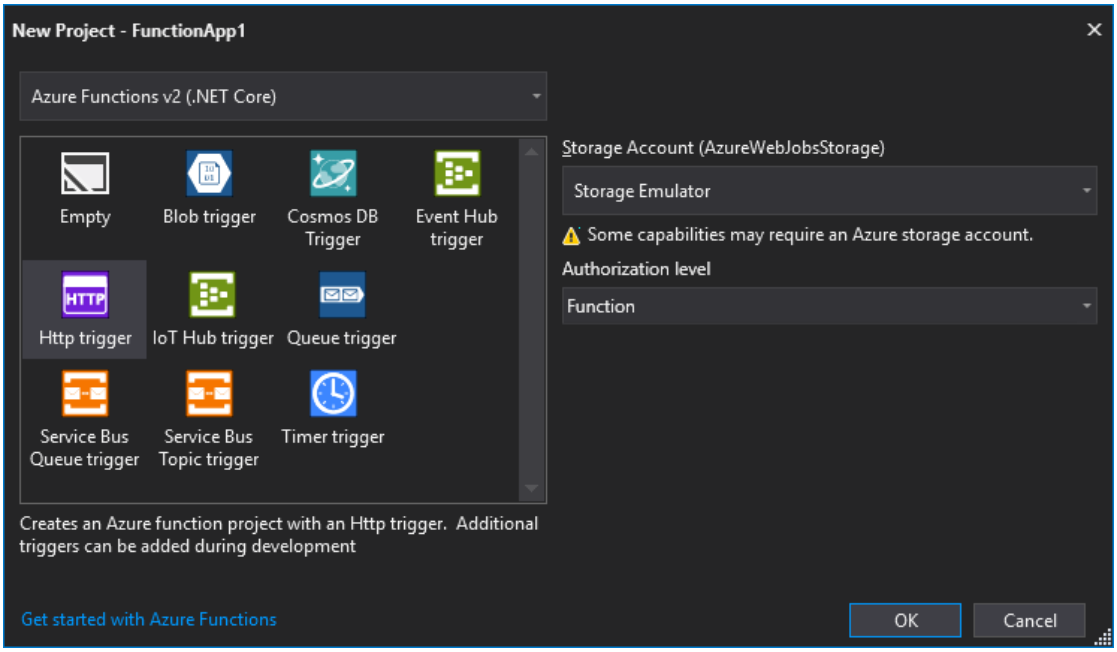
Sovelluksen nimi tulee olla uniikki, koska se toimii myös sovelluksen URL-osoitteena. Resurssiryhmä auttaa järjestämään kaikki tilin resurssit, jotta niitä on helpompi löytää portaalissa. Sovellukselle voi tehdä uuden ryhmän tai sen voi lisätä jo olemassa olevaan ryhmään. Sovellukseen voi valita yhden kahdesta maksusuunnitelmasta. Consumption-suunnitelmassa sovelluksen käytöstä maksetaan käytön mukaan. App Service -suunnitelmassa sovelluksesta maksetaan kuukausittain kiinteä määrä käytöstä riippumatta. Sovelluksen sijainnin voi valita eri palvelinsaleista ympäri maailmaa. Sovelluksen runtime vaihtoehdot ovat .NET Core (C#), Node.js (Javascript), Java ja PowerShell Core. Sovellukselle voi liittää tallennustiliin tai sille voi tehdä uuden tallennustilin. Tallennustiliin voi tehdä tietokantoja, tiedostoja ja jonoja.

Visual Studio -ohjelmistoon asennettiin Azure-kehitystyökalut valitsemalla asennusvaiheessa ominaisuuslistalta Azure development -vaihtoehdon. Työkalut voi myös lisätä vanhaan Visual Studio -asennukseen valitsemalla Visual Studio Installer -ohjelmassa asennetun Visual Studio -version kohdalta More ja sen jälkeen Modify. Kuvassa 12 näkyvästä valikosta valittiin Azure development ja sen jälkeen Modify, mikä aloittaa työkalujen asennuksen.



Kuva 12. Azure-työkalujen asennus

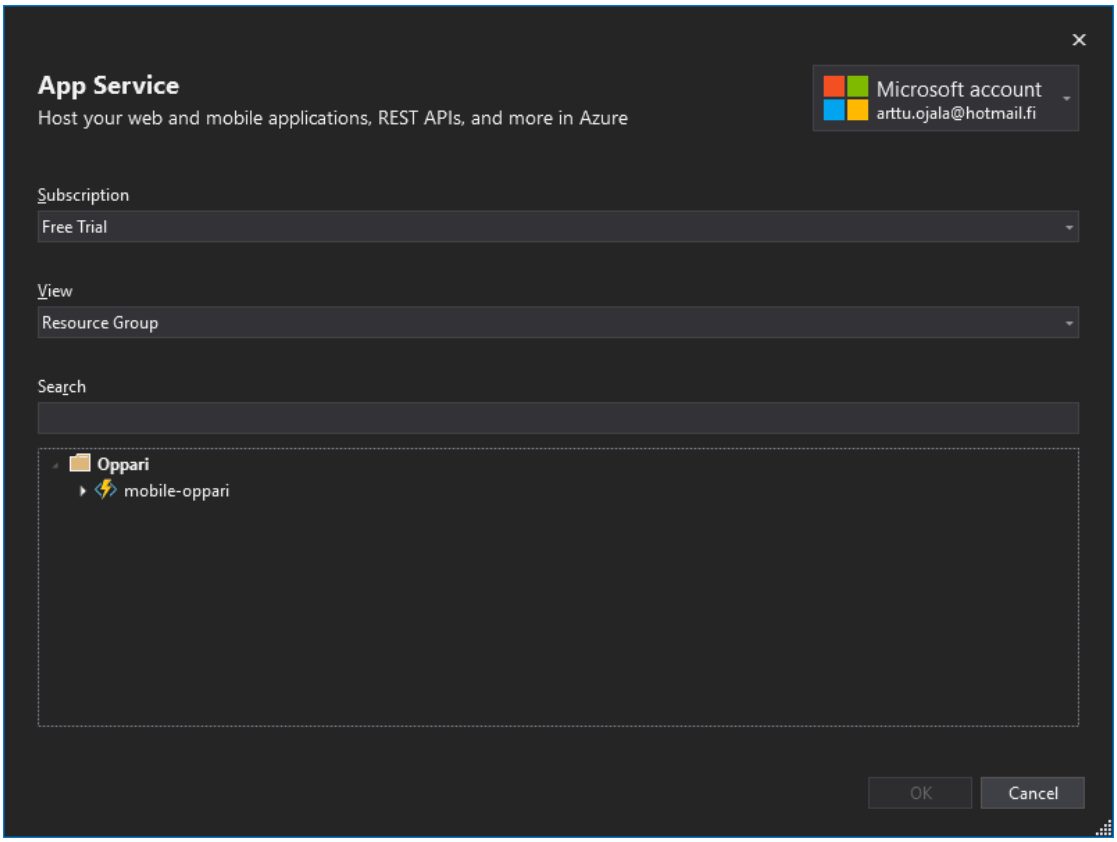
Funktio-sovelluksen luominen tapahtui Visual Studio -ohjelmassa valitsemalla New Project ja sen jälkeen Azure Functions. Kuvassa 13 olevasta ikkunasta projektiin valittiin suoritustyppi, lokaalisti käytettävä tallennustili sekä auktorisointitaso. Auktorisointitasolle on kolme vaihtoehtoa: funktio, anonyymi ja ylläpitäjä. Anonyymiä funktiota voi kutsua kuka tahansa, funktio auktorisointitaso vaatii funktioavaimen ja ylläpitotaso vaatii master-avaimen, jotta funktiota voi käyttää.



Kuva 13. Visual Studio -projektin luonti

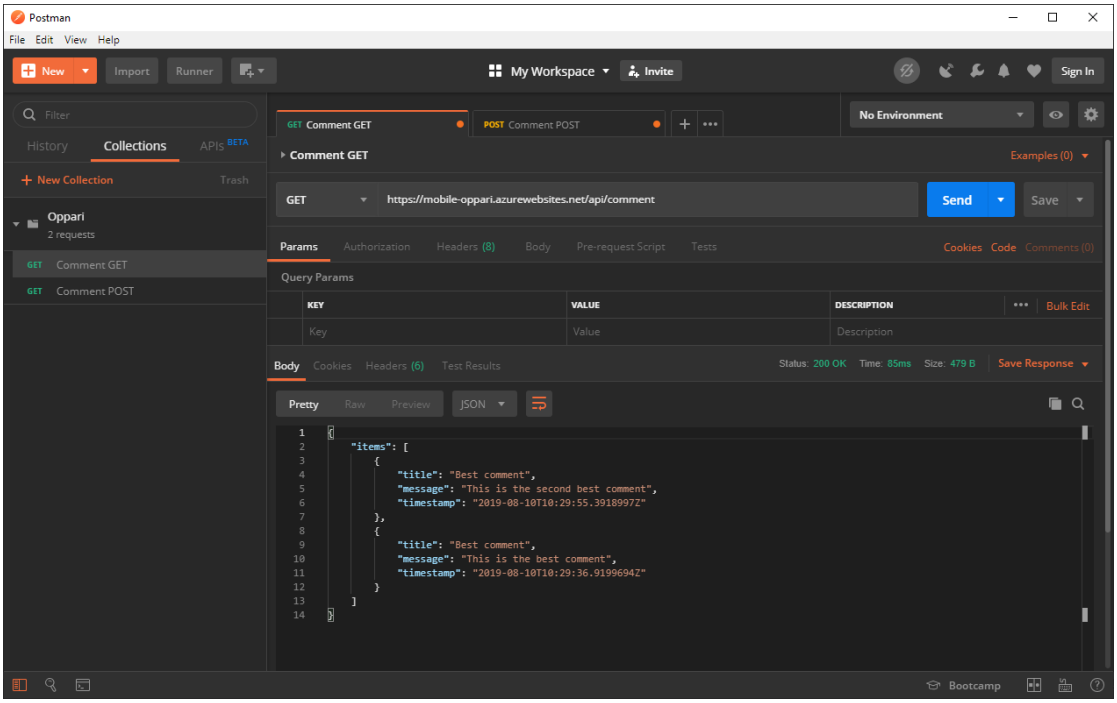
Luotuun funktioon valittiin HttpTrigger, mikä tarkoittaa sitä, että funktio suoritetaan joka kerta, kun sen osoitteeseen tulee http-pyyntö. Funktiota voi kutsua kahdella eri http-metodilla: GET ja POST. GET-metodilla lähetetty pyyntö hakee tietokannasta tallennetut kommentit ja palauttaa ne listana JSON-muodossa vastauksessa.

POST-metodia käytetään uusien kommenttien tallentamiseen tietokantaan. Kutsun tulee sisältää JSON-muodossa oleva kommentti, mikä sisältää otsikon ja viestin. Jos kutsu sisältää oikeamuotoisen kommentin, kommentti tallennetaan tietokantaan ja funktiosta palautetaan viesti onnistumisesta.



Kuva 14. Publish profiilin luonti

Valmis rajapinta lähetettiin Azure-palveluun valitsemalla Build-valikosta Publish. Projektille luotiin ensin Publish profile, mikä määrittää mihin sovellukseen projekti lähetetään. Visual Studio -ohjelmaan piti kirjautua samalla Microsoft-tilillä mitä portaalissa käytettiin resurssin luomiseen. Publish-valikosta valittiin New Profile ja seuraavaksi Select Existing. Kuvassa 14 on valikko, missä näkyy kaikki resurssit mitä tilillä on olemassa. Listalta valittiin haluttu resurssi ja sen jälkeen valittiin Publish.

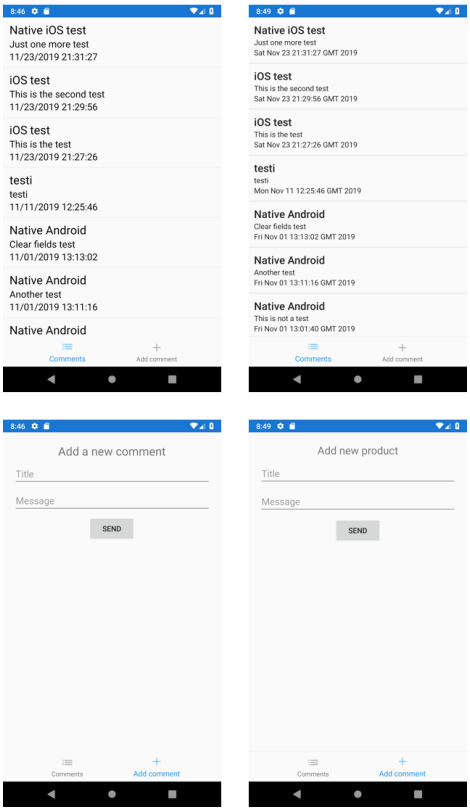


Kuva 15. Rajapinnan testaus Postman-ohjelmalla

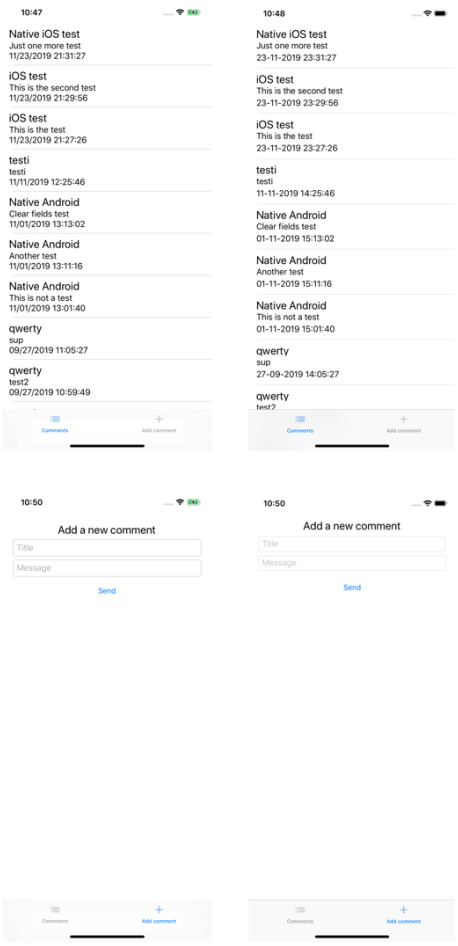
Julkaisun jälkeen rajapinta oli valmis käytettäväksi. Kuvassa 15 on kuvankaappaus Postman-ohjelmasta, mitä käytettiin rajapinnan toiminnan testaamiseen. Ohjelmalla lähetettiin http-pyyntöjä rajapintaan, ja tarkistettiin, että rajapinta palautti oletetun vastauksen.

5 TULOKSET JA JOHTOPÄÄTELMÄT

Opinnäytetyön tuloksena syntyi neljä lähes identtistä mobiilisovellusta. Toteutetusta ohjelmakoodista ei ole toimeksiantajalle suoraa hyötyä, ellei toimeksiantaja päätä tehdä natiivisovelluksia tulevaisuudessa. Kuvista 16 ja 17 voidaan päätellä, että Xamarin- ja natiivityökaluilla voidaan toteuttaa visuaalisesti yhdenvertaisia sovelluksia, joten natiivisovellusten ulkonäkö ei ole peruste toimeksiantajalle natiivisovelluksiin siirtymiseen.



Kuva 16. Valmis Android-sovellus. Vasemmalla Xamarin.Android ja oikealla natiivi Android



Kuva 17. Valmis iOS-sovellus. Vasemmalla Xamarin.iOS ja oikealla natiivi iOS

Toimeksiantajan Azure-rajapinnat on toteutettu C#-kielellä, joten datamalleja voi joissain tapauksissa kopioida suoraan rajapinnasta Xamarin-mobiilisovelluksiin, tai toisinpäin. Natiivisovellusten kanssa tätä hyötyä ei ole, koska ne käyttävät eri ohjelmointikieliä. Yhtenäisen ohjelmointikielen hyötynä on myös se, että kehittäjien ei tarvitse opetella useita eri ohjelmointikieliä, jotta he voivat toteuttaa mobiilisovelluksia ja rajapintoja.

Natiivityökalut sisältävät visuaalisia käyttöliittymätyökaluja, mitä Xamarin ei tarjoa. Natiivi iOS sisältää mielestäni parhaat visuaaliset työkalut, koska Android-sovellusta toteuttaessa käyttöliittymää piti välillä muokata myös suoraan xml-tiedostosta visuaalisen editorin puutteiden takia.

Xamarin Xaml Hot Reload on uusi ominaisuus, jolla Xaml-tiedostojen muutokset päivittyvät sovellukseen, ilman että sovellusta tarvitsee käynnistää uudelleen (Microsoft 2019f). Ominaisuus on opinnäytetyön kirjoitushetkellä vasta kehitysvaiheessa, ja se aiheuttaa usein sovellusten kaatumisia vähänkään monimutkaisemmissa sovelluksissa. Ominaisuus nopeuttaisi käyttöliittymien toteuttamista Xamarin-sovelluksissa huomattavasti.

Xamarin-kehityksessä tulee usein vastaan bugeja, joita ei natiivityökaluilla ole. Xamarin on avoimen lähdekoodin projekti, ja sen GitHub-sivulla on kirjotushetkellä raportoitu yli 1 800 avointa ongelmaa. Yleiset ongelmat, mitkä tulevat vastaan usein, korjataan yleensä nopeasti, mutta harvinaisten ongelmien korjauksissa saattaa kestää jopa vuosia.

Toimeksiantaja ei päätynyt vaihtamaan mobiilisovellusten toteutustapaa Xamarin-työkaluista natiiveihin. Jaettu ohjelmakoodi eri alustojen ja rajapinnan kesken todettiin tarpeellisemmaksi kuin natiivisovellusten tarjoamat hyödyt.

LÄHTEET

Apple. s.a. Xcode Overview. WWW-dokumentti. Saatavissa: https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/index.html [viitattu 27.11.2019].

Ducrohet, X., Norbye, T., Chou K. 2013. Android Studio: An IDE built for Android. Blogi. Päivitetty 15.5.2013. Saatavissa: https://android-developers.googleblog.com/2013/05/android-studio-ide-built-for-android.html [viitattu 29.9.2019].

Haase, C. 2019. Google I/O 2019: Empowering developers to build the best experiences on Android + Play. Blogi. Päivitetty 7.5.2019. Saatavissa: https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html [viitattu 29.9.2019].

Microsoft. 2016. Asynchronous programming. WWW-dokumentti. Päivitetty 20.6.2016. Saatavissa: https://docs.microsoft.com/en-us/dotnet/csharp/async [viitattu 27.11.2019].

Microsoft. 2017a. The Model-View-ViewModel Pattern. WWW-dokumentti. Päivitetty 7.8.2017. Saatavissa: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm [viitattu 7.12.2019].

Microsoft. 2017b. Xamarin.Forms XAML Basics. WWW-dokumentti. Päivitetty 25.10.2017. Saatavissa: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/xaml/xaml-basics/ [viitattu 27.11.2019].

Microsoft. 2018a. Supported languages in Azure Functions. WWW-dokumentti. Päivitetty 2.8.2018. Saatavissa: https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages [viitattu 25.8.2019].

Microsoft. 2018b. XAML Compilation in Xamarin.Forms. WWW-dokumentti. Päivitetty 22.8.2018. Saatavissa: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/xaml/xamlc [viitattu 27.11.2019].

Microsoft. 2019a. Azure Functions triggers and bindings concepts. WWW-dokumentti. Päivitetty 18.2.2019. Saatavissa: https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings [viitattu 25.8.2019].

Microsoft. 2019b. Simple Animations in Xamarin.Forms. WWW-dokumentti. Päivitetty 24.10.2019. Saatavissa: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/animation/simple [viitattu 6.12.2019].

Microsoft. 2019c. Task-based asynchronous pattern (TAP). WWW-dokumentti. Päivitetty 26.2.2019. Saatavissa: https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap [viitattu 27.11.2019].

Microsoft. 2019d. What is Xamarin? WWW-dokumentti. Päivitetty 16.9.2019. Saatavissa: https://docs.microsoft.com/fi-fi/xamarin/get-started/what-is-xamarin [viitattu 27.11.2019].

Microsoft. 2019e. Xamarin.Forms Requirements. WWW-dokumentti. Päivitetty 16.10.2019. Saatavissa: https://docs.microsoft.com/en-us/xamarin/get-started/requirements [viitattu 6.12.2019].

Microsoft. 2019f. XAML Hot Reload for Xamarin.Forms (Preview). WWW-dokumentti. Päivitetty 13.8.2019. Saatavissa: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/xaml/hot-reload [viitattu 7.12.2019].

Microsoft. s.a. Azure Functions Pricing. WWW-dokumentti. Saatavissa: https://azure.microsoft.com/en-us/pricing/details/functions/ [viitattu 25.8.2019].

KUVALUETTELO