Toni Hannula

UNITY MOBILE APPLICATION WITH A SERVERLESS FIREBASE BACKEND

Bachelor's Thesis

Information Technology / Game Programming

2021

Kaakkois-Suomen
ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Insinööri (AMK)
Tekijä/Tekijät	Toni Hannula
Työn nimi	Unity mobiiliapplikaatio palvelittomalla firebase backendillä
Toimeksiantaja	Kukouri Mobile Entertainment Ltd.
Vuosi	toukokuu 2021
Sivut	42 sivua
Työn ohjaaja(t)	Marko Oras

TIIVISTELMÄ

Opinnäytetyön ensisijaiset tavoitteet olivat demonstroida Unity-pelimoottorin Android-kehityksen käyttöönotto ja palvelittoman backendin kehitys kyseiseen applikaatioon Firebase-palveluita hyödyntäen. Opinnäytetyö myös havainnollistaa, mitä Firebase ja sen palvelut ovat, mitä ne tekevät ja miten niitä käytetään.

Tämä saavutetaan yhdistämällä teoriaa ja dokumentaatiota helposti lähestyttävään kokonaisuuteen sekä analysoimalla käytännön projektia, jossa käytettiin samoja palveluita ja työkaluja. Opinnäytetyö selittää, miten otetaan käyttöön Android-projekti Unity-pelimoottorissa ja Firebase-projekti Firebase-konsolissa, sekä miten käyttää Firebase SDK-toimintoja Unity-pelimoottorissa.

Projekti, jolla tämä demonstroidaan, on lo-fi-teemainen Android-mobiiliapplikaation prototyyppi. Applikaation tarkoitus on tarjota käyttäjille sosiaalinen alusta, jossa keskustella, kuunnella suoratoistettua musiikkia, pelata yksinkertaisia pelejä ja nauttia animoiduista taustakuvista.

Tutkimuskysymyksiin vastattiin ja prototyyppiapplikaation päätavoitteet saavutettiin. Ydinominaisuudet ja lähes kaikki muutkin pyydetyt ominaisuudet toteutettiin onnistuneesti muutamaa useista syistä tekemättä jätettyä ominaisuutta lukuun ottamatta.

Hyödyntämällä palvelitonta backendiä, kuten Firebasea, voidaan edesauttaa ketä tahansa perinteisen palvelimen ominaisuuksia kaipaavaa henkilöä tai tiimiä. Tällä voidaan välttää lähes kaikki ylläpitotarpeet ja huomattavasti yksinkertaistaa käyttöönottoa. Kaikki backendin käyttöönottoa Unity-pelimoottorin kanssa harkitsevat tahot voivat löytää tästä opinnäytetyöstä merkittävän hyödyn, erityisesti jos kohdealustana on Android.

Asiasanat: Firebase, Unity, Android, palvelimet, mobiilisovellukset

South-Eastern Finland University of Applied Sciences

Degree	Bachelor of Engineering
Author (authors)	Toni Hannula
Thesis title	Unity mobile application with a serverless firebase backend
Commissioned by	Kukouri Mobile Entertainment Ltd.
Time	May 2021
Pages	42 pages
Supervisor	Marko Oras

ABSTRACT

The primary objectives of this thesis were to demonstrate how to set up the Unity game engine for Android application development, and how to build a serverless backend for said application using Firebase services. The thesis also explored the usage and operation of Firebase and some of its services.

The thesis provides a conceptual basis for the study of Firebase and its services with an analysis of Firebase documentation and a practical project using Firebase services and tools. This study also explains how to set up a Unity project for Android and a Firebase project in the firebase console, and how to then use the Firebase SDKs in the Unity application.

The result of this thesis study was a lo-fi themed Android mobile application prototype. The purpose of this app is to provide a social environment where users can chat, listen to streamed lo-fi music, play simple games and enjoy some animated backgrounds. All core features of the functional prototype application and most of the other requested features, save a few that were discarded for various reasons, were successfully completed.

Utilizing a serverless backend service, such as Firebase can be an enormous help for anyone needing the functionality provided by a traditional backend server without the need for maintenance and with a massively simplified setup. This thesis can be utilized by those looking to set up such a backend for Unity, especially for the Android platform.

Keywords: Firebase, Unity, Android, servers, mobile applications

CONTENTS

TERMS AND ABBREVIATIONS

C#: A programming language, also known as C-Sharp. Based on C. C# is the main language used with Unity to develop the client-side application for this project.

CLI: Command-line interface. A program interface with no GUI, that is entirely text-based and used by typing in commands.

Client-side/Front-end: Application or code that runs on the end-user's device. Client-side and front-end are often used interchangeably, though front-end is usually used to refer to a browser-based client-side web application.

Function (programming): A self-contained block of code to accomplish a specific task. Functions can be re-used and called inside other functions.

GUI/UI: (Graphical) User interface. The interface used to interact with an application.

HTTP: Hypertext Transfer Protocol is a protocol for transferring data between a client and a server.

JavaScript/JS: A popular interpreted programming language. The language used on the Firebase Cloud Functions platform to run server-side code.

(Open)JDK: Java Development Kit. By definition essentially the same as SDK, but specific to Java. OpenJDK is an open-source implementation of the standard Java Platform.

JSON: JavaScript Object Notation. A file format for storing data in a structured human-readable text format.

Lo-fi (music): "Lo-fi" is short for "Low Fidelity" and is technically defined as music with the inclusion of undesirable or unprofessional elements such as misplaced notes or interference. That is however a poor way to describe lo-fi, which is in reality not quite so easy to define. Lo-fi music can, but does not have to follow a recipe something along the lines of: light distortion, samples, jazzy or hip-hop-like calm beats, usually with no lyrics. (Shaffer 2020.)

MonoBehaviour (Unity): The class that provides the framework for attaching scripts to game objects and for the use of many core features of the Unity engine. All Unity scripts derive from this class by default. (Unity Technologies 2021e.)

(Android) NDK: Android Native Development Kit. Much like the Android SDK, but it is needed to write apps or parts of them in native code such as C or C++ or to use libraries that use them.

NodeJS: An asynchronous event-driven JavaScript runtime designed to build scalable network applications (OpenJS Foundation 2020).

NPM: Node Package Manager is a software registry and a way to manage packages of code.

Prefab (Unity): In Unity, prefabs are a way of storing a game object and all of its child objects, components and properties in an easily reusable asset (Unity Technologies 2021f).

Scene (Unity): In Unity, scenes are the main asset in which all or parts of the application are contained (Unity Technologies 2021h).

ScriptableObject (Unity): In Unity, a ScriptableObject is a data container. It doesn't store game objects or components like a prefab, nor does it need to be instantiated every time at runtime like one. Instead, an instance can be made into the asset folder, which can then be referenced at any point. (Unity Technologies 2021i.)

SDK: Software Development Kit. A collection of tools and frameworks for software development.

Server-side/Back-end: Application or code that runs on a server separate from the end-user's device.

TypeScript/TS: A programming language that builds on JavaScript. TS helps avoid some errors by enforcing cleaner code and perhaps most notably, adds typing to the otherwise untyped JavaScript language. The server-side code for this project is written in TS, before being used to generate JavaScript by the Firebase CLI, which is then uploaded to the Cloud Functions service.

Unity: A development platform, mainly used for developing games. It can, however be used to make many kinds of applications.

1 INTRODUCTION

1.1 Preface

The project this thesis was built on was born from an app idea by Kim Soares, the CEO of Kukouri Mobile Entertainment Ltd. There is a fairly popular concept of so-called “lo-fi” music which many people use to relax and as background entertainment while chatting with others online. Perhaps most commonly, this can be seen on popular YouTube lo-fi livestreams and the related chat.

The objective of this study was to build a prototype of an app that could capture that market. The purpose of the app was to serve as a platform where people could spend time chatting with others while simultaneously playing simple games all while listening to relaxing lo-fi music. The app as a whole was intended to be themed around the concept of lo-fi and the main view was to include a chat, an animated image chosen from a small library by the user, music and optionally a simple game, also chosen from small selection by the user. The aim was to split the chat into rooms, which the user can create and choose from. The aim was to have public rooms available for all and private rooms created by users. In order to facilitate a better chatting experience, the purpose was to create other supporting features, such as the ability to add users to a friend- or blocked-list and invite friends to private rooms. The music would be streamed from a large library of lo-fi music online using a service and plugin called *Valinta*.

The programming and technical implementation of the app was made by the author of this thesis, while the UI and visual design, as well as some of the visual assets, were created and implemented by another project team member. Leadership and guidance were provided by Kim Soares.

The client-side app was developed with Unity, and the backend was predominantly developed using Firebase services. This thesis is focused on the said development and primarily the communication between the client application and the back-end services, using Authentication, Firestore and Cloud Functions.

1.2 Kukouri Mobile Entertainment Ltd.

The app prototype was made for a Finnish game development company called Kukouri. They are based in Kotka and were founded in 2011. Kukouri has a small team of professional developers, and their games have amassed over 50 million downloads across all platforms. The project this thesis is based on was developed under the guidance of the CEO Kim Soares.

1.3 Purpose and objective

This design-based thesis presents and reports the practical development made on the lo-fi app, referencing to theoretical information and material gathered from various sources.

The main objective of the thesis is to introduce and explain the development of an Android mobile application using Firebase services and Unity. Although not all Firebase services and platforms are very thoroughly explored in this thesis, the content will also be advantageous to readers interested in implementing those services.

1.4 Research methods and questions

The primary research method was to first collect findings during the development phase and report these findings. In order to support this, API documentation and other official sources of information provided by the services and development platforms were used.

The essential research questions this thesis attempts to answer are:

- How to develop applications for Android using Unity?
- How to implement a serverless backend with Firebase?
- How to use and implement Firebase services with Unity?

2 FIREBASE SERVICES AND UNITY

Firebase started as a real-time chat service called “Envolve”. The administrating team then decided to separate the real-time architecture from the chat ser-

vice and start another company to focus on real-time architecture service instead. The new company was named “Firebase”. (Ha 2012.) Acquired by Google in 2014, Firebase is now a flagship service platform for mobile and web app development that offers many tools and services needed for a modern application (Lardinois 2014). Including many services, for example, a database, an authentication service, cloud storage and web hosting.

Only the Firebase services that are relevant to this thesis are introduced at length. For this study, the *Firestore* database service is used to store data, the *Authentication* service to process and store user identities and *Cloud Functions* to run server-side code on dynamically allocated server instances. The other major services Firebase provides include *Cloud Storage*, *Machine Learning* and *Hosting*. The main subject matter of this thesis is the communication between Firebase and Unity. The focus is on Android development, but most of the solutions used can also be adapted for other platforms with minimal changes.

2.1 Asynchronous programming in Unity

When code is synchronous, each line of code is executed in order, with only one operation running at a time. Each action is only executed after the previous one finishes. Conversely, asynchronous code can run in any order and in parallel. Asynchronous functions are commonly used in web programming. (MDN contributors 2021.) Figure 1 illustrates the difference.

```
print("1")
AsynchronousPrint("2")
print("3")

async function AsynchronousPrint(x) {
    await Wait3Seconds()
    print(x)
}
```

Figure 1. Asynchronous code sample

If it is assumed that the *print* method prints something on to the screen, the sequence of characters one would see on the screen would be “132”. This is because the methods are called synchronously, one after another. However,

the *AsynchronousPrint* method on line two is running asynchronously, and even though it is called before line three, the asynchronous method does not reach the inner print method before line three has already been executed.

In this study, most of the asynchronous code pertains to the communication between the application's front- and back-end operations or more precisely, between Unity and different Firebase services. That communication is not instant, which necessitates asynchronous functions.

When developing games with Unity, the code is traditionally synchronous. There are use-cases for asynchronous functions for many operations, but most of the time, developers can make do with synchronous code. As an example, even an object continuously moving in a circle is still considered synchronous code. The object's position is set repeatedly in such a way as to appear like smooth movement. This is achieved by running the same set of operations many times a second, but synchronously. Operations executed this way are usually run for every frame. That is, for every image produced for the screen to display, some set of operations is executed.

2.1.1 Coroutines

Unity has a solution for running operations synchronously, or something like it, called *Coroutines*. Coroutines are single-threaded and, despite behaving much like it, are not actually parallel operations. They can be called and the operations in them can be executed similarly to normal functions, but the operations can be split across multiple frames. So, if a method calls a coroutine, the tasks in that coroutine are executed normally until an instruction to "yield" is reached. At that point, control flow is returned to the calling method. Then, after waiting the time allocated by the yield instruction, before the next frame is rendered, the coroutine continues from where it was stopped last time. This continues until the entire coroutine has finished. An example of this can be seen in Figure 2. (Unity Technologies 2021c).

```
0 references
IEnumerator ExampleCoroutine()
{
    DoThings("1"); //happens immediately after coroutine is started
    yield return null; //stops the coroutine execution with no wait instructions
    DoThings("2"); // coroutine execution continues from here the next frame
    yield return new WaitForSecondsRealtime(5); //stops coroutine with instructions to wait for 5 seconds
    DoThings("3"); // coroutine execution continues from here the next frame after the wait
}
```

Figure 2. Coroutine example

2.1.2 Async-await

With the release of Unity 2017, a new feature was introduced in Unity called “async-await”. Async-await and similar methods had already existed in other environments, but unlike in many of them, in Unity, the async operations had to be run on the main thread to have access to the Unity API. Unity does this automatically, and mostly developers need not worry about it. Despite this possible limitation, async-await offers many advantages over coroutines. For example, unlike coroutines, async-await functions can return values, which helps avoid bulky and complicated blocks of code and allows instead the splitting of these blocks into many smaller functions. It is also much easier to process exceptions compared to coroutines which do not allow yielding inside a try-catch block or stack tracing more than one level down. (Vermeulen 2017.)

There are, nevertheless, pit-falls, such as the fact that unlike coroutines, async operations are not tied to the lifetime of the MonoBehaviour, which means that if a game object is destroyed or disabled, the async operation keeps on going. This is both useful and a nuisance. Methods that do not need anything from the MonoBehaviour to finish can run to their logical end even if the object it is tied to is destroyed. In many cases, this can be helpful. Sometimes, for example if a task needs anything from the MonoBehaviour, this is undesired. In these instances, the task must be manually cancelled, which can be a laborious process. Thus, both coroutines and async-await have their place in asynchronous processing. (Ahvenniemi 2019.)

2.1.3 UniTask

In order to optimize asynchronous operations even further, *UniTask*, an efficient allocation-free async-await implementation for Unity can be used. The reasons to use UniTask rather than the standard Unity async-await include:

zero allocation, ability to await Unity's built-in AsyncOperations and Coroutines, as well as async-based replacements for all Coroutine features such as *Delay* and *DelayFrame* (Kawai 2020.)

2.2 Realtime Database and Firestore

There are different types of databases, but in a general sense a database is where applications can store and organize data to be easily accessed again. Firebase offers two types of NoSQL database services: Firebase Realtime Database and Cloud Firestore. The Realtime Database is the older of these two services and the product for which Firebase was originally founded (Ha 2012). Firestore, on the other hand, is a newer database service released in October 2017. It aimed to address some of the concerns encountered with the original Realtime Database, such as scalability and processing of complex queries. (Kumar 2018, chapter 13: Flexible NoSQL and Cloud Firestore; Lardinois 2017.)

Despite the improvements Firestore provides compared to the Realtime Database, the latter still remains a viable and sometimes better alternative, which is why the Realtime Database is still actively supported and updated. For example, the Realtime Database stores data simply in one enormous JSON tree, whereas Firestore stores it as collections of documents. The documents are stored somewhat similarly to JSON, but the values retain a data type, such as a boolean, number, or string and can be accessed individually, without fetching all the subcollections under the selected document. The data model used by Firestore is more complex but also more flexible and scalable. The advantages Firestore has over the Realtime Database include more complex queries, automatic scaling across multiple data centres and a higher maximum concurrent connection limit. The Realtime Database can have slightly lower latency and, unlike Firestore, it has native online presence status tracking, so if a client connection drops, it can be automatically reflected on the database. (Kumar 2018, chapter 13: Flexible NoSQL and Cloud Firestore; Firebase Documentation 2021a.)

2.3 Authentication

Firebase Authentication is a back-end service that makes the development of authentication for nearly any application easier by providing an SDK that can be used to authenticate users with email, as well as many providers such as Google, Facebook and Apple ID. It also integrates easily with other Firebase services such as Firestore and Cloud Functions. Even unregistered users can acquire an anonymous authentication profile that can be used to store user data to which a password and email, or other provider data can later be added when the user registers for the service. (Kumar 2018, chapter 2: Safe and Sound – Firebase Authentication; Firebase Documentation 2020.)

Authentication is often one of the backbones of any application that stores any kind of user data, and in Firebase, the stored user data is commonly tied to the user ID provided by Authentication. The Realtime Firebase and Firestore both have access rules that can be written by the developers to their liking, usually to maintain security. The rules are usually tied to Authentication to limit a user's access for example to only their own data. (Firebase Documentation 2020.)

2.4 Cloud Functions

2.4.1 Introduction

Traditionally, to build an application such as the one this thesis examines, one would likely need a back-end server of some sort to keep operations more secure as well as to process functions that are not appropriate to be executed on the client application, such as database maintenance or any operations that need to be performed reliably and independent of the client app. These servers need to be set up properly, maintained and scaled according to traffic. All that takes time, effort and money.

With Cloud Functions, however, developers can write server-side code in contained functions that can then be deployed onto the service. This is sometimes called a serverless framework, because as far as the developer using the service is concerned, there are no servers to manage. The virtual servers on

which the code is run are entirely managed by the service, including automatic scaling.

2.4.2 How Cloud Functions works

Each function runs in isolation in their own separated environments. As the server load increases or decreases, additional server instances are loaded and unloaded to process traffic as needed. The functions can be directly called by a client application with an HTTPS request, executed reactively to a set trigger or scheduled to run at set intervals. This allows for an easy-to-set-up, zero-maintenance solution for running server-side code. (Kumar 2018, chapter 4: Genie in the Cloud – Firebase Cloud Functions; Firebase Documentation 2021b.)

2.4.3 Writing a cloud function

The functions are written using NodeJS with either JavaScript or TypeScript and deployed to the cloud with Firebase CLI. In order to write a function for the cloud, the following syntax is typically used: “exports.[NameOfYourFunction] = functions.[serviceOrLibrary].[eventType]”. The main body of the function is passed as a parameter to the last method. An example function using the syntax can be seen in Figure 3.

```
exports.ExampleFunction = functions.https.onCall((data, context) => {
  // function body
});
```

Figure 3. Cloud Functions, generic function syntax example

The *context* and *data* parameters contain the context of the https request and the data passed to it, respectively. Both, but especially the context, can be used to authorize the request. The context object contains the authentication information of the user that made the request, so the function can for example be written to only allow a user to edit their own data.

3 PROJECT SETUP

3.1 Setting up Unity for Android

When installing Unity, the Android Build Support module must be selected. This installs the Android SDK and NDK as well as OpenJDK, which all are used for Android development. After Unity and the modules are installed, Unity must be configured to build an application for the Android platform. In order to achieve this, a Unity project template must be chosen. All templates support the same features and server the purpose of skipping some unnecessary setup procedures regarding settings and accompanying packages (Unity Technologies 2021g).

This thesis study uses the newer *Mobile 2D* template which is currently not officially documented but provides some settings mobile-friendly values and includes packages such as *Mobile Notifications* and *Android Logcat* by default.

3.1.1 Build Settings

The first setting that must be defined in a newly created Unity project is the build platform. Obviously, to build for Android, one should choose the Android platform from the build settings. During development, it is best to select the *Development Build* option which enables the *Profiler*, a window that helps with optimization and *Autoconnect Profiler* which, as the name suggests, automatically connects said profiler to the build. These options help with debugging. (Unity Technologies 2021b.)

The used compression method affects file size and read speeds. The three different methods available are *Default*, *LZ4* and *LZ4HC*. The default compression method for Android builds is ZIP which provides a smaller file size than LZ4 but requires more time for decompression. LZ4 compression is faster, and decompression is completed in real time as the file is read, which means it might be faster or slower than ZIP depending on the disk read speeds of the device (Unity Technologies 2021j). The faster compression makes LZ4 a good choice for development builds, whereas for a release build,

LZ4HC is better. Its compression is slower but also more efficient. The aforementioned settings and their locations can be seen in Figure 4. (Unity Technologies 2021b.)

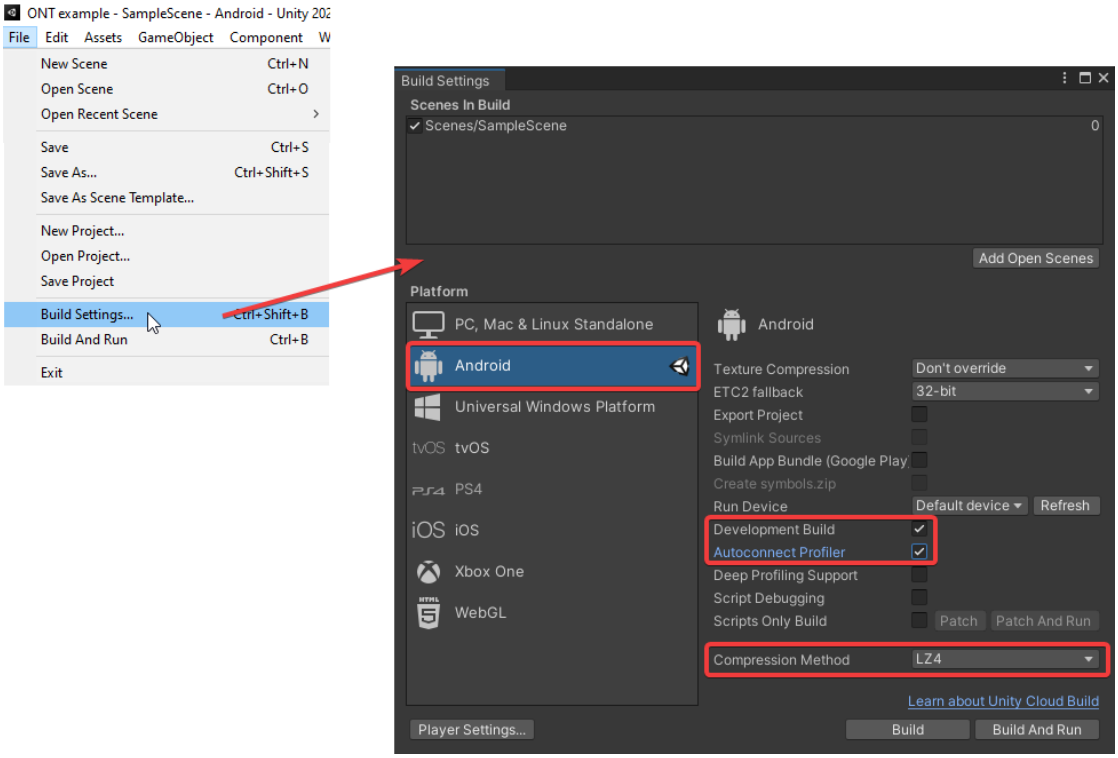

Figure 4. Unity Build Settings

3.1.2 Player Settings

The *Player Settings* section lets the developer choose various options for the final build. There are many options under Player Settings, but some of the more relevant ones will be explored here.

Under the subsection *Other Settings* and the *Identification* header, one should define the *package name*, the name used by the operating system and Google Play Store to identify the app. Version numbers are also defined here. The *minimum API level* to support should normally be the lowest possible version to maximize device compatibility. Some third-party libraries may require features that only a certain minimum version supports, in which case that API version should of course be chosen. In the case of this project, the API level is set at 19, or Android 4.4 “Kitkat”. (Unity Technologies 2021a.)

In the section *Configuration,* for the best compatibility with the Firebase SDK, the *API Compatibility Level* should be set to *.NET 4.x*. The settings and their locations can be seen in Figure 5. (Unity Technologies 2021a.)

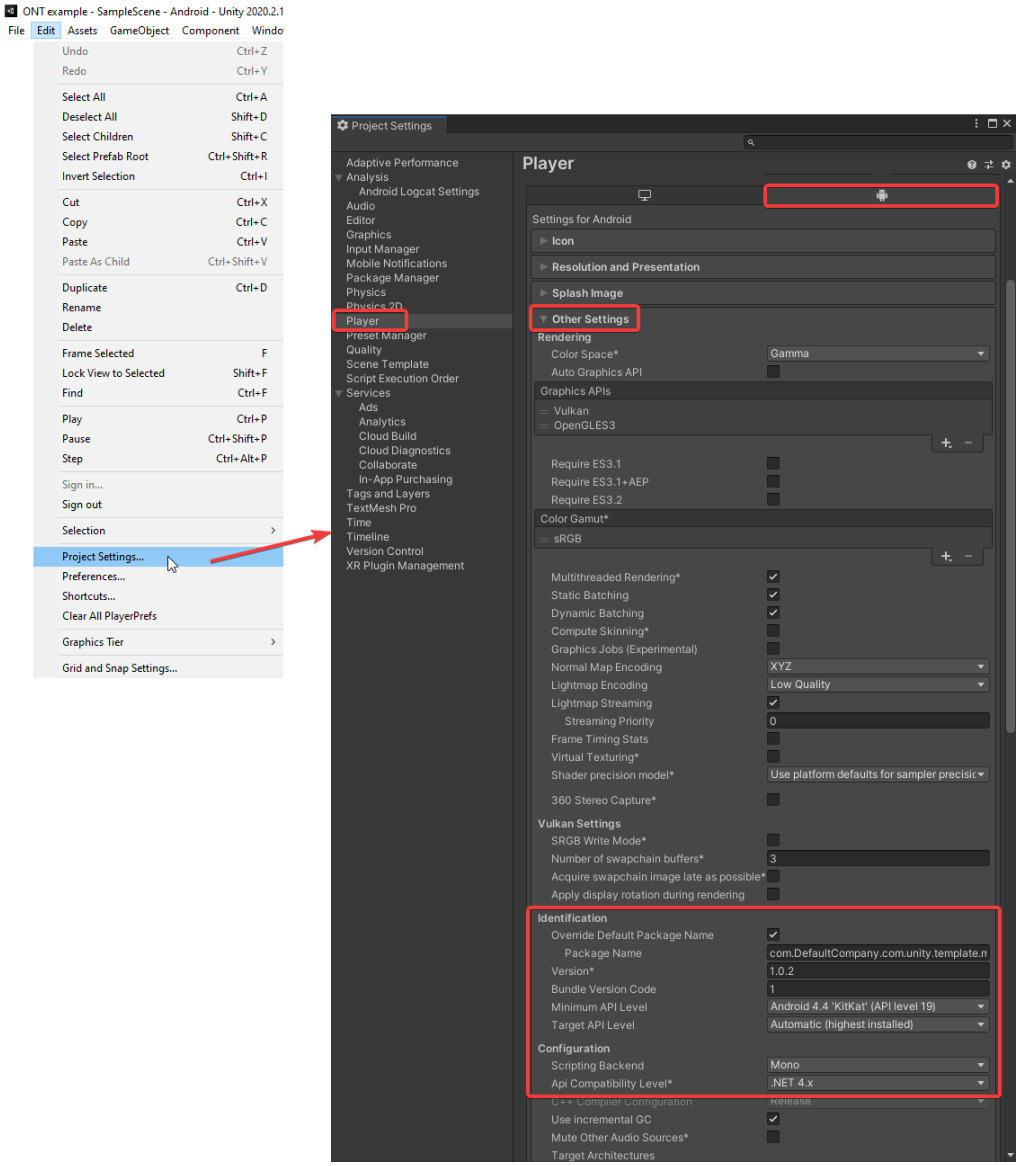

Figure 5. Player Settings, Other Settings, Identification and Configuration

Minification may be necessary if the class definitions used by the Android application build exceed the *Dalvik Executable*, or *dex-file* limit of 65,536 methods. Using minification automatically removes unused class references to make it easier to remain inside the class definition limit and shrinks and obfuscates code. Unity offers the option to use R8, a newer process for minimizing dex files, or the older Proguard method. Minification settings can be found under *Android Publishing Settings.* In this study, the use of Firebase SDKs increases the number of class definitions past the standard limit, necessitating minification. Due to compatibility concerns, the older Proguard method is

used. Minification settings and their locations can be seen in Figure 6. (Unity Technologies 2021a.)

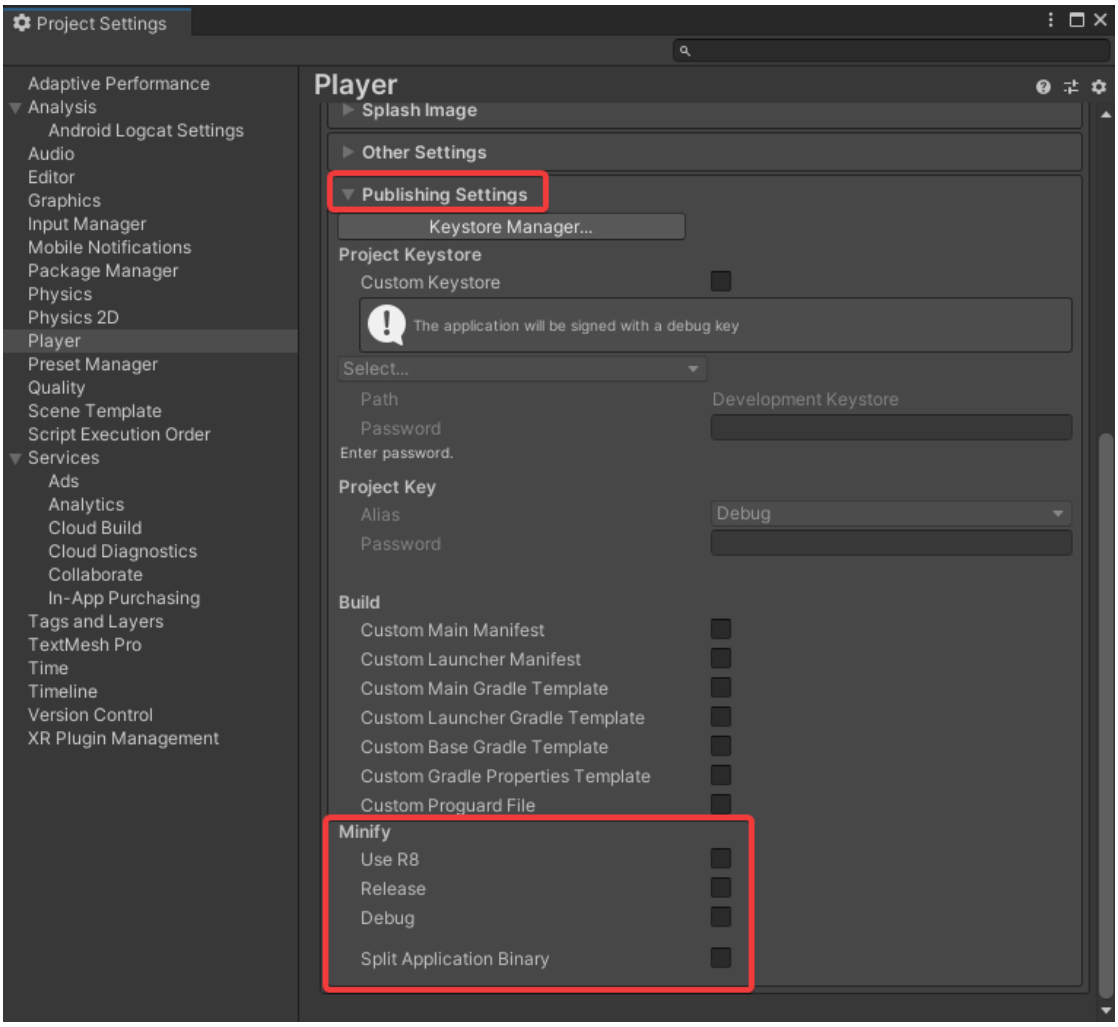

Figure 6. Player Settings, Publishing Settings, Minify

3.1.3 Gradle for Android

Gradle is a build automation tool used by Unity for all Android builds to automate processes, help prevent common errors, and reduce method references in DEX files. Unity can either build an application package or export a Gradle project that can then be built externally if necessary. For more advanced use-cases, a custom Gradle build template can be provided, but that will not be explored at length in this thesis as it is not relevant in most cases. (Unity Technologies 2021d.)

The DEX file can only hold a limited amount of method definitions. As mentioned in the previous chapter, minification can be used to bypass said limit.

However, if minification cannot be used for whatever reason, or if it is not sufficient, *multidex* is another way. Multidex can be enabled using a custom Gradle build template, which must be enabled from Unity settings. That being said, the use of multidex should be avoided unless absolutely necessary, as it can cause *Application Not Responsing* errors, as well as further problems on API levels preceding level 21, or Android 5.0. (Android Developers 2021.)

3.1.4 Testing on a real device

It is helpful to connect a device via USB to quickly test builds on a real Android phone. In order to do this, the device must have *USB debugging* enabled. USB debugging allows the Android device communicate with the development computer via USB using the *Android Debug Bridge* which is a tool incorporated in to the Android SDK. USB debugging can be enabled from the Android phone's developer settings. Prior to Android version 4.2, the developer options are available by default. In later versions, they are hidden and to gain access one must tap on the build number in the *About Phone* section of the settings seven times. (Android Studio User Guide 2021.)

When enabled and connected via USB to the development computer with the required SDK installed, the device should be available in the build settings under *Run Device* on Unity. With the device selected, choosing *Build And Run* from Unity will cause the app to be built and run directly on the connected device. With the *Development Build* option enabled, any logging from the running build will also appear in the console on Unity.

3.1.5 Device Simulator

It is not always worthwhile to build onto a real device for testing every little update. This is where Unity's *Device Simulator* package comes into play. At the time of writing this thesis, the package is still in a preview phase and it has not yet been thoroughly tested and is not officially recommended for production use. However, in the case of the Device Simulator, there is little risk of affecting the final version as it is entirely used in the editor and none of the features are used in the app itself. This provides a useful insight into the application's likely appearance and behaviour on a real device. In order to use preview packages, they must be enabled separately as shown in Figure 7.

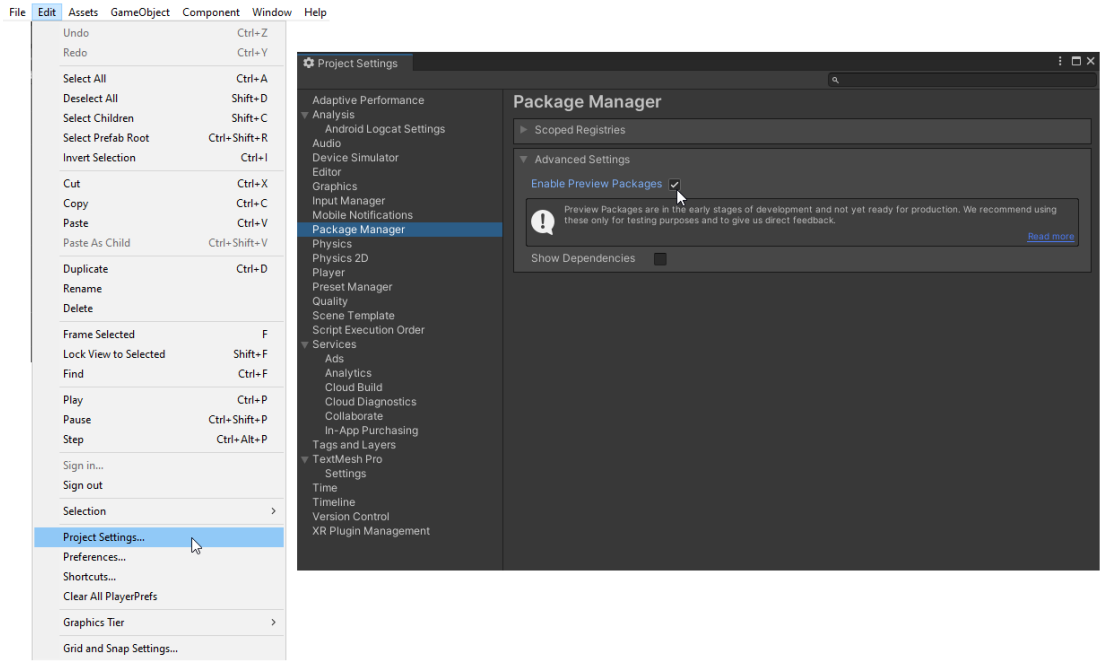

Figure 7. Enabling preview packages in Unity

Device Simulator, as the name suggests, attempts to simulate different devices in the Unity editor. It essentially renders the standard Unity *Game View* into a visualized representation of a device and simulates many internal system classes such as *Screen* and *SystemInfo*. In addition, it can simulate screen orientation changes and visually represent the safe area which is the area of the screen not disrupted by notches, cameras or rounded edges. It also allows developers to choose from a wide selection of devices to quickly see how the app would behave on different screen types. (Unity Technologies 2020.)

3.2 File and folder hierarchy

A proper file and folder hierarchy and structure is perhaps more important than it may sound. Depending on the scale of the project at hand, the number of files can grow extremely high. If project assets are not properly organized, the management of the project becomes more difficult. This may result in a situation where finding the right file will take more time, assigning references becomes a laborious process and mistakes such as accidentally recreating something that was already done by someone else become increasingly prevalent.

Due to the fact that Unity imports asset packages directly into the *Assets* folder, all assets made for this app are completely separated into a single location in the main Assets folder. The root folder of the structure containing said assets is "*_MyAssets*". In hindsight, the folder should, perhaps, have been renamed, but the name, after all, is not as important as the structure within. The underscore aims to ensure the folder is at the top of the hierarchy in alphabetical ordering. The inner structure is divided into types of assets and categories, and even further by page or other criteria. The folder structure can be seen in Figure 8.

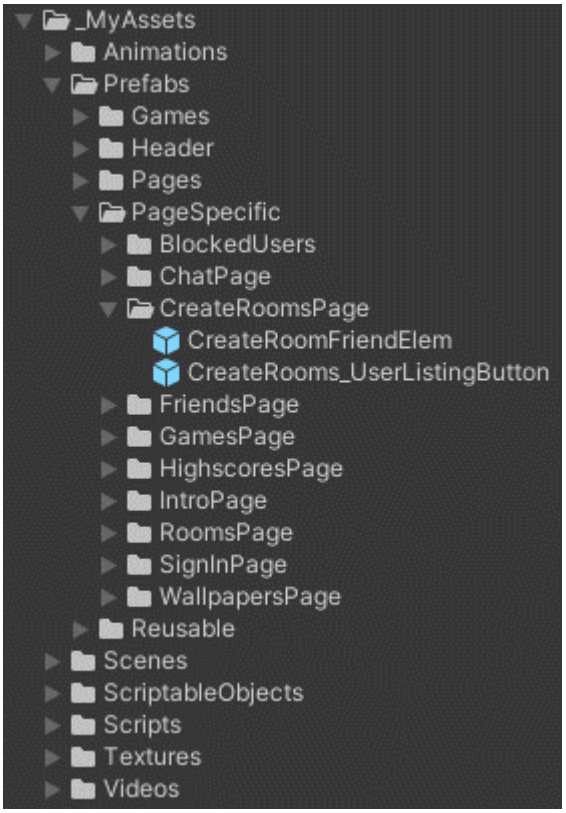

Figure 8. The file structure and hierarchy of the lo-fi project

3.3 Setting up a Firebase project for Unity

3.3.1 Basic setup

In order to use Firebase with Unity, one only needs a Google account. However, for the Cloud Functions service, a payment account needs to be created, even if the usage remains within the limits of the free tier. A Firebase project can be created on the Firebase console webpage by pressing *Add project* and following the instructions. Firebase makes the setup process fairly clear, allowing the user to follow on-screen instructions step by step.

In this process, after giving the project a name and enabling analytics if so desired, the project is created. An app should then be added to the project. By choosing Unity, the setup for an app of that type is started. The app needs to be registered as an Android app or iOS app, or both. Firebase asks for a nickname of the app and the *package name* or *bundle ID* which can be found in the *Player Settings* of the Unity project, under *Other Settings* titled as *Bundle Identifier*. Firebase allows then to download a json file which must be placed in the Assets folder of the Unity project. Next, the SDK package must be downloaded and selected SDKs imported into Unity. For best compatibility with the SDKs, the *API compatibility level* should be set to .NET 4.x in Unity's *Player Settings*.

3.3.2 Adding services and the Firebase CLI

After setting up the Firebase project, different Firebase services can be added with relative ease. For some of these services, such as Firestore or Authentication, the procedure is as simple as that of importing the associated SDK to Unity and enabling the service from the Firebase console webpage. For some others, however, the Firebase CLI is needed. It can be installed and used with NodeJS and NPM with the command "*npm install -g firebase-tools*". The Firebase CLI must be authenticated via a browser by running the command *firebase login* in the CLI. Then, to initialize a project in the chosen directory, the command *firebase init* is used. This starts an interactive setup in which the user can choose which features to set up and with which Firebase project to associate it.

3.3.3 Setting up Cloud Functions

After running the *firebase init* command with the firebase CLI and choosing Cloud Functions as one of the features, functions can be written in the *index.js* file inside the newly created project directory. The functions can then be sent to the cloud using the Firebase CLI with the command *firebase deploy*.

4 PROGRAM LOGIC STRUCTURE

4.1 General app structure

The entire app is built in a single scene with a header, main page container, banner ad container at the bottom and some other less relevant objects such as the main background and holders for handler scripts. This can be seen in Figure 9.

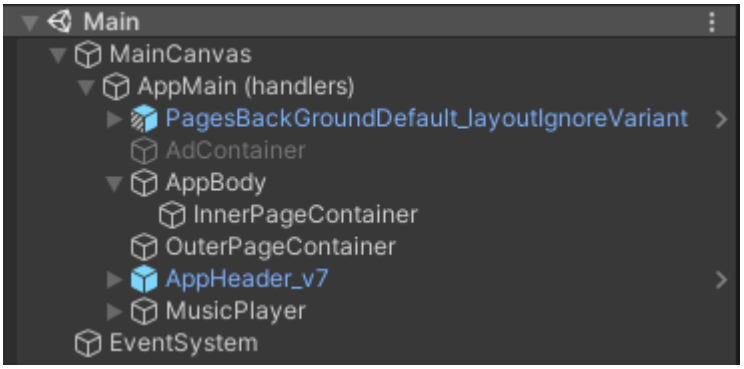

Figure 9. Main scene structure

All views of the app are separate prefabs that could be described as “pages”. These pages are loaded and unloaded into the main page-container in the scene. Each view has its own instance of a page *ScriptableObject*, which holds a reference to a prefab containing the page asset as well as some settings specific to that page. An example of a page *ScriptableObject* instance, a friend page in this case, can be seen in Figure 10.

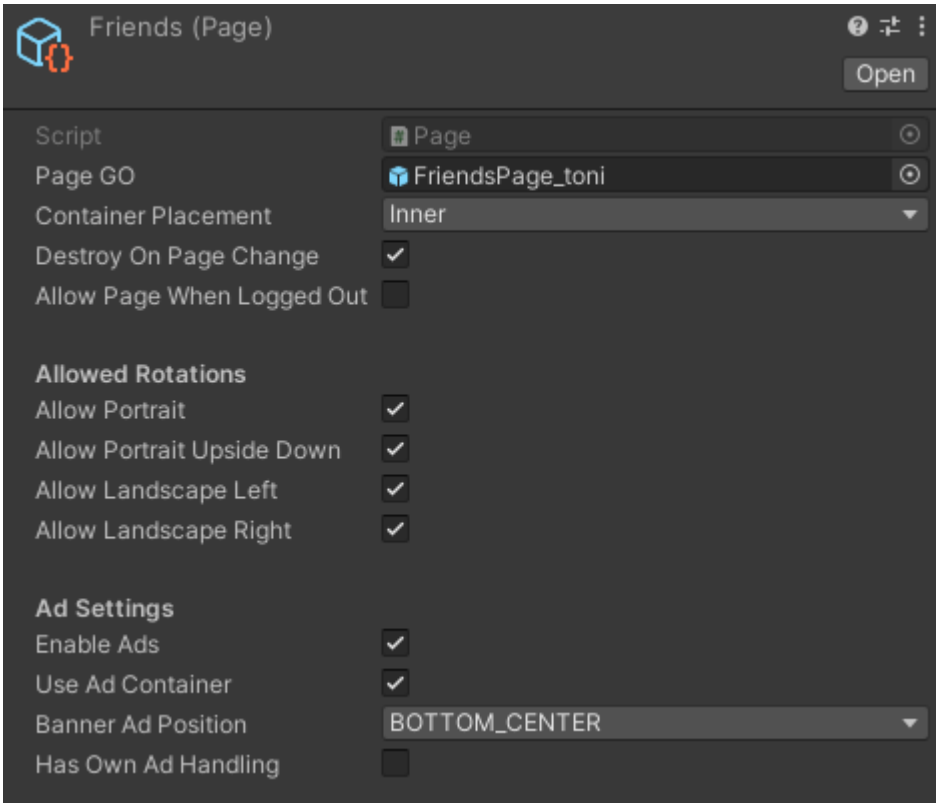

Figure 10. Friends page ScriptableObject instance example

4.2 Top level scripts

On the top level of the application's scene hierarchy, in other words outside the constantly changing page container section, there are scripts that process most of the overarching logic of the program during the entire life-time of the app. These are mostly singleton-like classes with static instance accessors. Singletons are a design pattern often used by Unity developers to make manager scripts where only one instance of that class ever needs to exist at a time. In this context, a static accessor to the instance of a singleton-derived class implies that any exposed methods of that class can be freely and easily accessed by outside scripts, making it a useful pattern for this use case.

Some of the scripts made this way include handlers for page changing logic, music streaming, Firebase service initialization, device orientation reactive logic, ad handling. The handler script which is used to change pages for instance, can be called from anywhere in the app to change the current page. The page changing script includes a great amount of conditional logic to allow or disallow page changes, as well as to enable and disable actions and operations depending on the page to be loaded.

4.3 Inner page logic

The inner logic of the separate pages that are loaded and unloaded is based on the communication between the aforementioned top level handler scripts, Firebase manager scripts and specific scripts inside each page.

In this study, the friend page is used as an example, and after this page is loaded, the main script inside the page searches the database by calling the Firestore manager script. The Firestore manager communicates with the Authentication manager script to retrieve data on the currently logged-in user. Using that data, it then retrieves the user's friend list, as well as any active friend requests from the database. The data returned by the database is parsed to a predefined format and passed along back to the script on the friend page to be displayed.

5 UNITY-FIREBASE IMPLEMENTATION

The three Firebase services the app created for this thesis study uses are Authentication, Firestore and Cloud Functions. The entire system is tied to authentication which is used for account management and to store and access some limited account data. The use of the app is technically almost fully restricted to registered users, but the unregistered users are allowed to create an anonymous account with one click, which they can later upgrade to a proper registered user account.

All other data except some settings and local data is stored in the Firestore database. Most of the functionality results from the communication between the app and Firestore.

For communication where more demanding or complex actions are required, or security is an issue, the Cloud Functions service is used as a middleman. It helps protect the database from malicious access and offload some of the data processing.

All of the firebase services have a manager script of their own, using the *ScriptableObject* system. When the app is started, a single handler script manages the initialization of each of the manager scripts.

5.1 Authentication manager

5.1.1 Initialization

As the app starts, the authentication system is initialized by calling the *Initialize* method of the authentication manager as can be seen in Figure 11. First, the method stores a reference to the default instance of the Authentication service, which is used to call most methods in the Authentication SDK. A method to process an authentication state change is then subscribed to the *StateChanged* event of the authentication SDK. Therefore, as a state change is detected, that method is used to process the new state. The initialization method then waits for the first state change which normally occurs immediately. At that point, the data of the currently signed in user should be available. However, since this state and associated data can come from the local cache, the user is reloaded immediately to ensure a data update. After this, the manager marks itself as initialized.

```
1 reference
public async UniTaskVoid Initialize()
{
    Debug.Log("Setting up Firebase Auth");
    auth = FirebaseAuth.DefaultInstance;
    auth.StateChanged += AuthStateChanged;
    await UniTask.WaitUntil(() => authChangeCalledOnce);
    await ReloadUserData();
    Debug.Log("Firebase Auth ready: " + auth);
    IsInitialized = true;
}
```

Figure 11. Authentication manager initialization method

5.1.2 Authentication state changes

Whenever a new authentication state is detected, at initialization or when the user logs in or out, the *AuthStateChanged* method (Figure 12), is called to process the new state. Depending on the new state, other methods are called to process a logout or login event. A separate event is also triggered at the end of this method to allow any other scripts listening to said event react to the new state if necessary. One such example would be the app's header which

enables and disables buttons depending on the new state, enabling, for example, the logout button if the user is signed in.

```
// Track state changes of the auth object.
2 references
private void AuthStateChanged(object sender, System.EventArgs eventArgs)
{
    Debug.Log("AuthStateChanged called");
    FirebaseAuth senderAuth = sender as Firebase.Auth.FirebaseAuth;

    //if sender auth is the originally initialized auth and the currentUser object...
    //...from the sent auth is not equal to the CurrentUser stored previously.
    //i.e. There was for sure an auth state change
    if (senderAuth == auth && senderAuth.CurrentUser != currentUser)
    {
        bool newStateIsSignedIn = senderAuth.CurrentUser != null;

        //if the new state isn't signed in and prev stored state isn't null, assume a logout event
        if (!newStateIsSignedIn && currentUser != null)
            HandleLogout();

        //update the stored CurrentUser
        currentUser = senderAuth.CurrentUser;

        if (newStateIsSignedIn)
            HandleLogin().Forget();
    }
    else
    {
        Debug.Log((senderAuth != auth ? "Sender Auth != Stored Auth. " : "") +
            (senderAuth.CurrentUser == currentUser ? "New state is same as stored state." : ""));
    }

    Debug.Log("Current state: " + (currentUser != null ? "Logged in" : "Logged out"));

    if(!authChangeCalledOnce)
        authChangeCalledOnce = true;

    if(OnAuthStateChange != null)
        OnAuthStateChange(currentUser != null);
}
```

Figure 12. Authentication state change handler method

The logout and login handler methods perform operations such as stopping or starting some listeners and in the case of a logout, resetting user related locally stored data and redirecting the users from the current page to the intro page from where they can sign in again.

5.1.3 Local temporary user data

The authentication manager stores user data locally, including for example a reference to the user object from the authentication SDK, as well as a friends list, blocked users list and user notifications.

Apart from the user object, the rest of the aforementioned data is kept up to date in real time using Firestore listeners. If, for instance, a new notification is added to the user's data in the database, the change is detected and the local list is updated immediately. This also allows the app to react to changes in the data by showing a visual notification to the user.

The local user data is employed in many ways by the app, commonly by verifying the current user's ID, as that is in many cases used to fetch relevant data from the backend. The friend and blocked user lists are, of course, used on the friends page, for instance, but if that was the only use case, it would not make sense to keep the list up to date in real time, since on the chat page, the new messages can be marked as friends or blocked as needed, without having to fetch the required data for each new message. Overall, this reduces the number of reads made to the database.

5.1.4 Other authentication manager methods

In addition to the aforementioned purposes, the authentication manager has other methods as well. Mainly these methods can be called from outside the manager to perform authentication related actions such as signing in, registering or sending a password reset. Some of these utilize built-in methods of the Authentication SDK and others call Cloud Functions methods.

The standard email login methods, for instance, must only use the *SignInWithEmailAndPasswordAsync* method of the SDK along with minimal error processing. On the other hand, there is another method used to upgrade an automatically created anonymous user to a proper registered user which is slightly more complex. It first calls a method written onto the Cloud Functions platform to run the upgrade process, and when this is succesfully completed, reauthenticates the user with the Authentication SDK.

5.2 Firestore manager

5.2.1 Initialization

Much like the authentication manager, the Firestore manager also has an initialization method that is called at the beginning of the app's lifetime immediately after initializing the authentication manager. The Firestore manager's initialization method is very simple and mostly only sets up the default instance reference to be used later.

5.2.2 Standard Firestore methods

The Firestore manager is used for all direct communication with the Firestore database. It is mainly used as a tool to consolidate all methods that deal with the database into one location that can be accessed and used by any script that needs it. The methods include, for example, *InsertNewGameScore*, *GetTopXScoresForGame*, and *GetRooms*.

Most of these methods work in a similar matter. As can be seen in Figure 13, some set of data is passed to the method via the parameters, be it some ID, or some new data to be inserted. That data is then constructed into a dictionary object that can be passed to various Firestore SDK methods to add, update, retrieve or delete data in the database. Exceptions are processed, and if relevant, the value is returned or parsed.

```
1 reference
public async UniTask<bool> BlockUser(string userId, string username)
{
    try {
        Dictionary<string, object> user = new Dictionary<string, object> {
            { "UserID", userId },
            { "Name", username }
        };
        await firestoreDB.Collection("UserData/" + AuthManSO.currentUser.UserId +
            "/BlockedUsers").Document(userId).SetAsync(user).AsUniTask();
        return true;
    }
    catch(Exception ex)
    {
        Debug.LogWarning("BlockUser ERROR: " + ex.Message + "\n" + ex.ToString());
        return false;
    }
}
```

Figure 13. Firestore manager's BlockUser method

5.2.3 Firestore listeners

The Firestore manager also deals with the listeners used by the authentication manager to retrieve up-to-date data from the database regarding the current user, such as user notifications. When the authentication manager detects a logged-in user, it calls Firestore manager's methods to start listeners.

All the listeners work function in a similar manner. For example, the friend listener is started by calling the method *StartListeningForFriends*. The method then performs some basic checks to see, for instance, if the user is signed in and if the database has been initialized. Next, the method makes sure that any other listeners of the same type, are stopped before starting a new one. Lastly, the method starts the new listener using the aptly named *Listen* method provided by the Firestore SDK and provides a callback method for the listener to call when changes are detected, which in this case is a method in the authentication manager that updates the friends list stored there. The *StartListeningForFriends* method can be seen in Figure 14 and the callback, which processes the changes returned by the listener, can be seen in Figure 15.

```
1 reference
public void StartListeningForFriends()
{
    if(!AuthManSO.IsLoggedIn)
    {
        Debug.LogError("StartListeningForFriends ERROR: Not logged in");
        return;
    }
    else if(firestoreDB == null)
    {
        Debug.LogError("StartListeningForFriends ERROR: no database");
        return;
    }

    StopFriendListener();
    ActiveFriendListener = firestoreDB.Collection("UserData/"+
        AuthManSO.currentUser.UserId+"/Friends").Listen(snapshot =>
    {
        //Debug.Log("Got friend snapshot");
        AuthManSO.HandleFriendsSnapshot(snapshot);
    });
}
```

Figure 14. Firestore manager's StartListeningForFriends method

```
1 reference
public void HandleFriendsSnapshot(QuerySnapshot snapshot)
{
    Debug.Log("Received a friends snapshot" + (snapshot.Metadata.IsFromCache ? " and is from cache" : ""));

    //We only need the changes. i.e. what's different from the local-client-state
    IEnumerable<DocumentChange> changes = snapshot.GetChanges();

    //Go through changes
    foreach (DocumentChange change in changes)
    {
        if (change.GetChangeType() == DocumentChange.Type.Added)
        {
            if(!FriendsLocal.ContainsKey(change.Document.Id))
                FriendsLocal.Add(change.Document.Id, change.Document.GetValue<string>("Name"));
        }
        else if (change.GetChangeType() == DocumentChange.Type.Modified)
        {
            if(FriendsLocal.ContainsKey(change.Document.Id))
                FriendsLocal[change.Document.Id] = change.Document.GetValue<string>("Name");
        }
        else if (change.GetChangeType() == DocumentChange.Type.Removed)
        {
            if(FriendsLocal.ContainsKey(change.Document.Id))
                FriendsLocal.Remove(change.Document.Id);
        }
    }

    if(OnFriendsLocalModified != null)
        OnFriendsLocalModified();
}
```

Figure 15. Authentication manager's HandleFriendsSnapshot method

5.3 Cloud Functions manager

The Cloud Functions manager is nearly identical to the Firestore manager in that it collects all associated methods into one place to be easily used by outside scripts. Its initialization method is similar as well, and is run immediately after the Firestore manager's initialization.

The Cloud Functions manager's methods behave much like the standard methods in the Firestore manager. They are passed parameters that are built into a dictionary object that is passed to the cloud functions. The result returned from the cloud function is then passed along to the caller of the manager's method. Apart from the initialization, the manager only uses the *GetHttpsCallable* method from the Cloud Functions SDK to select what function to use from the backend, and the *CallAsync* method to call the said function. No other methods of the Cloud Functions SDK are used. One of the manager's methods can be seen in Figure 16, namely the *CreateRoom* method which is used to create a new chat room.

```
1 reference
public UniTask<HttpsCallableResult> CreateRoom(string name, string desc, List<string> friendsToInvite)
{
    // Create the arguments to the callable function.
    Dictionary<string, object> data = new Dictionary<string, object> {
        { "Name", name },
        { "Desc", string.IsNullOrEmpty(desc) ? name : desc },
        { "AuthorName", AuthManSO.currentUser.DisplayName },
        { "Invites", friendsToInvite }
    };

    return functions.GetHttpsCallable("CreateRoom").CallAsync(data).AsUniTask();
}
```

Figure 16. Cloud Functions manager's CreateRoom method

The Cloud Functions manager is a very simple script. This is perhaps to be expected as its purpose is merely to call functions located in another place, namely, the cloud. Most of the related complexity is in the backend code.

5.4 Cloud Functions backend

The backend code is written in TypeScript in an entirely separate environment from the client app and Unity using NodeJS. The functions are then converted to JS and uploaded into the cloud when deployed using the Firebase CLI. All three types of triggers for functions are used: callable, event-based and

scheduled. Of these functions, in the majority of cases callable functions are used.

5.4.1 Scheduled functions

A scheduled function is a type function that is triggered by a predefined schedule. The only scheduled function used in the study is for clean-up purposes. The function is run once a day to remove old or unused data, such as old inactive anonymous accounts and messages. Due to the way Firestore indexes data, the presence of massive collections of documents does not actually slow operations down significantly. For performance, the size of the query result is what matters, not the total size of the document collection (Firebase Documentation 2021a). That being said, it is usually better to not let the database grow completely unhampered.

5.4.2 Event-based functions

An event-based function is a type of function that is triggered by an event chosen from a set of options by the developer. There are three different event-based functions used in this study. *CreateUserData* is triggered on user creation and as the name suggests, it creates some additional user data that the Authentication service does not save by default. That data is saved in the Firestore database using the user ID. *DeleteUserData* and *DeleteRoomData* are functions that trigger on user and chat room deletion, respectively. Both functions essentially automate the deletion of all relevant associated data.

5.4.3 Callable functions

A Callable function is a type of function that can be called by the SDK or by an HTTPS call. All of the cloud functions used in this study are callable by the client app directly, such as *RegisterUserWithEmail*, *CreateRoom*, *SendFriendRequest* and *GetRoomsForUser*. Most of these functions start with a set of authentication and data validation verifications. A set of operations is then executed, usually involving a manipulation of the database or user accounts.

5.4.4 Example function

Writing good backend code could be a thesis subject on its own, but one function is chosen here as an example so the basic methods and function structure can be explained. The selected function is *AcceptRoomInvite* and its purpose, as is quite evident from the name, is to accept an invitation to a chat room. This function can be seen in Figure 17.

```
exports.AcceptRoomInvite = functions.https.onCall((data, context) => {
  if (context.auth == null) {
    throw new functions.https.HttpsError(
      "unauthenticated",
      "Can't accept room invites if not logged in"
    );
  } else {
    if (!(data.NotifID as string)) {
      throw new functions.https.HttpsError(
        "invalid-argument",
        "NotifID missing"
      );
    }

    return db.runTransaction(async (t) => {
      const notifRef = db.collection("UserData/" + context.auth?.uid + "/Notifications").doc((data.NotifID as string));
      const doc = await t.get(notifRef);
      if (doc.get("Type") == "room") {
        console.log("before update");
        await t.update(db.collection("Rooms").doc(doc.get("RoomID")), {
          AllowedUsers: admin.firestore.FieldValue.arrayUnion(context.auth?.uid),
        });
        t.delete(notifRef);
      } else {
        throw new functions.https.HttpsError(
          "invalid-argument",
          "ID given was not for a room invite"
        );
      }
    });
  }
});
```

Figure 17. Cloud Functions, AcceptRoomInvite

The function starts by verifying that the request was sent by an authenticated user and that a notification ID was provided. The user that accepted the invite is added to the list of allowed users for the room in question, and the invitation is then deleted. These two operations are wrapped in a transaction to make the entire process atomic, that is, every operation made to the database succeeds or else they all fail. This is important as it would not be acceptable to have only one of the operations succeed. If the transaction or one of the verifications at the start fail, an exception is raised, and the error is returned for the client to process.

6 CONCLUSIONS

The purpose of this thesis was to describe the setup and implementation of an Android app in Unity, as well as its serverless backend using Firebase services. The setup section of the thesis depicts the required processes in relative detail, and the implementation section illustrates how the services were used in the lo-fi app project which this thesis is based on.

Despite a few problems and a couple of missing features, the lo-fi project managed to meet its objectives overall. It serves as a working prototype and demonstration of the features of a possible future version. The core features and most of the requested minor features are in working order.

The choice of Firebase as the provider for the backend services can be generally considered good decision. Being able to tie multiple different services and SDKs to each other seamlessly under the same umbrella of services helped streamline the development process enormously. Additionally, the Firebase Authentication service can be connected with all the other services for an easy setup and secure whole. The choice of preferring Firestore over the Realtime Database is somewhat more difficult to justify, but the advantages described in Chapter 2.2 make it the better choice in theory. The Cloud Functions service was a great way to avoid the set-up of any kind of server and, after the initial learning curve was overcome, proved to be an excellent service to use.

The implementation methods used in the study can be considered fairly effective. It makes sense to separate the backend methods into their own self-sufficient manager scripts to be easily usable by other outside scripts. In hindsight, the use of *ScriptableObjects* for said managers might have been unnecessary. Though the use of the *ScriptableObject* system for manager scripts is certainly a viable option, a simple singleton pattern, as was used by some other manager-type scripts in this study, would have sufficed for the Firebase manager scripts as well.

All the knowledge and experience gained in the process of developing the app and writing this thesis will be invaluable in the future, especially regarding any backend service implementation needs. This thesis should serve as a great

tool for anyone looking to build a backend for their app, especially if they happen to be developing with Unity or even more so if their target platform of choice is Android.

Making a backend for any app can be a massive undertaking that can in some cases require an entire team of developers. The utilization of a serverless backend service such as Firebase is an enormous help in making the entire process easier as well as saving both time and money. By using a conglomerate of many types of backend services such as Firebase, the additional benefit of having many different kinds of services and SDKs working in relative harmony together with little to no compatibility issues can be achieved.

REFERENCES

Ahvenniemi J. 2019. Best practices: Async vs. coroutines - Unite Copenhagen. Unity. Video. Published 27 September 2019. Available: https://www.youtube.com/watch?v=7eKi6NKri6 [Accessed: 14 April 2021]

Android Studio User Guide. 2021. Configure on-device developer options. WWW-document. Updated 24 February 2021. Available: https://developer.android.com/studio/debug/dev-options [Accessed: 29 March 2021]

Android Developers. 2021. Enable multidex for apps with over 64K methods. WWW-document. Updated 4 May 2021. Available: https://developer.android.com/studio/build/multidex [Accessed: 10 May 2021]

Firebase Documentation. 2020. Firebase Authentication. WWW-document. Updated 17 November 2020. Available: https://firebase.google.com/docs/auth [Accessed: 26 March 2021]

Firebase Documentation. 2021a. Choose a database: Cloud Firestore or Realtime Database. WWW-document. Updated 22 January 2021. Available: https://firebase.google.com/docs/firestore/rtdb-vs-firestore [Accessed: 26 March 2021]

Firebase Documentation. 2021b. Cloud Functions for Firebase. WWW-document. Updated 17 February 2021. Available: https://firebase.google.com/docs/functions [Accessed: 29 March 2021]

Ha, A. 2012. Firebase Raises $1.1M For Real-Time App Infrastructure. TechCrunch. WWW-document. Updated 22 May 2012. Available: https://techcrunch.com/2012/05/22/firebase-funding/ [Accessed: 26 March 2021]

Kawai, Y. 2020. UniTask v2 — Zero Allocation async/await for Unity, with Asynchronous LINQ. WWW-document. Updated: 11 June 2020. Available: https://neuecc.medium.com/unitask-v2-zero-allocation-async-await-for-unity-with-asynchronous-linq-1aa9c96aa7dd [Accessed: 24 March 2021]

Kumar A. 2018. Mastering Firebase for Android Development. E-book. Birmingham: Packt Publishing. Available: https://kaakkuri.finna.fi/Record/nelli29_mamk.4100000005116224 [Accessed: 16 April 2021]

Lardinois, F. 2014. Google Acquires Firebase To Help Developers Build Better Real-Time Apps. TechCrunch. WWW-document. Updated 21 October 2014. Available: https://techcrunch.com/2014/10/21/google-acquires-firebase-to-help-developers-build-better-realtime-apps/ [Accessed: 24 March 2021]

Lardinois, F. 2017. Google launches Cloud Firestore, a new document database for app developers. TechCrunch. WWW-document. Updated 3 October 2017. Available: https://techcrunch.com/2017/10/03/google-launches-cloud-firestore-a-new-document-database-for-app-developers/ [Accessed: 26 March 2021]

MDN contributors. 2021. JavaScript. WWW-document. Updated 21 February 2021. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript [Accessed: 24 March 2021]

OpenJS Foundation. 2020. About Node.js. WWW-document. Updated: 8 January 2020. Available: https://nodejs.org/en/about/ [Accessed: 6 May 2021]

Shaffer, N. 2020. The Origins of Lo-Fi. The Music Origins Project. WWW-document. Updated 5 April 2020. Available: https://www.musicorigins.org/the-origins-of-lo-fi/ [Accessed: 24 March 2021]

Unity Technologies. 2020. Unity – Manual: Device Simulator – Introduction. WWW-document. Updated 30 November 2020. Available: https://docs.unity3d.com/Packages/com.unity.device-simulator@2.2/manual/index.html [Accessed: 5 April 2021]

Unity Technologies. 2021a. Unity – Manual: Android Player settings. WWW-document. Updated 29 March 2021. Available: https://docs.unity3d.com/2021.1/Documentation/Manual/class-PlayerSettingsAndroid.html [Accessed: 5 April 2021]

Unity Technologies. 2021b. Unity – Manual: Build Settings. WWW-document. Updated 22 March 2021. Available: https://docs.unity3d.com/Manual/BuildSettings.html [Accessed: 29 March 2021]

Unity Technologies. 2021c. Unity – Manual: Coroutines. WWW-document. Updated 22 March 2021. Available: https://docs.unity3d.com/Manual/Coroutines.html [Accessed: 24 March 2021]

Unity Technologies. 2021d. Unity – Manual: Gradle for Android. WWW-document. Updated 10 May 2021. Available: https://docs.unity3d.com/Manual/android-gradle-overview.html [Accessed: 10 May 2021]

Unity Technologies. 2021e. Unity – Manual: Important Classes - MonoBehaviour. WWW-document. Updated 11 April 2021. Available: https://docs.unity3d.com/Manual/class-MonoBehaviour.html [Accessed: 15 April 2021]

Unity Technologies. 2021f. Unity – Manual: Prefabs. WWW-document. Updated 11 April 2021. Available: https://docs.unity3d.com/Manual/Prefabs.html [Accessed: 15 April 2021]

Unity Technologies. 2021g. Unity – Manual: Project Templates. WWW-document. Updated 22 March 2021. Available: https://docs.unity3d.com/Manual/ProjectTemplates.html [Accessed: 29 March 2021]

Unity Technologies. 2021h. Unity – Manual: Scenes. WWW-document. Updated 11 April 2021. Available: https://docs.unity3d.com/Manual/CreatingScenes.html [Accessed: 15 April 2021]

Unity Technologies. 2021i. Unity – Manual: ScriptableObject. WWW-document. Updated 11 April 2021. Available: https://docs.unity3d.com/Manual/class-ScriptableObject.html [Accessed: 15 April 2021]

Unity Technologies. 2021j. Unity – Scripting API: BuildOptions.CompressWithLz4. WWW-document. Updated 30 April 2021. Available:

https://docs.unity3d.com/ScriptReference/BuildOptions.Compress-WithLz4.html [Accessed: 10 May 2021]

Vermeulen S. 2017. Async-Await instead of coroutines in Unity 2017. WWW-document. Updated 23 September 2017. Available: http://www.stevever-meulen.com/index.php/2017/09/using-async-await-in-unity3d-2017/ [Accessed: 24 March 2021]

FIGURES