[Background 3](#_Toc6328787)

[Saving data in games 5](#_Toc6328788)

[Introduction 5](#_Toc6328789)

[Commercial and contractual considerations 5](#_Toc6328790)

[Saving data in Unity3D 5](#_Toc6328791)

[Storing data locally 6](#_Toc6328792)

[Data serialisation in Unity3D 7](#_Toc6328793)

[3rd party solutions 7](#_Toc6328794)

[Creating the level editor 8](#_Toc6328795)

[Requirements 8](#_Toc6328796)

[Level editor functionality 8](#_Toc6328797)

[Level objects 10](#_Toc6328798)

[Serialising level object data 12](#_Toc6328799)

[Custom serialisation 12](#_Toc6328800)

[Using JSON for serialisation 12](#_Toc6328801)

[Saving object data 14](#_Toc6328802)

[Loading a game level 14](#_Toc6328803)

[Storing level data in the database 15](#_Toc6328804)

[Using a BaaS 15](#_Toc6328805)

[Choosing the service 15](#_Toc6328806)

[Google Firebase products 17](#_Toc6328807)

[Realtime Database 17](#_Toc6328808)

[Authentication 18](#_Toc6328809)

[Integrating Firebase with Unity3D 19](#_Toc6328810)

[Data structure 19](#_Toc6328811)

[Storing data 19](#_Toc6328812)

[Retrieving data 19](#_Toc6328813)

[Security considerations 20](#_Toc6328814)

[Financial considerations 21](#_Toc6328815)

[Conclusion 21](#_Toc6328816)

[Further development 21](#_Toc6328817)

[References 22](#_Toc6328818)

Vocabulary

Game engine

Unity3D

A game engine developed by Unity Technologies. (has a free version and offers ability to develop for different platforms, including mobile, also offers services for monetisation (ads))

Firebase

IAP

In-game currency

Game level

Serialisation

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Background

The purpose of the thesis was to study saving and serialising data in games.

The game “Interdimentional Nuisance” (sic) was developed in one week in 2016 as part of the Castle Game Jam event. In the game, the player aims and shoots projectiles at ghosts. The goal in the game is to hit, and subsequently destroy, all the ghosts in the level. The game was demoed to the audience at the end of the game jam event. Based on the feedback, the game would be suitable for further development and release for mobile devices.



Picture 1 First level in the game “Interdimentional Nuisance”

The largest workload for such a released game would be designing and building content, i.e. levels. Since this would be a huge undertaking for a sole developer, it would help immensely if some of the content was created by the players themselves.

There are also other benefits to allowing players to create their own levels. People who create their own levels, would most probably share their creations with their friends, thus simultaneously creating word of mouth -marketing for the game.

-reference to games where people can create their own things

-more info on the game itself

-how and where was it made and how would it be released as a commercial game

-link to itch.io

Työn tavoite

Levels created with the editor are stored on a server. Players can then download and play them.

Ease of maintenance as well as commercial profitability were important factors in the design.

The game engine that was used for this project was Unity3D.

-saving and sharing levels with other players

-cost effectiveness and optimization as part of the game design

-scalability: possible for a sole developer to maintain, but doesn’t hinder further development

# Saving data in games

-how and why data is saved in games in general

-levels, settings, user info, game progression

## Introduction

The very first computer games didn't store any data due to limits in hardware (reference). Games such as TicTacToe (reference) and Pong (reference) were short and the game would end after the play session. Even after the games became longer, games didn't necessarily store game state data on the computer. Instead, in games such as (reference), the game would provide a password after finishing a level. This password could then be used next time to start the game at the level the player had previously reached. Some games used the inability to save the game state as a design feature; Treasure of Tarmin had four difficulty settings which were measured as the number of levels required to reach the final treasure. The gameplay at the easiest setting would be 5 minutes, while on the most difficult setting the game would last for 5 hours.‎2

Taito’s Space Invaders was the first game to store the player’s high score. (reference)

-what kinds of data is used now

-save files and sharing them (physically)

-examples of games (Super Mario Maker)

## Commercial and contractual considerations

Both Google Play store and the App Store require (reference) that the developer provide a way for the user to restore any purchases they've made through the game. This means that the data for the purchases needs to reside on somewhere other than locally on the user's device. This ensures that the purchase is not device dependent and users will not lose what they have bought in case they switch devices. Other situations where restoring might be needed is if something happens to the user's device or if they uninstall, and later reinstall, the game on their device.

Storing data locally can also be a commercial liability. Even mobile games can be hacked using different hacking tools. An example of a game that can be hacked is the popular Imangi Studios' Temple Run (reference)(reference to hack). By using a hacking tool, a user could access and change the locally stored game data. A hacker could add in-game currency into their game and use it to unlock features that would otherwise cost real money. Games such as Supercell's Clash of Clans (reference) have made this impossible. The game will connect to their server every time it's launched, and will make sure that the locally stored data is in sync with the data stored on their servers. The only way to get more in-game currency is to make an actual purchase.

## Saving data in Unity3D

Unity3D game engine offers different ways to save data. Additionally, there are different commercial third-party solutions.

### Don’t Destroy On Load

In Unity3D, the content of the game can be split to different scenes. As an example, one level in a game can be one scene. As the game progresses, certain data needs to be shared between these scenes. As an example, the current score might need to be saved during runtime. This information can be saved in a particular game object which is loaded during each scene.

When a new scene is loaded, Unity3D destroys the currently loaded game objects and creates new game objects that are associated with the new scene being loaded. DontDestroyOnLoad() -function tells Unity, that the object should not be destroyed during new scene load. This way the data stored in the object will be accessible in the new scene as well.

When using DontDestroyOnLoad(), it is important to remember to take into account the possibility, that the scene where the game object was originally created might be loaded again. In this case, we might end up with two of the same object. This is why it’s important to use the Singleton-method to ensure that there will only ever be one of this particular object.

### Player preferences

PlayerPrefs stores data locally in the game’s folder. The data stored in PlayerPrefs could be information that is needed in between game sessions; for example, high score.

By default, data stored in PlayerPrefs is not encrypted. This can become a problem if PlayerPrefs is used to store things like the amount of in-game currency. A user with sufficient abilities can “hack” into the file and change the information, thus bypassing the requirements to obtain this currency.

This was a problem with the popular Temple Run mobile game. Users could install a “hacking” app to their phones and change their high score and the amount of owned game currency.

PlayerPrefs data can be encrypted using third-party assets or by creating a proprietary encryption solution. However, this doesn’t ensure that the data would not be corrupted.

### XML and JSON

-JSON serialization

### Binary files

### Other solutions

-INI files

-custom serialization

-3rd party addons

# Creating the level editor

## Requirements

The requirements for the level editor were such that it should be intuitive and easy to use on a mobile device. However, the level editor is only supposed to be a prototype for testing how to store and restore the level data.

## Level editor functionality

The level editor enables the players to create their own versions of the game levels.

Future development options would include testing the level in a separate play mode. Only levels that can be played through by the level creator can be saved and shared with others. This would prevent sharing of levels which are impossible to complete. This kind of design is used for example in Nintendo’s Super Mario Maker [lähde].

The prototype of the level editor has a simplified UI for testing purposes. The most important function of the prototype is to be able to place objects and store information about the placed objects and their position. The level editor is also used for the purpose of loading previously saved level information.

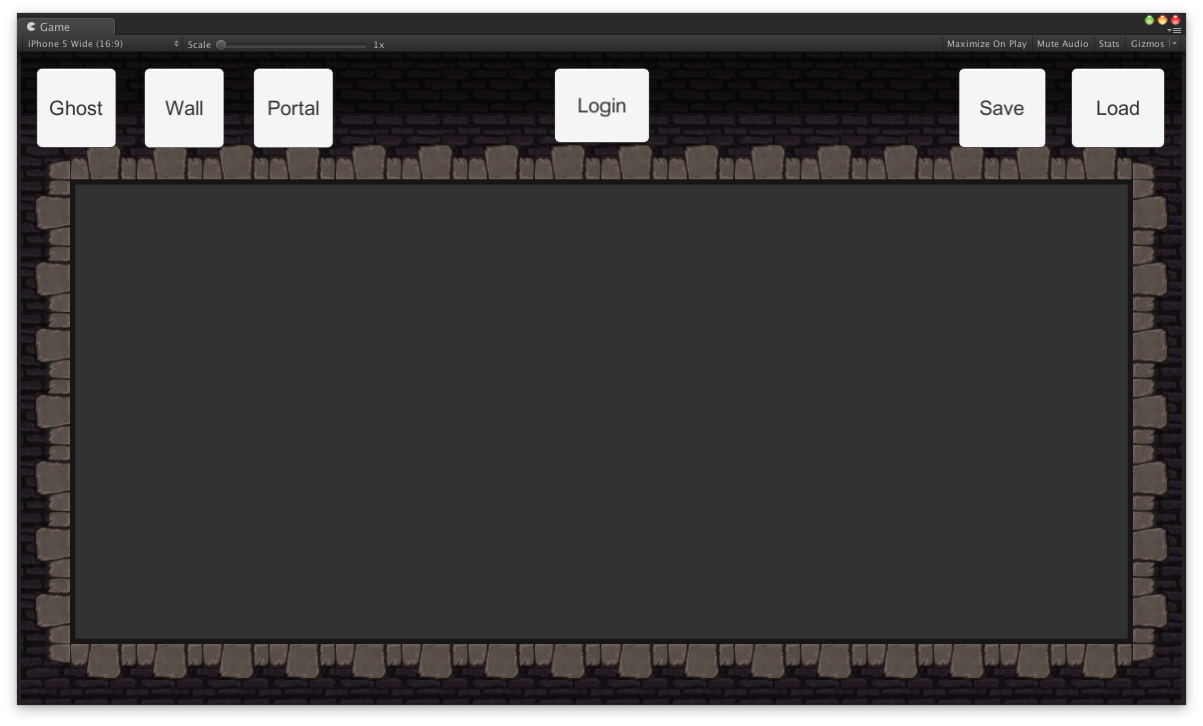


Figure 1

In the level editor UI, the buttons for different objects are located on the upper left corner.

The area where the objects can be placed is the empty space with grey background in the middle of the level editor screen. If an object is placed outside of this area, the object will be destroyed.

The login button in the UI was for testing login into the Firebase server using an e-mail address and password. In future development, this would be replaced with a different kind of authentication system.

The save button on the upper right corner is used when the user wants to save the level data. The load button is used for loading an existing level data.

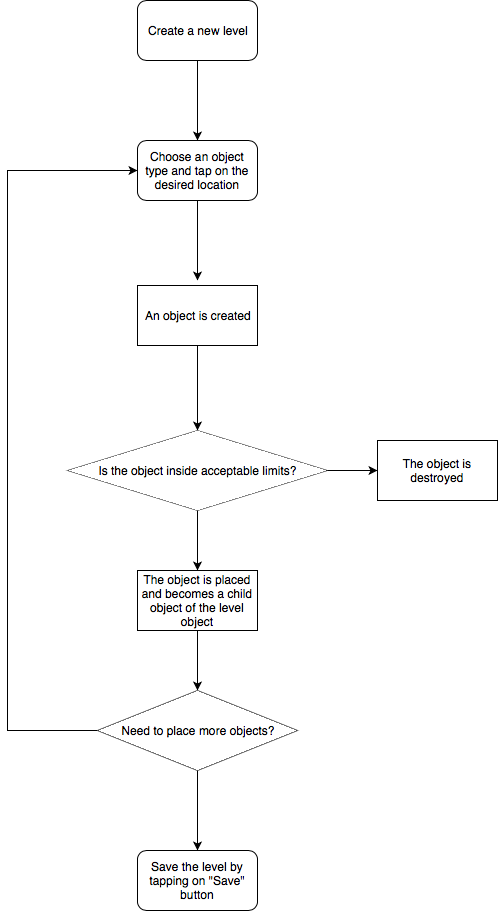


Figure 2

When a user places a new object in the editor, an actual game object is create in the level editor scene and placed as a child object underneath a game object called Level.

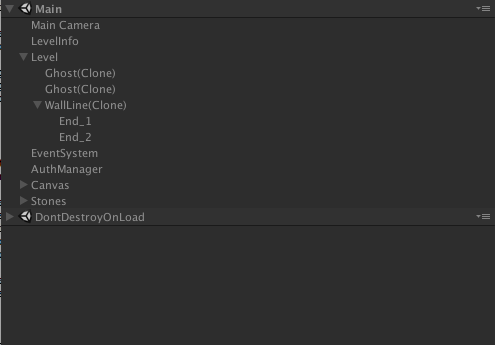


Figure 3

### Level objects

Since the objects in the existing game were fairly simple, static objects, some new ones were created for the sake of testing and to support future development.

The objects that can be placed in the level editor are as follows:

* Ghost - a static object
* A wall - an object which has a starting point and an ending point
* A portal - two objects which are linked together

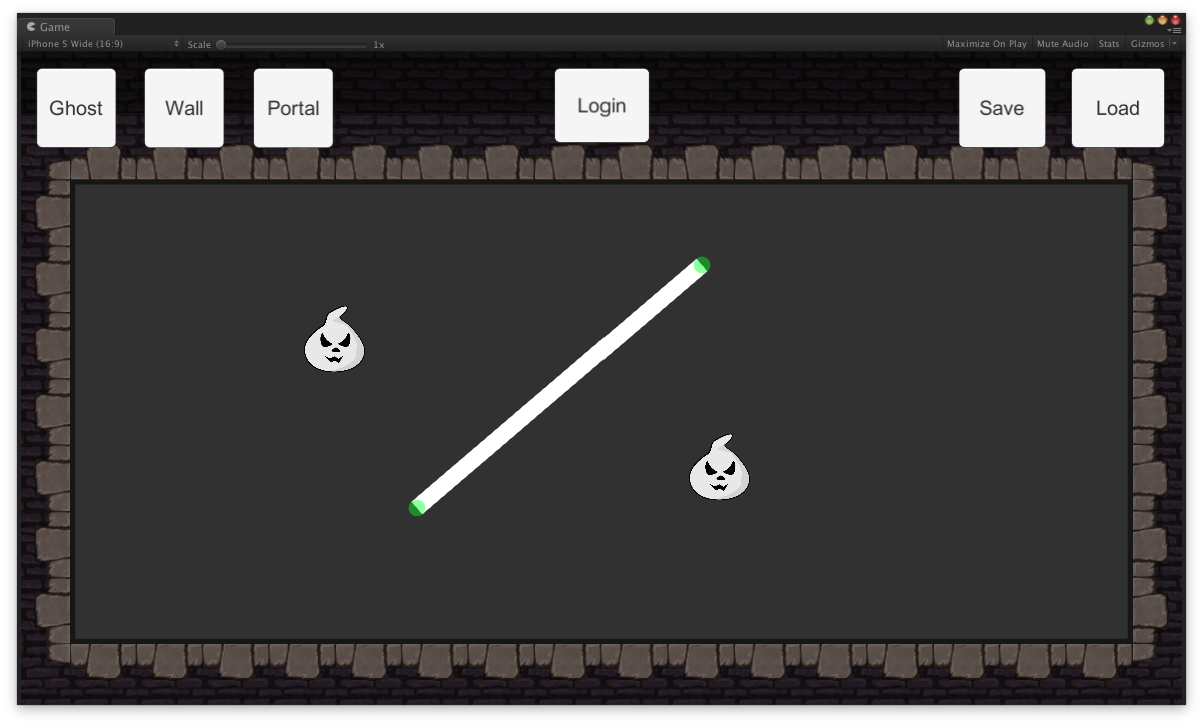


Figure 4

Placing a static object is fairly straightforward: the user presses on the object button and then taps on where they want to place the object.

When placing a wall object, the user first presses on the corresponding button and then taps on where they want to place the wall object. After the wall is placed, the user can then hold their finger on either end of the wall and drag the end point to where they want the wall to being or end.

-tähän vielä juttu portalista

## Serialising level object data

Different objects within the level require different types of information to be stored. A singular, static object only needs information about what type of object it is and where it’s positioned. The wall object requires information about where the object begins and where it ends, i.e. there has to be two different positions. The portal object is technically similar to the wall object, as it also requires information of two different positions. The difference between the objects is in how they are placed during the level editing.

### Custom serialisation

Since the information stored is fairly simple, a custom serialisation method was first developed. This was used for testing the functionality of the editor and saving data locally.

Each object had an identifier such as “ghost”, followed by it’s x and y position. The information for each object was delimited using a comma. Each object was delimited using a semicolon. The whole string was stored in a text file. When the level was loaded, the string in the text file was parsed and the objects were created according to the information in the file.

|  |
| --- |
| ghost,-3.921875,0.3720486;ghost,0.1465282,3.237326;  ghost,4.468577,0.07899284 |

The custom serialisation method wasn’t very elegant, but it worked and it produced a small file size. The file size could’ve been reduced even further, if the object identifier would’ve been just a number instead of a word. Another way to further reduce the file size would’ve been to design the level editor to have a grid. If the object can only be placed in a grid, the x and y coordinates could’ve been integers, instead of the longer floats.

### Using JSON for serialisation

Since the chosen cloud service provider, Google Firebase, uses JSON for storing data and Unity3D already offers JSON serialisation, it was decided that the serialisation should be done using JSON. While the custom made string could’ve also been stored on the Firebase server, using JSON seemed like a more elegant solution compared to creating and parsing strings.

|  |
| --- |
| {  “dataList”: [  {  “objType”: “0”,  “objPos”: {  “x”: “-0.21970559656620027”,  “y”: “0.24093663692474366”  “z”: “2.0”  }  },  {  “objType”: “2”,  “objPos”: {  “x”: “-0.04537254944443703”,  “y”: “0.2841036021709442”  “z”: “2.0”  },  “startPos”: {  “x”: “4.466002941131592”,  “y”: “0.048967257142066959”  “z”: “-9.0”  },  “endPos”: {  “x”: “-5.744858741760254”,  “y”: “-14.92019271850586”  “z”: “-7.0”  },  }  ]  } |

Figure 4

Even though the prototype only has three different types of objects, the plan was to design something that can be easily used with any future additions. For this reason, each object handles its own data serialisation, which is invoked through a common interface.

-luokkakaavio

Interace class ISaveLevelData has two functions: GetObjectJSON(), which returns the object specific JSON as a string and RestoreObject(string objJSON), which restores all the needed information for the recreated object during the level loading phase.

## Saving object data

When the user has placed all the objects they want in the level editor, they press on the “Save” button. Pressing the button executes a function called SaveData(), which is located in the Serializer.cs script.

The SaveData() function creates a new LevelData object and a new list for storing strings. It then loops through each child object of the Level game object and adds them to a list. This list is then looped through. Each object has a common interface, which enables the script to get the corresponding object’s data. After each objects data has been fetched and stored in a list, this list is then stored as a JSON file using Unity3D’s JSON utility.

-kuva koko jutusta tähän

## Loading a game level

When loading the saved game level, the JSON file will be read into a string variable. Using the Unity3D’s JSON utility, the string variable is then turned into a new LevelData object with a new list of the saved game objects.

The list with all the level’s game objects is looped through and depending on the type of object, a corresponding new game object will be instantiated. This object will then be added as a child object of the level. It will then receive the information about its position through the ISaveLevelData interface.

-kuva koko hommasta tähän

# Storing level data in the database

## Using a BaaS

BaaS, an acronym for Backend-as-a-Service, is a cloud based service model. It allows software and application developers to connect to backend systems through an API. A backend service aimed at mobile developers can also be called MBaaS, or Mobile Backend-as-a-Service.

BaaS often includes services like cloud storage, push notifications, user and file management, social networking integration and user management (‎3).

## Choosing the service

Different services providers were considered to be used for storing the level data on a server. The main criteria for the service were as follows:

* being able to store and retrieve data from Unity3D
* free tier for testing and for a game that doesn’t have many players yet
* ease of use and maintenance; something that one person can handle
* safe to use

The following service providers were considered:

* Amazon AWS
* Google Cloud
* Google Firebase
* Microsoft Azure
* Heroku

|  |  |  |
| --- | --- | --- |
| **Service provider** | **Positives** | **Negatives** |
| Amazon AWS | huge amount of different services  possibilities for expansion | big learning curve  difficult to determine costs |
| Google Cloud | easier to understand than Amazon AWS | only usable for registered companies within the EU |
| Google Firebase | designed to be used with games  easy to use | currently only for mobile and web player games  client driven, not server driven |
| Microsoft Azure |  | big learning curve |
| Heroku | cheap | not a lot of ready-made solutions |

In the end, Google Firebase was chosen mainly because of the ease of use. All other services would’ve required a lot of time to be spent on learning how to use and create cloud services. For just one person that would’ve been too big of an overtaking. Especially considering that the service has to have proper security measures when taken to production.

Google Firebase offers solutions that are aimed at game developers and has built-in authentication. At the time of writing this thesis, Google Firebase was designed to only work with mobile and web-player games made in Unity3D. This was acceptable, as the game was designed to be released on mobile.

## Google Firebase products

Products and services offered by Google Firebase are mainly aimed at mobile app developers. Firebase offers easy integration when developing apps with Swift, Objective-C, Java, JavaScript, C++ or in Unity3D. In Unity3D, Firebase is only supported when developing games for iOS, Android or web. This means that if the game is later released on PC, Mac or on a game console, Firebase can't be used as a backend system.

Google Firebase offers pricing tiers for different needs. The free "Spark Plan" was used for the prototype developed in this thesis ‎3.

Out of the different services offered, Realtime Database and Authentication was used with the prototype created during this thesis.

### Realtime Database

The Firebase Realtime Database is a NoSQL database hosted in Google Firebase’s cloud.

-real time syncing between users and devices

-offline usage

-challenges (too much data to be synced, data structure needs to be designed well)

-challenges with non-relational database

### Authentication

Google Firebase offers different types of authentication for identifying end-users. For the sake of simplifying the process, the prototype created during this thesis uses email and password pairs for authentication. A possible end-product would most likely use Game Center on iOS and Google on Android.

The purpose of the authentication is to allow end-users to create and manage their own level designs. Authentication can also be used for other purposes, such as storing information on any in-app purchases.

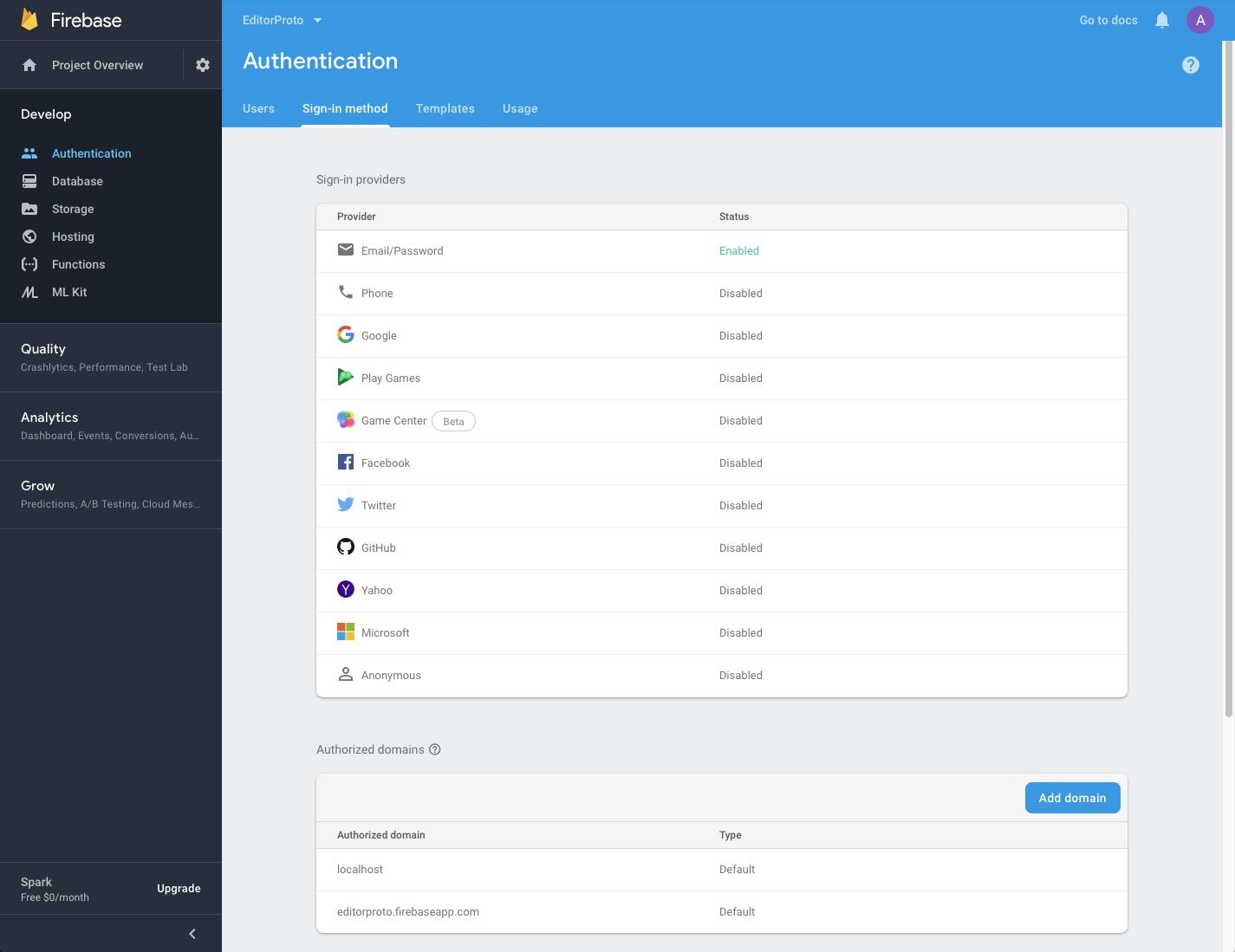


Figure 5

*-kuva autentikointi-tapahtumasta (tilakaavio)*

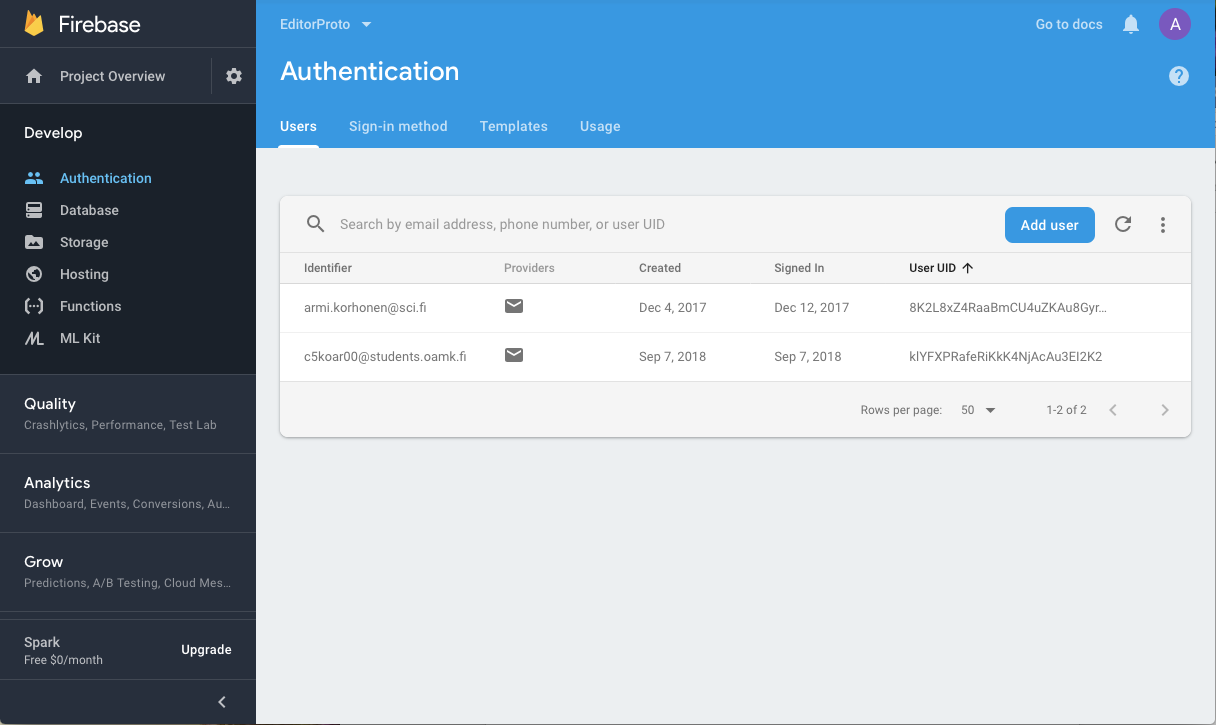


Figure 6

## Integrating Firebase with Unity3D

Most of the Firebase services can be integrated with native iOS or Android applications. The game used in this thesis was developed with the Unity3D game engine, so the services that can be used are more limited. The services that can be used with Unity3D are Realtime Database, Cloud Storage, Authentication and Cloud Functions.

-console settings

-plist file

-Unity3D files

## Structuring the data

Designing the data structure is one of the most challenging parts of creating and using a database. When designing a relational database, data redundancy is avoided when possible. This isn’t necessarily the case with the Firebase Realtime Database. The database is a JSON tree, which should be kept as flat as possible. This means that even though the database JSON tree can have up to 32 levels of nested data, every time a node is retrieved, all its child nodes are also retrieved.

The security of the Realtime Database heavily relies on rules, which are combined with authentication. If a person has access to one node, they will have access to all the child nodes as well.

-tähän kuva nodeista

To avoid too much nested data, some data needs to be duplicated. This enhances the security of the database, as well as performance. (lähde)

-kuva tähän esimerkistä

Because of these requirements for the data structure, it was decided that the user data would be in its own node and the levels in a separate node. If each user’s own levels would’ve only resided within the user nodes, other users would not have been able to access that data without also having access to other user data. This would’ve posed a potential security risk.

Having all the level data nested within a dedicated node produced further problems. The original idea was to let users get, for example, 10 random new levels created by other users. The Firebase SDK does not offer a way to access a random node within the database. It also doesn’t offer a neat way to know how many nodes there are, so creating a random number on the client side wouldn’t have worked either.

Another idea was to just get the first 10 levels from the list, and later the next 10 and so on. It’s possible to filter the retrieved data to the first 10.

-tarkista vielä että säilyykö ensimmäiset kymmenen paikallisesti ja jos hakee seuraavat kymmenen, niin riittääkö silloin update?

One way to get the levels would be to create just a list with all the level names, without the level data. This list would then be downloaded to all devices. Each device would then locally keep track of which levels they have played and retrieve new level data based on the list. The issue with this approach is that each device would download a large amount of data, regardless of if they would end up playing the game or not. Since the game would be a free to play game, there would be no guarantee that the advertising revenue would cover the costs of the Realtime Database service.

-why the data structure has been designed the way it is

## Storing data

-using the router

## Retrieving data

## Security considerations

Due to the way Google Firebase has been designed to work, certain security measures should be taken into consideration before moving into production. The reference to the database and it's API key reside on the client side. A knowledgeable hacker can access that information and use it for accessing and tampering with the database. (reference) Because of this, the way to secure the database is to use the Firebase Realtime Database Rules.

The Firebase Realtime Database Rules determine who can read and write data in the database, what indexes exist and how the data is structured (ref. https://firebase.google.com/docs/database/security/) By default, the rules do not allow anyone to access the database. When creating a new database, it's also possible to start in "test mode", which enables all access to the database. Using the test mode can make it easier to focus on developing the data structure and the client side functionality. However, before moving into production, the rules should be set to only allow authorised clients to access their own data.

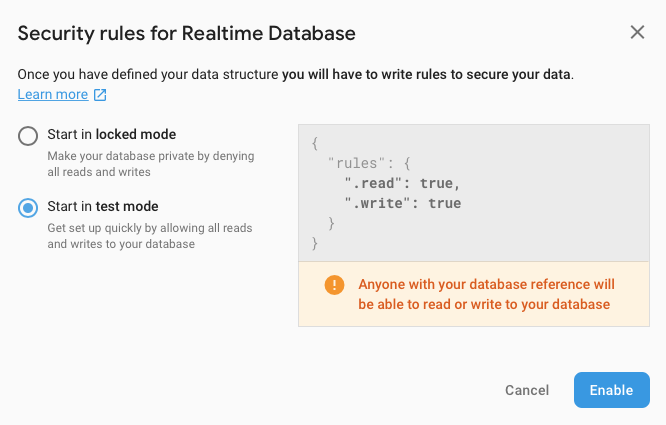


Figure 7

# Financial considerations

Estimated expenses

Revenue requirements

-how many Unity ads should be shown to make profit?

# Conclusion

## Further development

# References

1. Interdimentional Nuisance. Cited 06.04.2019   
   <https://northernerd.itch.io/interdimentional-nuisance>
2. Williams, A. 2017. History of Digital Games. Focal Press. Retrieved on 16.04.2019 from <http://proquest.safaribooksonline.com.ezp.oamk.fi:2048/book/programming/game-programming/9781317503804/altering-time-in-home-console-games/sec115_html?uicode=ouluuas>
3. Technopedia. Backend as a Service. Cited 16.04.2019  
   <https://www.techopedia.com/definition/29428/backend-as-a-service-baas>
4. IGN’s Top 10 Most Influential Games. Cited 16.04.2019  
   <https://www.ign.com/articles/2007/12/11/igns-top-10-most-influential-games>
5. Unity: Working with Google Firebase. Cited 05.04.2019  
   <https://www.linkedin.com/learning/unity-working-with-google-firebase/firebase-101>
6. Firebase Products. Cited 16.04.2019  
   <https://firebase.google.com/products/>
7. Google Firebase Pricing Plans. Cited 05.04.2019   
   <https://firebase.google.com/pricing/>
8. Firebase Realtime Database. Cited 06.04.2019  
   <https://firebase.google.com/products/realtime-database/>
9. Understand Firebase Realtime Database Rules. Cited 07.04.2019  
   <https://firebase.google.com/docs/database/security/>