ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Ανάπτυξη εργαλείου Web για τη σύνταξη και λειτουργία παιχνιδιών τοποθεσίας με χρήση αφαιρετικών προγραμματιστικών δομών

ΔΗΜΗΤΡΙΑΔΗΣ ΒΑΣΙΛΕΙΟΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

ΔΑΔΑΛΙΑΡΗΣ ΑΝΤΩΝΙΟΣ
ΕΠΙΚΟΥΡΟΣ ΚΑΘΗΓΗΤΗΣ

ΣΥΝΕΠΙΒΛΕΠΩΝ

ΚΟΖΥΡΗ ΜΑΡΙΑ
ΕΠΙΚΟΥΡΗ ΚΑΘΗΓΗΤΡΙΑ

Λαμία Οκτώβριος 2020

ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Ανάπτυξη εργαλείου Web για τη σύνταξη και λειτουργία παιχνιδιών τοποθεσίας με χρήση αφαιρετικών προγραμματιστικών δομών

ΔΗΜΗΤΡΙΑΔΗΣ ΒΑΣΙΛΕΙΟΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΥΠΕΥΘΥΝΟΣ

ΔΑΔΑΛΙΑΡΗΣ ΑΝΤΩΝΙΟΣ
ΕΠΙΚΟΥΡΟΣ ΚΑΘΗΓΗΤΗΣ

ΣΥΝΕΠΙΒΛΕΠΩΝ

ΚΟΖΥΡΗ ΜΑΡΙΑ
ΕΠΙΚΟΥΡΗ ΚΑΘΗΓΗΤΡΙΑ

Λαμία Οκτώβριος 2020

UNIVERSITY OF THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE and TELECOMMUNICATIONS

Development of a Web portal for authoring and deploying position-based games using abstract domain specific programming language structures

DIMITRIADIS VASILEIOS

FINAL THESIS

ADVISOR

DADALIARIS ANTONIOS
ASSISTANT PROFESSOR

CO ADVISOR

KOZIRI MARIA
ASSISTANT PROFESSOR

Lamia October 2020

«Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις (1), που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

1. Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί ***χωρίς να τα περικλείω σε εισαγωγικά*** *και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

2. Δέχομαι ότι η αυτολεξεί ***παράθεση χωρίς εισαγωγικά****, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια

4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.

Ημερομηνία: 20/10/2020

Ο - Η Δηλ.

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση
του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων
σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.»

ΠΕΡΙΛΗΨΗ

Τα παιχνίδια βάσει τοποθεσίας είναι ένα νέο είδος παιχνιδιού, συνδυάζοντας τον φυσικό και ψηφιακό κόσμο, οδηγώντας τους χρήστες με έξυπνες κινητές συσκευές που έχουν την δυνατότητα GPS να επισκέπτονται φυσικές γεωγραφικές τοποθεσίες και να αλληλεπιδρούν ψηφιακά μαζί τους. Το γεγονός ότι ο χώρος παιχνιδιού συνδυάζει τόσο τον ψηφιακό όσο και τον φυσικό κόσμο δίνει ένα μοναδικό πλεονέκτημα στα παιχνίδια βάσει τοποθεσίας σε τομείς εκπαίδευσης και μάρκετινγκ, ενώ ταυτόχρονα επανασυνδέει τον χρήστη με τον φυσικό κόσμο και βελτιώνει έμμεσα τις φυσικές του δραστηριότητες. Αυτό το είδος έχει ήδη αποδειχθεί επιτυχημένο σε άτομα νέας ηλικίας με ορισμένους μεγάλους τίτλους παιχνιδιών να γίνονται πρωτοσέλιδα, οδηγώντας σε οικονομική ανάπτυξη. Ωστόσο, η ανάπτυξη τέτοιων παιχνιδιών απέχει πολύ από το να είναι ασήμαντη καθώς απαιτεί υψηλές δεξιότητες προγραμματισμού, ενώ η επιβάρυνση της διαχείρισης που προκαλείται από την ανάπτυξη παιχνιδιού μεγάλης κλίμακας είναι επίσης εμπόδιο. Με κίνητρο τα παραπάνω, παρουσιάζουμε μια επισκόπηση του σχεδιασμού και της αρχιτεκτονικής μιας υπηρεσίας, της Πλατφόρμας GeoMakeIt!, σε μια προσπάθεια μείωσης του χάσματος μεταξύ της γνώσης ανάπτυξης παιχνιδιών και των δεξιοτήτων προγραμματισμού που απαιτούνται για την ανάπτυξη τέτοιων παιχνιδιών. Η προτεινόμενη πλατφόρμα έχει σχεδιαστεί με ανοιχτό τρόπο, ενεργώντας ως μία αγορά για προσαρμόσιμα στοιχεία, Plugins, που μπορούν να συντεθούν και να παράγουν διαφορετικές εμπειρίες παιχνιδιών. Μειώνοντας απαιτήσεις στις γνώσεις και δεξιότητες, οραματιζόμαστε το GeoMakeIt! θα προσελκύσει το ενδιαφέρον τόσο των έμπειρων προγραμματιστών παιχνιδιών όσο και των ερασιτεχνών, να διαφοροποιήσουν και να εμπλουτίσουν αυτό το νέο είδος παιχνιδιών.

ABSTRACT

Location based games is a new and primitive gaming genre, blending the physical and digital world by driving users with GPS enabled smartphone devices to visit physical geographical locations and interact with them digitally. The fact that the playing area combines both the digital and natural world gives a unique advantage to location-based games in fields of education and marketing while concurrently reconnecting the user with the physical world and indirectly improving his physical activities. This genre has already been proven successful in the youth target group with certain big gaming titles making headlines, leading to financial growth. However, developing such games is far from being trivial requiring high programming skills, while the administration overhead induced by large scale deployment is also impediment. Motivated by the above, we present an overview of the design and architecture of a service, the GeoMakeIt! Platform, in attempt to lowering the gap between game development knowledge and the programming skills required for developing such games. The proposed platform is designed in an open-ended manner, acting as a marketplace for customizable plugin components that can be synthesized and output diverse game experiences. By lowering the knowledge and skill gab, we envision GeoMakeIt! will attracting the interest of both experienced game developers and amateurs to diversify and enrich this new and primitive genre.

Table of Contents

ΠΕΡΙΛΗΨΗ IX
ABSTRACT XI

TABLE OF CONTENTS 12

CHAPTER 1: INTRODUCTION 16

(GAMES 1.1) 16
(HISTORY OF GAMES 1.1.A) 16
(MOBILE GAMES 1.1.B) 16
(LOCATION-BASED GAMES 1.1.C) 17
(GAME DEVELOPMENT 1.2) 18
(DEFINITION AND PROCESS 1.2.A) 18
(GAME ENGINES 1.2.B) 18
(GAME ENGINE VS AUTHORING TOOLS 1.2.C) 19
(ANDROID OPERATING SYSTEM 1.3) 19
(INTRODUCTION 1.3.A) 19
(ANDROID ARCHITECTURE 1.3.B) 20

CHAPTER 2: GEOMAKEIT! 22

(CONCEPTS 2.1) 22
(INTRODUCTION 2.1.A) 22
(AUDIENCE 2.1.B) 22
(USER ROLES 2.1.C) 22
(PLATFORM: GEOMAKEIT! STUDIO 2.1.D) 23
(API: GEOMAKEIT! API 2.1.E) 24
(TECHNOLOGIES 2.2) 24
(APPLICATION DEVELOPMENT PLATFORMS 2.2.A) 24
(GEOMAKEIT! STUDIO LANGUAGE AND FRAMEWORK CHOICES 2.2.B) 25
(MAPS API CHOICES 2.2.C) 28
(DATABASE CHOICES 2.2.D) 32
FINAL THOUGHTS 33

CHAPTER 3: GEOMAKEIT! API 35

(INTRODUCTION TO THE API 3.1) 35
(OVERVIEW 3.1.A) 35
(API WORKFLOW 3.1.B) 35
(API COMPONENTS 3.2) 36
(THE “APP” COMPONENT 3.2.A) 36
(THE “GAME” COMPONENT 3.2.B) 37
(THE “PLUGINS” PACKAGE 3.2.C) 37

(THE "ACTIVITIES" PACKAGE 3.2.D) 39
(THE "SERVICE" COMPONENT 3.2.E) 42
(THE "COMMANDS" PACKAGE 3.2.F) 42
(THE "PLACEHOLDER" PACKAGE 3.2.G) 45
(THE "CONFIGURATION" COMPONENT 3.2.H) 47
THE PLUGIN.JSON FILE 47
(THE "UTILITIES" COMPONENT 3.2.I) 49
(THE "MODEL" COMPONENT 3.2.J) 50
(OTHER GEOMAKEIT! COMPONENTS 3.2.K) 50
(GEOMAKEIT! DEFAULT PLUGIN 3.3) 52
(THE "MAIN" CLASS 3.3.A) 52
(ANDROID FILES AND RESOURCES 3.3.B) 53
(COMMANDS 3.3.C) 54
(PLACEHOLDERS 3.3.D) 55
(CONFIGURATIONS 3.3.E) 56
(MODELS 3.3.F) 57

CHAPTER 4: GEOMAKEIT! STUDIO 70

(INTRODUCING THE STUDIO 4.1) 70
(OVERVIEW 4.1.A) 70
(LANDING PAGE 4.1.B) 70
(SIGNING UP / SIGNING IN 4.1.C) 70
(FIRST VIEW TO THE STUDIO 4.1.D) 71
(SIGNING IN / SIGNING UP 4.2) 72
(SIGNING UP AS A GAME CREATOR 4.2.A) 72
(BECOMING A PLUGIN DEVELOPER 4.2.B) 73
(SIGNING IN TO GEOMAKEIT! STUDIO 4.2.C) 73
(PLUGINS 4.3) 74
(REGISTERING AN IDENTIFIER 4.3.A) 74
(UPLOADING A PLUGIN 4.3.B) 75
(PLUGIN MANAGEMENT 4.3.C) 76
(DELETING A PLUGIN 4.3.D) 76
(GAMES 4.4) 76
(CREATING A GAME 4.4.A) 77
(CONFIGURING GEOMAKEIT! API AND OTHER PLUGINS 4.4.B) 78
(MANAGING OTHER PLUGINS 4.4.C) 80
(BUILDING A GAME 4.4.D) 80
(WEBSITE COMPONENTS 4.5) 81
(PLUGIN UPLOAD PROCESS 4.5.A) 81
(GAME BUILDING PROCESS 4.5.B) 82

CHAPTER 5: GEOQUIZ – IN DEPTH LOOK INTO A COMPLETE PLUGIN 85

(INTRODUCTORY 5.1) 85
(HOW TO INSTALL GEOQUIZ 5.1.A) 85
(GEOMAKEIT! STUDIO OPTIONS 5.1.B) 85
(FINAL BUILT QUIZZES EXAMPLES 5.1.C) 86
(SOURCE CODE AND COMPONENTS 5.2) 86

(DIRECTORY STRUCTURE 5.2.A) 86
(RESOURCES AND MANIFESTS 5.2.B) 87
(CONFIGURATIONS 5.2.C) 89
(GEOQUIZ MAIN CLASS 5.2.D) 96
(ACTIVITIES 5.2.E) 97
(MODELS 5.2.F) 100
(FUTURE IMPLEMENTATIONS 5.3) 104
(FINAL THOUGHTS 5.4) 104

CHAPTER 6: CREATING A PLUGIN FROM SCRATCH 106

(PREREQUISITES 6.1) 106
(OPERATING SYSTEM 6.1.A) 106
(PRIOR KNOWLEDGE 6.1.B) 106
(SIGN-IN/SIGN-UP TO GEOMAKEIT! PLATFORM 6.1.C) 106
(HELLO WORLD 6.2) 107
(CREATING YOUR PLUGIN 6.2.A) 107
(HELLO WORLD 6.2.B) 107
(CHOOSING AN IDENTIFIER 6.2.C) 108
(RUNNING / TESTING YOUR PLUGIN 6.2.D) 109
(COMMON COMPONENTS 6.3) 110
(DIRECTORY STRUCTURE 6.3.A) 110
(ACTIVITIES 6.3.B) 111
(CONFIGURATIONS 6.3.C) 113
(MODELS 6.3.D) 116
(PLACEHOLDERS 6.3.E) 117
(COMMANDS 6.3.F) 119
(CLEANING UP 6.3.G) 121
(UPLOADING AND PUBLISHING YOUR PLUGIN 6.4) 121
(THE PLUGIN.JSON FILE 6.4.A) 121
(REGISTERING YOUR IDENTIFIER 6.4.B) 121

CHAPTER 7: THE FIRST GEOMAKEIT! GAME. 123

(INTRODUCTION 7.1) 123
(GAME CREATION PROCESS 7.2) 123
(CREATING A GAME 7.2.A) 123
(INSTALLING PLUGINS 7.2.B) 124
(CONFIGURING GEOFIGHTING! 7.2.C) 125
(CONFIGURING GEOQUIZ 7.2.D) 126
(CONFIGURING GEOMAKEIT! 7.2.E) 127
(GAME BUILDING PROCESS 7.3) 134
(INSTALLING AND PLAYING OUR GAME 7.4) 134

CHAPTER 9: FUTURE PLANNING 136

REFERENCES 139

CHAPTER 1: INTRODUCTION

(Games 1.1)

(History of Games 1.1.a)

From as early as 3000 BC[1] and inevitably for the rest of humanity's history, games have been entangled to our lives and engraved in all cultures. Usually undertaken for entertainment and fun, while other times they end up becoming a valuable method of education. Games can be played alone, distinguished as single-player games, or in teams of people, usually labeled as multi-player games. They are a form of entertainment enjoyable both by being an active member of the game, a player, or even a passive one, the audience.

Games generally involve mental or physical stimulation, and more often than not; both. They are usually composed of a set of rules, a goal that needs to be achieved, interaction between components or other players and must usually present a form of challenge.

(Mobile Games 1.1.b)

Mobiles Games are played in a mobile device, usually a smartphone or a tablet, but it isn't unheard for games to be played or developed for smartwatches, PDAs, portable media players or even calculators! The earliest instance of a mobile game being developed would be a Tetris[2] variant for the MT-2000 device. Mobile games were destined to become one of the gaming platforms major players as mobiles were becoming an affordable every-day use item and a powerful computer machine.

Proof of that was made as early as 1997, when Nokia launched the well-known game Snake[3]. This game has become one of the most played games and could be found in more than 350 million devices.

Today, mobile games are usually downloaded through an app-store, commonly the "Google Play Store" in Android devices and "App Store" in Apple Devices, or even pre-loaded in some devices.

The revenue model[4] has diverged significantly away from traditional physical games such as football, basketball etc. or console/computer games. Unlike the usual pay-to-watch or pay-to-own respectively, mobile game developers have opted for a different model of revenue generation. There are 5 revenue models for mobile games.

"Premium", a model reminding the traditional model, where the user pays for a game to own it upfront. Additional downloadable content may be available either by free updates or micro-transactions.

"Freemium", where the user tries for a limited time a game and then proceed to make a one-time purchase similar to the "Premium" model.

“Free to play”, where the user can download and play the game for free, but usually there is a slow factor such as limited attempts per day or stamina to slow down the user’s progress. These limitations can be usually appealed with micro-transactions.

“Advertisement supported”, where the user can play the game freely but is occasionally shown advertisements. Game Developers can earn revenue by either from an advertising network, such as Google Ads, or from micro-transactions that fully disable these ads.

“Subscription”, where the user can freely play the game but must pay a monthly subscription fee to access an extra set of premium features.

Most of the games usually combine the aforementioned methods to increase revenue. Typically, most of the independent developers prefer the “Free to play” with “Advertising supported” combination to make the game accessible to a wider audience and be able to monetize it.

Lastly, due to their nature, mobile games tend to have some unique limitations. Unlike consoles and computers, mobile games have limited storage and memory which place constraints on quality, size and direct migration from computer or console to mobile development. Being also mobile means that the battery life is limited and as such playtime can be constrained by the battery available. Additionally, intensive games that drain battery quickly or require multiple recharges reduce a battery’s lifetime. Finally, since mobile games are usually tethered to mobile data, in some countries the cost of being constantly online to play games may prove very expensive.

(Location-based Games 1.1.c)

Mobile games that utilize and mainly use GPS as a core mechanic can be titled Location-based games. Unlike other types of games, this genre of mobile games considers the player’s position into the game concept and uses his coordinates and movements as main elements of the game.

There are a few commercial successes to this genre, while most of the games and project of this type are developed as scientific researches[5] or educational[6] material. The most widely known mobile game that utilizes GPS as a core concept is Pokemon Go[7], developed by Niantic. It was released in 2016 and had over 147 million active users in May of 2018, while generating more than 3$ billion in revenue by 2019. Another popular game is GeoCaching[8], a global treasure hunt type of game which allows users to visit places, hide and seek containers called “caches” containing a logbook and sometimes a pen or pencil, gift items and read daily adventures of other people. GeoCaching can be thought as people planting their own sentimental small time-capsules and letting other people physically go and read their stories.

Other available location-based games would be Chase games, ex. [9] and [10], targeting to improve an individual’s health and popular games like PacMan[11][12], Tic-Tac-Toe[13] and Snake[14] are now available also as location based games.

Location-based games make a tiny percent of the global game market. This can be explained by the knowledge and skill-level gab between people wanting to develop such games, the pricey cost to develop and maintain, the lack of standardized tools to develop

this type of genre and more importantly the fact that it hasn't been yet adopted and tested by many people with different options of game styles.

(Game Development 1.2)

(Definition and Process 1.2.a)

Video Game Development is the process of developing a video game. Games can be developed by companies or individuals. In the first case, usually the effort is undertaken by a big team of programmers, designers, artists, sound engineers and testers, and funded by publishers. Alternatively, games are designed by individuals, either single person or a small indie team, where usually the resources are limited.

The latter one has been lately on a rise, with independent game developers using advanced game engines such as Unity and Unreal Engine to develop complex games that can be then published in online distribution platforms such as Steam, UPlay, Android and IOS.

Development process[15] is standardized and can be separated in stages, but it is also time-consuming. The first stage is called "Pre-production", where a concept is formed, basic game design and mechanics are decided, and a prototype is either created in a Game Engine or envisioned using pen and paper. Next, we move to the "Production" stage, where the actual Game Design, Programing, Level Creation, Art Production, Audio Production take place. These then are synthesized and thoroughly tested. Following that, there is a cycle between "Production" and our next stage, "Milestones", were the First playable is released, moving to Alpha, Beta and Full release. Lastly, "Post-production" stage is all about maintaining the game to survive the long run, with typical updates for fixing issues and sometimes updating or upgrading the game mechanics and extending the concept.

(Game Engines 1.2.b)

Game Engines or Game Frameworks are software designed to aid people to build video games[16]. Thy can export games for a variety of platforms such as personal computers, consoles, mobile devices etc., and provide most of the game components out of the box for both 2D and 3D games. The most common subcomponents that make a functioning and powerful game engine is the rendering engine for graphics, the physics engines, sound, animation, threading, scripting and memory management.

There is a plethora of available free, proprietary or even in-house developed game engines available, some of which are developed for specific applications and hardware in mind while others can export in multiple platforms. Some examples of common Game Engines used today are Unreal Engine, Unity, Godot Engine, RPG Maker and Game Maker Studio.

Even when we select only the Android Platform, there are still plenty of available choices. One of the most common Game Engines for Android is Unity where Pokemon Go, LaraCroft Go and Angry Birds were released. Unreal Engine is the second contender, with well-known games such as Heart at Attack and Lineage 2: Revolution.

Other available engine choices would be GameMaker Studio, Corona SDK for 2D Games, Marmalade SDK, App Game Kit, Construct 2, Fusion, Cocoon JS, all of which have different price ranges, strengths and weaknesses.

(Game Engine vs Authoring Tools 1.2.c)

Authoring Tools are software or services that contain a set of pre-programmed elements/components for developing multimedia or software applications[17][18]. Most Game Engines contain pre-programmed elements as well to aid in the Game Development. The big difference between those two is the usual target audience. Game Engines usually target people with some background knowledge in Game Development and usually a certain skill level in some programming or scripting language. Authoring tools are developed usually for non-programmers, usually as an educational software, to easily create software with programming features.

Authoring Tools hide programming features behind buttons, simple text commands and other tools so the author does not need have any programming skills. Usually authoring systems provide many typical graphics, interactions and pre-made scripts that the author might need. Additionally, authoring tools may manage tasks such as compiling behind the scenes the end-program, hosting and delivering it to the end-users. Similar to Game Engines, Authoring tools may include an authoring language but built only for representing the system with limited commands to not overwhelm the author.

(Android Operating System 1.3)

(Introduction 1.3.a)

Android[19][20] is a mobile operating system based on a modified version of Linux Kernel designed primarily for touchscreen devices such as Smartphones and Tablets but is also used as an operating system for Smart-TVs, Smart-watches, game consoles, digital cameras and PCs. It is free and open source, licensed under the APACHE License and is developed by a consortium of developers and commercially sponsored by Google.

Software packages are usually distributed through proprietary application-stores like Google Play Store, Samsung Galaxy Store, Huawei AppGallery, Aptoid or F-Droid, using the APK file format.

Android has been the best-selling mobile operating system for smartphones since 2011 with more than 2 billion monthly active users and 3 million apps.

Many phone manufacturers use Android OS directly, while some of them choose to customize its look and add additional features to differentiate themselves from other competitors.

Applications developed for the Android OS are developed using the Android Software Development Kit, commonly known as Android SDK, often using the Kotlin Language which recently replaced Java as the Android’s programming language choice.

(Android Architecture 1.3.b)

The whole Android Architecture[21] can be divided into 4 layers, as seen in the Figure 1.1.

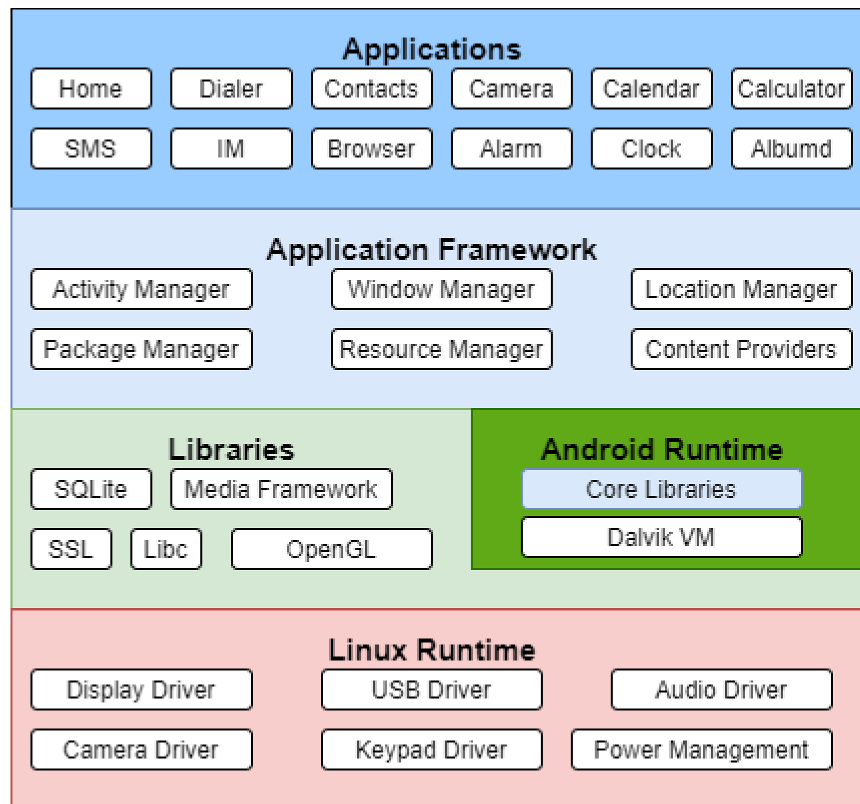

Figure 1.1 Android Architecture

Starting from the bottom layer, "Linux Kernel" provides a level of abstraction between a devices hardware and provides all necessary drivers such as Display, Camera, Bluetooth, USB, Audio etc.

The next layer contains common "Libraries" used both by the Android itself as well as most of the third-party developed applications such as WebKit, the open source Web browser engine, SQLite for reading and storing data, Media Frameworks to play and record audio and video, the common library Libc, SSL Libraries, Graphic Libraries etc. Additionally in this layer all Java-based Android Libraries for thid-party applications are stored, such as "android.app", "android.os", "android.view", "android.widget", "android.opengl", "android.webkit" etc. Furthermore, this layer also contains the Dalvik Virtual Machine, a Java Virtual machine designed and optimized to run Android developed applications and a set of C/C++ based core libraries.

"Application Framework" is built on top of the "Libraries" layer and contains higher-level services that Android Developers take leverage of to develop their applications. Key services include the "Location Manager", "Activity Manager", "Content Providers", "Resource Manager", "Notifications Manager" and "View System".

The last layer is the "Applications" layer, which contain all developed applications. These include preinstalled applications like the Dialer, SMS, Camera, Clock, Alarm etc. and third-party applications that can be downloaded by an App Store or installed externally.

RESEARCH PAPER

Parts of this Thesis, “Development of a Web portal for authoring and deploying position-based games using abstract domain specific programming language structures” have been accepted in SMAP2020[22]

CHAPTER 2: GeoMakeIt!

(Concepts 2.1)

(Introduction 2.1.a)

GeoMakeIt! Is a platform for developing Geo-based applications, typically games, without prior knowledge of Computer Software Programming, Game Development, use of Game Engines or developing in the Android ecosystem. To achieve that, GeoMakeIt! provides an extendable API were Android Developers, Game Developers and Programmers can create modular and reusable plugins that can be used by a wider, non-programmer, audience. These plugins can be implemented into multiple games and can range from mildly to extensively configurable. As such, a single plugin can have different graphical design, contents, actions and results depending on the implementation chosen by the end-user. Typical examples of plugins are:

- Statistics (Such as Score, Health, Energy etc.)
- Mechanics (Such as PvP, PvE Combat, Quests, Quizzes etc.)
- UI Elements (Such as Alert Dialogs, Popup Messages etc.)
- Map Elements (Such as Circular and Polygon Zones, Markers etc.)

With a combination of well configured plugins, a game developed by GeoMakeIt! can provide unique aesthetics and gameplay.

(Audience 2.1.b)

The target audience for this platform are people with little to no prior experience with the Android Ecosystem, Software Development and Game Engines. Expected end-users would be kids, teachers or people exploring simple ways to prototype and release a game. Subsequently, another use-case would be quick prototyping of geo-applications and thus a secondary target audience could be Android developers for prototyping game ideas.

(User roles 2.1.c)

We consider that there could be 3 available user roles in the platform which are as follows:

1. **Creators (aka Game Developers or end-users):** People with little to no prior experience of coding or the Android eco-system, looking to prototype or release a Geo-based Android game.
2. **Plugin Developers:** People with prior experience to both coding and the Android eco-system, looking to create modular, reusable plugins for the community of GeoMakeIt! and to improve the base API of GeoMakeIt!.
3. **Players:** Users that download and play the released game.

Users can obtain multiple roles and are not necessary entitled to one. For example, a Creator can choose to additionally become a Plugin Developer and create plugins for

GeoMakeIt!. A Plugin Developer is usually also a Creator since he must test his plugins by creating GeoMakeIt! games, while both a Creator and a Plugin Developer become Players when testing their games or playing other people's games.

(Platform: GeoMakeIt! Studio 2.1.d)

GeoMakeIt! serves as a tool to quickly create prototypes or fully-fledged games, depending on the plugin implementation choices, the amount of the customization and configuration efforts of the end-user. To achieve that we created a platform, called GeoMakeIt! Studio that each user can use depending on their role.

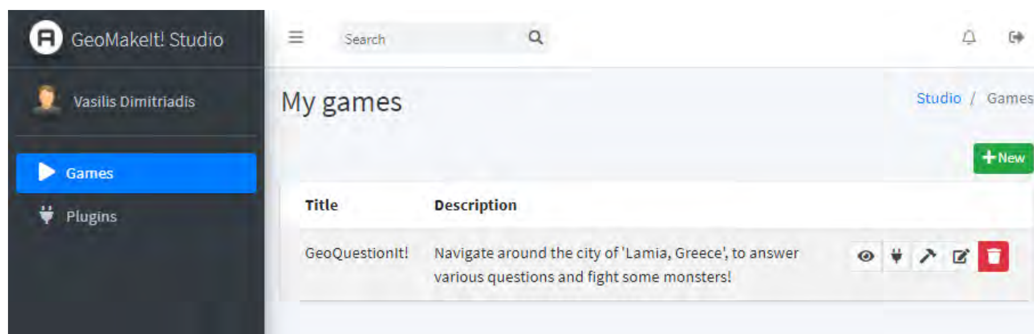

Figure 2.1 GeoMakeIt! Studio Games page

Creators

The GeoMakeIt! Studio for Creators is optimized for Game Development. A Creator can very fast prototype and release a game. He can create a blank project, select the plugins that he desires to be included in the game and configure them. Plugins, including their documentation and available version, can quickly be found from inside GeoMakeIt! Studio.

Plugin Developers

The GeoMakeIt! Studio for Plugin Developers is optimized for Plugin Development. Plugin Developers can register and upload their plugins in the platform, while they develop them using the Android Studio. GeoMakeIt! Studio also includes information such as documentation, tutorials and example sources of basic plugins. This way Plugin Developers can understand how to create, implement and publish their plugins, stay up to date about changes and new features, while they can also easily manage their published plugins.

Players

Players do not have any specific environment in the platform, but rather the actual game itself. The only environment a user could encounter is the download page of the APK which contains information such as the title, description and statistics about the game, as well as the download link of the game for Android. This page is described in

Chapter 9 “Future Planning”, as this environment could be considered a cosmetic to GeoMakeIt! and it is not directly associated with this thesis.

(API: GeoMakeIt! API 2.1.e)

GeoMakeIt! API is the core of any GeoMakeIt! developed game. It stands side by side with the Android API and takes part of in many processes, but its main features can be highlighted as the following:

- Plugin Management (Enabling/disabling plugins, managing dependencies).
- Database Management (Providing the ability to communicate with the games database).
- Core Android Features (Providing information about core features, such as current active activities).
- Command Management (Providing the ability for a Creator to execute plugin features).
- Placeholder Management (Providing the ability for a Creator to grab and reuse and share plugin information between plugins, such as player username, score, health, statistics etc.).
- Permission Management (Requesting permissions and executing actions if no required permissions are given).
- User Authentication (Using Googles API).
- User Location Information.
- Predefined Models (Alert Dialogs, Markers, Quests and Zones).
- Predefined User Interfaces (ex. Login/Register using Googles API).

Each GeoMakeIt! Plugin includes an instance of GeoMakeIt! API to serve two purposes; attach to any GeoMakeIt! Game and ensure plugin compliance.

(Technologies 2.2)

(Application development platforms 2.2.a)

The GeoMakeIt! platform consists of two core elements; The GeoMakeIt! API, for developing plugins to be used in games, and the GeoMakeIt! Studio, a web studio for creating, configuring, publishing and managing GeoMakeIt! Plugins and Games.

GeoMakeIt! API

GeoMakeIt! API is developed using the Kotlin Language and the Android Studio 3.61 as its development platform. Correspondingly, any GeoMakeIt! Plugin is recommended to use the same tools to keep consistency. It is possible to develop a GeoMakeIt! Plugin using the Java Language and any other compatible development platform, ex. Eclipse, but that is not covered by any Chapter in this thesis.

More details about the specific design of GeoMakeIt! API can be found on Chapter 3.

GeoMakeIt! Studio

GeoMakeIt! Studio is developed primarily with PHP, using the framework Laravel 7.0[23]. Laravel is simple and easy to learn, favored by the web-development community and chosen since there was prior experience with its library. GeoMakeIt! Studio could be built with a different framework, but Laravel was favored by this project since it has some promising features and is a framework widely known. This can ensure that GeoMakeIt!, as an open-source project, will be improved in time both by us, the creators, as well as third-party contributors.

More information about the GeoMakeIt! Studio framework choices are described below.

(GeoMakeIt! Studio Language and Framework choices 2.2.b)

Currently, there are 3 common ways of developing an application using a Web Framework. These are: Laravel (PHP), Django[24] (Python), and Node[25] (JavaScript). Obviously, these are not the only options, but these are the ones we're going to choose from since they are more suited for out project.

NodeJS (and other common JavaScript Frameworks)

NodeJS has recently been acquiring a lot more space into the Web Development industry since it's an extremely fast and economic way to create a Web Application. NodeJS itself should not be considered a framework but rather a backend JavaScript server. It supports most of the features our project requires, and it stands out as a possible candidate by its unique built design. It has been built to work on a single thread. That means there is no multithreading and its learning curve is considered higher than other options since there has been no prior experience with such design. With all things considered, NodeJS would be a tradeoff of speed vs learning curve.[25]

Reasons to choose NodeJS

- Built on a single thread → No more multithreading.
- Event-driven, Asynchronous IO APIs.
- Extended support in form of libraries.
- Performance exceeding every other option.
- Works great for building APIs.
- Excellent package manager.
- Easy to handle concurrent requests.
- Great for cross-platform applications.

Reasons to avoid NodeJS

- Asynchronous programming is harder to understand, and it is difficult to deal with early.
- Learning curve higher than other suggestions.

- Weak for CPU-intensive applications.
- Callbacks lead to tons of nested callbacks.

Similar design choices have been made for other common JavaScript frameworks, which typically tend to share the pros and cons of NodeJS. Other JavaScript frameworks that were considered were Angular, React and Vue.

Even though NodeJS is a good solution for our project and is perfect for building API's and real-time applications, which is considered a requirement in the Geo-Location based games, it was not chosen due to the time constrains at the time of writing. This choice though isn't definitive since our final framework choice can incorporate any JavaScript framework, typically Vue, and work together, which could potentially mean leveraging NodeJS's, ReactJS's or Vue's beneficial features.

Django: The web framework for perfectionists with deadlines.

Django is a Python-based Web Framework using the MVT pattern design. Django is at the loved by many developers and companies such as Instagram, Mozilla, Bitbucket and Pinterest. Django uniquely handles other technologies providing seamless integration with third-party solutions, reducing development time, costs and provides a lot of security features. What set it aside though is its long-lasted support by major companies and the Django Software Foundation, proving stability and security features in the long run.

Reasons to choose Django [26][27]

- Security → Django doesn't easily allow security breaches through a long list of checks and constant notifications to the application developer.
- Seamless collaboration with relational databases.
- Seamless integration with other solutions → Applications developed faster.
- Can easily support PostgreSQL and Geographical Data.
- Easy learning curve.
- Highly scalable out of the box.
- Hugely supported by community and companies.
- Detailed and full documentation.
- Better for CPU-Intensive work.

Reasons to avoid Django [26][27]

- Monolithic → The whole framework is a single-tiered application.
- Bad for small-scale applications.
- Even with its easy learning curve, it still requires good knowledge of python.
- Doesn't support REST API's out of the box. It is provided as a third-party solution, but the third-party solution is a powerful one.
- Potentially could be more expensive to run (Requires custom web-services such as PythonAnywhere, or a custom VPS/Dedicated server).

Python is a language that we had some previous encounters with, as well as the Django Framework, and as such this framework could be a more favorable solution. Django can handle a lot of CPU-intensive applications, such as building and managing a Game, it can scale both vertically and horizontally and it is secure. Django is also suited for larger scale applications, which is what GeoMakeIt! can be categorized as when completed but is also a more expensive and time-consuming option condemning early development stages. Finally, it requires expertise on some parts of the Python Language and some very specific design choices.

Laravel: The PHP Framework for Web Artisans

Laravel is a PHP Framework that has lately increased in popularity supposedly due to the PHP 7 improvements. Designed for the Artisans Web-Developers in mind and with the latest trendy design choices, Laravel is an MVC Framework feature rich and easy to develop on. While prior PHP knowledge is required, the learning curve of this framework seems to be a lot easier than other competitors. It is usually used for Content Management Systems or powerful API's and it is supported by its 2 creators, Taylor and Otwell, as well as the open-source community.

Even though the primary design of Laravel is to create powerful Content Management Systems, that's not its only intent. As such we can harvest some of the rich-full features on our project.

Reasons to choose Laravel [26][27]

- Considered the ideal framework for PHP developers.
- Simple and Easy to learn.
- Detailed Documentation.
- Based on the MVC pattern, it is easy to understand, and it eliminates the need of writing basic HTML codes.
- Supported by ORM → Helps in abstraction and automation.
- Blade Template Engine → Easy migration from HTML/PHP templates.
- Easy testing and Automation features, as well as dependency injection.
- Supports React and other JS frameworks out of the box.
- Powerful migration system.
- Favored by more developers (More GitHub stars than Django).

Reasons to avoid Laravel [26][27]

- No in-built tools and requires third party integration for custom website development.
- Laravel is slow and good knowledge of PHP is usually required to make it faster, but this has changed after the release of PHP 7.0 which supposedly can be up to 3x faster than Python.

Laravel doesn't necessary match all the requirements to complete GeoMakeIt! Studio and is missing some features that could be helpful for future integrations, but since there have been prior experiences with Laravel and alongside with the PHP 7.0

claims of 3x speeds than Python, Laravel does become a true competitor to the other Frameworks.

An advantage of choosing Laravel though is the easy integration with JavaScript Frameworks such as ReactJS and Vue, which could allow easier integration of these and their real-time capabilities on the future.

Framework Comparison

To begin, NodeJS as well as other JavaScript Frameworks are excluded from the Framework selection pool due to the inexperience with the framework and the language. Although the real-time capabilities and the increasingly larger open-source community could provide a lot of support for the GeoMakeIt! project, due to the time constraints the choice have been made to work with a language that has been previously encountered and feel more comfortable with. As such, the following comparison will be between Django and Laravel.

1. **`Scalability`** – Django has scalability out of the box, while Laravel has a set of features that can keep a website one step ahead in the market. Per se, Laravel does provide scalability solutions but at an extra cost with their services such as Laravel Forge, Laravel Vapor and Laravel Envoyer.
2. **`Architecture`** – Django uses the MVT pattern, while Laravel uses the MVC.
3. **`Security`** – Django is considered the most security-proof framework and does its job outstandingly well. Laravel is secure, but still is inferior when considering its opponent; Django.
4. **`Customizations`** – Django is a complex framework and takes time and good architecture when someone needs to customize it. Laravel, on the contrary, needs third-party tools to personalize the website.

Community voting and Trends

Lastly, we got to consider the development community's opinions and trends. As such we are going to check how does the community votes on Django and Laravel, based on slant.co[28]. Django is voted as a better backend web framework and it is recommended as it can develop a prototype faster, provides customization, contains great documentation, contains an admin panel out of the box and has simple database management. Laravel is voted as the better web-framework for designing REST API and it is recommended as easier to start with. It comes with an integrated CLI, there is a lot of great documentation available, it gives freedom on the choices and has a powerful templating system.

As for current trends, according to builtwith.com, Django has around 7.500 live websites while Laravel dominates with 157.000 live websites at the time of writing. Django's popularity seems to have fallen quite dramatically according to statistics and online threads.

(Maps API Choices 2.2.c)

There is a plethora of solutions when it comes to mapping APIs, but there are some clear winners when it comes to implementing them into the Android API and their ease of use. First, let's discuss about some of the most known Map APIs.

Google Maps

Google Maps[29] is the most common choice for maps when used under the Android Operating System and it is considered one of the best documented Map APIs.

According to Google[30], "With the Maps SDK for Android, you can add maps based on Google Maps data to your application. The API automatically handles access to Google Maps servers, data downloading, map display, and response to map gestures. You can also use API calls to add markers, polygons, and overlays to a basic map, and to change the user's view of a particular map area. These objects provide additional information for map locations and allow user interaction with the map. The API allows you to add these graphics to a map:

- Icons anchored to specific positions on the map (`Markers`).
- Sets of line segments (`Polylines`).
- Enclosed segments (`Polygons`).
- Bitmap graphics anchored to specific positions on the map (`Ground Overlays`).
- Sets of images which are displayed on top of the base map tiles (`Tile Overlays`)."

But Google Maps comes with a price when it comes to GeoMakeIt!. Depending on the design choices of the project, there could be significant price-differences both in the developing phase as well as the GeoMakeIt!'s public launch phase.

Pricing

Google Maps provides a 200$ coupon each month to be used with its Map Services. As such, the first 200$ spent on the service is considered as a "free tier". If the API calls exceed the 200$ free tier coupon, Google Maps bills per 1000 requests on each API call. More specifically:

- Static Maps and Dynamic Maps are free if used on Mobile, but if used under any web-based application they cost $2 and $7 per 1K requests accordingly.
- For the Places API, prices for Geocoding, Autocompletion, Place Details, Find place, Current place, Geolocation and Time Zone cost from $5-$30 per 1K requests, where the "Current Place" API which is responsible for discovering what the place at a reported location is, and is vital for our project costs 30$ per 1K requests.
- Road Speed Limits cost $20 per 1K elements.
- Static Street View and Dynamic Street View (360 panoramic view) cost $7 and $14 per 1K requests accordingly.
- For the Routing API, prices for Directions, Distance Matrix, Roads (Route to travel and Nearest Road) range from $5 to $10 per 1K requests/elements depending on the details required.

With that being said, the last 3 APIs, Road Speed Limits, Static Street View and Dynamic Street View as well as the Routing API, will not be used at all in GeoMakeIt! API, and as such will be excluded from the pricing.

Considering the aforementioned pricing, Google Maps could be considered an expensive choice if the GeoMakeIt! API was chosen to work cross-platforms or generally if it was not only Android OS Based. When it comes to the API required by our project, Google Maps provides the best implementation. It is also considered the most wide-known Maps API which makes it easy for plugins to use it. An extra bonus of its high-usage rate could be that the information returned by the Google's API could also be the most accurate ones. Finally, it should be noted that there can be discounts for 100K+ requests, so Games requiring heavy Maps API calling could benefit from that.

According to the thread "5 Most Powerful Alternatives to Google Maps API" [31], other choices could be Open Layers, TomTom, MapBox and HERE. The article also mentions a service called Mapfit, but as of 2020 it seems that this service has been discontinued. When considering the other options, depending on each application, there could be a good reason to select each one of these.

MapBox – Best for customization

When it comes to features and customization, MapBox is considered one of the top alternative choices to Google Maps. It comes with Tiled Maps, Vector Maps, Markers, Location Search, Route Finding out of the box. The most important feature of MapBox is its ability to create customizable maps.

It offers a stricter pricing considering the alternatives, with the first 50.000 requests per month free, while each subsequent 1.000 requests cost $0.50.

HERE – Map Visualization and Coverage

HERE Maps API provides similar features with other Map APIs but stands out when it comes to Map Visualization and Coverage. Some of its features include Tiled Maps, Vector Maps, Markers, Location Search, Route Finding, Custom Maps, Map Visualizations and Public Transits.

HERE also provides the most flexible free tier, with 250.000 free monthly requests to its API, while it is also the most expensive when it comes to each subsequent 1.000 request, priced at 1$ each. It must be noted though that it also comes with a 1.000.000 request package priced at $449, or $0.449 per 1.000 requests, which is near the base price of every competitor.

TomTom – Navigation and Functionality

TomTom provides some unique features such as traffic density and it is primarily focused on Map Navigation. It also provides common features such as Tiled Maps, Vectored Maps, Markers, Location Search and Route Finding, but due to its primary focus on Navigation it doesn't necessary fit our criteria.

When it comes to pricing, TomTom has a generous free tier of 2.500 daily requests, or 75.000 per month, and a cost of $0.42-$0.50 for each subsequent 1.000 requests.

Open Layers – DIY and Affordable

Open Layers is widely known as the Free Maps API and it is a common choice for anyone who's looking for a free and simple Maps API. When it comes to features it is considered of the simplest API's, since it comes standalone with Tiled Maps, Vector Maps and Markers only. Unlike though other options, it provides third party open source extensions, Open Layers can be a strong competitor since it has a big community and a lot of information about it are available online. Due to its nature of DIY, it would take a considerable amount of time to correctly implement it in the Android Operating System and implementing some of its core and extra functionality.

When considering its price point, of course it is marked the best, since it is free with unlimited requests.

Final thoughts

When choosing between the Map APIs, we can some everything up to these basic choices:

- Google Maps API if we chose to go with the standard way for Android Native Application Development.
- MapBox API if we chose to use all 250.000 requests monthly and ignore the fact that each subsequent 1.000 requests cost $1 and that it is considered the second most expensive API after Google Maps.
- TomTom if we choose to harvest the already in place powerful Navigation and Functionality that is provided by the service.
- Open Layers if we choose to go the DIY way and create some core-functionality from the start or by using free libraries. There should be also noted that we did not immediately find any Android specific SDK/Implementation for Open Layers, but rather numerous HTML and JavaScript examples in the Open Layers Examples website[32]. As such, if this approach is chosen then it would be required by GeoMakeIt! to implement and generate its own HTML and JavaScript backend Map Services.

An extra API that wasn't mentioned above was Open-Street Map API. There are some Android implementations for Open Street Maps API, such as OSMDroid (Open Street Maps Android), but since Open Street Maps is better for providing raw data, it shares similar characteristics with Open Layers and it would not be chosen since it would require even more customization and time efforts.

Our choice for GeoMakeIt! is to implement the basic Google Maps API using directly the Android Native SDK. By that, we are leveraging all the power of the basic Google Maps API, without extending to the paid version. This way we can do all the necessary testing for GeoMakeIt! for simple applications and leave room for future upgrades. It should be also noted that a single choice is never always the option and to provide better future results and functionality for a Map API it should be considered that GeoMakeIt!

could provide a Maps API wrapper leveraging the power of all common Maps API while letting the end-user to make the final choice depending on his application.

(Database Choices 2.2.d)

The initial choice for the database software was MySQL[33]; but as the GeoMakeIt! API matured another option would become more applicable; Google's Firebase and Firestore API[34].

This choice was made since Google's Firebase and Firestore is already widely supported in the Android Development community and it provides a generous free tier package, but unlike MySQL, it is not completely free. This choice will put an extra burden to the Game Creator, but it will provide the tools to bootstrap GeoMakeIt! API and try out different architectures before finalizing the design. Due to the nature and features of Google's Firebase, it will also speed up development progress since some packages come already out of the box, such as Authentication and Crash analytics.

We will list the pros and cons that about the two different database designs.

MySQL

MySQL is an open source, free, relational database management system, widely known to the development community and dominated the database market as one of the primary free choices for a relational management system.

Pros

- Open source and free.
- Can create server-side triggers.
- Relational data and safe transactions.
- Cheaper to maintain since it is more widely used as a general-purpose database.

Cons

- Not as flexible as Firebase.
- Requires for our specific needs to create complex Database Management and relationships between the GeoMakeIt! API and GeoMakeIt! Studio.

SQL and MySQL are older and more established in the programming community but in this specific application heavy modifications would be required to establish a secure, modular and scalable design between GeoMakeIt! Studio, GeoMakeIt! API and the GeoMakeIt! Plugins. To understand further the problem, a Plugin Developer should have the freedom to create, store, modify data in his own personal tables in the Game's Database. There would also need to be a secure way to communicate with other plugins and share its own data with other plugins data. Since these are not included directly in SQL or MySQL, an extra API would need to be created that would manage such transactions safely.

As result, we decided to experiment with Google's Firebase and Firestore.

Google's Firebase and Firestore

Google's Firebase is a cloud-hosted NoSQL Database with a very flexible way to store data using JSON files. It's Schema-less, meaning there are no tables, columns or rows like in MySQL, but rather a key-value pair. Google's Firebase was designed with real-time data in mind, which does meet the GeoMakeIt!'s criteria quite well. It is also designed by Google and as result, it is well implemented inside the core of Android, while it also is surrounded by a big community of developers and support. Unlike though MySQL, Google's Firebase does come with some extra tools to increase productivity and doesn't require us to re-invent the wheel by implementing a lot of useful features such as Cloud Storage and Authentication.

Pros

- Cloud—hosted relative document storage.
- Fast and suitable for real-time applications.
- Widely used by the Android community.
- Schema-free, meaning a 'Plugin Developer' can append, update data or modify the structure of their documents and data easier.
- Can be live monitored for changes.

Cons

- No foreign keys, which means extra work for the developer to search data through different places of a database.
- Limiting functionality with server-side scripts.
- Commercial, so it can be costly if there are a lot of transactions. Our implementation will contain a lot of transactions so there needs to be future optimizations to lower usage.
- Not immediate, but eventual consistency if the client is offline.
- Supports offline data storage and synchronization when internet connection becomes available again.

Google's Firebase is going to be a costly solution for GeoMakeIt!'s implementation, but it should be noted that it is used as a temporary implementation for prototyping. The best solution here would be to create a database wrapper like the one described in Map choices: Final thoughts, to give the end-user more choices when selecting a database provider, such as Google's Firebase. This way the user can select a provider based on his needs.

Final thoughts

When it comes to the Database choices, it is never an easy choice and those choices will affect the design of the final product. For GeoMakeIt! we've decided the following.

GeoMakeIt! Studio will be designed using an SQL based design, MySQL. This choice is perfect for GeoMakeIt! Studio since its data will be predictable and Plugin Developers will not have access to the GeoMakeIt!'s Studio directly. Other choices such as PostgreSQL have been considered, but where scrapped since their features are not

necessary at this stage of development and if necessary, migration in this case would be easier.

GeoMakeIt!'s API and any plugin developed using it will use a NoSQL Document based designed. Currently that is being done using Google's Firebase and Firestore, but optimally, that would be done with a wrapper to allow multiple NoSQL provider implementations. The main reasoning behind choosing NoSQL document-based design was its ability to store arbitrary data, schema-less and do that well for real-time applications.

CHAPTER 3: GeoMakeIt! API

(Introduction to the API 3.1)

(Overview 3.1.a)

GeoMakeIt! API is the core of GeoMakeIt!. It enables the development of GeoMakeIt! Plugins using the Android Studio API and it links those plugins with GeoMakeIt! Studio. On this chapter we'll understand how the GeoMakeIt! API functions, how it works alongside with Android Studio and how its inner components are designed. Then, after we also introduce GeoMakeIt! Studio on Chapter 4, we will discuss about how to use both to create our own GeoMakeIt! Plugin.

The GeoMakeIt! API provides implementations for managing some of the Android API's features hybrid bottom-up and top-to-bottom, instead of the classic top-to-bottom approach. This way instead of the classic approach of the main app managing all the libraries and their executions, GeoMakeIt! API allows the plugins to make some critical decisions about how the GeoMakeIt! Game would run, while allowing the end-user to contribute to that run-time flow with simple commands.

To achieve that, GeoMakeIt! API handles tasks such:

- Managing Android Activities.
- Managing Android Services.
- Managing Android Permissions.
- Managing Database and Storage.
- Managing run-time exceptions.
- Provides implementations for GeoMakeIt! Commands.
- Handles plugin registration and execution.
- Provides basic UI for common features.
- Provides common utilities for handling plugin configuration.
- Manages placeholders for player feedback such as player name, score etc.
- Provides utilities for managing Users, HTTP Requests and JSON files.

It should be noted that GeoMakeIt! is under continuous development so its core features are updated and improved and as such more features will be available further down the road. The features described above though, even with their basic, primitive implementation, serve to test and verify the feasibility of the API.

(API workflow 3.1.b)

GeoMakeIt! API serves as the intermediary between the Android API and the GeoMakeIt! Game. It handles common requests by libraries to the Android API, provides simple implementations for common tasks such as managing configuration and the database.

The image shows possible ways of communication between the vital components of a GeoMakeIt! Game. These components are the `Android Application`, `Android API`, `GeoMakeIt! API` and a series of `GeoMakeIt! plugins`.

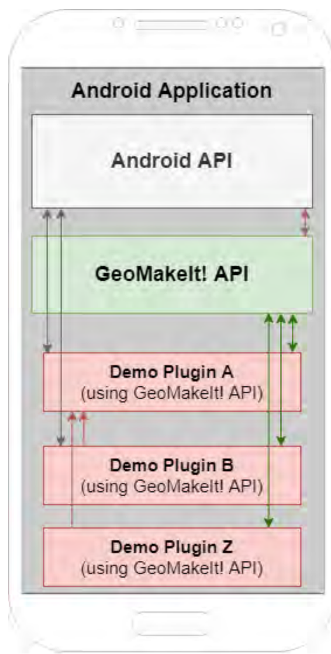

Figure 3.1 GeoMakeIt! Communications and Dependencies

The Android Application serves just as the front-face of the application and provides very basic functionality for showing the application's pre-loading screen "Made by GeoMakeIt!", authenticating the user using Google's Firebase Authentication, and initializing the `GeoMakeIt! API`. Hidden under the Android Application are the other vital components.

The `GeoMakeIt! API` starts loading the application's implemented plugins, `Plugin A`, `Plugin B` and `Plugin Z`. The plugin loading order is calculated based on each plugin's dependencies, starting with the ones requiring no other plugins and finishing up with the ones requiring most of them. In this case, `Plugin A` requires `Plugin B` and `Plugin Z`, and as such will be loaded last.

When all the plugins are loaded, the application can start, and the plugins can execute their functions. The plugins can start activities, register commands and placeholders or access classic Android functions with no limitations, and as such can communicate with the Android API directly. When though actions are required to be seen in the app, usually information pass first through the `GeoMakeIt! API` which then forwards them to the app. Furthermore, plugins can communicate between each other, after they verify their existence in the application using the `GeoMakeIt! API`. Finally, the end-user can issue commands to execute plugin functionality that passes from the `GeoMakeIt! API` to the `GeoMakeIt! Plugins` directly.

(API Components 3.2)

(The "App" component 3.2.a)

The `App` component is the first thing that gets is registered by the application and contains vital information about the application's run-time features and data. The `App` component is an extension of the Android's "MultiDex Application" and shares all the available methods. Furthermore, it checks if GeoMakeIt! API was initialized correctly, makes the database and the current activity publicly available to plugins, as well as loads the main map at the start of the application.

A GeoMakeIt! Plugin can call the `App` component to:

- Get the current active activity of the application using `getCurrentActivity()`.
- Get information about the application, such as the current context or application context using `getContext()` and `getApplication()` accordingly.
- Get the main Google Map component by accessing the `map` variable.
- Get information from the database, append or modify data using the `database` variable.

- Start activities without require the current context by calling `startActivity(Intent)`.

(The “Game” component 3.2.b)

The `Game` component contains information about the currently built game. It is a simple way for a plugin to grab the `title`, `description`, `version` and `built type` of the game, `development` or `release`, for statistical information. The methods provided by this component are `getTitle()`, `getDescription()`, `getVersion()`, `getBuildType()` correspondingly.

(The “Plugins” package 3.2.c)

The `Plugins` package contains all the necessary tools for GeoMakeIt! API and Plugin Developers to create, instantiate and manage their plugins and dependencies. It contains 3 important components, the `Plugin` component, the `Plugin Manager` component and the `Plugin Logger` component.

The `Plugin` component is the core of each plugin, managing basic functions such as the actions to be taken as soon as the plugin gets enabled or disabled, and registering each plugin’s appropriate sub-components.

The `Plugin Manager` manages all those plugins. It is one of the first classes called by the GeoMakeIt! API and its main purposes are to register, enable and keep track of plugins installed by a GeoMakeIt! Game.

Finally, the `Plugin Logger` component handles all the logging and debugging information output by plugins, by forwarding their messages to the Android’s Logcat, the GeoMakeIt! Studio and any other necessary location.

Plugin component

Every plugin extends the `Plugin` component. This component contains information about the current status of the plugin such as if it is initialized, enabled or disabled. It also contains information about the plugin such as its current name, description, version, hard and soft dependencies of other plugins. Furthermore, it contains 2 important methods of each plugin, `onEnable()` and `onDisable()`, which declare the actions taken by the plugin as soon as it is enabled or disabled. Finally, it contains shortcuts for the plugin to access its logger using the `logger` variable, the current application context with the `context` variable and its configurations using `getConfig()` and `getConfig(Path)`.

Let’s talk a bit more about the `onEnable()` and `onDisable()` methods. As soon as a GeoMakeIt! Plugin is detected by the GeoMakeIt! API and all its dependencies are met, GeoMakeIt! API calls the `onEnable()` method of the plugin. Usually, the `onEnable()` method is called once and when it is called a plugin must initialize primary components such as registering commands and placeholders, loading models

from json files and storing them to the database if necessary, starting services and loading configurations. Similarly, when the plugin is no longer needed, the `onDisable()` will be called to handle all the unloading.

It of vital importance to understand the `identifier` variable of the plugin component. The `identifier` is a **unique lowercase alphanumeric string that identifies a plugin** in GeoMakeIt! platform. The `identifier` is chosen by the Plugin Developer and must be registered inside the GeoMakeIt! Studio before a plugin can be uploaded, shared or used by a GeoMakeIt! Game. Usually the `identifier` is self-explainable depending on the plugin, so if for example a Plugin Developer was creating a scoring system plugin, the identifier could be one of these: *score*, *scoring_system*, *epic_score*, *pro_score* etc. This way any plugin can be distinguished from others and common components such as commands and placeholders that depend on it. For example, a placeholder for the player's score with the plugin *epic_score* could look like this: {`epic_score::player_score`}.

Other methods provided by the `Plugin` component is the `getConfig()`and the `logger` variable. Logger can log plugin messages on the Android console for the developer, for other plugins to monitor or for debugging purposes. To get the plugin's configuration the method `getConfig()` or `getConfig(Path)` is used, which returns the configuration requested. Further information about these methods will be given shortly.

Plugin Manager component

The `Plugin Manager` component handles the registration, enabling and disabling of plugins that will be included by the GeoMakeIt! Game. When the GeoMakeIt! API is initialized, it calls the Plugin Manager's `registerPlugins()` method to register all the plugins included in the `plugins.json` file.

The plugins.json file

The `plugins.json` file is a JSON file generated by GeoMakeIt! Studio at the compilation time of the Game. This file contains all the necessary information to enable a plugin by finding its package and main class that *extends* the `Plugin` Component.

Here is an example of the `plugins.json` file:

```
[
    {
        "package": "com.example.pluginA",
        "main": "Main"
    },
    {
        "package": "com.example.pluginB",
        "main": "MyMain"
    },
                .....
]
```

As soon as the registration of the plugins is complete, all the plugins are enabled based on their hard dependencies.

Lastly the `Plugin Manager` component can be used to check if a specific plugin is registered, enabled, disabled or even to enable/disable other plugins by a GeoMakeIt! Plugin. This way, a GeoMakeIt! Plugin can check if a soft-dependent plugin is available or to disable itself if it finds that it conflicts with other plugins.

Plugin Logger component

The `Plugin Logger` allows all plugins to uniformly log messages in the `Android Logcat`. This way, any GeoMakeIt! Plugin message can be easily discovered by both the plugin's author as well as any other Plugin Developer when using dependencies.

Additionally, the `Plugin Logger` will be used to log messages to the creator's console inside GeoMakeIt! Studio or even store it in Google's Analytics extension.

The `Plugin Logger` component can be called directly only by GeoMakeIt! API, while any plugin developed with the GeoMakeIt! API has access to the `logger` through the Plugin Component's `logger` variable. Another way to access the `Plugin Logger` would be initializing its class and passing as arguments a plugin object.

When calling the Plugin Logger component, a developer can log a message using one of the following priorities: `VERBOSE`, `DEBUG`, `INFO`, `WARN`, `ERROR` and `ASSERT`. Additionally, inside the GeoMakeIt! Studio the end-user can choose the log level for debug purposes, thus some messages can be visible or hidden depending on the end-user's choices. When it comes to developing a plugin, the default option is to show all messages, this way a GeoMakeIt! Plugin Developer has full knowledge of all the current plugins states.

(The "Activities" package 3.2.d)

The `Activities` package contains all necessary components for handling Android activities by the GeoMakeIt! API. It consists of two components, `JActivity` and `Activity Registry`, while there will be future additions to cover a wide range of other Activity classes.

`JActivity` is an extension of the Android's Activity class to provide extra implementations for GeoMakeIt! API, while `ActivityRegistry` registers activities to be used by GeoMakeIt! Commands and to uniquely identify them in the GeoMakeIt! API. More information about GeoMakeIt! Commands can be found in the upcoming section, Commands package.

JActivity component

`JActivity` *extends* the Android's `AppCompactActivity` and provides extra capabilities to an activity of a GeoMakeIt! Plugin. When developing a GeoMakeIt! Plugin, instead of the normal `AppCompactActivity` it is recommend switching it with a `JActivity` class. This way, implementations such as `Services` and `Commands` can be run when an activity is open.

The primary function of the `JActivity` class is to execute `Services` described in the upcoming section `Services` package. A quick description of services would be *actions that need to be executed in the run-time of an activity*; Services that need to use the `onPause()`, `onResume()` and `onDestroy()` methods of the `AppCompactActivity` class. When the Activity is paused, resumed or destroyed these methods are called, making the services aware of the state of the current activity.

Additionally, the `JActivity` contains functions of requesting permissions from the player and executing GeoMakeIt! commands issued by other plugins or the Game Creator.

The `requestPermission(Context, Permissions, Request Code, Execute Function, Fallback Function)` method is a quick method to uniformly and easily request Android permissions by a plugin using an `Activity`. It takes as arguments the `current context` of the application, a *`list of permissions`* to be requested by the user, a *`Request Code`* to handle the activity returned, an *`execute function`* that will run if all the permissions are given by the user and a *`fallback function`* if they are not. If no *`fallback function`* is provided, a predefined one is executed that prompts the user informing him why the requested permission is necessary and why he should grant that permission. If the permission is still not granted though, the plugin must manage the rest, such as denying the player a specific functionality or view of the plugin. These permission requests are handled by the `onActivityResult()` method which is provided by the Android's `AppCompactActivity` class. If more customized functionality is required by the Plugin Developer, overriding this method is recommended.

Lastly, the `executeCommandFromView(View)` function is a necessary method for allowing the Creator to execute specific commands from the activity's view. A good example would be a fab button that signs out the user when pressed.

ActivityRegistry component

The `Activity Registry` component allows plugins to register their activities with a unique name, thus, allowing the Game Creator to call an activity using the command "`geomakeit::activity`" which will be discussed later in the `Commands` package section.

This command enables the Game Creator to start a registered activity with a set of parameters. An example of this in action would be issuing a command to start an activity called *QuizActivity* of a plugin called `GeoQuizzes`, with parameters which quiz to show. Such a command could look like this: `geomakeit::activity geoquizzes::`*`quizactivity`* `quiz=`*`beginner_quiz`*.

Let's see more in action how the `Activity Registry` works. The `Activity Registry` is a *data class* that contains a private constructor with 3 arguments, the `plugin` that is registering an activity, a unique per plugin `identifier for the activity` and `the activity` which must be or extend a `JActivity` class.

It is also important to understand that **<u>the unique activity identifier is "unique per plugin"</u>**, meaning two plugins, *pluginA* and *pluginB* <u>can both register an identifier</u> `my_activity` since these two plugins are uniquely identifiable by their own identifier, `plugina` and `pluginb` respectively. Each plugin though individually cannot share the same identifier for two activities.

`ActivityRegistry` can be called to check if a registered activity identifier exists, to get an activity from its registered identifier or to register a specific activity. Some of the methods of the `ActivityRegistry` can be seen below:

The methods `isRegistered(Class<JActivity>)` and `isRegistered(Plugin, Activity Identifier)` can be called to check if an activity or an identifier is registered. It can be either called by passing as arguments the `plugin` that an activity belongs to and the `identifier`, or the activity itself.

The method `register(Plugin, Activity Identifier, Class<JActivity>)` can be called to register an activity, as long as it is not already registered. To register an activity the Plugin Developer needs to pass as arguments the `plugin` itself, a `unique identifier` and `the activity` to be registered under that identifier. It should be noted that an activity can be registered only once. It cannot be re-registered under an alias.

The method `get(Plugin Identifier, Activity Identifier)` returns a registered activity if it exists. To get a registered activity a Plugin Developer needs to pass the `third party's plugin` as the first argument and the `activity's unique identifier` as second. If there is a registered activity under that name it returns the `class of that activity`, otherwise it returns `null`.

Finally the method `getAll(Plugin Instance)` works similarly to the `get()` function, but rather takes a `plugin instance` or `plugin identifier` and returns a list of all registered activities under that single plugin.

(The "Service" component 3.2.e)

The Service component is created to run alongside any JActivity and JActivity inherited class. It allows a developer to run services between activities without restrictions, such as global timers, location tracking etc. It consists of 3 methods to register, unregister and check if the service is registered. These methods are register(), unregister() and isRegistered() respectively. Furthermore it contains 4 methods that can be overridden and are necessary to create a Services workflow: onCreate(), onResume(), onPause(), onDestroy(). Each one of these methods are called first on an activity's onCreate(), onResume(), onPause() and onDestroy() function respectively.

This way, a service can actively run between multiple activities. An example of a service in this case would the Location Service which is provided by the GeoMakeIt! API to track the player's position so it can later be used to represent him on a map.

Location Manager Service

The Location Manager Service handles information about the current players location. It *extends* the GeoMakeIt!'s Service component and uses the Android's Fused Location Provider Client, Location request and Location callbacks.

By using the Service component, the Location Manager Service can be enabled as soon as an activity is active and paused when it is not so it can conserve the battery of the user.

When a player is moving, he's position is stored in an interval of 3 seconds, allowing the plugins to know when a player is moving and updating the current player's location. The player's location is monitored between every 1-3s with high accuracy by default, although these settings will be configurable by the Game Creator from the GeoMakeIt! Studio when configuring the Game. Finally, all the changes are monitored using the Android's FusedLocationClient, so all the settings of the Fused Location Client can be configured through this service.

(The "Commands" package 3.2.f)

The Commands package contains all necessary tools to allow GeoMakeIt! Plugins register commands for the Game Creator to use in the GeoMakeIt! Studio, as well for plugins to dispatch them when necessary. It consists of 3 components: The Command component, the Command Manager and the Command Map.

The Command component contains information about how a command looks like, such as its identifier, description and usage message, as well as the actions it executes.

The Command Manager is responsible for dispatching commands for all the plugins to list.

The `Command Map` is a solution to group commands based on their responsibilities, as well as declares the way a command is structured.

Command component

The `Command` component contains information about how a command is structures, like its `identifier`, `description` and `usage message`. Additionally, it declares the actions that must be run once executed.

Let's understand the inner workings of a command. To create a command a Plugin Developer must *extend* the `Command` class, calling its constructor which takes 3 arguments. The `label`, also known as the identifier of a command and optionally its `description` and `usage message`. The last two are provided so another developer using third-party commands can quickly understand their functionality, as well as in the GeoMakeIt! Studio to assist the Game Creator acknowledge information about this command.

The `Command` component contains also an `execute` method which a Plugin Developer *extends* to implement his command functionality. When a `command` is issued and its `label` matches the label of the `command` class, the `execute` method is called passing as a parameter the available `arguments`. For example, if the `pluginA` with a `unique identifier` of `plugina` registers a command with label `my_command` then every time a Game Creator dispatches the command "`plugina::my_command argumentA argumentB argumentC`" the `execute()` method will be called and an arguments list containing 3 arguments in the form of strings: `argumentA`, `argumentB` and `argument` parsed as parameters. The Plugin Developer then returns success or failure in the form of a Boolean, `true` or `false`, which is the forwarded back to the `Command Dispatcher` which we will see later in this section.

Lastly, a `Command` is not registered directly in the plugin, but rather a `Command Map` which allows the grouping of commands into groups of similar actions. As such, there `Command` provides 3 methods for the registration which are `isRegistered()`, `register()` and `unregister()` which respectively check if a command is registered, register the command to a `Command Map`, or unregister a command from a `Command Map`. Additionally, there is a private function that checks if the command can be registered/unregistered, which depends by the command map that requested the change.

Command Manager component

The `Command Manager` component handles the execution of commands and registration of Command Maps. There are 4 methods inside the `Command Manager`: `dispatch()`, `registerExpansion()`, `unregisterExpansion()` and `isExpansionRegistered()`.

The 3 last methods described above are self-explainable, they take a `CommandMap` as a parameter and attempt to register, unregister or verify that it is registered or not.

The `dispatch()` method executes a given command line, ex. "`myplugin::command arg1 arg2`" by broadcasting it to all registered `Command Maps` and awaiting for results. It returns back a Boolean value, `true` or `false`, which is the result of the Command's `execute()` method. This way a Plugin Developer can allow the Game Creator to both execute functions and do some simple Boolean calculations. A simple example would be a plugin that handles creating zones which contains a command that checks if a player has entered a zone. Such a command could look like this: "`zoneplugin::hasVisited zone_a`" which would return `true` or `false` depending on if a player has visited *zone_a* or not. A Plugin Developer could take this one step further and add some more complex logic to the command such as "`zoneplugin::hasVisited zone_a < weeks:3`" which could check if a player has visited the *zone_a* within the last 3 weeks.

Finally, it must be noted that the `dispatch()` method is publicly available and very useful when a Plugin Developer wants to implement a way for the Game Creator to execute commands on specific events. For example, on the previously mentioned zone plugin, there could be events such as `onZoneEnter()`, `onZoneLeave()`. In such cases the Plugin Developer could execute some specific commands given by the Game Creator from a configuration that he has created. Examples of using the dispatch commands are plenty, from the already implemented models inside the GeoMakeIt! Plugin which we will see shortly, up to the example Quiz plugin shown on Chapter 5.

ICommand Map and Simple Command Map components

The `Command Map` serves as a way to group commands and describe the way those commands can be constructed. When it comes to `Command Maps`, there are two components: An interface called `ICommandMap` and a class that *extends* the `ICommandMap` called `SimpleCommandMap`.

The `ICommandMap` contains just instructions about how a `Command Map` should look like, and declares that a `Command Map` must contain the plugin of which the commands belong to, and 5 methods: A method to register a single command, a method to register multiple commands, a method to clear those commands registered, a method to get a command based on the label of the register command and a dispatch method that forwards a command line to the registered commands inside the `Command Map`.

More important though is the `Simple Command Map`, which declares how to structure a command, such as the following: "`plugin_identifier::command_label arg0….argN`". Even though the freedom of creating a `Command Map` is given to the Plugin Developer, doing so is not necessary the best idea. That freedom is only given so a Plugin Developer can extend the basic `Command Map` by adding extra features. If though there is no such desire be the Plugin Developer, it is recommended that he uses the default `SimpleCommandMap` to keep consistency between plugins.

For GeoMakeIt! API commands though, the pattern "`plugin_identifier::command_label arg0….argN`" must not necessary be followed. Any command dispatched to a `SimpleCommandMap` without a `plugin_identifier` is considered as a default GeoMakeIt! Command, so the

GeoMakeIt!'s `identifier` is applied, "`geomakeit`". Meaning, "`geomakeit::signout`" and "`signout`" commands are considered the same.

(The "Placeholder" package 3.2.g)

Placeholders allow Plugin Developers and Game Creators to display GeoMakeIt! API and GeoMakeIt! Plugins information to players easily, and thus allowing to create modular, customizable and personalized messages. Think of the following example. Imagine that you want to greet a player when he first joins a GeoMakeIt! Game. The creator could write "Welcome to my awesome game!". But what if he would like to personalize this message with the player's username? By using the `Placeholder` component, he could write the following: "*Welcome {username} to my awesome game!*", and thus if a player named *AwesomePlayer123* joined the game it would greet him by saying "*Welcome AwesomePlayer213 to my awesome game!*". Now, let's take it a step further and think that there is a plugin keeping scores for the game with a leaderboard. A Game Creator could write the following into an alert: "*Awesome {{username}}! You've reached a new highscore: {{leaderboard::my_highscore}}. The top highscore globally is: {{leaderbpard::global_highscore}}.*"

Placeholders are vital to make a game more enjoyable and personalized.

Placeholder Interface

The `Placeholder Interface` contains a single method, the `onRequest(Placeholder Identifier)` method. This method takes a single argument, the identifier of a placeholder which is of type string and returns a `string value` or `null`.

For a plugin to create a placeholder, they require to create a class ex. `DefaultPlaceHolder` which takes one argument, the `plugin` which the placeholder belongs to. The Plugin Developer then needs to *implement* the `Placeholder Interface`'s method `onRequest()` in a similar way as seen here:

```
class DefaultPlaceholders(plugin: IPlugin) : Placeholder(plugin) {

    override fun onRequest(placeholderId: String): String? {
        when (placeholderId) {
            "username" -> return LocalUser.displayName
            "email" -> return LocalUser.email ?: "N/A"
            "displayname" -> return LocalUser.displayName
        }
        return null
    }
}
```

When a Game Creator or a Plugin Developer wants to personalize a message by using placeholders, the GeoMakeIt! API forwards that message through the `Placeholder Manager` which cycles through the registered placeholders and check if one fits the description. If it does, then they are replaced by the string calculated `onRequest()` method.

For example, in the image above, we can see that the `username` placeholder returns the player's *display name*. Since those placeholders are defined in GeoMakeIt! API, it means that every time we call a personalizable message like "*Hello there {{geomakeit::username}}*" it is replaced by the string provided, in this case something like this: "*Hello there AwesomeUser123*".

Placeholder Class

The `Placeholder` class is the way for a GeoMakeIt! Plugin to register a specific placeholder. As mentioned in the previous section, `Placeholder Interface`, when a Plugin Developer wants to create his own Placeholders he needs to *extend* this class.

This class contains a `plugin` variable, which is the plugin that the placeholder belongs to, as well as a `register` method to register the placeholder and an `isRegistered()` method to verify the placeholder is registered.

Placeholder Manager

The `Placeholder Manager` handles the parsing of placeholders and keeping track of the available placeholders.

Methods `register()`, `unregister()` and `isRegistered()`are provided for registering, unregistering and checking if a placeholder is registered.

A method `getRegisteredIdentifiers()` returns the registered plugin identifiers, so a Plugin Developer can check which plugins have registered at least one identifier.

The `containsPlaceholder()` is a helper function that checks if a string matches the classic placeholder pattern which is the following `{{plugin_identifier::placeholder_identifier}}`.

Finally, there are 4 `parse()` methods. The simple `parse(Text)` method that takes a an argument a single string, which the unpersonalized message and returns the personalized message back. A `parse(List<Text>)` method that take a list of unpersonalized messages and returns a list of personalized messages back. A `parse(List<Text>, Pattern)` method that takes a list of unpersonalized messages and a pattern for the placeholder, which returns the a list of personalized messages. Finally, a `parse(Text, Patern)` method that takes a single unpersonalized message and a pattern for the placeholder, which returns a single personalized message back. The last two `parse()` methods allow a Plugin Developer to change the Placeholder pattern, giving the freedom if necessary to change the pattern from **`{{`**`plugin_identifier::placeholder_identifier`**`}}`** to something different, for example: **`[[`**`plugin_identifier::placeholder_identifier`**`]]`**. It should be noted though that if there is not a very important reason to change a pattern, it is recommended that a Plugin Developer shall not.

(The “Configuration” component 3.2.h)

The `configuration` component allows plugins to create simple, yet powerful, configurations that are visible on the GeoMakeIt! Studio platform and easily configurable by the Game Creator. The GeoMakeIt! Plugin configurations are JSON files which reside in a folder inside the plugins directory, specifically in *src/main/assets/plugin_identifier/* directory.

All plugins have at least one configuration file called `plugin.json`, which contain information about the developed plugin.

The plugin.json file

This file contains all the necessary information about the plugin for the GeoMakeIt! Studio to configure it on the online platform and make it publicly available for download and installation. There are 9 required information that must be configured on this file.

```
{
    "main": "com.package.myplugin.Main",
    "title": "My Plugin's Title",
    "short_description": "My plugins short description",
    "description": "A bigger description containing more important information about what this plugin is all about and what it can offer the user that includes it",
    "version": "1.0.0",
    "author": "The name of the author",
    "gradle_implementations": [],
    "soft_dependencies": [],
    "hard_dependencies": []
}
```

Main

The main describes the location of the first class to be executed by the GeoMakeIt! API when the plugin is included. The Main is a class that *extends* the GeoMakeIt! API's `plugin` class described in *The “Plugins” package*.

Title

The `title` of the plugin that will be visible in the GeoMakeIt! Studio. Not to be confused by the plugin's identifier, the plugin's `title` can be a descriptive name not limited by the identifiers format. For example, if we create a plugin with an `identifier` *plugina*, the title could be: “*Plugin A: Awesome Plugin*”.

The title must be a `String` with no less than `3` characters but `30` characters at most.

Short Description

The short description of the plugin will be visible in the GeoMakeIt! Studio and used to persuade someone to view or install the plugin.

Description

The `description` of the plugin will be visible in the GeoMakeIt! Studio and it is used to example both the plugin's functionality as well as be as a tutorial for some of the plugin's commands and placeholders.

In a future implementation the description will be evolved into a fully functional wiki describing the whole functionalities of the plugin.

Version

The `current version` of the uploaded plugin. In a future implementation this will be used to version control the plugin through the GeoMakeIt! Studio.

Author

The `author` of the plugin will be visible in the GeoMakeIt! Studio. It can be a single author or a list of multiple authors who've assisted into developing this plugin. If left `null`, the username of the user that uploaded the plugin will be used instead.

Gradle Implementations

In some cases, there might be required for the main app to require some specific `gradle implementations`. In order to do that the `gradle implementations` setting is used to append such settings.

Soft and Hard dependencies.

`Soft` and `Hard` `dependencies` are both shown in the GeoMakeIt! Studio as well as used by the GeoMakeIt! API to calculate the loading order of the plugin. It is a list of other GeoMakeIt! Plugin dependencies identified by their unique identifier on the site.

`Soft dependencies` are dependencies that may or may not be present in a Game and are not required for the plugin's functionality.

`Hard dependencies` are dependencies that are required to be present in a Game otherwise the plugin will not function.

`Hard dependencies` requirements must be met, otherwise the plugin cannot be built.

Back to the `Configuration` component, this component is designed to load JSON Files, similar to the one discussed beforehand, `plugin.json`.

The `Configuration` component has the following methods: `exists()`, which checks if a JSON file exist in a specific path, `isJSONArray()` which checks if the JSON File is a JSON Array, `isJSONObject()` which checks if the JSON File is a JSON Object, `isValidJSON()` which checks if the JSON File is a valid JSON File and `asString()`, `asJSONObject()` and `asJSONArray()` which converts the loaded file into a String, a JSON Object or a JSON Array respectively.

There are multiple ways the Configuration class can be used to open a file. The most common way is by using the plugin's main class and calling the `getConfig()` method. For example if we need a configuration file named `my_example_configuration.json` which is stored in *src/main/assets/<plugin_identifier>/my_example_configuration.json* we could get its contents by calling the plugin's main class, `Main.getConfig("my_example_configuration")`. It should be noted that we do not need to use the .json file extension, since the only configurations currently allowed are .json files.

Here is a small example of a json configuration file:

```
{
  "question_1": {
    "question": "When you mix red and blue paint, what color does it make?",
    "question_type": "multiple_choice",
    "possible_answers": [
      "Aqua",
      "Purple",
    ],
    "correct_answer": 1,
    "image": "@android:drawable/btn_star_big_on"
  },
  "question_2": {
    "question": "Which is the first american company to reach 1$ trillion evaluation",
    "question_type": "multiple_choice",
    "possible_answers": [
      "GeoMakeIt!",
      "Microsoft",
      "Apple",
    ],
    "correct_answer": 2,
    "allowed_wrong_answers": 1,
    "image": "@drawable/a_random_picture"
  },
}
```

(The "Utilities" component 3.2.i)

The `Utilities` component described helper functions for common actions using the GeoMakeIt! API.

The `JSONUtil` object contains methods to read files stored in the `assets` folder as well as to load them as `Models` or `Maps`.

The `HTTPUtil` object contains methods to connect to the web and read data in the form of restful APIs.

Lastly, the `HelperUtil` contains methods to generate random model unique `identifiers`.

(The “Model” component 3.2.j)

The `Model` component is one of the most important classes when it comes to GeoMakeIt! API, but it is described at the end since it can incorporate any GeoMakeIt! Component.

The `Model` component is the representation of a singular object inside the NoSQL, in this case Google’s Firestore, database. It is an abstract class containing the fundamental methods and variables for a database object.

There are `4 constructors`, two of which can be used to generate a single object with common Java and Kotlin constructor practices, and two which can construct the object from a map. The two last constructors are usually used when loading an object from a configuration.

The `unique id` variable which is contained by the model is used as the primary id of that object. This identifier can be used to save, update, search the object when it comes to the database of choice.

A `db` variable also exists, giving the model instant access to the database.

The `toMap()` method exists for converting an object to a Map→JSON representation when saving to the database is required.

Finally the `save()` method saves the model instance to the database.

Multiple examples of model implementations will be used on this and the upcoming Chapters.

(Other GeoMakeIt! components 3.2.k)

There are other components in GeoMakeIt! that contribute directly or indirectly but do not necessary require a big section to understand. Such components are UI’s, Enumerators, Interfaces etc.

User Interfaces

User Interfaces that contribute to the GeoMakeIt!’s functionally are usually stored in the “*UI*” package. GeoMakeIt! API provides common user interfaces that may be used by any Game, thus providing some uniformity. Such interfaces are the “permissions” activity, which is responsible for notifying the user when not all the necessary permissions are given, and a simple Item List Activity used for the Quest Model that we will see in a later chapter.

Enumerators

Generic enumerators such as the `EAuthProviders` and `EBuildType` are stored in "*enums*" package. It should be noted that all enumerators start with a capital **E** to symbolize the "**E**numerator".

EAuthProvider

The `EAuthProvider` contains all possible ways for a user to authenticate using the Google's Authentication API. Google Authentication API can authenticate a user in one of these ways: `EMAIL`, `PHONE`, `GOOGLE`, `FACEBOOK`, `TWITTER`, `GITHUB`, `MICROSOFT`, `APPLE` and `ANONYMOUS`. Additionally, the `EAuthProvider` can be used to get the `IdpConfig` to build the authentication screen by using the Google's API.

EBuildType

The `EBuildType` enumerator contains the possible ways that the Game has been built. It currently supports two types, `DEVELOPMENT` and `RELEASE`. Plugin Developers can use this to check the build type of a Game.

Exceptions

The `Exceptions` package contains all possible exceptions that can be thrown by the GeoMakeIt! API. Such exceptions are `CommandException`, `GameInitializationException`, `InvalidCallerClassException` and `ModelMissingException`. Some of these exceptions are handled by GeoMakeIt! API, while others must be managed by the Plugin Developer.

Interfaces

Generic interfaces that may be on different places in the GeoMakeIt! API are stored in the "Interfaces" package. All interfaces created by GeoMakeIt! API have as a first letter the capital "**I**" which symbolizes the "**I**nterface".

Currently, the interfaces that exist in this package are the following:

ILoadable, IDBLoadable, IFileLoadable

The `IFileLoadable` contains a single method, `loadAllFromFile()`, which is usually used by a `Model` when it is necessary to load settings or game data from a configuration file.

The `IDBLoadable` contains a single method, `loadAllFromDB()`, which similarly to the `IFileLoadable`, it is used by a model when it is necessary to load settings or game data from the database.

Finally, the `ILoadable`, contains the two interfaces mentioned above, `IFileLoadable` and `IDBLoadable`, and can be used when both of those methods are necessary and required.

IModelFactory

The `IModelFactory` contains the usual methods that are required by a factory to create, add and modify data of a `Model`. It contains the following methods: `create()`, `add()`, `contains()`, `get()` and `getAll()`, for when needed to create a new model, add a new model to the models list, check if a model exist in the model list, get a specific instance of a model or get all the model's instances saved.

IExpansible

The `IExpansible` interface can be used when a class is necessarily expansible. You can imagine such classes being the Commands class, the Placeholders class and the Services class. Those classes require to be extended and as they implement the IExpansible interface. This interface contains 3 methods, registerExpansion(), unregisterExpansion() and isExpansionRegistered() for registering an expansion, unregistering an expansion and checking if an expansion is registered accordingly.

IDrawable

The `IDrawable` interface is used when a class need to be drawn into a Google Map. It contains two methods, draw and erase, each one of which contain a single argument, the Google Map. This interface can be seen used into some default models, discussed in GeoMakeIt! Default Plugin, specifically regarding the markers and zones.

IOverlappable

The `IOverlappable` interface can be used when an object can be overlapped. Such example can be seen similarly in the GeoMakeIt! Default Plugin, specifically in the zone models. This interface contains two methods, `onBeginOverlap()` and `onEndOverlap()` which are used when a player begins overlap with a zone or when a player stop's overlapping with that zone.

(GeoMakeIt! Default Plugin 3.3)

The GeoMakeIt! Default Plugin resides inside GeoMakeIt! API. It is the first plugin that it is loaded since all other plugins could depend on it. Its identifier is "`geomakeit`", and it provides some of the main functions that a Game Creator could use in their Games, such as starting specific activities, signing out a user, using the basic placeholders etc.

(The "Main" class 3.3.a)

The `main` class is the first class loaded when GeoMakeIt! Default Plugin is initialized. The `Main` class in this case is called "`GeoMakeIt`". Once opening though

GeoMakeIt.kt we can see that it is actually not a class, but rather an Object. When enabling a plugin there is only one active instance of it. Changing from class to an object is a standard solution for that design and makes it easier for us to call later on methods such as GeoMakeIt's logger, ' GeoMakeIt.logger.d("Example debug message") '.

```
object DefaultPlugin: Plugin() {
    override val identifier: String = "geomakeit"

    override fun onEnable() { ... }

    override fun onDisable() { ... }

                ...
}
```

The first thing we notice about GeoMakeIt object is that it *extends* the Plugin() class, and in doing so we need to include some extra methods and variables.

Firstly, we are required to set the plugin's identifier, "geomakeit". This specific identifier is reserved for this plugin, since it was registered in the GeoMakeIt! Studio as we will see in Chapter 4.3.

```
override val identifier: String = "geomakeit"
```

Next, we can see the onEnable() method, which register the Placeholders, the CommandMap, Activities, Services and finally loads all necessary files from settings.

```
override fun onEnable() {
    DefaultPlaceholders(this).register()
    CommandMap(this).register()
    ActivityRegistry.register(this, "quests", QAct::class.java)
    LocationManager.register()

    /* Load Defaults */
    AlertDialog.loadAllFromFile()

    /* Other */
    runStartupCommands()
        ...
}
```

Finally, the onDisable() method which unloads the whole plugin if necessary.

(Android Files and Resources 3.3.b)

Something that hasn't been discussed previously is about what do we do with the Android Default Files and Resources. When developing any GeoMakeIt! Plugin we must keep the following thing in mind. A GeoMakeIt! Plugin can use all the Android features and must contain an Android Manifest as well as a Resources Folder. In this case, GeoMakeIt! Default Plugin does too.

The `Android Manifest` is stored in the "*src/main*" of the plugin and requires no different configuration than the casual `Android Manifest` configurations described on the Android Documentation.

Resources though are slightly changed to contribute to the changed hybrid top-up and bottom-down design.

For "`Values`" Resources, contained in the values resource folder such as styles xml, config xml and strings xml the naming conventions remain the same, but for the content inside those files, names should be prefixed with the identifier of the plugin. For example, in `styles.xml`, this naming convention would require the Plugin Developer to do the following: `<style name="geomakeit_something">...</style>`. Furthermore, the final compiled strings xml if the name is prefixed with `geomakeit_public`, it makes that specific resource available for editing through the GeoMakeIt! Studio.

For other resources, such as `drawables`, `layout`, `navigations`, every file must be prefixed by the plugin's identifier to avoid conflicts on compilation time. For example, consider that you would usually create an `activity_main` and `activity_secondary`. When developing for GeoMakeIt! These files must be prefixed with the plugin's identifier, in this case for example **`geomakeit_`**`activity_main` and **`geomakeit_`**`activity_secondary`.

Similar situations can be seen in GeoMakeIt!'s Default Plugin's `drawables`, with the file named `geomakeit_logo.png`, the layout resource directory, with all the xml files being prefixed with "`geomakeit_`" and finally the same thing happening for `geomakeit_nav_quests` which can be found in the navigation directory.

(Commands 3.3.c)

The GeoMakeIt! Default Plugin registers 5 commands, which can be seen inside the file `CommandMap.kt`. Those commands are the `sign out` command, `activity` command, `alert dialog` command, `quest` command and `zone` command. We are going to discuss the command specific implementations when we see each specific command's model design, in this case the `user` models, the `activity` models, the `alert dialog` models, the `quest` models and `zone` models respectively.

```
class CommandMap : SimpleCommandMap {
    constructor(plugin: IPlugin) : super(plugin) {
        setDefaultCommands()
    }

    private fun setDefaultCommands() {
        registerCommand(SignOutCommand())
        registerCommand(ActivityCommand())
        registerCommand(AlertDialogCommand())
        registerCommand(QuestCommand())
        registerCommand(ZoneCommand())
              ...
```

```
    }
}
```

Let's first discuss about the `Command Map`. We can see that the `Command Map` extends the `Simple Command Map` discussed in Chapter 3.2. Next, for each command the `command map` uses the method `registerCommand(command)` to register that command.

```
class SignOutCommand : Command {
    constructor() : super("signout") {
        this.description = "something";
        this.usageMessage = "/geomakeit::signout";
    }

    override fun execute(commandLabel: String,
                         args: Array<String>): Boolean {
        UserAuth.signOut()
        return true
    }
}
```

Finally, lets open just one command to see its generic structure. We will open the `sign out` command since it is the simplest one to explain. By opening it we can see that we have declared the `description` and `usage message`. The `description` explains what this command does, while the `usage message` shows the example execution of that command. If that command is executed, the `execute()` method will run, which in this case it will sign out the user. We finally *return* `true`, declaring that this command was successful, and that the user has been signed out.

(Placeholders 3.3.d)

When opening the `DefaultsPlaceholders.kt`, we can see how to create a generic placeholder.

```
class DefaultPlaceholders(plugin: IPlugin) : Placeholder(plugin)
{
    override fun onRequest(placeholderId: String): String? {
        when (placeholderId) {
            "username" -> return LocalUser.displayName
            "email" -> return LocalUser.email ?: "N/A"
            "displayname" -> return LocalUser.displayName
        }
        return null
    }
}
```

The class `DefaultPlaceholders` *extends* the default `Placeholder` mentioned in Chapter 3.2 and all we are required to do is override the `onRequest()` method. We then cycle through the possible placeholders for this plugin, in this case `username`, `email`, and `display_name`. As such, the GeoMakeIt! Default Plugin, once registered, tells the GeoMakeIt! API to register these placeholders: `geomakeit::username`,

geomakeit::email and geomakeit::display_name. In doing so, if a personalizable message like "*Hello there {{geomakeit::username}}, how are you?*" it will be parse as the following: "*Hello there AwesomePlayer123, how are you?*" where AwesomePlayer123 is the name of the player. We finally see that if nothing is matched, null is returned.

(Configurations 3.3.e)

GeoMakeIt! Default plugin contains some predefined configuration files. All these files are saved as JSON configuration files under the path "*src/main/assets/geomakeit/*". By opening this path we can see the following JSON files: alert_dialogs.json, markers.json, plugin.json, quests.json, startup.json, zones.json and config.json.

We'll take a look first on the file we are familiar with, while the rest we will discuss when we see their represented models. As such, let's take a quick pick on the plugin.json file.

```
{
  "main": "com.geomakeit.api.defaults.GeoMakeIt",
  "title": "GeoMakeIt!",
  "short_description": "The main API",
  "description": "GeoMakeIt! is the main api to create GeoMakeIt! games. Every game that is created through our platform must require it to be able to create the final game.",
  "version": "1.0.0",
  "author": "Vasilis Dimitriadis",
  "gradle_implementations": []
}
```

Knowing from the previous chapters how the plugin.json is formatted, we can identify that this plugin has its main located in the com.geomakeit.api.defaults.GeoMakeIt location, called GeoMakeIt. We can also see that the title of this plugin on GeoMakeIt! Studio is going to be "*GeoMakeIt!*", alongside with its short description "*The main API*" and it's full description "*GeoMakeIt! is the main api to create GeoMakeIt! games. Every game that is created through our platform must require it to be able to create the final game.*". Finally, we can see that its current version is "*1.0.0*", authored by "*Vasilis Dimitriadis*" requiring no extra gradle implementations and having no soft or hard dependencies.

Under the "*com.geomakeit.api.defaults.configurations*" package we can also see the plugin's default configuration file.

```
object ConfigFile {
    val authentication: AuthenticationFileSection

    init {
        val configuration = GeoMakeIt.getConfig()

        if (!configuration.exists()) { // Config.json NOT FOUND
            PluginLogger.e("File 'config.json' is missing. Loading default configurations")
            authentication = AuthenticationFileSection()
```

```
            } else {
                // Load all JSON
                val json = configuration.asJSONObject()

                // Load Authentication Section
                val authObject = json.optJSONObject("authentication")
?: JSONObject()

                @Suppress("UNCHECKED_CAST")
                authentication = AuthenticationFileSection(
                    Gson().fromJson(authObject.toString(),
Map::class.java).filterKeys { it is String } as Map<String, *>
                )

                // Load ...
            }
        }
    }
```

We can see that this file is calling the `Configuration` class to load the `config.json` and does checks to verify if the file exists and is a valid JSON file or if not. If it is a valid JSON file it loads its data to the `ConfigFile` Object, as well as any other object sections specific such as the `Authentication Section` to keep everything cleaned up and tied up. If no such file exists or if the JSON file is invalid, then the configuration is loaded with its default values.

Currently, any configuration saved in the root of "`/assets/<plugin_identifier>/`" directory is considered `public` and will be uploaded to GeoMakeIt! Studio. Any sub-directories will be considered `private` and can be used by developers to store other private information.

```
    {
      "authentication": {
        "enabled": true,
        "providers": ["anonymous", "email"]
      }
    }
```

The current configuration file looks like this, containing information about if the game is going to contain authentication methods, which one of these it will contain, the startup commands etc.

(Models 3.3.f)

GeoMakeIt! Default plugin provides 5 initial models that a Plugin Developer can later use or extend upon them and a Game Creator can use them to design his game. These are an `Alert Dialog` Model, a `Marker` Model, a `Quest` Model, a `User` Model and a `Zone` Model. These 5 models are provided as an initial proof of concept for the GeoMakeIt! API and GeoMakeIt! Default Plugin. Further down the line, some of the models might be extracted to separate plugins of their own, while new ones might also emerge. Until then, let's discuss the ones currently implemented in this version of the GeoMakeIt! Default Plugin.

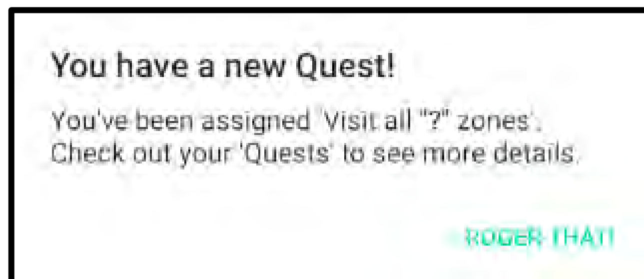

Figure 3.2 Alert Dialog example inside a GeoMakeIt! game

The `Alert Dialog` is an Android based model for a small window that allows the user to make a decision or enter additional information. In this case, the `Alert Dialog` that GeoMakeIt! Default Plugin provides is a small wrapper around the `Android's dialog` that can be loaded from a JSON configuration or from the database. GeoMakeIt! Default Plugin's `Alert Dialog` consists of 3 classes, the `Alert Dialog` itself, the `Alert Buttons` and an `Alert Dialog` command.

Alert Dialog class

This is the wrapper around a basic `Android's Alert Dialog` implementation. It consists of a `unique id` like any model, this way it can be stored and managed through the database. Additionally, it contains the `title` of the Alert Dialog, the `message` that it contains, an option for making it `cancelable` or not and the options for the `buttons`, a positive, a neutral and a negative button.

```
val uniqueId: String
var title: String
var message: String?
var cancellable: Boolean
var positiveButton: AlertButton?
var neutralButton: AlertButton?
var negativeButton: AlertButton?
```

There are 3 available constructors, all leading to the main constructor containing most of the available variables

```
constructor(uniqueId, title): this(uniqueId, title, null)
constructor(uniqueId, title:, message): this(uniqueId, title,
        message, false)
constructor(uniqueId, title, message, cancellable,
        positiveButton, neutralButton, negativeButton) {

    this.uniqueId = uniqueId
    this.title = title
    this.message = message
    this.cancellable = cancellable
    this.positiveButton = positiveButton
```

```
        this.neutralButton = neutralButton
        this.negativeButton = negativeButton
    }
```

There are two method, `show()` and `dismiss()`, that both take as arguments the current context which by default is the current activity's context. Show makes the dialog visible to the player while dismiss dismisses this dialog.

```
    fun show(context: Context = App.getCurrentActivity())
    fun dismiss(context: Context = App.getCurrentActivity())
    private fun asAlert(context: Context = App.getCurrentActivity())
       : AndroidAlertDialog
    private fun setupButton(button: AlertButton,
       dialogBuilder: AndroidAlertDialog.Builder, buttonType: Int)
```

There are two private methods, `asAlert()` and `setupButtons()`, both for setting up the alert dialog. The method `setupButtons()` prepares the positive, neutral and negative buttons with some default actions. These actions can be set up from the data configurations files that we will see shortly. There are 5 possible actions.

Actions "`cancel`" and "`dismiss`" cancel or dismiss, subsequently, the current alert dialog.

Action "`shutdown`" dismisses the current dialog and proceeds to shut down the whole application.

Action "`nothing`" does nothing.

Finally, when it comes to actions, if none of the above are specified, the user can simply enter a `command`. For example, if the Game Creator sets as a button action the command "*geomakeit::signOut*", in this case, if a user clicks the button he will be signed out.

The other private function, `asAlert()`, converts the GeoMakeIt!'s Alert Dialog to an Android's Alert Dialog.

```
    companion object: IModelFactory<AlertDialog>, IFileLoadable {
       const val BUTTON_POSITIVE: Int = 0
       const val BUTTON_NEUTRAL: Int = 1
       const val BUTTON_NEGATIVE: Int = 2

       private val models: MutableMap<String, AlertDialog>
          = hashMapOf()
       override fun create(): AlertDialog? { ... }
       override fun create(uniqueID: String?): AlertDialog? { ... }
       override fun add(model: AlertDialog): Boolean {
           if(contains(model.uniqueId)) return false
           models[model.uniqueId] = model
           return true
       }
       override fun contains(uniqueID: String): Boolean {
           return models.contains(uniqueID)
       }
```

```
    override fun contains(model: AlertDialog): Boolean {
        return models.contains(model.uniqueId)
    }
    override fun get(uniqueID: String): AlertDialog? {
        return models[uniqueID]
    }

    override fun getAll(): Map<String, AlertDialog> {
        return models.toMap()
    }

    override fun loadAllFromFile(): Boolean {
        return try {
            var config =
                    DefaultPlugin.getConfig("alert_dialogs.json")
            val json = config.asJSONArray()
            for (i in 0 until json.length()) {
                val jsonAlert = json.getJSONObject(i)
                val alertDialog =
                    Gson().fromJson(
                        jsonAlert.toString(),
                        AlertDialog::class.java)

                add(alertDialog)
                DefaultPlugin.logger.i("AlertDialog
                      '${alertDialog.uniqueId}' loaded from file.")
            }
            true
        }catch (ex: Exception) {
            DefaultPlugin.logger.e("Failed to load AlertDialogs
                                   from file: $ex")
            false
        }
    }
}
```

Lastly since it is an `AlertDialog` *extends* the GeoMakeIt!'s `Model` class, it also contains a static object that *extends* the `ModelFactory` and `IFileLoadable`. The first one is extended to define necessary methods to create, add, check if an Alert Dialog model already exists, while the second is used to load alert dialogs both from the database as well as from data files.

Alert Button class

```
data class AlertButton (val text: String, val action: String)
```

This is a simple data class containing just the text of a button and its action.

Alert Dialog Command class

This class is responsible for the `geomakeit::alert_dialog` command. It is currently responsible for displaying a specific alert dialog back to the player. The Game Creator can execute this command for example, "*geomakeit::alert_dialog show dialog_1*",

to show to the current player the `dialog_1` which is saved in the `alert_dialogs.json` file. If such command doesn't exist, a warning will be logged to the Android's log.

Configuration File: alert_dialogs.json

The `alert_dialogs.json` file located in the assets folder contains instances of an Alert class, constructed as a json file.

```
[
  {
    "unique_id": "alert_0",
    "title": "Hello {username}",
    "message": "Do you want to {score::score} this application?
{username} {email}",
    "cancellable": true,
    "positive_button": {
      "text": "Success",
      "action": "nothing"
    },
    "neutral_button": {
      "text": "Erm Okay Close me",
      "action": "shutdown"
    },
    "negative_button": {
      "text": "Negative",
      "action": "close"
    }
  }
]
```

The `Game Creator` must specify a `unique id` for the alert and can then append the alert details.

The `title`, `message` and the `text` variable contained inside the buttons can all take placeholders.

By specifying the `positive`, `neutral` or `negative` button the `Game Creator` enables the alert buttons and executes the `actions` described.

Users Model

The `Users` Model contains information about the player, such as his username and location. It is structured by 4 classes, the `IEntity`, `User`, `LocalUser` and `UserAuth`.

IEntity interface

The `IEntity` interface is just a requirement that every entity must contain a location on the map.

User class

```
abstract class User : IEntity {
    abstract val id: String
    abstract val displayName: String?
    abstract val email: String?
    override var location: Location? = null

    fun hasDisplayName(): Boolean { ... }
    fun hasEmail(): Boolean { ... }

    companion object {
        private var players: MutableMap<Int, User> = hashMapOf()
    }
}
```

The `User` class is an abstract design of how a GeoMakeIt! Player will look like. The player contains a `unique id`, an optional `display name`, an optional `email` and optional `location`. This class has currently no functionality, but it is implemented to provide future multiplayer functionality.

Local User object

```
object LocalUser : User() {
    private val firebaseUser: FirebaseUser
        = FirebaseAuth.getInstance().currentUser!!
    override val id: String
        get() = firebaseUser.uid
    override val displayName: String?
        get() {
            return ...
        }
    override val email: String?
        get() = ...
}
```

The `Local User` object is the representation of the currently authenticated or anonymous user. Currently, the information of this authenticated user is taken from the Google's Firebase Authentication, commonly called `FirebaseAuth`,.

Markers Model

The `Markers` model exists to add functionality to the map. It is wrapper for the API of the `Android's Map Markers`, but since it is using the GeoMakeIt! logic it is also configurable.

The `Markers` model contains a single class to declare the model and a command class to allow the `Game Creator` to execute commands on markers.

Marker class

The Marker class includes the following configurable variables through the marker.json file which we will see shortly.

```
var position: GeoPoint = GeoPoint(0.0, 0.0)
var title: String = ""
var snippet: String? = null
var alpha: Float = 1.0f
var flat: Boolean = false
var draggable: Boolean = false
var visible: Boolean = true
var rotation: Float = 0.0f
var zIndex: Float = 0.0f
```

The position describes the exact position the marker will be placed on. Each marker can also contain a title and a snippet/short description. Lastly visibility, rotation, transparency and zIndex can also be configured.

Marker Command class

The Marker Command allows a Game Creator to draw or erase specific markers, depending on his needs.

Commands that are currently supported:

- geomakeit::marker draw *<Marker ID>* - Draws the Marker ID on the map.
- geomakeit::marker erase *<Marker ID>* - Erases the Marker ID from the map.

Zones Model

Like the way the Markers model exists to add functionality to the Map, the Zones model is here to add zones. This API is a wrapper of the Android's Map Zones classes but infused with some GeoMakeIt! logic to make it configurable. Opening the zones package we can see that it contains 6 classes that we will see in detail. The Zone class, the Circle Zone class, the Polygone Zone class, a Zone Monitor Service, a Zone Visit class responsible for storing zone visits and the Zone Command.

Zone class

```
abstract class Zone : IOverlappable, IDrawable, Model {
    override val dbModel: String
        get() = DB_MODEL

    var title: String = "Untitled Zone"
    var clickable: Boolean = false
    var visible: Boolean = true
    var zIndex: Float = 1.0f
    var fillColor: String? = null
    var strokeColor: String? = null
```

```
    var strokeWidth: Float = 10.0f
    var onEnter: List<String> = listOf()
    var onExit: List<String> = listOf()
    var isHidden: Boolean = true
        ...
}
```

The `Zone` class describes an abstract structure of the Android's Map Zone. It consists of the following elements:

The `title`, which is required and can be used to describe the zone.

The settings `clickable` and `visible`, which determine if a zone can be clicked and if it is visible or not.

The `zIndex`, which determines which zone is drawn on top of which. Example: For two zones, Zone A with `zIndex` 1, and *ZoneB* with `zIndex` 2, the *ZoneB* is drawn on top of Zone A since it has bigger `zIndex`.

The `fillColor`, `strokeColor` and `strokeWidth` determine the hexademical color code for the inside of a zone and the stroke of a zone respectively, while `strokeWidth` determines how thick is the zones outer layer in pixels, which stays the same independently from the map's zoom factor.

Lastly, for variables, there are two lists, `onEnter` and `onExit`, which determine the commands to be executed when a player enters the zone and when a player leaves the zone.

Moving on to the methods of the abstract Zone:

```
                    ...
    constructor(): super()
    constructor(uniqueID: String): super(uniqueID)

    abstract fun zoneOpts(): Any
    abstract override fun draw(gMap: GoogleMap): Any
```

There are two abstract methods, `zoneOpts()` and `draw()`. The first one is the link between the GeoMakeIt's zone and the Android's one. It converts a GeoMakeIt `Zone` to an `android zone options`. The `draw()` method draws the `zoneOpts()` in a map.

```
    override fun onBeginOverlap() {
        if(isHidden) return
        DefaultPlugin.logger.d("${LocalUser.displayName} entered
             zone $uniqueId")

        // Store user's visit to zone
        val zone = ZoneVisit.create(LocalUser, this, Date())
        ZoneVisit.add(zone)

        onEnter.forEach { CommandManager.dispatch(it) }
```

```
    }

    override fun onEndOverlap() {
        if(isHidden) return
        DefaultPlugin.logger.d("${LocalUser.displayName} left
             zone $uniqueId")
        onExit.forEach { CommandManager.dispatch(it) }
    }
```

Next, there is the `onBeginOverLap()` and `onEndOverlap()` methods, which check first if the zone is hidden or not and if not, they proceed to execute the `onEnter()` / `onExit()` commands as well as log a zone visit.

```
    fun startMonitoring(): Boolean {
        return ZoneMonitor.startMonitoring(this)
    }

    fun stopMonitoring(): Boolean {
        return ZoneMonitor.stopMonitoring(this)
    }
```

Finally there are the `startMonitoring()` and `stopMonitoring()` methods to start or stop monitoring a zone for zone overlaps, and the classic static `ModelFactory` for creating, adding, checking if a zone exist or loading one from file or database.

Circle Zone and Polygon Zone class

```
class CircleZone : Zone {
    var center: GeoPoint
    var radius: Double = 10.0
              ...
}
```

```
class PolygonZone : Zone {
    var points: ArrayList<GeoPoint> = arrayListOf()
              ...
}
```

These are implementations of the abstract `Zone` class and are created for declaring circle and polygon zones. When implementing a `Circle Zone`, we introduce two new variables, the `center position` and a `radius`, while when we implement a `Polygon Zone`, we introduce a single variable, a `list of positions` with start and end the Initial position to create a full polygon.

```
               ...
override fun zoneOpts(): CircleOptions {
     val opts = CircleOptions()
     opts.clickable(clickable)
     opts.visible(visible)
     opts.zIndex(zIndex)
     opts.strokeWidth(strokeWidth)

     fillColor?.let{ opts.fillColor(Color.parseColor(it)) }
```

```
        strokeColor?.let{ opts.strokeColor(Color.parseColor(it)) }

        opts.center(getLatLng())
        opts.radius(radius)
        return opts
    }
                    ...
```

Since these implementations *extend* the abstract `zone`, we also need to declare and fill the abstract functions, so for the `Circle Zone` the `zoneOpts()` return a `CircleOptions` object while the `Polygon Zone` returns a `PolygonOptions`. Lastly we define how these shapes are drawn on a map and how they are removed.

Zone Monitor class

The `Zone Monitor` is a service which registers active zones and monitors them for player activity such as entering or exiting the zone. This service runs every 3 seconds and when it detects change in the position of a player it checks for new zone collisions.

Zone Visit class

```
abstract class Zone : IOverlappable, IDrawable, Model {
            ...
    fun onBeginOverlap() {
            ...
        val zone = ZoneVisit.create(LocalUser, this, Date())
        ZoneVisit.add(zone)
            ...
    }
            ...
}
```

The `Zone Visit` class stores information about when a user visited a specific zone. Each time a user enters a zone which is visible, the `onZoneEnter()` method which is triggered by the `Zone` stores a new `Zone Visit` instance in the database for that specific player.

Zone Command class

The `Zone Command` allows a Game Creator to verify if a user has visited a zone and to draw or erase specific zones.

Commands that are currently supported:

- `geomakeit::zone visited` *`<Zone ID>`* `-` Returns `true` or `false` if the player has visited or not the `Zone ID`.
- `geomakeit::zone draw` *`<Zone ID>`* `-` Draws the `Zone ID` on the map.
- `geomakeit::zone erase` *`<Zone ID>`* `-` Erases the `Zone ID` from the map.

Quests Model

The last model included by GeoMakeIt! Default Plugin is the `Quest` model. This model is here just for the prototyping stages and should be a separate plugin in a future release. With this fact mentioned, it might not seem so weird that we have a Quest system implemented directly inside the GeoMakeIt! API.

The `Quests` Model contains a few classes and UI Elements. The UI elements are stored in the `res` folder of the GeoMakeIt! API and are prefixed with "`geomakeit`". Let's see and explain some of the classes included.

Quest class

```
class Quest: Model {
    var title: String
    var description: String
    var onAssignment: ArrayList<String> = arrayListOf()
    var actions: ArrayList<String> = arrayListOf()
    var onComplete: ArrayList<String> = arrayListOf()

                   ...
}
```

The `Quest` class describes the structure of a quest. Each quest contains a required `title` and `description`, as well as optional `action` to be executed `on assignment`, `after assigned` and `on completion`. The last 3 are important because these are the settings a Game Creator changes to decide how a quest behaves, more importantly:

The "`on assignment`" option contains a list of actions, commands, that will be executed as soon as the quest is assigned to a player. Usually on assignment commands such as an alert dialog or a notification are executed. A quest gets manually assigned by a command which we will see shortly.

The "`actions`" options contain a list of commands that all must return `true` before the quest is considered complete. Consider that we are making a quest for a player to go from point a to point b. We can accomplish that by creating two zones and checking if a user has visited both point a and point b with the "`geomakeit::zone visited`" command.

Finally, the "`on complete`" options contains a list of commands that will be executed as soon as the quest is considered completed. Such commands usually are considered rewards to the player, for example a message congratulating him, giving him extra health or score, or could be more functional such as assigning him to the next quest.

EQuestStatus Enumarator

```
enum class EQuestStatus(val status: Int) {
    STATUS_UNASSIGNED(0),
    STATUS_ASSIGNED(1),
    STATUS_COMPLETED(2);
}
```

The `EQuestStatus` contains all possible statuses for a Quest. These are "`UNASSIGNED`", "`ASSIGNED`" and "`COMPLETED`". These statuses are used by other Plugin Developers when extending the plugin or manually assigning quests through scopes.

PlayerQuest and PlayerMonitor class

Similar to the aforementioned `PlayerVisit` on a Zone, the `PlayerQuest` stores data about the status of each player's quest. Basically, it keeps track of what quests are currently active for a player, what quests has he completed and what are his upcoming quests.

The `PlayerQuestMonitor` on the other hand is a service similar to the `ZoneMonitor` and in a specific interval checks the status of the current players active quests, making sure to classify them as completed when necessary.

Quest Command class

The `Quest Command` class contains all the commands regarding the `Quest` model. Currently there are two commands in this class.

The "`assign`" command assigns a specific quest to a player. To execute this command the Game Creator can call "`geomakeit::quest assign` *`<quest_id>`*", which automatically assigns the Quest to the player if it exists.

The "`geomakeit_activity_quests`" command, which opens a simple UI to show active quests. If a Game Creator wants to execute this command he may call "`geomakeit::quest geomakeit_activity_quests`".

The same functionality could be also accomplished using "`geomakeit::activity geomakeit::quests`", which is also recommended. We are only introducing this to demonstrate the ability of a third-party developer opening activities outside of `ActivityRegistry`, thus having more control on his command.

Quest Placeholder class

Contains slightly more advanced placeholders regarding the information about the quest using regex.

This class registers the following placeholders:

- `{{ geomakeit::quests.latest_assigned.<info> }}`
- `{{ geomakeit::quests.latest_completed.<info> }}`
- `{{ geomakeit::quests.<quest_id>.<info> }}`

Where `<info>` can be the `title` or `description` of a quest for prototyping reasons.

As such, this placeholder class shows that a placeholder can be a more dynamic if needed and can take new forms by using regex.

UI package

The `Quest` Model also contains a UI package which includes a simple user interface for displaying currently assigned quests and information about them.

CHAPTER 4: GeoMakeIt! STUDIO

(Introducing the STUDIO 4.1)

(Overview 4.1.a)

GeoMakeIt! Studio is the platform that connects Game Creators, Plugin Developers and Players all together. The GeoMakeIt! Studio is the online tool for building GeoMakeIt! Games and for uploading GeoMakeIt! Plugins. It is the way for a Game Creator to create games, add plugins, configure, build and release them to the plugin and the way for a Plugin Developer to register, upload, publish and share their plugins.

(Landing Page 4.1.b)

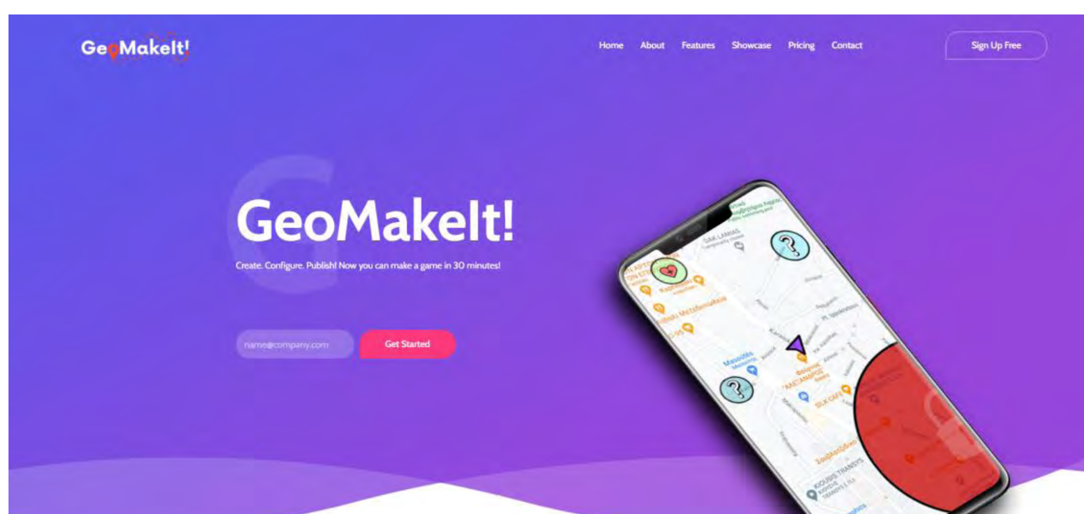

Figure 4.1 GeoMakeIt! Studio Landing page

The Landing Page is the first page someone looking for GeoMakeIt! will find. It can be considered a portal to a giant workshop of makers, makers of Geographic Positioning based games.

Although the landing page has no functionality other than introducing the tool, it is considered important since it will be the decision of someone using or not the tool. As such the landings page design must be elegant enough to understand it's a powerful tool, yet playful enough to show the worlds of wonders it can create.

The landing page is a free template designed by ColorLib[35] and customized to fit the needs of this thesis.

(Signing up / Signing In 4.1.c)

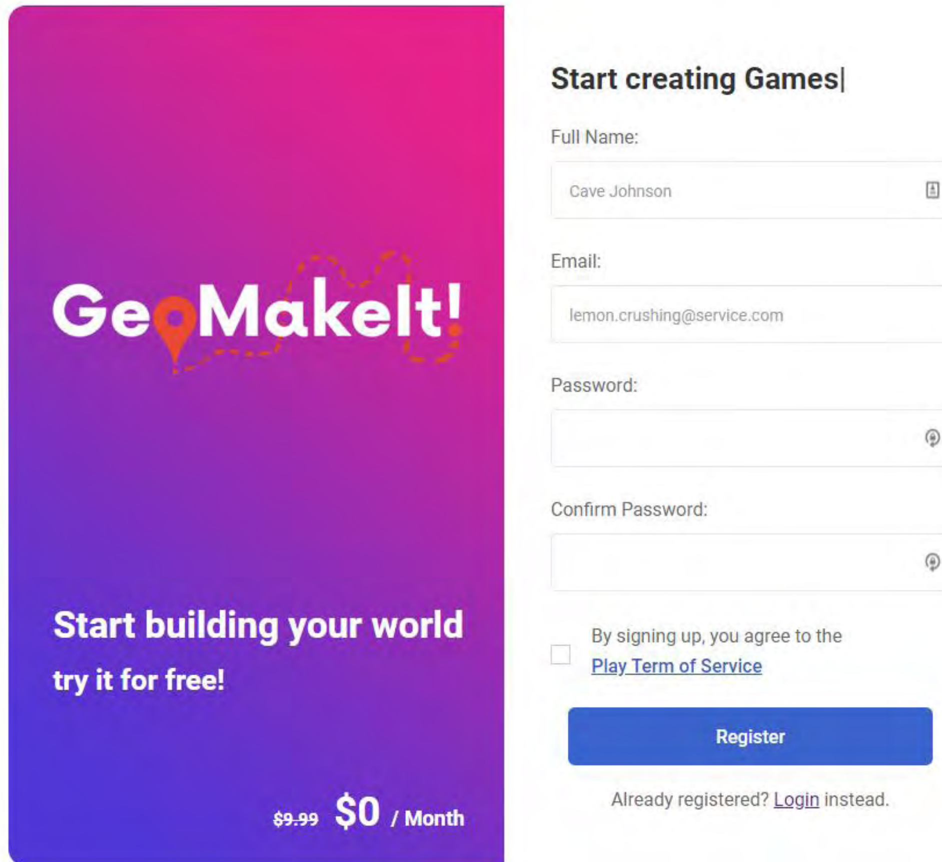

Figure 4.2 GeoMakeIt! Studio sign up page

The sign in and sign up provide the casual features as any other sign in or sign up page. More specifically, the user can:

- Login or Register to the service as a Creator.
- Password reset, if necessary.

The sign in / sign up page is a free template designed by ColorLib and customized to fit the needs of this thesis.

(First view to the Studio 4.1.d)

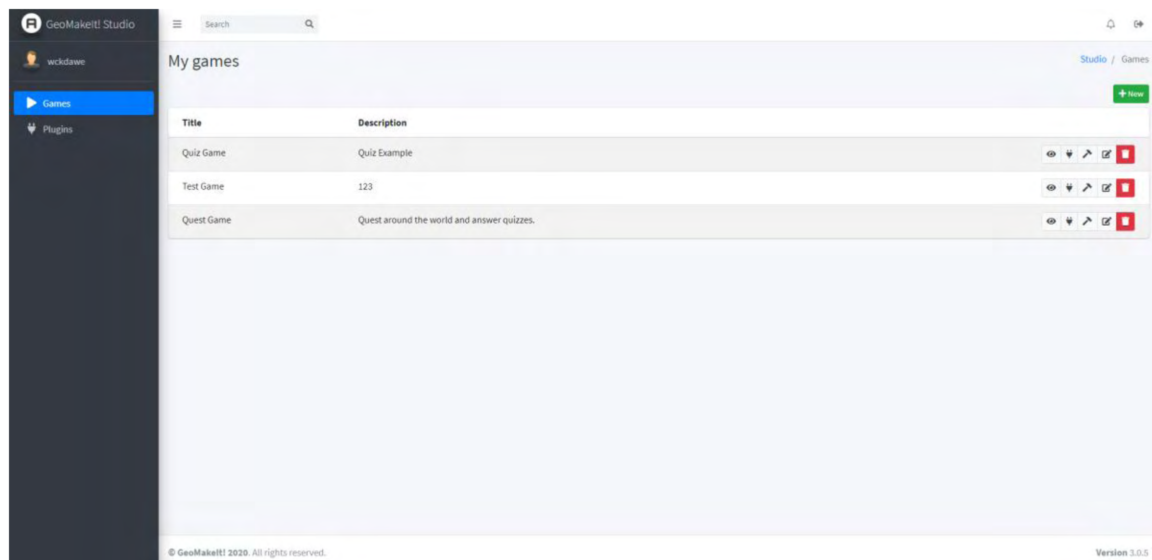

Figure 4.3 GeoMakeIt! Studio Games browse page

After signing in or up to GeoMakeIt! we get redirected to the GeoMakeIt! Studio. Currently, the following pages have been developed.

- "BREAD" for Games, accessible by Game Creators.
- "BREAD" for Plugins, accessible by Plugin Developers.
- Per-Game Plugin Showcase, showing available plugins that can be installed for a game.
- Per-Game Plugin Management, showing available actions that can be performed per plugin, such as editing configurations or reading plugin information.

Where BREAD symbolizes Browse, Read, Edit, Add and Delete.

The landing page is a free template designed by and customized to fit the needs of this thesis, using AdminLTE 3[36] and Voyager[37].

(Signing in / Signing up 4.2)

(Signing up as a Game Creator 4.2.a)

To sign up to GeoMakeIt! Studio, the user must go to the Sign up page, located in http://geomakeit.com/account/signup. From there the user must follow the instructions and fill the available form with his personal information, including the following:

- **Full Name** – An alphabetic only representation of the user's full name. Ex. *Vasilis Dimitriadis*.
- **Email Address** – The user's email address, ex. *vdimitriadis@uth.gr*.
- **Password** and **Password Confirmation** – The password that will be used to login to the account associated with the email address filled. The password must meet some minimum requirements for security precautions.

Before submitting the form, the user must also accept the services Terms and Conditions of Service. At this point and time, there are no Terms for this service and this option is simply a placeholder for the Terms of Service that will be added in the future.

After submitting the form, the user will be registered to the service and he will be given the role of Game Creator. In the next section we will discuss on how to register as a Plugin Developer for GeoMakeIt!.

(Becoming a Plugin Developer 4.2.b)

Obtaining the role of a Plugin Developer in GeoMakeIt! can be easily done through GeoMakeIt! Studio. A user must first register for a Game Creator account, which gives him access to the GeoMakeIt! Studio itself. After the registration is completed, the user must sign in to his GeoMakeIt! Studio account and can select to become a Plugin Developer.

After proceeding to the next page, the user will be required to accept some extra Terms of Service, written specifically for Plugin Developers. Once that step is completed the user can submit the form and will be immediately given the extra role of Plugin Developer.

Currently, any user is assigned automatically a Plugin Developer role and no Terms of Service have been written since the project has not been finalized.

(Signing in to GeoMakeIt! Studio 4.2.c)

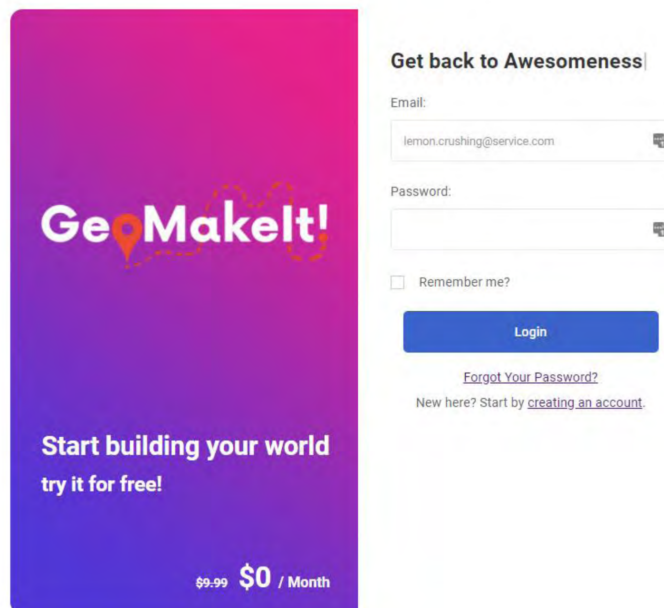

Figure 4.4 GeoMakeIt! Studio Sign in page

The user can sign in to GeoMakeIt! Studio by going to the sign in page located in http://geomakeit.com/account/login. From there, the user must follow the instructions and fill in the form accordingly. The user can sign in by providing his `email address`

and the password he registered with. If the user has forgotten his password, he can request a password reset, as seen in Figure 4.4.

(Plugins 4.3)

When signed in as a Plugin Developer in GeoMakeIt! Studio, the user has access to the Plugins tab in the main menu located on the far left of the page. Opening that tab will redirect the user to his plugins Browse view.

The user has access to browse, read, edit, add or delete his own plugins. We will see all these actions in the upcoming sections. For all these sections we will consider as a starting point for the user actions his plugins Browse view, which can be seen in Figure 4.5.

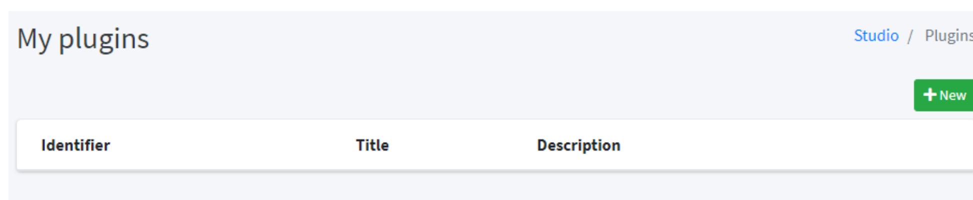

Figure 4.5 GeoMakeIt! Studio plugins browse page

(Registering an identifier 4.3.a)

Plugin Developers can register their plugin's identifier inside the GeoMakeIt! Studio by creating and uploading their plugin. Registering an identifier can be considered equal to creating a plugin object in the GeoMakeIt! Studio.

The Plugin Developer can create a plugin by clicking the green "+ New" button located on the top-right of the page. From there, he will be redirected to the plugin Add view, where he must provide his selected identifier as seen in Figure 4.6.

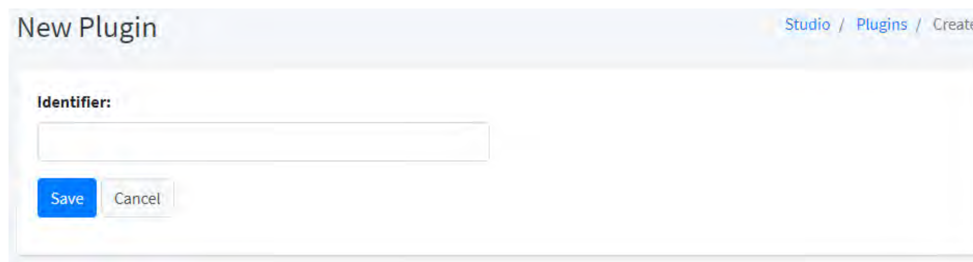

Figure 4.6 Selecting an identifier inside GeoMakeIt! Studio

The identifier must follow the rules declared in Section 7.2.c, "Choosing an Identifier". In sort, the following rules apply:

- Must be at least 3 characters length and at most 32.
- Must be all lowercase.
- Must start with an alphabetic character.

- `After that, it can contain any alphanumeric character and underscores.`

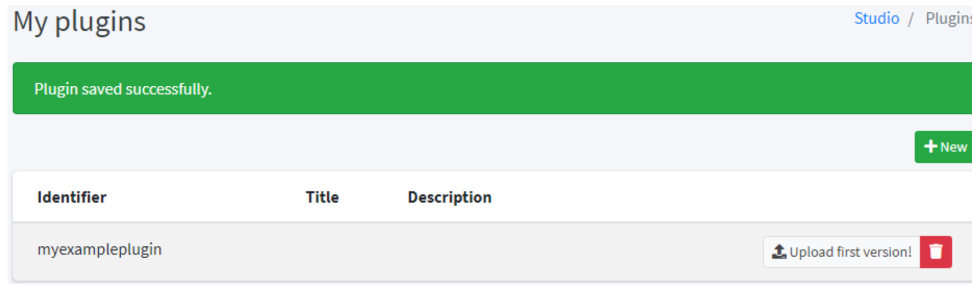

Figure 4.7 Example of saving a plugin

If successful, the plugin will be created, the identifier will be registered, and a success message will appear on the browser as seen in Figure 4.7. If unsuccessful, the user must verify that he followed the rules mentioned above, and in the case that the identifier is taken the user must choose a different one.

(Uploading a Plugin 4.3.b)

After a successful plugin creation, the Plugin Developer must proceed into uploading his compiled plugin in "`.aar`" format. When uploading a newly created plugin, the `Browse` view will contain a "`Upload first version!`" button. Clicking that button will redirect to the `Edit` view which can be seen in the Figure 4.8. The user now searched for the `.aar` file by clicking the `Browse` button and finally uploads it by clicking `Upload Plugin`.

The plugin gets uploaded, verified and has its data extracted. This process is described in detail in Chapter 4.5.

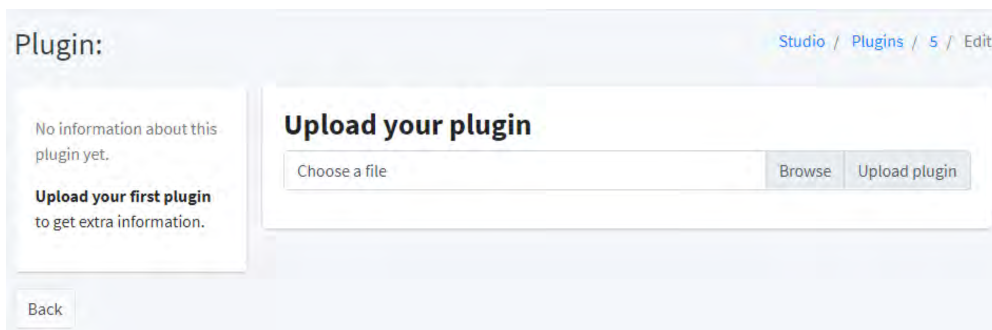

Figure 4.8 Uploading a plugin to GeoMakeIt! Studio

If the plugin succeeds in all the verification steps, the plugin details page gets updated containing new information gathered from the extracted plugin as seen in Figure 4.9.

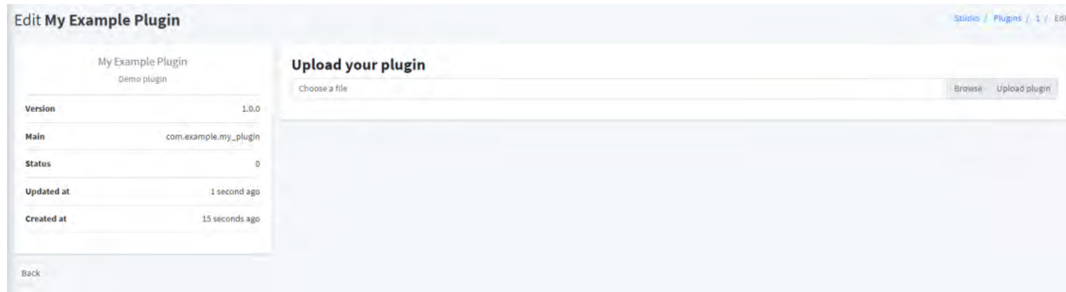

Figure 4.9 Uploaded plugin with new information

(Plugin Management 4.3.c)

In this version of GeoMakeIt! Studio, Plugin Management is limited and is done all directly from the "`.aar`" file itself once uploaded. Decisions on the exact Plugin Management system haven't been taken into account yet, but most likely the plugin details will be added automatically by the "`.aar`", such as the `title`, `short description`, `version` and `authors` while other plugin details similar to `wiki`, `tutorials`, videos and `template projects` will be added through the web platform.

(Deleting a Plugin 4.3.d)

Deletion of a plugin is possible if the author decides to abandon it. As seen in Figure 4.7, a red `Trash` button is provided to perform this action. If a Plugin Developer chooses to abandon a plugin, the plugin will be archived for a period to allow Game Creators to either remove it completely or choose an alternative option. This ensures that no Game breaks in its current version and that it can continue to work normally with no disruptions. When the period passes or no other Games are using the abandoned plugin, the plugin can be permanently deleted by the Plugin Developer.

In future versions more options will be available to the Plugin Developer, such as allowing another developer to take over the project or finding people to temporary support it.

(Games 4.4)

When signed in as a `Game Creator` in GeoMakeIt! Studio, the user has access to the Games tab in the main menu located on the far left of the page. Opening that tab will redirect the user to his games `Browse view`.

The user has access to browse, read, edit, add or delete his own games. We will see all these actions in the upcoming sections. For all these sections we will consider as a starting point for the user actions his games `Browse` view, which can be seen in Figure 4.10.

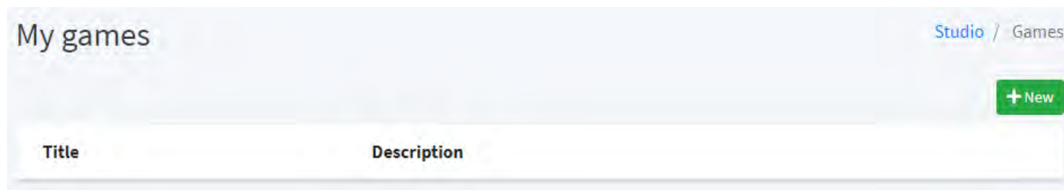

Figure 4.10 Games browse page in GeoMakeIt! Studio

(Creating a Game 4.4.a)

Creating a Game in GeoMakeIt! Studio can be done through the games `Browse` view. Pressing "`+ New`" will redirect the Game Creator to the Create view of the Game's "`BREAD`". The user must fill in the required information displayed in the form seen in Figure 4.11. There are no rules regarding game creation as long as the `title` and `description` are not blank and does not exceed the character upper limit.

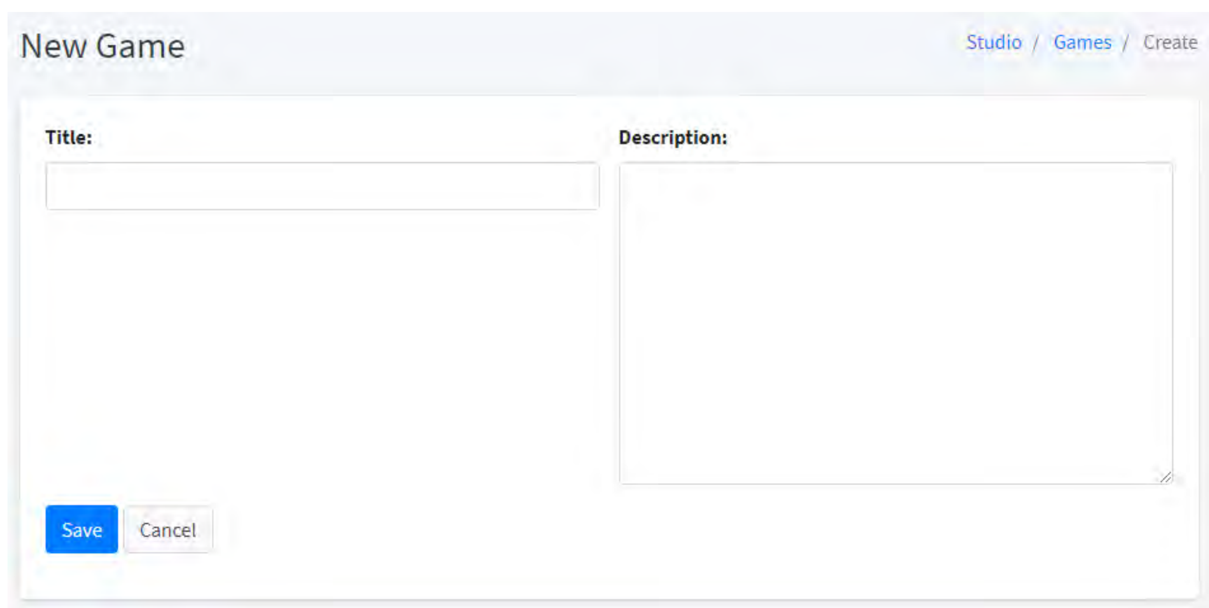

Figure 4.11 Creating a new game

If the Game is successfully created the `Browse` view will be updated and include the game similarly to Figure 4.7. On the right of the newly created game we can see 5 action buttons. These buttons are explained below:

- **`View/Read (Eye)`** – Redirects to Read view of the game. This page contains details about the game including its `title`, `description`, `version` and `installed plugins`.
- **`Plugins (Plug)`** – Redirects to the Game's `Plugins` view. This page contains all the installed plugins including possible configurations that can be made per plugin.
- **`Build and Download (Hammer)`** – Redirects to the Game's `Build` view. This page contains actions for `building` and `downloading` a playable version of the game.

- **`Edit (Notepad)`** – Redirects to the Game's `Edit` view. This page contains details about the game, including its `title`, `description` and `installed plugins` and allows the Creator to perform `edits` and `updates` on these elements.
- **`Delete (Trash)`** – Deletes the Game.

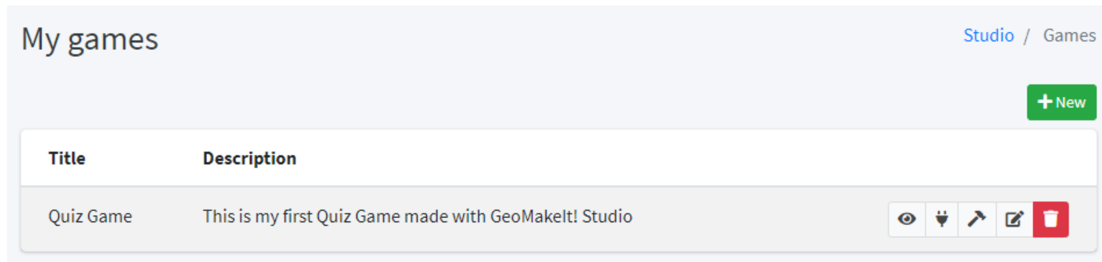

Figure 4.12 Games browse page in GeoMakeIt! Studio

(Configuring GeoMakeIt! API and other Plugins 4.4.b)

The GeoMakeIt! API is built as any other GeoMakeIt! Plugin, with the only exception that it is required to be installed in every GeoMakeIt! Game. We will be using the GeoMakeIt! API to see how we can configure it and any other GeoMakeIt! Plugin inside the GeoMakeIt! Studio. Everything mentioned below can apply to any other GeoMakeIt! Plugin.

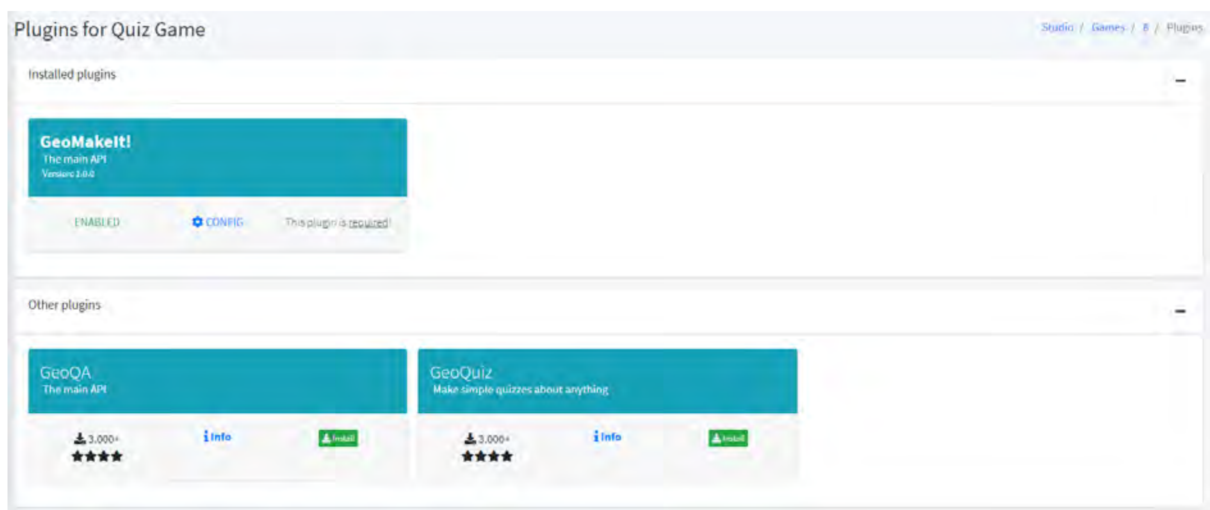

Figure 4.13 Installed and available plugins per game

By clicking the `Plug` button from the `Browse` view page, the user is redirected to the `Game's Plugins` view. From this location the Game Creator can `install` or `uninstall plugins`, `read their information` or `edit their configurations`. In this case, we will proceed to opening the GeoMakeIt! API's configuration by clicking the "`Config`" under the GeoMakeIt! Plugin panel seen in Figure 4.13.

Figure 4.14 Per plugin information and configurations

We get redirected to the GeoMakeIt! plugin's information page, which contains info about the currently installed plugin, such as its `title`, `description`, `author` and `version`. On the left side we have 3 tabs. The first, called "`Information`", is the place we got redirect at and contains the plugin's information. The second one called "`Configs`" contains available configurations that must be overridden by the Game Creator and the last one, called "`Strings`", contains localizable words or phrases. We will focus only in the Configs section, since this is the most important for configuring any plugin.

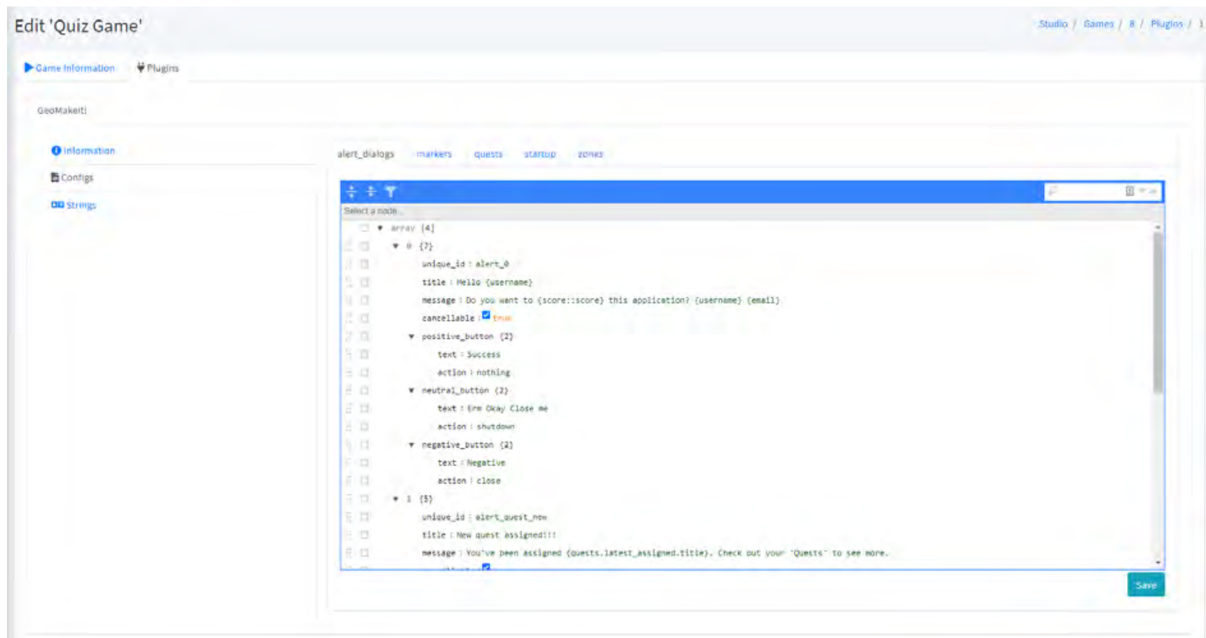

Figure 4.15 Configuration page for the GeoMakeIt! API

The "`Configs`" section offers a flat catalog of configurable `JSON` settings provided by the Plugin Developer for the Game Creator to personalize and setup. On the top of the panel we can see the available configurations, in this case `alert dialogs`, `markers`, `quests`, `startup` and `zones` which are provided by the GeoMakeIt! API. Selecting any of these panels will load the JSON configuration contained in that file for the Game Creator to personalize to his needs.

It must be noted that this design is only for the concept version of the thesis and a more user friendly and richer environment will be designed for a future version of GeoMakeIt!.

(Managing other Plugins 4.4.c)

Other plugins can be managed in a similar manner as the GeoMakeIt! API. The only difference provided between these plugins and GeoMakeIt! API is the inability to delete it. Visible in Figure 4.16 and Figure 4.17 is the GeoMakeIt! API with the message “`This plugin is required`”, while other available plugins can be `installed` or `uninstalled` at the Game Creator’s will.

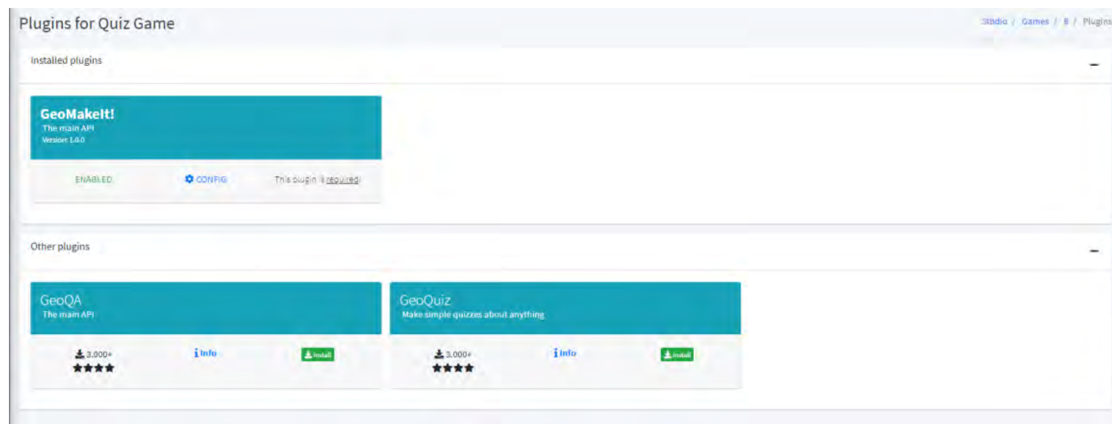

Figure 4.16 GeoMakeIt! API is required

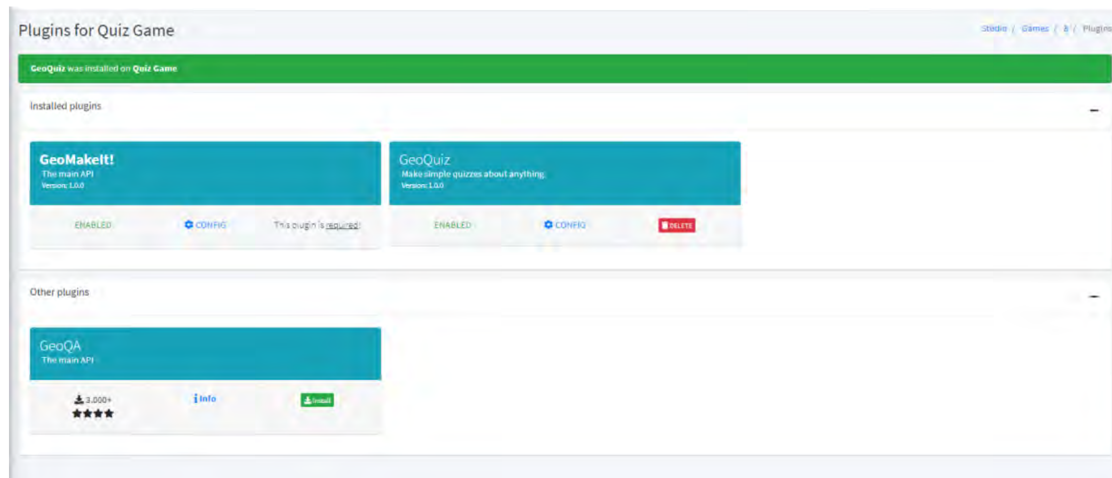

Figure 4.17 Other plugins can be uninstalled

(Building a Game 4.4.d)

The last step before downloading a playable edition of a Game is building it. To perform this action the Game Creator must browse to the Game’s `Build` view by clicking the “`Build (hammer)`” action that is visible in Figure 4.12. The user will then be redirected to a page similar to Figure 4.18.

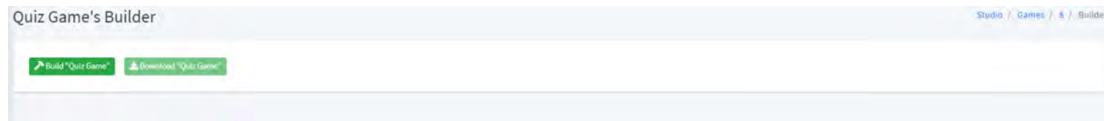

Figure 4.18 Game builder page

The Game Creator is presented with two options, `Build Game` and `Download Game`. The latter one is disabled at first as for it to be available the Game must be built first. To build the game the Game Creator clicks the `Build Game` button and the building process begins. This process is not instantaneous since there are a lot of processes that take place to build the game. As such this job is queued to be run in a GeoMakeIt! Server and when the build succeeds or fails the user is alerted. If the build is successful the `Download Game` button is enabled allowing the Game Creator to download and share the APK.

An in-detail view of how a game is build can be seen in (Game Building Process 4.5.b).

(Website Components 4.5)

There are a lot of components included in GeoMakeIt! Studio to make everything run smoothly and correct. Most of these components are provided by Laravel or other third-party libraries and will not be discussed in detail but rather mentioned in (Libraries used 4.6). Instead, in this section we will only go in detail of a few interesting designs by GeoMakeIt! Studio, such as the `Plugin Upload Process` and the `Game Building Process`.

Both `Plugin Upload Process` and `Game Building Process` are designed as `Laravel Jobs` because they may execute for long periods of times. These processes happen in the server-side and the users will be alerted for their results with the `Laravel's Notification` System.

(Plugin Upload Process 4.5.a)

In this section we will be discussing the `Plugin Upload Process` and the job that handles information extracting. We've seen how a Plugin Developer can upload a compiled GeoMakeIt! Plugin to GeoMakeIt! Studio in (Uploading a Plugin 4.3.b). When the Plugin Developer pressed "`Upload Plugin`", the selected file is uploaded to a temporary folder inside the GeoMakeIt! Studio and using `Laravel's Validation` system is verified that the file is a zipped `.jar/.aar` file. Immediately, the system triggers an "`ExtractUploadedPlugin`" Job which handles further the `plugin's verification` and `extraction of` valuable `data`. This job can take a significant time, so it is queued to be executed by a worker in the background and once finished the user will be alerted with the results.

```
        public function handle()
        {
```

```
            if(!$this->pluginExists()) return;
            $this->unzipPlugin();
            DB::beginTransaction();
            try {
                $this->extractPluginInformation();
                $this->plugin->data()->delete(); // Cleanup old
configs
                $this->extractStrings();
                $this->extractAssets();
                DB::commit();

                $this->cleanupExtracts();

                $this->plugin->user->notify(new GenericNotification(
                    'Plugin upload success',
                    "{$this->plugin->identifier} has been uploaded,
processed and is ready to be used!",
                    GenericNotification::TYPE_SUCCESS,
                    route('studio.plugins.edit', ['plugin' => $this-
>plugin])
                ));
            } catch(Exception $e) {
                DB::rollback();
                $this->fail($e);
            }
        }
```

The “`ExtractUploadedPlugin`” Job executes the following algorithm.

Step 1. **`Check if plugin exists`** – There is a small chance that something may go incorrect while uploading the plugin so we must verify a plugin exists both in database and store physically in the server before we proceed.

Step 2.**`Unzip plugin`** – The plugin is in `.aar` or `.jar` format, which is essentially a compressed folder containing multiple files. We `extract` these files in a temporary folder and separate source from configurations.

Step 3.**`Begin Transaction`** – We must start a database transaction to ensure that if something goes wrong we can rollback our changes and no harm is done.

Step 4.**`Extra Plugin Information`** – From the configurations extracted, we read the `plugin.json` file and update the database if necessary.

Step 5. **`Delete old plugin configurations`** – Delete any previously stored configurations of the plugin.

Step 6.**`Extract Strings and Assets`** – Upload new `configuration data` and `templates` of the plugin.

Step 7.**`Finish up`** – `Commit` database changes, `notify` the Plugin Developer that the plugin upload was successful and finally `cleanup` the temporary folders.

(Game Building Process 4.5.b)

The Game Building Process works similarly to the Plugin Upload Process, but its job is rather the opposite. Instead of extracting the plugin data, the Game Building Process

must take multiple plugins, merge overridden configurations and compress them back to workable .aar / .jar files. Finally, these files must be moved inside the game and built into a complete Android Application using Gradle.

Before talking about the Game Building Process though, we must become aware of some background knowledge of Android Applications. Each Android Application is built upon a base project. We can imagine that base project as the main() of a C program, the first piece of code that will be called once the application is run. In our case, GeoMakeIt! provides a default, hardcoded, base project on which plugins are hooked into. This base project is stored as a prototype inside the GeoMakeIt! server and is deep copied to a temporary location when necessary to create a new game.

Let's take a deeper look into the algorithm for building the game.

```
public function handle()
{
    $this->game->release_file = null;
    $this->game->status = Game::STATUS_BUILDING;
    $this->game->save();

    // TODO: Clean old file
    $this->prepareDirectory();
    $this->configureStrings();
    $gradle_plugins = $this->assemblePlugins();
    $this->assembleGradleConfig($gradle_plugins);
    $this->buildRelease();
}
```

Step 1. **Set status to building –** The first step to building the project is to clean any data regarding the old release and to set the game status as building. There are 4 statuses regarding the game build:

a. **STATUS_NOTHING** – Symbolizes that no status has been assigned yet.
b. **STATUS_BUILDING** – Symbolizes that the game is being built.
c. **STATUS_RELEASE** – Symbolizes that the game has been built and is available for download.
d. **STATUS_ERROR** – Symbolizes that an error has occurred while building.

Step 2. **Prepare Directory** – Creates a deep copy of the `prototype/base` GeoMakeIt! project which will be the final Android Application.

Step 3. **Configure Strings** – Appends some game data into `Strings.xml` of the Android Project, such as the name of the game, `description`, `version` and `build type`.

Step 4. **Assemble Plugins** – For each installed plugin, `extract` its data, take the `source code` and `merge` it with the `overridden configurations` for that plugin. Finally, `re-compress` the plugin and move it to the cloned project's library folder.

Step 5. **Assemble Gradle Config** – Override `gradle config` with extra data provided either by GeoMakeIt! Plugins or by GeoMakeIt! API.

Step 6. **Build Release** – Build the final APK using `Gradle's assemble command`. This action can take a significant amount of time. When the command reaches the end of its execution, the Game Creator is `notified` of

the results and if successful, the Game Creator can `initiate the download` of the built APK.

CHAPTER 5: GeoQuiz – In depth look into a complete plugin

(Introductory 5.1)

GeoQuiz is the first complete plugin created for GeoMakeIt!, ready for release and modification. GeoQuiz allows a Game Creator to make quizzes, ask a player questions and reward them when correct while punishing them when they make mistakes. Even time could be of an issue since players will have specific time to answer each question. GeoQuiz provides a highly configurable environment, where every option can be changed, from time, to hints, to button texts.

(How to install GeoQuiz 5.1.a)

GeoQuiz is provided on the GeoMakeIt! Studio available directly for implementation into any game! Simply install it to your game and a list of example quizzes will be available for you to try out.

(GeoMakeIt! Studio options 5.1.b)

GeoQuiz provides 4 configurable files for the Game Creator to personalize his game. These files will be discussed shortly. Since those configurable files are exported to GeoMakeIt! Studio, the following interface is available for the Game Creator.

Figure 5.1 Example of a GeoMakeIt! Studio configuration

(Final built Quizzes examples 5.1.c)

Figure 5.2 Quiz activity

In this Figure we can see an example of a question that is included in a quiz. This contains the following components:

- Timer that counts down the time left to answer.
- An optional image that can be shown to the player.
- The question that is being asked.
- A set of available answers, which can be either in a form of a multiple-choice answer or a fill the blank answer.
- An optional hint that will be visible if the player chooses a wrong answer.

All these components can be edited online inside the GeoMakeIt! Studio and configured depending on the desired gameplay by the Creator.

(Source Code and Components 5.2)

Now that we have understood what GeoQuiz is and how a Creator can implement and use it in his game, we can have a closer look at the source code and how it works exactly. We will cover more of the plugin-specific implementations while showing of in parallel how GeoMakeIt! API assists into releasing the final form of the plugin.

(Directory Structure 5.2.a)

Here we have an oversight of how the GeoQuiz plugin is organized. Some Plugin Developers will choose to keep this same directory structure since it is simple and easy to understand, while others will choose a slightly different structure to fit specific needs. With that being said, GeoMakeIt! recommends keeping a similar structure so the plugins can be uniformly made and even managed, updated and upgraded by the community if chosen for a plugin to be open source.

- ➔ `AndroidManifest.xml` – Contains permissions and registers activities
- ➔ `assets/geoquiz` – Contains data and configuration files
  - ➔ `defaults.json` – Default configurations for quizzes, questions, activities etc.
  - ➔ `messages.json` – Default messages for common quiz actions
  - ➔ `plugin.json` – Information for GeoMakeIt! Studio
  - ➔ `questions.json` – Question data for database or offline usage
  - ➔ `quizzes.json` – Quizzes data for database or offline usage
- ➔ `java/com/geomakeit/geoquiz` – Package: com.geomakeit.geoquiz
  - ➔ `activities` – Package: com.geomakeit.geoquiz.activities
    - ➔ `MainActivity.kt` – Source for Main Activity
    - ➔ `FragmentQuestion.kt` – Source for Question Fragment

- ➔ `configurations` – Package: com.geomakeit.geoquiz.configuration
  - ➔ `defaults` – Package: com.geomakeit.geoquiz.defaults
    - ➔ `DefaultsConfigFile.kt` – Provides access inside defaults.json
    - ➔ `FileViewSection.kt` – likewise
    - ➔ `HintingSection.kt` – likewise
    - ➔ `MultipleChoiceView.kt` – likewise
    - ➔ `QuestionsSection.kt` – likewise
    - ➔ `QuiizesSection.kt` – likewise
    - ➔ `TimerSection.kt` – likewise
  - ➔ `Messages.kt` – Provides access inside messages.json
- ➔ `enums` – Package: com.geomakeit.geoquiz.enums
  - ➔ `EQuestionType.kt` – Available types of questions (multiple choice, fill the blank etc.)
- ➔ `models` – Package: com.geomakeit.geoquiz.models
  - ➔ `questions` – Package: com.geomakeit.geoquiz.models.questions
    - ➔ `AQuestion.kt` – Abstract structure for a Question
    - ➔ `AMultipleChoiceQuestion.kt` – Abstract structure for Multiple Choice Question (MCQ)
    - ➔ `MultipleChoiceQuestion.kt` – Defines structure for MCQ
    - ➔ `MultipleChoiceAllQuestion.kt` – Defines structure for MCQ (All type)
    - ➔ `MultipleChoiceAnyQuestion.kt` – Defines structure for MCQ (Any type)
    - ➔ `FillQuestion.kt` – Defines structure for Fill Question
  - ➔ `quizzes` – Package: com.geomakeit.geoquiz.quizzes
    - ➔ `Quiz.kt` – Define structure for a Quiz
  - ➔ `results` – Package: com.geomakeit.geoquiz.results
    - ➔ `AResult.kt` – Abstract structure of a result
    - ➔ `QuestionResult.kt` – Structure for a question result
    - ➔ `QuizResult.kt` – Structure for a quiz result
- ➔ `GeoQuiz.kt` – Plugin's main (extends GeoMakeIt! Plugin)

➔ `res` – Android's resource folder
- ➔ `layout` – Android's layout folder
  - ➔ `geoquiz_activity_main.xml`
  - ➔ `geoquiz_fragment_question.xml`
- ➔ `values`
  - ➔ `strings.xml`

(Resources and Manifests 5.2.b)

We'll start taking apart the GeoQuiz starting of the Android's common resource files. These are normal resource files and require no significant changes for them to work with GeoMakeIt!, other than prefixing a few of them with the plugin's `identifier`.

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.geomakeit.geoquiz">
    <application>
        <activity android:name=".activities.MainActivity">
        </activity>
    </application>
</manifest>
```

In GeoQuiz's case the `Android Manifest` only does one thing, registers the `Main Activity` as an Android Application Activity.

strings.xml

```
<resources>
    <string      name="geoquiz_public_msg_question_on_correct">Your
answer was correct!</string>
    <string
name="geoquiz_public_msg_question_on_wrong">Unfortunately,        you
failed to answer this one.</string>
</resources>
```

The `Strings.xml` contain two public messages available for the Game Creator to configure. These are currently not functional but will be in a future version allowing the user to also create localized messages depending on the user's country.

Since GeoQuiz needs to make these public and editable to the GeoMakeIt! Studio, all strings are prefixed with "`geoquiz_public_`" to make it public.

geoquiz_activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/FrameLayout_Fragment_Question"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".activities.MainActivity"/>
```

This is the `main activity` of the GeoQuiz plugin. It serves just as a frame to contain inside it any `Quiz` or `Question`.

geoquiz_fragment_question.xml

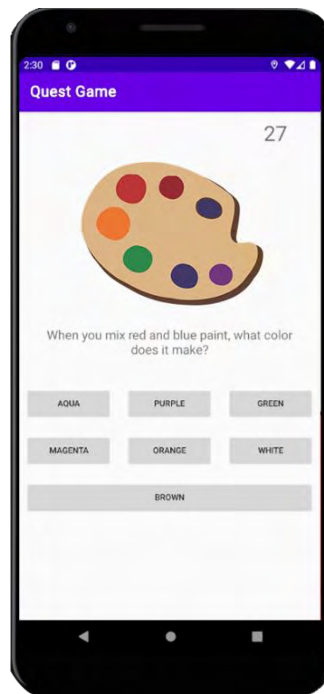

Figure 5.3 Quiz activity

This contains the design for any `Quiz` or `Question`. It is a `Vertical Linear layout`, split into `10 vertical parts` that are weighted differently depending on how much space is required by each one.

(Configurations 5.2.c)

To continue, let's have a closer look to all the configuration files, explaining some of their usages and then seeing how we can pass them to the plugin and use them.

plugins.json

```
{
  "main": "com.geomakeit.geoquiz.GeoQuiz",
  "title": "GeoQuiz",
  "short_description": "Make simple quizzes about anything",
  "description": "Challenge your players in your own quiz game. 
Create multiple-choice questions or let the players fill the 
blank. Add timers, hints or even allow players to make a few 
mistakes till they get the answer right. Reward or punish players 
per correct/incorrect answer or quiz!",
  "version": "1.0.0",
  "author": "Vasilis Dimitriadis",
  "gradle_implementations": []
}
```

Default Options

defaults.json file

The defaults.json contains all the global settings for the GeoQuiz plugin. We can see that it separates the file into 6 sections: questions, quizzes, timer, hinting, multiple choice view and fill view.

All these settings can be overridden on per quiz or per question in the upcoming files quizzes.json and questions.json

The question section defines global settings for all questions.

```
{
  "questions": {
    "allowed_wrong_answers_per_question": 0,
    "reward_on_correct": [],
    "reward_on_wrong": []
  },
  "quizzes": {
    "next_question_delay": 3,
    "ratio_for_success": 1.0,
    "reward_on_success": [],
    "reward_on_failure": []
  },
  "timer": {
    "enabled": true,
    "default_time": 30
  },
  "hinting": {
    "enabled": true,
    "wrong_answers_to_show_hint": 1
  },
  "multiple_choice_view": {
    "answers_per_line": 3
  },
  "fill_view": {
    "case_insensitive": true,
    "normalize_spaces": true,
    "can_answer_blank": false
  }
}
```

A detailed explanation of the above options is given below:

➔ allowed_wrong_answers_per_question: How many wrong answers can a player give before the question is considered as "failed to answer". Usually this setting must be 0, unless hinting is available or if there are hard to answer questions.

➔ reward_on_correct: Commands to execute if a question is answered correctly.

➔ reward_on_failure: Commands to execute if a question is answered wrongly.

The quizzes section defines global settings for all quizzes.

➔ next_question_delay: Delay in seconds to wait before proceeding to next question after successful or failed answer is provided.

➔ `ratio_for_success`: What percentage of questions must be answered in order to consider the whole quiz a success. For example, if we have 4 questions and we need at least two correct answers to consider this quiz correctly answered, then we must provide a ratio for success of 0.5.

➔ `reward_on_success`: Commands to execute if a quiz is completed successfully. These rewards are executed on top of the per answer rewards.

➔ `reward_on_failure`: Commands to execute if a quiz is failed. These rewards are executed on top of the per answer rewards.

The `timer section` defines global settings regarding the time available per question to answer.

➔ `enabled`: Defines if the timer system is enabled or not globally. If it is disabled and a specific question has a timer, that timer would be ignored. If it is enabled and a specific question requires unlimited timer, the time in that question must be set to 0.

➔ `default_time`: The default time a player has, to answer, per question.

The `hinting sections` defines properties regarding the hinting functionality.

➔ `enabled`: Defines if the hinting mechanism is enabled globally, similarly to the timer options.

➔ `wrong_answer_to_show_hint`: Defines the number of wrong answers a player must make per question to show the hint

The `multiple-choice view and fill view` define properties that affect how the user interface works depending on the question type.

➔ `answers_per_line`: Defines how many button (answer options) are available per single line when a player is answering a multiple-choice question.

➔ `case_insensitive`: Defines if the answer that is given by the player isn't case sensitive.

➔ `normalize_spaces`: Defines if the answer that is given by the player is affected depending on double spaces. A true value would convert an answer such as "*Hello____there*" (with 4 spaces) to the answer "*Hello there*" (with one space), hence, normalizing it.

➔ `can_answer_blank`: Defines if the user can answer nothing or spaces.

Defaults Config File class

```
object DefaultsConfigFile {
    var questions: QuestionsSection
    var quizzes: QuizzesSection
    var timer: TimerSection
    var multipleChoiceView: MultipleChoiceViewSection
    var fillView: FillViewSection
    var hinting: HintingSection

    init {
        try {
            val configuration = Main.getConfig("defaults.json")
            // Load all JSON
```

```
                val json = configuration.asJSONObject()
                // Load Sections
                val questionsJSON = json.optJSONObject("questions")
                                    ?: JSONObject()
                questions = QuestionsSection(
                                JSONUtil.jsonTo(
                                questionsJSON.toString()))

                val quizzesJSON = json.optJSONObject("quizzes")
                                  ?: JSONObject()
                quizzes = QuizzesSection(JSONUtil.jsonTo(
                            quizzesJSON.toString()))

                        ...

            } catch (e: Exception) {
                questions = QuestionsSection(mapOf())
                quizzes = QuizzesSection(mapOf())
                timer = TimerSection(mapOf())
                        ...
            }
        }
}
```

The `Defaults Config File` Class can be found under the package `com.geomakeit.geoquiz.configurations.defaults` and it is responsible for loading the global options for GeoQuiz. It attempts to first load the `defaults.json` file and then parses each subsection to its own class that reads the contents of the JSON file. If it fails to load due to the `defaults.json` missing or invalid configurations, the `defaults.json` will load its own default values instead of the ones provided by the Game Creator until the issue is fixed.

```
    class TimerSection {
        companion object {
            var enabled: Boolean = true
                private set
            var defaultTimer: Int = 30
                private set
        }
        constructor(map: Map<String, Any>) {
            enabled = map["enabled"] as? Boolean ?: true
            defaultTimer  =  (map["default_time"]  as?  Double)?.toInt()
?: defaultTimer
        }
    }
```

When the configuration is loaded, the Plugin Developer can use it to check contents of the `config.json` file by calling the specific section in question. For example, if we would like to see if a the timer is enabled, the Plugin Developer could call “`DefaultsConfigFile.timer.enabled`” or “`TimerSection.enabled`”.

Messages

messages.json configuration file

```
{
  "question_on_correct_answer": "Your answer was correct!",
  "question_on_wrong_answer": "Unfortunately, you failed to
                              answer this one.",
  "timer_out_of_time": "You've run out of time!",
  "fill_view_answer": "Answer!"
}
```

This file contains default messages that are given by the user interface. This file will be merged with `Strings.xml` in a future version to provide localization.

➔ `question_on_correct_answer`: Message to show when an answer is correct.
➔ `question_on_wrong_answer`: Message to show when an answer is wrong.
➔ `timer_out_of_time`: Message to show when the timer runs out.
➔ `fill_view_answer`: The text of the answer button when a question is of type fill.

Messages object

```
object Messages {
    private var map: Map<String, Any>
    init {
        map = try {
            val json = Main.getConfig("messages.json").asString()
            JSONUtil.jsonTo(json);
        } catch(e: Exception) {
            mapOf()
        }
    }

    val QUESTION_ON_CORRECT_ANSWER =
            map["question_on_correct_answer"] as? String
            ?: "Your answer was correct!"
    val QUESTION_ON_WRONG_ANSWER =
            map["question_on_wrong_answer"] as? String
            ?: "Unfortunately, you failed to answer this one."
    val TIMER_OUT_OF_TIME =
            map["timer_out_of_time"] as? String
            ?: "You've run out of time!"
    val FILL_VIEW_ANSWER = map["fill_view_answer"] as? String
            ?: "Answer!"
}
```

Messages object attempts to get the `messages.json` file and convert its information to a map object. If it fails to load the file or if an option is missing the default options will be loaded.

questions.json

Moving on to the questions data file, here all the possible questions the game might show reside, along with example templates to show a Game Creator the possible choices for each question.

This file contains 11 templates to display some of the possible combinations that a Game Creator might do, but in our case we will compress them into all the possible options and values a question might have. To keep everything short we will show only the first two options in this case.

```
  "question_1": {
      "question": "When you mix red and blue paint, what color does
                 it make?",
      "question_type": "multiple_choice",
      "possible_answers": [
        "Aqua",
        "Purple",
        "Green",
        "Magenta",
        "Orange",
        "White",
        "Brown"
      ],
      "correct_answer": 1,
      "image": "@android:drawable/btn_star_big_on"
    },
    "question_2": {
      "question": "Which American company was the first to reach
                 1$ Trillion?",
      "question_type": "multiple_choice",
      "possible_answers": [
        "Microsoft",
        "GeoMakeIt!",
        "Apple",
        "Samsung",
        "Xiaomi"
      ],
      "correct_answer": 2,
      "allowed_wrong_answers": 1,
      "image": "@drawable/geomakeit_logo"
    },
                        ...
```

A detailed explanation of the above options is given below:

➔ **(required)** `question_identifier`: Replace this with the unique identifier of the question object, such as "question_1", "awesome_question", "how_much_does_1_plus_1_equal_to" etc.

➔ **(required)** `question`: This is the question that will be shown to a player. Keep it sort and simple.

➔ **(required)** `question_type`: What type of a question is this? There are 4 types supported in this version of GeoQuiz:

- ➔ `multiple_choice` → Classic multiple choice, where one of the answers is correct.

    - multiple_choice_any → Multiple choice, but any of X answers could be correct and selecting one of them will result on question success.
    - multiple_choice_all → Multiple choice, but X answers are correct, and all must be selected in order for the quest to be considered successful.
    - Fill → The player must write the answer.
- **(required)** correct_answer: Determines which answer is the correct one depending on the question_type.
    - multiple_choice → The index of the possible answer is the correct one, where the starting index is 0.
    - multiple_choice_all and multiple_choice_any → A list of indexes, answers, is provided.
    - Fill → A string containing the correct answer is provided
- **(required if multiple choice)** possible_answers: Available only when the question_type is one of the multiple_choice ones, a list of possible answers is provided. Each one of these will be formed into a single button to for the player to choose from.
- image: An optional path for the question to show an available image. If not provided, no image will be shown. Such could be the "@drawable/geomakeit_logo"
- allowed_wrong_answers: Optional variable for overriding the allowed wrong answers that can be made. If 0, no wrong answers are allowed, while if anything more than that gives X changes for the player to answer the question. If -1 is selected, the player can answer with no limits.
- hint: Text to be shown if a player makes a wrong mistake as a hint to guide him to success. Option allowed_wrong_answers must be at least "1" for this to work.
- timer: Time in seconds the user has available to answer the question. If 0, the user has unlimited time. If any value above of 0 is selected and timer is disabled globally through defaults.json, no timer will be shown.
- case_insensitive, normalize_spaces and can_answer_blank: Options used when a question is of type fill, overriding the default settings and checking if the answered provided by the player is correct even if the capitalization doesn't match or there are extra spaces.
- reward_on_correct and reward_on_wrong: List of commands to be executed if a player answers correctly or wrong a single question.
- message_on_correct_answer and message_on_wrong_answer: Overrides the default messages shown if the correct or wrong answer is selected.
- view_answers_per_line: Overrides the defaults answers per line provided globally when a view is of multiple choice.

quizzes.json

The final data file is the quizzes.json file. This contains a list of quizzes that can be assigned to a player. Quizzes are simply collections of questions and as such their representation is very easy and understandable.

Similarly to the questions.json, we will provide a compressed representation of quizzes since there are multiple ways that they can be configured.

```
{
  "quiz_1": {
    "questions": [
      "question_1",
      "question_2",
      "question_3",
              ...
    ],
    "next_question_delay": 1,
    "ratio_for_success": 0.7
  },
  "quiz_2": {
    "questions": [
      "question_1",
      "question_2",
      "question_4",
      "question_6",
      "question_8",
      "question_10"
              ...
    ]
  },
              ...
}
```

A detailed explanation of the above options is given below:

➔ **(required)** `quiz_identifier`: Replace this with the unique identifier of the quiz object, such as “`quiz_1`”, “`10_question_quiz`”, “`awesome_quiz`” etc.

➔ **(required)** `questions`: A list of questions that will be shown in the quiz. These are the unique question identifiers that were created in the `question.json` file.

➔ `next_question_delay`: Delay in seconds after each successful or failed question to move to the next one.

➔ `ratio_for_success`: Percentage of questions that must be answered correctly for a quiz to be considered successful or failed. A value of `0.7` for example, means for every 10 questions, 7 must be correct.

➔ `reward_on_success`: List of commands to be executed as soon as a quiz is finished by a player.

➔ `reward_on_failure`: List of commands to be executed as soon as a quiz if failed by a player, typically for punishing the player.

(GeoQuiz Main Class 5.2.d)

The main class for GeoQuiz is located in package `com.geomakeit.geoquiz` and it is called GeoQuiz. Inside this class we can see the identifier of the plugin, “`geoquiz`” as well as the `onEnable()` method.

```
object Main : Plugin() {
    override val identifier: String
        get() = "geoquiz"

    override fun onEnable() {
```

```
        // Load Configurations
        DefaultsConfigFile
        Messages

        // Load Models: From Files
        AQuestion.loadAllFromFile()
        Quiz.loadAllFromFile()

        // Load Models: From database
        AResult.loadAllFromDB()

        // Register Activities
        ActivityRegistry.register(this, "quiz",
                                  MainActivity::class.java)
    }

    override fun onDisable() { ... }
}
```

In the `onEnable()` method we can see that GeoQuiz loads all configuration files first, then proceeds with loading the data models and finally registering the quiz activity so it can be callable with the command "`geomakeit::activity geoquiz::quiz` <*arguments*>".

(Activities 5.2.e)

GeoQuiz contains a single activity providing a simple `Quiz User Interface`. The activity itself is a wrapper for a `Question Fragment`, leaving room in the future for multiple implementations of different User Interface designs and templates.

Main Activity

The `Main Activity` is the entrance to the `GeoQuiz's User Interface`. It is a blank template containing inside information only about the `Quiz` or `Question` that is about to get loaded and a mechanism to forward the necessary `Quiz` or `Question` to the final fragment.

```
class MainActivity : JActivity(),
                     FragmentQuestion.OnQuestionResultListener {
    companion object {
        const val TYPE_QUESTION = 0
        const val TYPE_QUIZ = 1
    }

    // Is it a question or a quiz?
    private var type by Delegates.notNull<Int>()
    private var force: Boolean = false

    /* For Question */
    private lateinit var question: AQuestion
```

```
        /* For Quiz */
        private lateinit var quiz: Quiz
        private var currentQuestionIndex: Int = -1
        private var correctQuestionsCounter = 0
                        ...
    }
```

Opening the `Main Activity`, we can see that it *extends* the GeoMakeIt! API's `JActivity`, as well as it contains a `Question Result Listener` which we will see shortly. It then defines the two types of data it can show, `Quizzes (TYPE_QUIZ)` and `Questions (TYPE_QUESTION)`.

Depending on the type of data that is loaded it can be either a question, thus attempting to load a `Question` model or a quiz, attempting to load a `Quiz` object. We can see that a question object requires no extra information other than the object itself, while in the case of a quiz we keep track of the `current question index` as well as the `correct question counter`. The first one is used to check our progress within the quiz and to load the next question in line, while the correct questions counter assists into calculating if the quiz was successful or not based on the ratio of success. Finally, a force options is supplied to, letting the activity know if the question or quiz can be forced even if it was successfully answered in the past.

As soon as the `Main Activity` is loaded, the `onCreate()` method is called which checks the intent data that have been passed. The possible data that can be passed in this case are:

➔ `force`: A Boolean value to let the activity know if the question/quiz can be forced to the user, even if he has successfully answered it before.

➔ `type`: The type of the `Main Activity` can be either quiz or question. This type is determined by if the activity has a bundle passed with the parameters of quiz or question. If none of these are provided, the `Main Activity` closes, failing to be launched.

As soon as the type is determined, the `Activity` chooses to either load the question or the quiz. If the load fails, which can happen if for example the quest/question doesn't exist, the again the activity closes. If a quiz is loaded and it exists but there are invalid questions included, those invalid questions are filtered off. Continuing off, if there is no force option and the quest or quiz has already been answered, then the activity is again closed.

If the `Quiz` or `Question` hasn't been closed by any of the previous measures then it is forwarded to the `Question Fragment` using the fragment manager. On each quiz response on the `Fragment Manager`, success or failure is returned using the `onQuestionResult` method. If the question was a part of a quiz, then the result is saved, and the quiz continues to the next question. If though the question was a standalone question the result is saved and depending on if or not it was successful the reward on correct or the reward on wrong commands are triggered.

Finally, when a quiz ends, the percentage of correctly answered per total questions is calculated and if it is bigger than the required ratio for success the reward on success or reward on failure commands are triggered and the result is saved.

The `Question Fragment` contains all necessary data to visually build a question from a quiz in a modular manner. Opening the `FragmentQuestion.kt` we can see that it is a simple Android Fragment with a lot of modularity written on top of it.

```
class FragmentQuestion : Fragment() {
    companion object {
        private const val ANSWER_STATUS_CORRECT = 0
        private const val ANSWER_STATUS_WRONG = 1
        private const val ANSWER_STATUS_TIMER = 2
        private const val ANSWER_STATUS_HINT = 3
    }
         ...
}
```

Starting from the top, we can see that there are some constants that define the answer status of a question, `ANSWER_STATUS_CORRECT`, `ANSWER_STATUS_WRONG`, `ANSWER_STATUS_TIMER` and `ANSWER_STATUS_HINT`. These variables are for forwarding information when a player responds to a question, informing the fragment and main activity of the current answer state, such the answer was correct, the answer was wrong, the time run out or that a hint must be forwarded.

```
                 ...
/* Timer Settings */
private var timer by Delegates.notNull<Int>()
private var timerHandler: Handler = Handler()
private val timerTask = object : Runnable {
    override fun run() {
        if(question.hasTimer() andand !hasAnswered) {
            timer -= 1
             requireView().findViewById<TextView>
                (R.id.TextView_Timer).text =
                     timer.toString()
            if(timer <= 0)
                showAnswerResultText(ANSWER_STATUS_TIMER)
            timerHandler.postDelayed(this, 1000)
        }
    }
}

/* Question Specific Settings */
private lateinit var question: AQuestion
private var wrongAnswersChancesLeft by Delegates.notNull<Int>()
private var wrongAnswerCount = 0
private var hasAnswered: Boolean = false

/* Question Result Listener */
var listener: OnQuestionResultListener? = null
interface OnQuestionResultListener {
   fun onQuestionResult(result: Boolean)
```

```
    }
                          ...
```

Following the answer status constants, we have some question specific settings. `Timer` settings check if the timer is enabled and if it is, it starts the countdown. Then there are variables about the question, checks if it has been answered or not and counters for wrong answers and chances left. Next, the question `result listener` is defined and following that, question specific settings such as which buttons for answers are generated, which text view is generated for fill answers and which buttons have been pressed.

Proceeding then to methods, we start with the `onAttach()` which verifies that the `Main Activity` has the `question listener` attached. Then comes the `onCreate()` method which verifies the question exists and loads the information for the question. The `onStart()` and `onPause()` methods make sure the timer is running correctly when enabled, while the `onCreateView()` inflates the fragment with the question and image.

When it comes to designing the view, the `designView()` method shows or hides the timer and then proceeds to design the view dynamically based on the question type. When the question is of `Multiple Choice` type, then the `designMultipleChoiceView()` will be called, while if it is of `Fill` type the `designFillView()` will be called.

Finally, other helpful private methods such as the `disableButtons()`, `destroyFragment()`, `fillClickListen` and `multipleClickListener` exist, but we will be explaining them here.

(Models 5.2.f)

There are 3 base models for GeoQuiz, `Questions`, `Quizzes` and `Results`.

Questions

`Questions` are the foundational element of any Quiz. Since there are multiple types of questions for GeoQuiz we must ensure a good abstract design for each type to *extend* from. As such, there are 6 Question classes, 2 of each are the Abstract ones and 4 that make the actual Question Types.

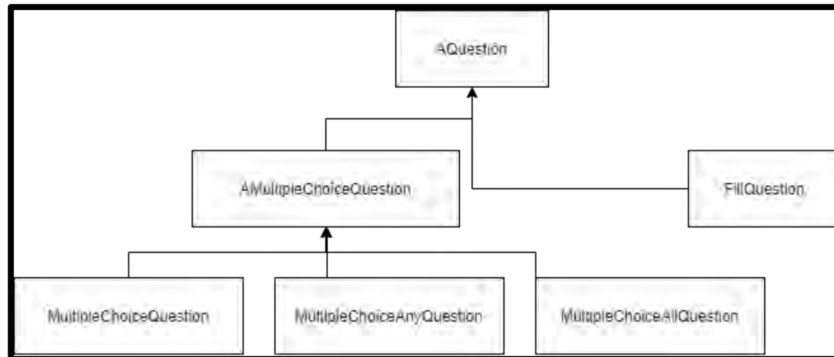

Figure 5.4 Question Hierarchy

Abstract Question class

The `Abstract Question` class is the base of the `Question` Model. It *extends* the GeoMakeIt!'s Model class and as such it can be saved in the database.

The variables inside this class have been already discussed previously in detail, specifically in the configuration section. To summarize those, the Abstract Question class contains configuration for setting up the `question text`, a `timer`, the `maximum number of allowed wrong answers`, customize `messages` for correct and wrong answers, customize `rewards` for correct and wrong answer, `hint` and to show specific images per question.

These settings are parsed to the question object through the `map constructor` and the configuration data previously discussed. The `AQuestion` class also contains private methods to `execute rewards on correct and on wrong`, a function to answer the question, functions to check if an answer is correct or wrong and if there can be unlimited answers for this question. Finally a `toMap()` method is defined to store the data in the database with that specific structure.

The `AQuestion` also contains a companion/static object that *extends* the GeoMakeIt!'s `IModelFactory` and `ILoadable` interfaces, providing the necessary methods to `create`, `add`, `store` and `save` question objects into the database or `load` them from there.

Fill Question class

The `Fill Question` class defines the structure for a `Question` of type "`Fill the blank`". There are just some extra variables to provide that functionality such as, adding the correct answer of type string, settings to check `for case insensitivity`, `normalizing spaces`, checking if can `answer blank` or even setting the answer `button text`. The default variables are again taken from the `Fill View Section` of the `defaults.json` file.

When it comes to overriding the `Question` class other than defining the variables two more methods must be changed. The `isCorrectAnswer()` method must do its necessary changes and check if the answer is correct or not by parsing the answer, checking if blank and normalizing it if necessary. The `toMap()` method must also be updated to add the extra variable options.

Abstract Multiple-Choice class

Since there are many multiple-choice options when it comes to question types it is wise to create an abstract class that incorporates common variables. As such the `AmultipleChoiceQuestion.kt` adds the possible answers list and the view answers per line option while updating the `toMap()` method again.

MultipleChoiceQuestion, MultipleChoiceAny and MultipleChoiceAll

The difference between those 3 implementations of `MultipleChoices` is the correct answer type and the `isCorrectAnswer()` method.

For the default `Multiple Choice` Question, the correct answer is the index of the correct answer and the only checking we need to do for the `isCorrectAnswer()` method would be to check if the answer provided is int and if it is the same as the correct answer.

For the `Any Multiple-Choice` Question, we follow a similar approach to the default `Multiple Choice` Question, but we change the correct answer to a list of integers and check if the index of the clicked button is one of the acceptable ones.

Lastly, for the `All Multiple-Choice` Question we follow exactly the similar approach as the `Any,` but we also store the selected answer in the `Question Fragment` to verify all the required answers have been clicked.

Quizzes

`Quizzes` are just a batch of questions in order and as such their structural representation is simpler than the Questions.

The Quiz class, located in `com.geomakeit.geoquiz.models.quizzes`, *extends* the GeoMakeIt! Model class. Similarly, to the aforementioned `AQuestion` class, since this is a GeoMakeIt! Model we implement some of its features such as the `toMap()`, `dbModel` and `constructors`. We will not go into detail about these since a lot of similar models have been explained.

The `Quiz` class contains a list of `AQuestions` and can incorporate any `question` model previously discussed. It also three extra variables, `reward on success` and `failure` for storing the commands that must be executed when a quiz `is completed` and the `next question delay` setting.

Finally, as most of the GeoMakeIt! Models, it contains a companion object extending the `IModelFactory` and `ILoadable` interface to load and unload data from the data files and database.

Results

The last models in GeoQuiz represent how the results are saved. There are two types of results, `Quiz Results` and `Question Results`. Since we have seen multiple models until now we won't go into detail and we'll just talk only about the data that these models store.

The common attributes between a `Question Result` and a `Quiz result` are stored inside the `AResult` class. These attributes would be the result itself, meaning if the player has completed or failed the quiz or question, the result type, determining if the result is a question result or a quiz result and the date the result was submitted.

The `Question Result` which *extends* the `AResult` class just adds a single attribute, the question itself. When saved into the database the question's unique identifier is also saved to show which question was answered.

Lastly, the `Quiz Result`, similarly to the `Question Result`, *extends* the `AResult` class and adds a single attribute, the quiz. When saved to the database, the quiz's unique identifier is also saved to show which quiz was answered.

(Future Implementations 5.3)

Figure 5.5 Alternative look to a Quiz Activity

GeoQuiz is a complete plugin but, as anything in this world, it is far from perfect. While there are no immediate plans for upgrading GeoQuiz, there are some thoughts on what must come next. GeoQuiz is indeed a complete plugin when it comes to its quiz functionality, but it is lacking appearance. When the time comes, a UI redesign would be essential to also prove its likability other than its functionality. As such the UI redesign must consider multiple factors to fit all necessary games and if possible even provide templates for Game Creators to customize for their needs. The configurations that could be created are endless, depending on the Game Creator's knowledge too. To grasp the unlimited possibilities let's just consider a single component of the GeoQuiz, the timer.

The timer currently is located on the top right of the Question Fragment. First, this must be able to change position, such as to be placed on the top left, bottom left, bottom right or even middle. Then, other than positioning we must consider multiple design options, such as what color do we want the text to be as? Finally, let's consider other complete redesigns on the timer such as the following. Instead of a text timer we might want to consider a progress bar timer, or maybe a circular progress bar timer. That circular progress bar timer could also contain inside the text number of that timer etc.

These and many other such UI redesign thoughts must be considered when the time comes. But for those redesigns to happen, GeoMakeIt! Must mature a bit as a software and create extra standards when it comes to developing plugins as well as provide functionality for easier configurations of that type.

(Final Thoughts 5.4)

When it comes to creating a fully modular and configurable plugin we can see that our imagination is the limit. A Plugin Developer must make choices to decide which components of his plugin can be modular or must be modular. The more modular and configurable a plugin is, the more personalized per game it is implemented to will be. But at what cost? The more we personalize a plugin, the more information we force to a Game Creator to understand. As such we want to find an equilibrium and allow a Game Creator to customize it as much as possible with

out making it though confusing. A nice way GeoQuiz has implemented this is with hiding configurations behind default values and allowing the user to do specific changes only when he is ready and understands what those changes mean. At the same time,

this option gives the Game Creators many choices when it comes to deciding how they can personalize this plugin to their needs.

CHAPTER 6: Creating a plugin from scratch

(Prerequisites 6.1)

GeoMakeIt! is a wrapper around the Android's API and as such it doesn't really have any requirements of its own. The operating systems required are based on the IDE of your choice, while the only actual requirement to create GeoMakeIt! Plugins is to sign up for an account at the GeoMakeIt! Studio.

(Operating System 6.1.a)

The requirements are set by the IDE of your choice. Here are some common IDE's used to develop Android apps and therefor GeoMakeIt! Plugins.

- `Android Studio`
  - **`Windows`** `7/8/10 (64bit)` or **`Mac OS`** `X 10.10-10.14`
  - **`4GB`** `RAM` minimum, `8GB RAM` recommended
  - **`2GB disk space`** minimum, `4GB` recommended
  - **`1280x800`** `minimum screen resolution`
- `Eclipse`

More information about specific IDE requirements can be found in each IDE's individual page.

This tutorial for GeoMakeIt! is based on the Android Studio IDE and using the `Kotlin Language`; Thus, it is recommended to use the `Kotlin Language` and if possible the Android Studio for your project.

(Prior Knowledge 6.1.b)

It is strongly recommended that before you start writing GeoMakeIt! Plugins there has been some prior experience with creating `Android Studio` Applications and that you've used the `Kotlin` or `Java Programming Language`, as we are not going to discuss in this tutorial a lot about those. We will only focus into creating an introductory GeoMakeIt! Plugin project from start to finish.

(Sign-in/Sign-up to GeoMakeIt! Platform 6.1.c)

This is not a required step for creating GeoMakeIt! Plugins but it is a required step if you'd like to publish them for the public to use later and since the whole idea for GeoMakeIt! Is to create `re-usable`, `modular` and `configurable` plugins why would you miss on that?

Signing-up for the GeoMakeIt! Platform is free and can be easily done here, http://geomakeit.com/signup.

If this is your first time signing up, make sure to sign-up for a developer account. If you have already signed-up for a Game Creator account previously.

(Hello World 6.2)

Now that you've downloaded the `Android Studio`, installed all necessary updates and probably have signed-up in the GeoMakeIt! Platform, we are ready to ready to start creating our GeoMakeIt! Plugin.

To do that, we'll first need to download the GeoMakeIt! API. There are two ways to create a plugin; Starting from a `Starter Project` or `Starting from Scratch`. The second one is considered a more advanced method and we will not use it in this tutorial, but you can freely check the documentation to see how that works too.

The first step is to download the GeoMakeIt! Starter Project located here:

https://github.com/GeoMakeIt/Demo-Project

After downloading the Starter Project, we must do the following:

1. Open Android Studio
2. Open Project
3. Navigate to the downloaded GeoMakeIt! Project

The starter project already contains a version of the latest GeoMakeIt! API located in the `app/libs` directory.

Keep in mind that if you choose to keep developing a specific project for a long time you might be required to update the GeoMakeIt! API manually from time to time.

This guide is also available on the GeoMakeIt! online documentation. Make sure to visit that webpage to get the latest guide according to the current GeoMakeIt! API release.

(Creating your plugin 6.2.a)

On Android Studio, while you're in the GeoMakeIt! Starter Project:

1. File > New > New Project.
2. Select "No Activity" and click next.
3. Fill name, package, save location and choose the language of your choice.
4. Make sure that your minimum SDK is "API 16" and click Finish.

When selecting the package name, please also make sure to choose a package different than `com.geomakeit.*`.

(Hello World 6.2.b)

Let's try to recreate the classic "`Hello World!`" example. Now, in our case, we are just going to log a message to our Logcat and thus prove our plugin can be loaded and is functional.

In `java > com.your_package.….` create a new Kotlin file `Main.kt`.

```
object Main: Plugin() {
    override val identifier: String
        get() = "hello_world_example"

    override fun onEnable() {
        logger.i("Hello World!!")
    }

    override fun onDisable() {
        logger.d("Bye World:(")
    }
}
```

From the source provided above we must take the following notes:

1. The variable `identifier` is very important! This "`identifier`" will be used in a lot of parts of your plugin and it is a `unique identifier` separating your plugin from others. As such keep in mind that this must be something unique and descriptive. It will be also used to execute commands so most of the times you'll need to choose something short. There is an upcoming section explaining how to choose a valid `identifier`, please read that carefully. Whenever we'll be talking about "`Plugin Identifier`", this variable is what we will be referring to.
2. The name `Main.kt` is optional, but it is just a convention between other GeoMakeIt! Projects. Another common convention would be to name your name as your project. For example, if you were making a project called `GeoQuiz` you could name your main instead of `Main.kt` to `GeoQuiz.kt`.
3. Instead of **`class`** `Main`, we are using **`object`** `Main`!
4. Make sure to *extend* `Plugin()` of `com.geomakeit.api.plugin.Plugin`

We could build the plugin right now and be done with it, but we haven't connected it to GeoMakeIt! Starter project. As such, there is no way to test what we've made yet. Let's do that next.

(Choosing an identifier 6.2.c)

Each plugin is required to have a `unique identifier` to be able to communicate with other plugins. On top of that, the identifier serves as the gate for your user interactions, for example, `commands` and `placeholders`. Therefore, make sure to choose the appropriate identifier for your plugin.

Any identifier must follow the following rules:

- Must be at least 3 characters length and at most 32.
- Must be all lowercase.
- Must start with an alphabetic character.
- After that, it can contain any alphanumeric character and underscores.

Keep in mind that people remember easier the following:

- Descriptive words
- Short words
- Common words
- Words that don't contain numbers or symbols

We can think of it as creating the world's most insecure password.

For example, if you were creating a plugin that can create, manage and display a player's score, you could name it one of these: "score", "score_api", "epic_score", "player_score".

After choosing your plugin's identifier, make sure to use that whenever we mention "plugin identifier". You'll get used to it, trust us.

Even though you have chosen your identifier, you haven't yet verified that it is unique or not. To do that you must register your plugin's identifier which will be one of our last steps, Uploading and Publishing your plugin. Typically, this step should be done first to ensure your identifier is unique and avoid accidental mishaps or need to change the identifier anywhere that it is necessary. Since though this is your first time developing, we are expecting that you'll be just trying out GeoMakeIt! and uploading a test application that will be uploaded and published as a test and rather not your final idea / plugin.

(Running / Testing your plugin 6.2.d)

To run or test our plugin, we will need to connect it with the GeoMakeIt! Starter project. It is simple and straightforward:

1. On Android Studio, select: File > Project Structure > Dependencies > Module: app.
2. + > Module Dependency.
3. Select your plugin and press ok.

Now your project is included in GeoMakeIt! Starter Project. Let's also tell GeoMakeIt! to load our plugin on startup.

1. Navigate to *app/assets/plugins.json*.
2. Fill the file with the following information inside:

```
[
  {
    "package": "com.<your_package_name>",
    "main": "<MainClass>"
```

```
    }
]
```

And now we're ready to run our project!

1. Run the project by clicking the play button
2. Go to Logcat and search for "`[GeoMakeIt!]`" as seen in the image below. Make sure to leave the "regex" box unchecked. This way you can see all the logs created by GeoMakeIt! and other plugins.

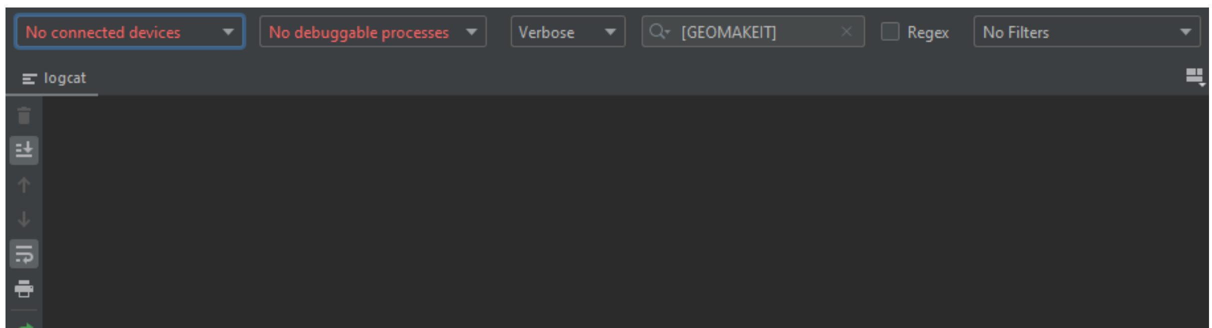

Figure 6.1 Android Studio Logcat console using [GEOMAKEIT] search filter

3. If you'd like to see only your outputs, search for "`[GeoMakeIt!][`***`<Plugin Identifier>`***`]`".

(Common Components 6.3)

Until now we've accomplished creating a simple "`Hello World`" and running it with the GeoMakeIt! Starter Project. This a great first step to understand how to create GeoMakeIt! Plugins, but currently our plugin doesn't really do anything other than logging a simple message. Wouldn't it be nicer to add some extra functionality to our "Hello world" plugin?

So, let's improve this plugin by adding the following extras:

1. Adding a welcome activity to greet a player when he first joins.
2. Creating a configuration for the Game Creator to personalize the messages.
3. Create a command to show the hello world message.
4. Create a model to represent how many times a user has seen this page.
5. Create a placeholder for showing how many times a player has seen this page.
6. Create a command to reset that number.

(Directory Structure 6.3.a)

We're going to keep a similar directory structure to the one used by GeoMakeIt!, as such our project will look like this:

- `<Plugin Name>/src/main` → The main source directory of your plugin
  - `AndroidManifest.xml` → Register's activities and permissions for your plugin.
  - `assets/<Plugin Identifier>/` → Contains configurations and data.
    - `plugin.json` → Information for GeoMakeIt! Studio.
    - `messages.json` → Personalized messages for greeting the player
  - `res/` → Will contain our plugin's resources.
  - `java/com.yourpackage.....` → Source code of your plugin.
    - `Activities package` → Will contain the activities source code that we will create.
    - `Configurations package` → Will contain configuration controllers for `messages.json`.
    - `Commands package` → Will contain commands for your plugin.
      - `CommandMap.kt` → The command map of your plugin and register your commands.
    - `Models` → Will contain all necessary models for your plugin.
    - `Main.kt` → The main of our plugin. It will extend GeoMakeIt!'s API Plugin Class.
    - `Placeholders.kt` → Will contain the placeholders for your plugin.

(Activities 6.3.b)

We'll begin by first creating our activity. In our case, we decided to name our activity "`Greetings Activity`". Therefor this will create a "`GreetingsActivity`" which we will place in the "`activities`" package. When it comes to the Greetings resource, make sure to name it as such: `<plugin_identifier>_activity_greetings.xml` . In our case again, since our plugin identifier is "`hello_world_example`", the xml will be named "`hello_world_example_activity_greetings.xml`". This prefix is appended to follow the rules of GeoMakeIt! API and avoid possible conflicts with other plugins.

GreetingsActivity.kt

```
class GreetingsActivity : JActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(
            R.layout.hello_world_example_activity_greetings)
    }
}
```

Make sure to extend the `JActivity()` class and not `AppCompactActivity()`.

hello_world_example_activity_greetings.xml

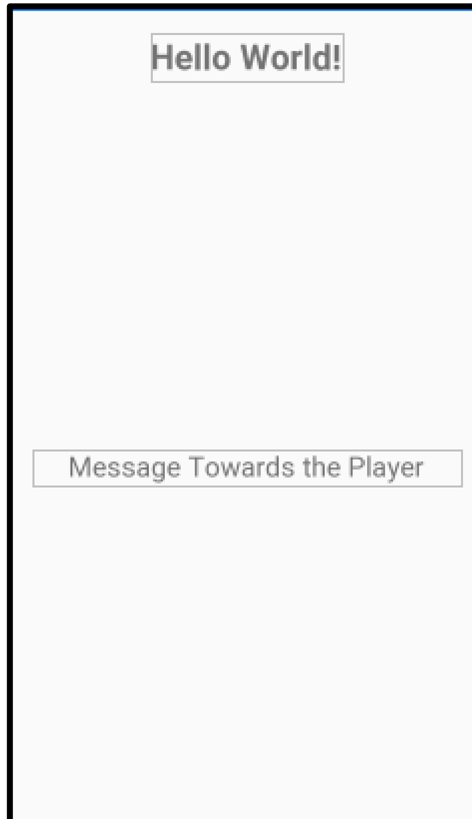

Figure 6.2 Initial Hello World Activity

```
   <?xml version="1.0" encoding="utf-8"?>
   <androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
       xmlns:app="http://schemas.android.com/apk/res-auto"
       xmlns:tools="http://schemas.android.com/tools"
       android:layout_width="match_parent"
       android:layout_height="match_parent"

tools:context="com.geomakeit.helloworldexample.activities.GreetingsA
ctivity">

       <TextView
           android:id="@+id/title"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:layout_marginStart="20dp"
           android:layout_marginLeft="20dp"
           android:layout_marginTop="20dp"
           android:layout_marginEnd="20dp"
           android:layout_marginRight="20dp"
           android:text="Hello World!"
           android:textSize="30sp"
           android:textStyle="bold"
           app:layout_constraintEnd_toEndOf="parent"
           app:layout_constraintStart_toStartOf="parent"
           app:layout_constraintTop_toTopOf="parent" />
```

```
        <TextView
            android:id="@+id/message"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginStart="20dp"
            android:layout_marginLeft="20dp"
            android:layout_marginEnd="20dp"
            android:layout_marginRight="20dp"
            android:gravity="center"
            android:text="Message Towards the Player"
            android:textSize="24sp"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/title" />
    </androidx.constraintlayout.widget.ConstraintLayout>
```

Main.kt

Finally, we will require to register our activity for the Game Creator to be able to execute it. We usually do that in the plugin's main class. So, let's use the `Activity Registry` to register our Activity on GeoMakeIt! API. We'll also use the `Command Manager` to force dispatch a command and check how our activity looks and if it starts.

```
    object Main: Plugin() {
                    ...
        override fun onEnable() {
                    ...
            ActivityRegistry.register(this, "welcome",
GreetingsActivity::class.java)

            CommandManager.dispatch("geomakeit::activity
hello_world_example::welcome")
                    ...
        }
                    ...
    }
```

The `Activity Registry's Register` method takes 3 arguments, the `plugins` `(this)`, a `unique identifier` for that activity (`welcome`) and the JActivity class (`GreetingActivity::class.java`). The `Command Manager` dispatch command execute a command immediately. In this case we are calling the "`geomakeit::activity`" command which starts a GeoMakeIt! Activity. We request the "`hello_world_example::welcome`" activity which is the activity we just registered.

(Configurations 6.3.c)

We'll continue by creating the `messages.json` configuration. This configuration will contain inside the default greetings that will be shown to the player. We are placing the

messages.json inside the "assets/hello_world_example/" directory to make it visible on the GeoMakeIt! Studio for the Game Creator to edit it and personalize it. Our configuration controller, the way to grab the messages, will be contained into the configurations package and named Messages.kt.

assets/hello_world_example/messages.json

The messages.json contains all the welcome messages that we will show to the Greetings Activity randomly when welcoming a player.

```
{
  "welcome_messages": [
    "Welcome player. What's up?!",
    "It’s an honor to meet you player!"
  ]
}
```

configurations/Messages.kt

The Messages class is a quick way to load and grab information from messages.json

```
object Messages {
    private var map: Map<String, Any>
    init {
        map = try {
            val json = Main.getConfig("messages.json").asString()
            JSONUtil.jsonTo(json);
        } catch(e: Exception) {
            Main.logger.e("Failed to initialize 'messages.json'.
                Loading defaults. (Reason: ${e.message})")
            mapOf()
        }
    }

    var WELCOME_MESSAGES =
        map["welcome_messages"] as? List<String> ?: listOf(
            "Welcome. What's up?!",
            "Hey there. Welcome to the game!",
            "How are you doing? Welcome to the game!"
        )
        private set

    init {
        if(WELCOME_MESSAGES.isEmpty())     WELCOME_MESSAGES     =
listOf("No default message has been set to greet you. That's a
shame, aint it?")
    }
}
```

Now that we have created a way to grab information from Messages, lets grab a random message every time and present it to the player when the `Greeting Activity` starts.

```
class GreetingsActivity : JActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        val msg = findViewById<TextView>(R.id.message)
        msg.text = PlaceholderAPI.parse(
                Messages.WELCOME_MESSAGES.shuffled().take(1)[0]
            )
        ...
    }
}
```

We will replace the text of view id “`message`” from the `GreetingsActivity` to a text from `welcome_messages` contained in `messages.json`. To grab that data, we will use the `Messages` class, calling the `WELCOME_MESSAGES` variable. We want to grab a random welcome message, so we shuffle the list and grab a random one.

Lastly, if we replace the text immediately like that the placeholders contained in it such as `{username}` will not work. For those placeholders to be replaced we need to parse them to the `Placeholder API`, so we call the `PlaceholderAPI.parse()` method.

Let’s run our program again and see what we get now!

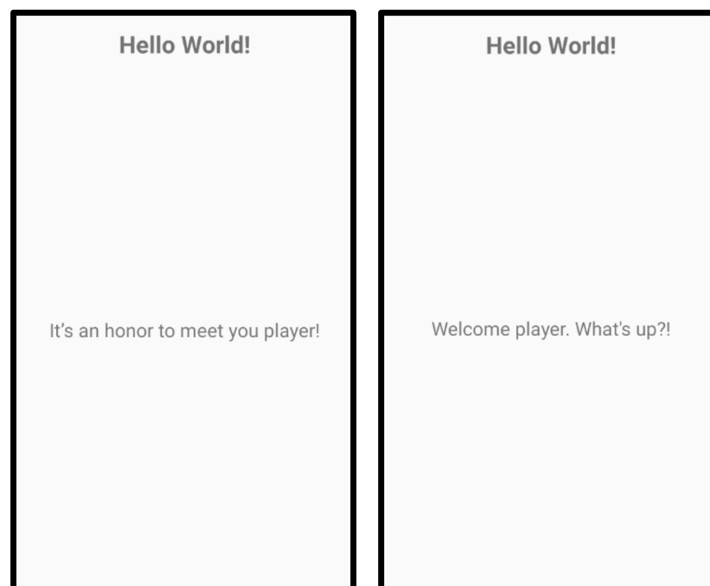

Figure 6.3 First and second opening returns two different outcomes

(Models 6.3.d)

Since we want to keep this example very simple, we are going to skip over models that have database support and create a very simple `Greetings Counter` object that will count the number of times a player has been greeted.

Because this will be a single instance, just for the local player and not support multiplayer we can create an object instead of class.

Greetings Counter

```
object GreetingsCounter {
    private const val PREF_NAME =
            "com.geomakeit.helloworldexample"
    const val PREF_COUNTER = "$PREF_NAME.counter"

    private val sharedPref =
    App.getCurrentActivity().getSharedPreferences(
        "com.geomakeit.helloworldexample",
        Context.MODE_PRIVATE
    )
}
```

We will be using the Android's `Shared preferences` to store the `greetings counter` and as such we must get the `shared preferences` from the current activity. We also know that the `getSharedPreferences()` method takes two arguments, the name and the `Context Mode` which we will supply from constant values as seen here. More about shared preferences can be learned by taking a look at the Android's Documentation.

Our `Greetings Counter` object has 3 methods, one to `greet`, one to get the `number of greetings` and one to `reset` them. All of them are just simple actions on top of the shared preferences variable.

```
    fun greet() {
        Main.logger.d("Player '${LocalUser.displayName}'
            has been greeted")
        with (sharedPref.edit()) {
            putInt(PREF_COUNTER, getGreetings() + 1)
            commit()
        }
    }

    fun getGreetings(): Int {
        return sharedPref.getInt(PREF_COUNTER, 0)
    }

    fun reset() {
        with (sharedPref.edit()) {
            putInt(PREF_COUNTER, 0)
            commit()
```

```
        }
    }
```

We can now use this class to increase the counter by calling `GreetingsCounter.greet()`, get the numbers of greetings by calling `GreetingsCounter.getGreetings()` and reset that number by calling `GreetingsCounter.reset()`.

```
class GreetingsActivity : JActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(
                R.layout.hello_world_example_activity_greetings)

        GreetingsCounter.greet()

        val msg = findViewById<TextView>(R.id.message)
        msg.text = PlaceholderAPI.parse(
                Messages.WELCOME_MESSAGES.shuffled().take(1)[0]
            )
    }
}
```

Since we want to increase the greetings counter every time we open the `Greetings Activity`, inside the `GreetingsActivity.kt` before we change the message text call `GreetingsCounter.greet()` as seen here.

(Placeholders 6.3.e)

Next, we need a way to allow the Game Creator to display how many times the user has been greeted on any text. We do that by creating a placeholder.

Inside the main package lets create a file called `Placeholders.kt` and insert the following code:

```
class Placeholders(plugin: IPlugin) : Placeholder(plugin) {

    override fun onRequest(placeholderId: String): String? {
        when (placeholderId) {
            "counter" -> return
                                GreetingsCounter.getGreetings()
                                                .toString()
        }
        return null
    }
}
```

With this we instruct GeoMakeIt! API that whenever someone calls on a personalized text "`{<plugin_identifier>::counter}`", in this case

“{hello_world_example::counter}” that number is going to be replaced by the number returned from GreetingsCounter.getGreetings().

To make though this placeholder functional, we are required to register it. We do that inside the plugin’s Main class, in this case Main.kt file. Before the ActivityRegistry write the following:

```
Placeholders(this).register()
```

We usually want any placeholder registration to happen before anything else since it may be used by the plugin itself later.

Now that we’ve registered the placeholders it is time to try them out! We can do that easily by modifying slightly the messages.json file to show also our placeholders. In this case let’s make something similar to this:

```
{
  "welcome_messages": [
    "Welcome {username}. What's up?! You've been greeted
       {hello_world_example::counter} times!",
    "Hey there {username}. Welcome to the game! You've been
       greeted {hello_world_example::counter} times!",
    "How are you doing {username}? Welcome to the game! You've
       been greeted {hello_world_example::counter} times!"
  ]
}
```

Remember, since in Greetings Activity we are parsing the text using the PlaceholderAPI.parse() method, the message is personalized and as such GeoMakeIt! API will attempt to replace anything inside {} with its appropriate placeholder.

Running our plugin now will produce the following:

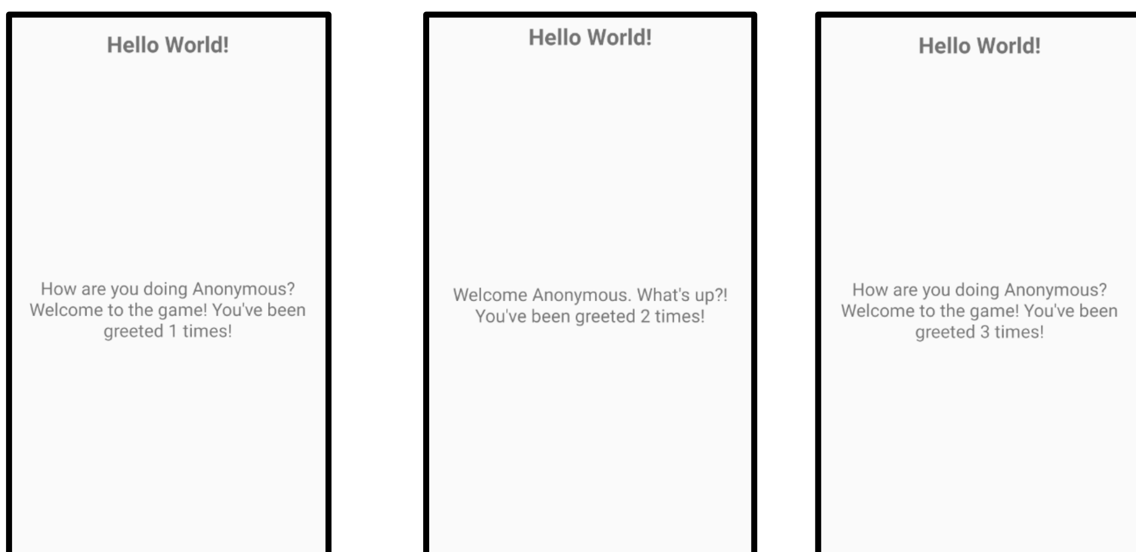

Figure 6.4 Three random openings of the application using Placeholders

The counter is gradually increasing since we are restarting every time our application.

(Commands 6.3.f)

Lastly we want to create a simple command to reset everything if the Game Creator chooses to do so. We shall create two classes under the command package. One named `CommandMap` and the other one `ResetCommand`.

Reset Command

```
class ResetCommand : Command {
    constructor() : super("reset") {
        this.description = "Reset the greetings
                            counter for a player";
        this.usageMessage = "hello_world_example::reset";
    }

    override fun execute(commandLabel: String,
                         args: Array<String>): Boolean {
        try {
            GreetingsCounter.reset()
            return true
        } catch(ex: Exception) {
            Main.logger.e("Encountered an error: ${ex.message}")
        }
        return false
    }
}
```

Here, we must take note of the `super()` call inside the constructor. We can see that we pass the "`reset`" string as a parameter. That is the identifier of the command. In our case, since our plugin identifier is "`hello_world_example`" and the command identifier is "`reset`" the command that will be created by this class would be "`hello_world_example::reset`".

Optionally you can add information about the description and usage message of this specific command as seen here.

The `execute()` method contains the code that will be executed once that command is typed, in our case, `GreetingsCounter.reset()`. We optionally can wrap this inside a try catch statement to catch any unexpected error, ex. an error caused by the Shared Preferences. Remember to always return true if the command was executed successfully or false otherwise.

Command Map

The command map registers any given commands under that specific command map.

```
class CommandMap : SimpleCommandMap {
    constructor(plugin: IPlugin) : super(plugin) {
        registerCommand(ResetCommand())
    }
}
```

Main

Lastly, never forget to register your command maps so all your commands can actually run. To do this, inside main function we must append the following code:

```
object Main: Plugin() {
    override val identifier: String
        get() = "hello_world_example"

    override fun onEnable() {
        logger.i("$identifier got enabled!")

        CommandMap(this).register()
        Placeholders(this).register()

        ActivityRegistry.register(this, "welcome",
                                  GreetingsActivity::class.java)

        CommandManager.dispatch(
            "geomakeit::activity hello_world_example::welcome"
        )
    }

    override fun onDisable() {
        logger.d("$identifier got disabled :(")
    }
}
```

We can verify that the commands work by doing some debugging inside the main's `onEnable()` method. Let's append the following code at the end which will reset our counter after 5s and reopen the Greetings activity.

```
object Main: Plugin() {
              ...
    override fun onEnable() {
              ...
        // Reset after 5s
        Handler().postDelayed({
            CommandManager.dispatch("hello_world_example::reset")
            CommandManager.dispatch("geomakeit::activity
hello_world_example::welcome")
        }, 5000)
    }
              ...
}
```

Let's now run our plugin and see the results.

(Cleaning up 6.3.g)

Before this plugin can get uploaded to the GeoMakeIt! API it would be a good thing to clean up all the unnecessary debug code, since it would interfere with our plugin's actual functionality for the Game Creator.

(Uploading and Publishing your Plugin 6.4)

(The plugin.json file 6.4.a)

Before we upload the plugin to the GeoMakeIt! Studio we must first create its initial `plugin.json` configuration. This configuration contains information about the plugin that will be visible when uploaded and published.

To create the `plugin.json` we must do the following:

1. In the Android Studio, switch to Project view.
2. Navigate to your project's main directory. It should be located on *<Plugin Name>/src/main*.
3. Inside the directory *assets/<plugin_identifier>/*, create a file named *plugin.json* and fill in the following contents:

```
{
  "main": "com.geomakeit.helloworldexample.Main",
  "title": "Hello World! By GeoMakeIt!",
  "short_description": "Hello World example created by GeoMakeIt!",
  "description": "This is just a big description to fill in the 70 character minimum for a description.",
  "version": "1.0.0",
  "author": "Vasilis Dimitriadis",
  "gradle_implementations": []
}
```

Remember to change some of the contents depending on your specific plugin settings, such as the main!

(Registering your identifier 6.4.b)

Your plugin might be unique but is your plugin's name unique? In order to figure this, you must register your plugin's identifier in the GeoMakeIt! Studio.

We assume you've read the "Choosing your identifier" section, so let's just have a look at the steps required to register that identifier.

1. Navigate to GeoMakeIt! Studio.
2. Make sure you are logged in and a plugin developer.

3. Select “Plugins”.
4. Select “+ New”.
5. Fill out the form. Remember the identifier rules described in “Choosing your identifier”.
6. Submit and make sure that you have corrected any possible error.

Congratulations! Now you have secured the `identifier` of your choice. If for some reason the selected identifier isn’t the same as the one you’ve selected when developing the plugin, make sure to update all references to the new identifier.

CHAPTER 7: The first GeoMakeIt! Game.

(Introduction 7.1)

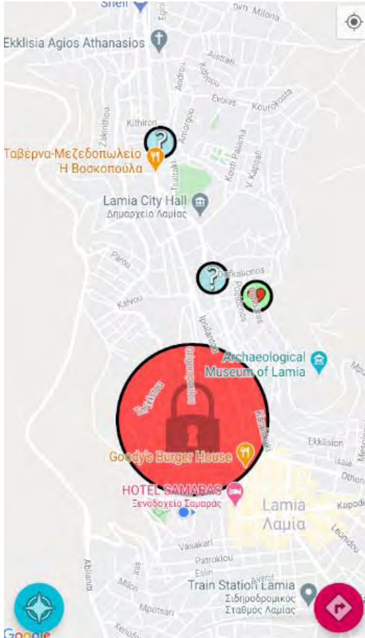

Figure 7.1 Snapshot of the first GeoMakeIt! Game

In this Chapter we will be creating a Demo Game from scratch using the GeoMakeIt! Platform and the two demo plugins `GeoQuiz!` and `GeoFighting!`. The concept of the game is as follows:

A player located in Lamia, Greece must visit some predefined geo-locations places. There are two types of locations, Quiz/Trivia zones and Fighting zones. The player must first visit and complete successfully all the Trivia Quizzes. Following that, the player must attempt to fight and win inside the Fighting zone. If successful, a previously locked area unlocks for the player to answer a few more Trivia questions. Once all tasks have been completed the game is over and the application closes.

(Game Creation Process 7.2)

To create a game a user must be registered to the GeoMakeIt! Studio as a Game Creator. We will consider that this step has been already fulfilled and the user is currently logged in and inside the GeoMakeIt! Studio.

To create a game a user must be registered to the GeoMakeIt! Studio as a Game Creator. We will consider that this step has been already fulfilled and the user is currently logged in and inside the GeoMakeIt! Studio.

(Creating a Game 7.2.a)

Inside the GeoMakeIt! Studio we must navigate to our `Games` tab, highlighted in blue on Figure 7.2. From there press the green `+ New` button.

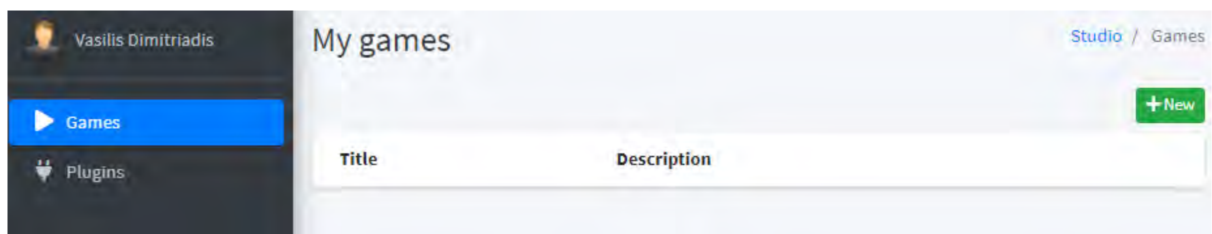

Figure 7.2 Browse Games in GeoMakeIt! Studio

We get redirected to the game creation page where we must fill our games `title` and `description`. These can be either the final title and description or just placeholders that we may change later. We fill the requested details and `save`.

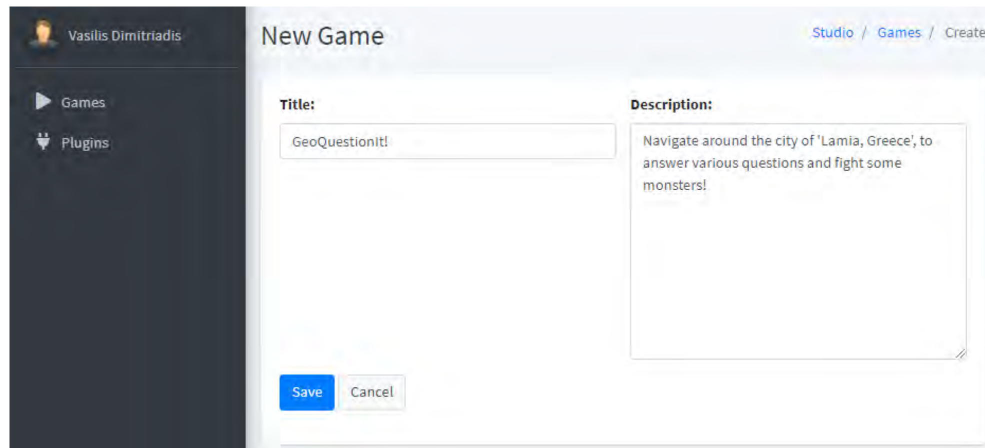

Figure 7.3 Game creation page in GeoMakeIt! Studio

(Installing plugins 7.2.b)

Our game has now been created successfully. If we were a new user we would stick around and try to explore the GeoMakeIt! API options, but since we already have some background knowledge of how everything works we can immediately add the plugins we require and start configuring them. To do that, we must navigate to the `game's plugins` which can be done by clicking the `plug` icon.

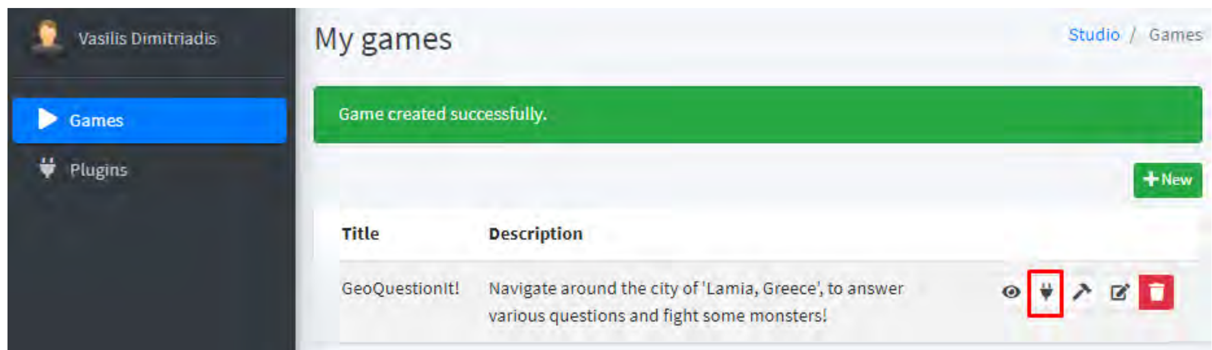

Figure 7.4 Successful creation of Game

We can see that GeoMakeIt! API is already included because it is a required plugin, so all we must do is install the rest plugins of our choice. In this case we install both `GeoQuiz` and `GeoFighting!` since we will need them both. To install them, we must click the `green install button`.

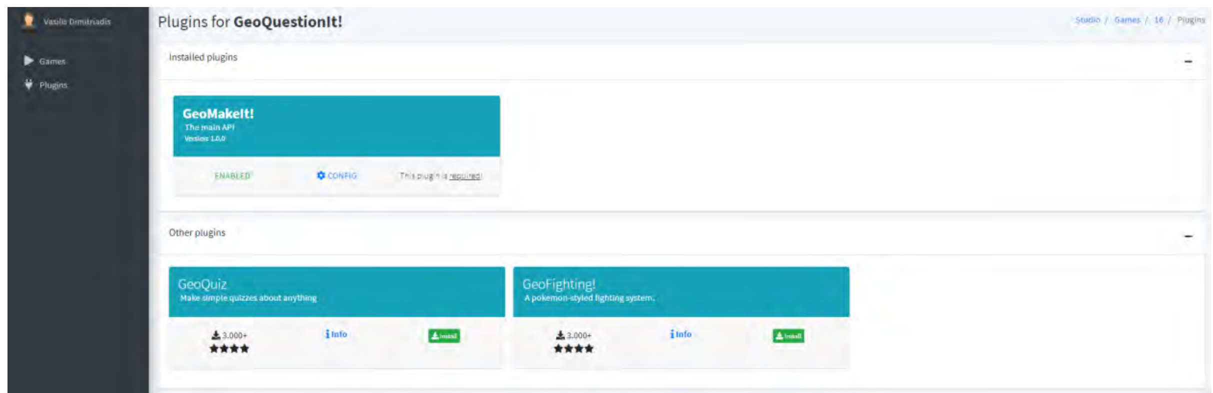

Figure 7.5 Plugin Installation page

(Configuring GeoFighting! 7.2.c)

We can then start configuring the plugins. We will start this process in reverse. Normally we would begin by configuring GeoMakeIt!, but in our case we will start by configuring the simplest plugin, `GeoFighting!`, then proceed to `GeoQuiz` and lastly `GeoMakeIt! API`.

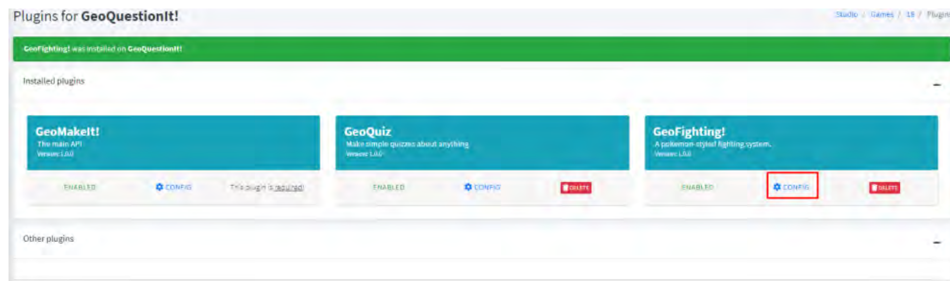

Figure 7.6 Navigating to GeoFighting! Configuration

We navigate to GeoFighting! plugin configurations by clicking the `config icon` and from there we select the `configs tab` as seen in the figure below. There are two available configuration files, the `config` file and `mobs` file.

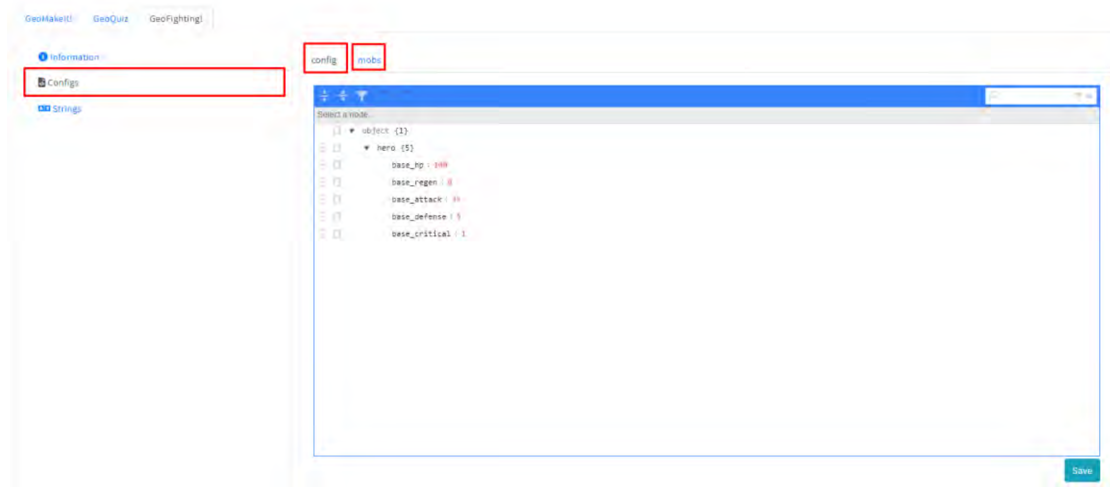

Figure 7.7 GeoFighting available configurations

The config file contains the hero default stats, meaning our player starting stats while the mobs file contains information about the mobs he may encounter. In our case we can leave these configurations as is and work with the defaults.

Figure 7.8 Contents of Mobs.json file

(Configuring GeoQuiz 7.2.d)

Similarly, using the tabs in the plugins section, we navigate to GeoQuiz. Here there are more configurations available to us.

The `defaults` contain default settings for every quiz, question and UI available in GeoQuiz. The `messages` section contains the defaults messages used on `success` and on `failure` of answer. The `questions` section contains all available questions that may be asked to the user alongside with their `individual configurations`. Lastly the `quizzes` section contains all the questions grouped into quizzes and overridable settings `per-quiz`.

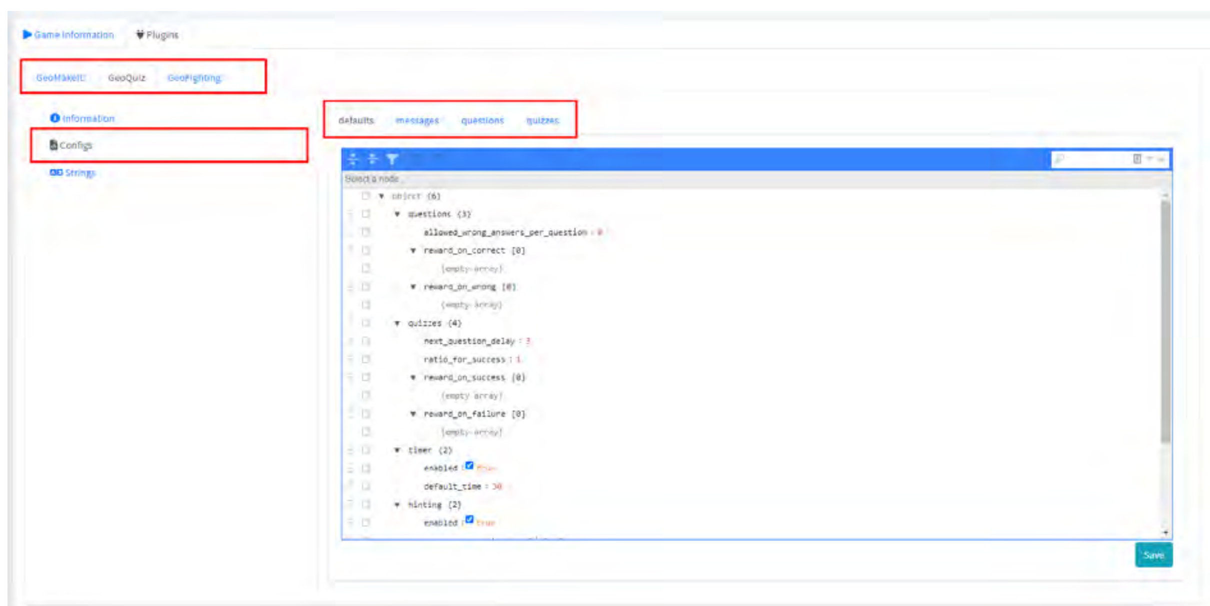

Figure 7.9 GeoQuiz and its available configurations

We will leave the `defaults` and `messages` sections as is.

We then override the questions and quizzes to the following files. We are not going into detail explaining the settings of these files, as it was thoroughly explained in Chapter 5.

The questions file contains 11 questions that will be asked to the player, their types such as multiple choice or fill the blank, possible answers, the correct answers, images and rewards.

The quizzes file groups the above questions into 5 quizzes, 4 small ones and a big one. These quizzes will be forwarded to the player when he enters a specific zone using commands as we will see shortly.

And with that, we have completed setting up GeoQuiz.

(Configuring GeoMakeIt! 7.2.e)

Lastly, we must connect all the quizzes and the mob fights together into a unified game and assign quests to our player. Since the `Quests` are implemented directly through GeoMakeIt! API, we must edit the GeoMakeIt! configurations to assign quests and unify the plugins.

The GeoMakeIt! plugin contains 6 configurable files, which we will investigate in more detail.

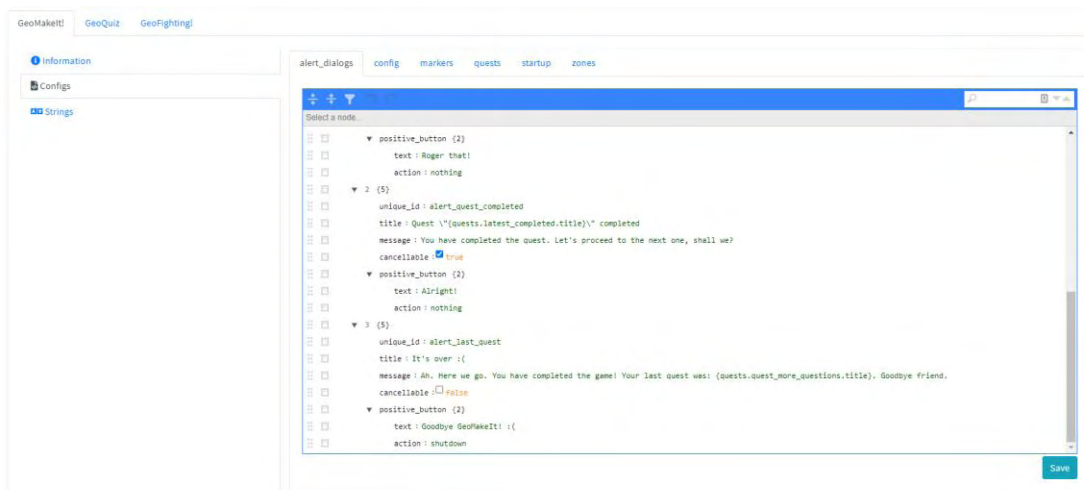

Figure 7.10 GeoMakeIt! API available configurations

The `Config` contains authentication methods for our application. In our case we can leave that as is, meaning the `authentication` is enabled and the user can authorize with his `email` or by `signing in anonymously`.

The `Markers` contains available markers that can be drawn into position. We will not be using markers in our project so we can either delete the contents or just leave it as is.

Alert Dialogs

The `Alert Dialogs`, which contains example alert dialogs that can be used inside the game. These dialogs usually contain `placeholders` inside to be personalized per player. The default settings will do for us, and we will use 3 out of the 4 alerts, `alert_quest_new`, `alert_quest_completed`, `alert_last_quest`. The `alert_0` is unnecessary, so we can go ahead and delete that.

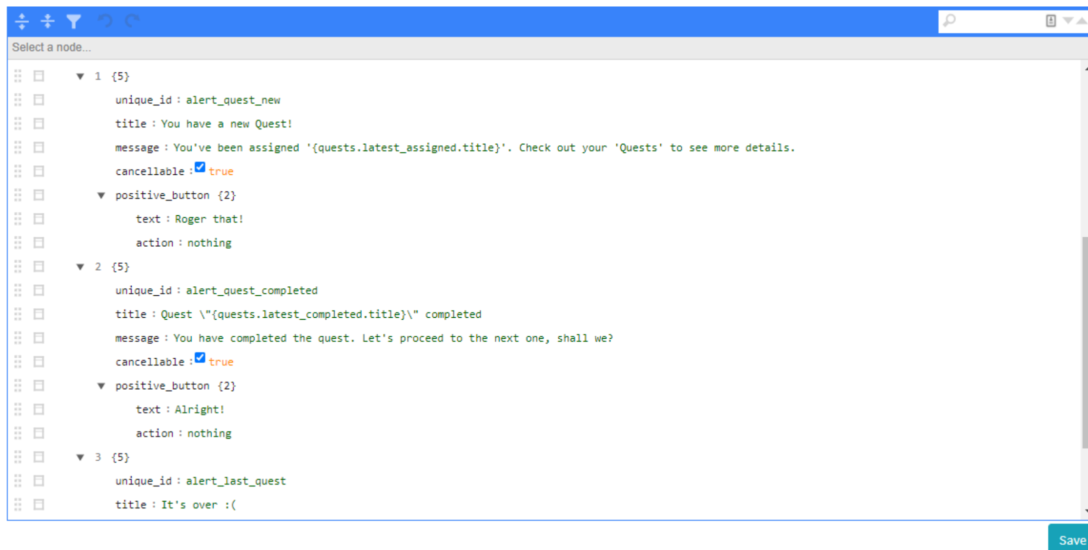

Figure 7.11 GeoMakeIt! Alert configurations

Let's start now modifying the rest. By default, GeoMakeIt! API contains configurations for a quiz and fighting game like the one that we want to create. The only difference though is that it doesn't know which plugins we have implemented and as such it just checks if the user enters a zone to complete the quest, but not if the user has actually answered correctly a quiz or if he won a battle.

Zones

By opening the `Zones` configuration, we can see 5 Question Zones, 1 Locked Zone, `1 Fighting Zone` and `1 Regeneration Zone`. These zones have prefilled `icons`, `colors` etc., so we will not bother with those, but rather just edit the `on_enter` to execute our desired commands when the user enters.

For example, this could be the `Question Zone 1`:

```
title : Question Zone 1
▼ center {2}
    latitude : 38.90817
    longitude : 22.430357
radius : 60
unique_id : zone_question_1
fill_color : #501CBFD1
stroke_width : 10
▼ on_enter [1]
    0 : geomakeit::activity geoquiz::quiz quiz=quiz_small_1
▼ icon {2}
    image : drawable/geomakeit_icon_question_mark
    width : 100
```

Figure 7.12 Closer look to a single question

What we are saying here is the following: When the player enters `zone_question_1`, we shall execute the command "`geomakeit::activity geoquiz::quiz quiz=quiz_small_1`", where `geoquiz::quiz` is the quiz activity loading the quiz "`quiz_small_1`". We fill similarly the rest zones with the same command but editing which quiz they are executing, ex. `quiz_small_2`, `quiz_small_3` etc.

For the fighting zone we add the following command `on_enter`: "`geomakeit::activity geofighting::fight mob=golem`".

Lastly for the Regeneration Zone we add this command `on_enter`: "`geofighting::regen_hp`".

We should end up having a file like this:

```
[
  {
    "title": "Question Zone 1",
    "unique_id": "zone_question_1",
     ...
    "on_enter": [
      "geomakeit::activity geoquiz::quiz quiz=quiz_small_1"
    ]
  },
  {
    "title": "Question Zone 2",
    "unique_id": "zone_question_2",
    ...
    "on_enter": [
      "geomakeit::activity geoquiz::quiz quiz=quiz_small_2"
    ]
```

```
    },
    {
      "title": "Fighting Zone",
      "unique_id": "zone_fight_1",
      ...
      "on_enter": [
        "geomakeit::activity geofighting::fight mob=golem"

    },
    {
      "title": "Regen Zone",
      "unique_id": "zone_regen",
      ...
      "on_enter": [
        "geofighting::regen_hp"
      ]
    },
    {
      "title": "Locked Area",
      "unique_id": "zone_locked_area",
      ...
    },
    {
      "title": "Question Zone 3",
      "unique_id": "zone_question_3",
      ...
      "on_enter": [
        "geomakeit::activity geoquiz::quiz quiz=quiz_small_3"
      ]
    },
    {
      "title": "Question Zone 4",
      "unique_id": "zone_question_4",
      ...
      "on_enter": [
        "geomakeit::activity geoquiz::quiz quiz=quiz_small_4"
      ]
    },
    {
      "title": "Question Zone 5",
      "unique_id": "zone_question_5",
      ...
      "on_enter": [
        "geomakeit::activity geoquiz::quiz quiz=quiz_big_1"
      ]
    }
  ]
```

Startup

Now that we have created available zones, lets draw our initial ones. Open the `startup` configuration. This file contains all the commands that will execute as soon as

the application starts. In our case we want to draw the areas that will be always there on the start. These would be the `zone_regen` and `zone_locked_area`.

```
array [2]
    0 : geomakeit::zone draw zone_regen
    1 : geomakeit::zone draw zone_locked_area
```

Figure 7.13 Commands to draw two specific zones at startup

Quests

The last configuration is `quests`. Quests declare what the player must do to progress in the game. In our case our quests would look like the following:

1. Visit all visible quizzes zones and answer the quizzes successfully. Then,
2. Visit all fight zones, fight and win. Then,
3. Unlock locked zone and answer a few more quizzes.
4. When all quests are complete, finish game.

To achieve these, our quests file will look like this:

```
{
  "on_first_join": [
    "quest_visit_questions"
  ],
  "quests": [
    {
      "unique_id": "quest_visit_questions",
      "title": "Visit all \"?\" zones",
      "description": "You must all zones marked with \"?\" to complete this quest.",
      "on_assignment": [
        "geomakeit::alert_dialog show alert_quest_new",
        "geomakeit::zone draw zone_question_1",
        "geomakeit::zone draw zone_question_2"
      ],
      "actions": [
        "geomakeit::zone visited zone_question_1",
        "geomakeit::zone visited zone_question_2",
        "geoquiz::has_completed_quiz quiz_small_1",
        "geoquiz::has_completed_quiz quiz_small_2"
      ],
      "on_complete": [
        "geomakeit::zone erase zone_question_1",
        "geomakeit::zone erase zone_question_2",
        "geomakeit::alert_dialog show alert_quest_completed",
        "geomakeit::quest assign quest_visit_fights"
      ]
    },
    {
```

```
        "unique_id": "quest_visit_fights",
        "title": "Visit all fight zones",
        "description": "You must all fight zones to complete this
quest.",
        "on_assignment": [
          "geomakeit::alert_dialog show alert_quest_new",
          "geomakeit::zone draw zone_fight_1"
        ],
        "actions": [
          "geomakeit::zone visited zone_fight_1",
          "geofighting::fight_results has_won"
        ],
        "on_complete": [
          "geomakeit::zone erase zone_fight_1",
          "geomakeit::alert_dialog show alert_quest_completed",
          "geomakeit::quest assign quest_more_questions"
        ]
      },
      {
        "unique_id": "quest_more_questions",
        "title": "Visit all \"?\" zones",
        "description": "You must visit the rest zones marked with
\"?\" to complete this quest.",
        "on_assignment": [
          "geomakeit::alert_dialog show alert_quest_new",
          "geomakeit::zone erase zone_locked_area",
          "geomakeit::zone draw zone_question_3",
          "geomakeit::zone draw zone_question_4",
          "geomakeit::zone draw zone_question_5"
        ],
        "actions": [
          "geomakeit::zone visited zone_question_3",
          "geomakeit::zone visited zone_question_4",
          "geomakeit::zone visited zone_question_5",
          "geoquiz::has_completed_quiz quiz_small_3",
          "geoquiz::has_completed_quiz quiz_small_4",
          "geoquiz::has_completed_quiz quiz_big_1"
        ],
        "on_complete": [
          "geomakeit::zone visited zone_question_3",
          "geomakeit::zone visited zone_question_4",
          "geomakeit::zone visited zone_question_5",
          "geomakeit::alert_dialog show alert_last_quest"
        ]
      }
    ]
  }
```

This is overwhelming, so let's see in detail only the first few lines and the rest will make sense:

```
  {
    "on_first_join": [
      "quest_visit_questions"
    ],
```

```
    "quests": [
      {
        "unique_id": "quest_visit_questions",
        "title": "Visit all \"?\" zones",
        "description":  "You  must  all  zones  marked  with  \"?\"  to
complete this quest.",
        "on_assignment": [
          "geomakeit::alert_dialog show alert_quest_new",
          "geomakeit::zone draw zone_question_1",
          "geomakeit::zone draw zone_question_2"
        ],
        "actions": [
          "geomakeit::zone visited zone_question_1",
          "geomakeit::zone visited zone_question_2",
          "geoquiz::has_completed_quiz quiz_small_1",
          "geoquiz::has_completed_quiz quiz_small_2"
        ],
        "on_complete": [
          "geomakeit::zone erase zone_question_1",
          "geomakeit::zone erase zone_question_2",
          "geomakeit::alert_dialog show alert_quest_completed",
          "geomakeit::quest assign quest_visit_fights"
        ]
      },
```

The first thing we see is the `on_first_join`. This declares which quests are assigned as soon as the player joins the game. We only want one quest to be assigned and that one is “`quest_visit_questions`”.

Next we declare the quests. We will look just the first quest, `quest_visit_zones`. We begin by giving it a title and a description. Next there are 3 sections: `on_assignment`, `actions` and `on_complete`. These sections are simply a list of commands that are executed in different times.

On the assignment of quest, we execute the following commands:

1. First, we alert the player that he has a new quest by showing him an alert dialog using the command “`geomakeit::alert_dialog show alert_quest_new`”. The argument `alert_quest_new` is the id of the alert which we previously show inside the `alert_dialogs` configuration.
2. Next we draw the zone `zone_question_1` and `zone_question_2`.

As soon as the quest is assigned the user must complete some actions in order to complete it. These are described here:

1. First we check if the user has visited the zone `zone_question_1` and `zone_question_2` by executing commands “`geomakeit::zone visited zone_question_1`”.
2. Then we check if the user has completed the quizzes `quiz_small_1` and `quiz_small_2`.

Similarly, on complete is executed as soon as the quest is completed. We will just describe the commands that are seen above to make sure that everything is well understood.

1. We erase the zones we have already visited, `zone_question_1` and `zone_question_2`.
2. We alert the player that he has finished the quest.
3. We assign the next quest to the player.

(Game Building Process 7.3)

The game logic has been created and now the last step is to build and release the game.

We must first go back to the GeoMakeIt! Studio Games tab and load our list of games. From there we must choose to build our game.

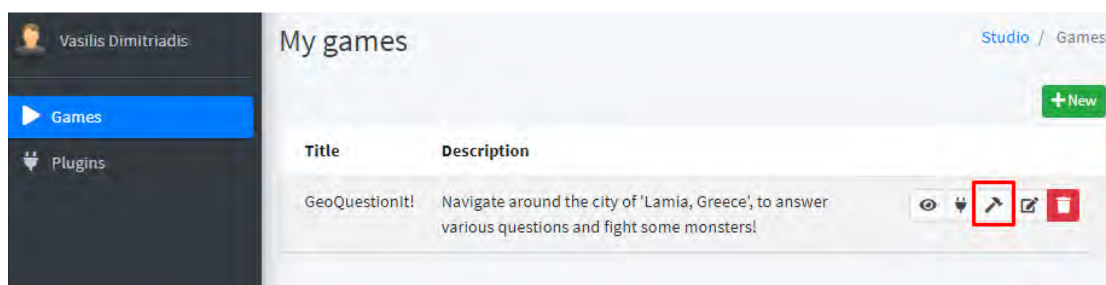

Figure 7.14 Navigating to Build Page

We will get redirected to our game's builder page. There are two options: `Build "<Game>"` and `Download "<Game>"`. We begin building our game by pressing the first button. This process usually takes 3-5 minutes so we will be alerted by the website as soon as the process is complete.

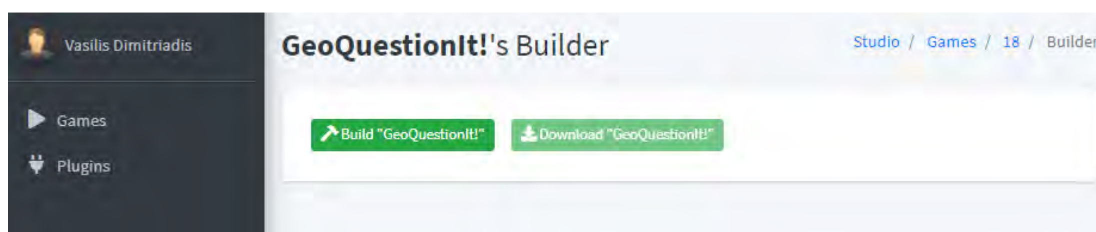

Figure 7.15 GeoMakeIt! Studio build page

Once completed and if successful, the download button will be available. We can press this button to download and share our game.

(Installing and Playing our Game 7.4)

After building and downloading our game the only thing left is installing it to our device and playing our game. This process can be done in multiple ways such as:

- Installing it over USB.
- Uploading it to a website such as Dropbox, Google Drive etc., downloading it and then installing it.
- Uploading to Google Play Store.
- Emulating it through an Android Emulator.

CHAPTER 9: Future Planning

At the current state, the core functionality and a proof of concept for the GeoMakeIt! Platform has been implemented and will be accessible over the internet. In the nature though of a proof of concept key feature are missing but are planned to be implemented in the long run.

Graphically rich environment

We believe that the most powerful tool over the GeoMakeIt! Platform should be the Studio. GeoMakeIt! Studio must have a user-friendly environment that can be easily grasped by non-experts and feel natural. JSON files and hard-coded commands must be modified to a dynamic system where all plugins make easily accessible to the GeoMakeIt! Studio their available commands and as such can be uniformly displayed to the Game Creator.

Version Control System

An application specific version control system must be built to handle all the plugin versions and configurations. Game Creators must have the freedom to choose or change versions when they desire and the configurations to be migrated automatically. Plugin Developers must be able to upload multiple versions and control them, mark possible bugs and even force creators into updating if a vulnerable issue is found.

Richer Plugin Information

More information about plugins must be visible in GeoMakeIt! Studio such as all their hard and soft dependencies with appropriate linking. Options to automatically install dependencies shall also be available. Plugin commands, placeholders and other common GeoMakeIt! functionality should be documented automatically when the plugin is uploaded, and an option should be provided for the plugin developers to enrich this information with examples or a full wiki.

Game Templates

Game templates should be available to allow a Game Creator prototype with different combinations of plugins fast, instantly creating the base of common games such as quiz games, fighting games etc.

Independent Database

The current state of GeoMakeIt! uses Google's Firebase for storing game specific information. This should be broadened to accommodate different NoSQL databases both for experts and non-experts.

Embedded UI Editor

A basic XML Based UI editor should be implemented to allow slightly more experienced Creators to modify activities and customize their appearance.

Statistics

A basic statistical analysis system should be implemented into the GeoMakeIt! Studio so Game Developers can track their games progress with total downloads, active players, daily interaction etc. At the same time Plugin Developers should be able to have access to statistics about their plugins too, such as total installs per version, how many games use them, creator feedback etc.

Monetization

Typical monetization methods must be implemented to allow Game Creators and Plugin Developers to earn a generous income out of their creations. Game creators should have the ability to add advertisements to their plugins in an easy way and Plugin Developers should be able to offer free and premium based plugins. A Plugin Store could be implemented to further assist those developers.

Game download page

Simple pre-built download pages for plugins should be available under the GeoMakeIt! Domain. This would allow Creators to easily share their games and at the same time provide a chance to generate new Creators for the GeoMakeIt! Platform.

Database and Map Wrappers

The current state of GeoMakeIt! provides hard-coded database and map choices, specifically Google's Firebase and Google Maps. Creators should have the freedom to choose which service they will use and be even able to migrate if they desire.

Localization

Plugins should offer localizable words and phrases to Creators easily. As such a game created with GeoMakeIt! can be used by different people in multiple countries without language restrictions.

Lastly, the plugin database should be enriched allowing game creators to make even more subgenres of location-based games.

REFERENCES

[1] “History of Games”, https://en.wikipedia.org/wiki/History_of_games, Accessed 2020-10-01

[2] Tetris, Andreas Elmenthaler (Elmi), "Hagenuk MT-2000 with Tetris", http://handy-sammler.de, Accessed 2020-10-01.

[3] "Snake is born: A mobile gaming classic" (in Dutch), Nokia, https://web.archive.org/web/20090209232201/http://www.nokia.com/A4303014, Accessed 2020-10-01

[4] “Revenue Model”, https://en.wikipedia.org/wiki/Video_game_monetization#Retail, Accessed 2020-10-01

[5] O. Ahlqvist and C. Schlieder, “Introducing geogames and geoplay: characterizing an emerging research field,” in Geogames and Geoplay, pp. 1–18, Springer, 2018.

[6] K. Vassilakis, O. Charalampakos, G. Glykokokalos, P. Kontokalou,M. Kalogiannakis, and N. Vidakis, “Learning history through location-based games: The fortification gates of the venetian walls of the city of heraklion,” in Interactivity, Game Creation, Design, Learning, and Innovation, pp. 510–519, Springer, 2017.

[7] J. Paavilainen, H. Korhonen, K. Alha, J. Stenros, E. Koskinen, and F. Mayra, “The pok´emon go experience: A location-based augmented reality mobile game goes mainstream,” in Proceedings of the 2017 CHI conference on human factors in computing systems, pp. 2493–2498, 2017.

[8] “GeoCaching”, https://www.geocaching.com/play, Accessed 2020-10-01

[9] M. Flintham, S. Benford, R. Anastasi, T. Hemmings, A. Crabtree, C. Greenhalgh, N. Tandavanitj, M. Adams, and J. Row-Farr, “Where on-line meets on the streets: experiences with mobile mixed reality games,” in Proceedings of the SIGCHI conference on Human factorsin computing systems, pp. 569–576, 2003.

[10] S. Benford, W. Seager, M. Flintham, R. Anastasi, D. Rowland, J. Humble, D. Stanton, J. Bowers, N. Tandavanitj, M. Adams, et al., “The error of our ways: the experience of self-reported position in a location-based game,” in International Conference on Ubiquitous Computing, pp. 70–87, Springer, 2004.

[11] A. D. Cheok, K. H. Goh, W. Liu, F. Farbiz, S. W. Fong, S. L. Teo,Y. Li, and X. Yang, “Human pacman: a mobile, wide-area entertainment system based on physical, social, and ubiquitous computing,” Personal and ubiquitous computing, vol. 8, no. 2, pp. 71–81, 2004.

[12] T. Chatzidimitris, D. Gavalas, and D. Michael, “Soundpacman: Audio augmented reality in location-based games,” in 2016 18th Mediterranean Electrotechnical Conference (MELECON), pp. 1–6, IEEE, 2016.

[13] C. Schlieder, P. Kiefer, and S. Matyas, “Geogames: A conceptual framework and tool for the design of location-based games from classic board games,” in International Conference on Intelligent Technologies for Interactive Entertainment, pp. 164–173, Springer, 2005.

[14] L. Chittaro and R. Sioni, “Turning the classic snake mobile game into a location–based exergame that encourages walking,” in International Conference on Persuasive Technology, pp. 43–54, Springer, 2012.

[15] Bates, Bob (2004), p. 204. Game Design (2nd ed.). Thomson Course Technology. ISBN 1-59200-493-8.

[16] “Game Engines”, https://en.wikipedia.org/wiki/Game_engine, Accessed 2020-10-01

[17] Martinez, Juan (27 October 2016). "The Best eLearning Tools". PC Mag.

[18] “Authoring Systems”, https://en.wikipedia.org/wiki/Authoring_system, Accessed 2020-10-01

[19] “Android”, https://www.android.com, Accessed 2020-10-01

[20] “Android OS”, https://en.wikipedia.org/wiki/Android_(operating_system), Accessed 2020-10-01

[21] “Android Architecture, https://source.android.com/devices/architecture, Accessed 2020-10-01

[22] V. Dimitriadis, P. Oikonomou, P. K. Papadopoulos, N. Panagou, A. N. Dadaliaris, and T. Loukopoulos, “GeoMakeIt!: A Platform for Developing Location Based Games,” Proc. 15th

Int. Workshop on Semantic and Social Media Adaptation and Personalization (SMAP 2020), IEEE, Oct. 2020 (to appear)
[23] "Laravel Framework", https://laravel.com, Accessed 2020-10-01
[24] "Django", https://www.djangoproject.com, Accessed 2020-10-01
[25] "NodeJS", https://nodejs.org/en, Accessed 2020-10-01
[26] "Django vs Laravel for 2019", https://www.quora.com/Should-I-have-a-plan-to-learn-Django-or-Laravel-in-2019, Accessed 2020-10-01
[27] "Django vs Laravel", https://hackr.io/blog/django-vs-laravel, Accessed 2020-10-01
[28] "Django vs Laravel", https://www.slant.co/versus/1746/9622/~django_vs_laravel-5, Accessed 2020-10-01
[29] "Google Maps", https://www.google.com/maps, Accessed 2020-10-01
[30] "Google Maps SDK Intro", https://developers.google.com/maps/documentation/android-sdk/intro, Accessed 2020-10-01
[31] "5 Powerful alternatives to Google Maps API", https://nordicapis.com/5-powerful-alternatives-to-google-maps-api, Accessed 2020-10-01
[32] "OpenLayers Examples", https://openlayers.org/en/latest/examples/, Accessed 2020-10-01
[33] "MySQL", https://en.wikipedia.org/wiki/MySQL, Accessed 2020-10-01
[34] "Google Firestore", https://firebase.google.com/docs/firestore, Accessed 2020-10-01
[35] "ColorLib", https://colorlib.com, Accessed 2020-10-01
[36] "AdminLTE 3", https://adminlte.io/themes/v3, Accessed 2020-10-01
[37] "Voyager", https://voyager.devdojo.com, Accessed 2020-10-01