ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΠΛΗΡΟΦΟΡΙΚΗ ΚΑΙ ΥΠΟΛΟΓΙΣΤΙΚΗ ΒΙΟΙΑΤΡΙΚΗ

Ανάπτυξη του GeoMakeIt! Alchemist και GeoMakeIt! Designers: Ένα σύστημα διαμοιρασμού και διάδοσης αλλαγών του συστήματος μέσα απο γεγονότα, και αυτόματη δημιουργία προσαρμόσιμων διεπαφών χρήστη για το GeoMakeIt! Studio από αντικείμενα.

ΔΗΜΗΤΡΙΑΔΗΣ ΒΑΣΙΛΕΙΟΣ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ
Επιβλέπων
ΛΟΥΚΟΠΟΥΛΟΣ ΑΘΑΝΑΣΙΟΣ

Λαμία, 2022

UNIVERSITY OF THESSALY

SCHOOL OF SCIENCE

INFORMATICS AND COMPUTATIONAL BIOMEDICINE

Development of GeoMakeIt! Alchemist and GeoMakeIt! Designers: An event driven system to share and propagate system changes, and an automatic and customizable UI generation system for GeoMakeIt! Studio from object classes.

DIMITRIADIS VASILEIOS

Master thesis

LOYKOPOYLOS ATHANASIOS

Lamia 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΔΙΑΤΜΗΜΑΤΙΚΟ ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΠΛΗΡΟΦΟΡΙΚΗ ΚΑΙ ΥΠΟΛΟΓΙΣΤΙΚΗ ΒΙΟΙΑΤΡΙΚΗ ΚΑΤΕΥΘΥΝΣΗ

«ΠΛΗΡΟΦΟΡΙΚΗ ΜΕ ΕΦΑΡΜΟΓΕΣ ΣΤΗΝ ΑΣΦΑΛΕΙΑ, ΔΙΑΧΕΙΡΙΣΗ ΜΕΓΑΛΟΥ ΟΓΚΟΥ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΠΡΟΣΟΜΟΙΩΣΗ»

Ανάπτυξη του GeoMakeIt! Alchemist και GeoMakeIt! Designers: Ένα σύστημα διαμοιρασμού και διάδοσης αλλαγών του συστήματος μέσα απο γεγονότα, και αυτόματη δημιουργία προσαρμόσιμων διεπαφών χρήστη για το GeoMakeIt! Studio από αντικείμενα.

ΔΗΜΗΤΡΙΑΔΗΣ ΒΑΣΙΛΕΙΟΣ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Επιβλέπων/σα
ΛΟΥΚΟΠΟΥΛΟΣ ΑΘΑΝΑΣΙΟΣ

Λαμία, 2022

«Υπεύθυνη Δήλωση μη λογοκλοπής και ανάληψης προσωπικής ευθύνης»

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, και γνωρίζοντας τις συνέπειες της λογοκλοπής, δηλώνω υπεύθυνα και ενυπογράφως ότι η παρούσα εργασία με τίτλο «Ανάπτυξη του GeoMakeIt! Alchemist και GeoMakeIt! Designers: Ένα σύστημα διαμοιρασμού και διάδοσης αλλαγών του συστήματος μέσα απο γεγονότα, και αυτόματη δημιουργία προσαρμόσιμων διεπαφών χρήστη για το GeoMakeIt! Studio από αντικείμενα» αποτελεί προϊόν αυστηρά προσωπικής εργασίας και όλες οι πηγές από τις οποίες χρησιμοποίησα δεδομένα, ιδέες, φράσεις, προτάσεις ή λέξεις, είτε επακριβώς (όπως υπάρχουν στο πρωτότυπο ή μεταφρασμένες) είτε με παράφραση, έχουν δηλωθεί κατάλληλα και ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο ΔΗΛΩΝ
ΔΗΜΗΤΡΙΑΔΗΣ ΒΑΣΙΛΕΙΟΣ

Ημερομηνία

Υπογραφή

Ανάπτυξη του GeoMakeIt! Alchemist και GeoMakeIt! Designers: Ένα σύστημα διαμοιρασμού και διάδοσης αλλαγών του συστήματος μέσα απο γεγονότα, και αυτόματη δημιουργία προσαρμόσιμων διεπαφών χρήστη για το GeoMakeIt! Studio από αντικείμενα.

ΔΗΜΗΤΡΙΑΔΗΣ ΒΑΣΙΛΕΙΟΣ

Τριμελής Επιτροπή:

Ονοματεπώνυμο, ...(επιβλέπων/σα)

Ονοματεπώνυμο, ……

Ονοματεπώνυμο, …….

Επιστημονικός Σύμβουλος:

Ονοματεπώνυμο…..

ΠΕΡΙΛΗΨΗ

Τα παιχνίδια βάσει τοποθεσίας είναι ένα νέο και πρωτόγνωρο είδος παιχνιδιών που συνδυάζουν τον φυσικό και τον ψηφιακό κόσμο. Η ανάπτυξη τέτοιων παιχνιδιών είναι αρκετά περίπλοκη, απαιτώντας υψηλές δεξιότητες προγραμματισμού. Με την εισαγωγή υπηρεσιών όπως της πλατφόρμα GeoMakeIt!, υπάρχουν σημαντικές προσπάθειες για να μειωθεί το χάσμα μεταξύ της ανάπτυξης παιχνιδιών και των δεξιοτήτων προγραμματισμού που απαιτούνται για να ξεκινήσει κάποιος χρήστης. Δημιουργώντας μια πλατφόρμα που χρησιμοποιεί στοιχεία τύπου προσθέτων, plugins, που μπορούν να συνδεθούν, να συνεργαστούν και να παράγουν διαφορετικές εμπειρίες παιχνιδιών, προσπαθήσαμε να μειώσουμε τις απαιτούμενες γνώσεις και δεξιότητες, προσελκύοντας το ενδιαφέρον τόσο των έμπειρων προγραμματιστών παιχνιδιών, όσο και των ερασιτεχνών, με τελικό σκοπό το είδος αυτών των παιχνιδιών να ευρυνθεί. Με κίνητρο τα παραπάνω, συνεχίζουμε παρουσιάζοντας τρία νέα εργαλεία, το GeoMakeIt! Alchemist, GeoMakeIt! Designers, και GeoMakeIt! Meta Injection System, για περαιτέρω απλοποίηση της διαδικασίας δημιουργίας παιχνιδιών. GeoMakeIt! Alchemist, ένα σύστημα που βασίζεται σε εντοπισμό γεγονότων, στα οποία μοιράζεται και διαδίδει αλλαγές καταστάσεων τους, επιτρέποντας στους δημιουργούς του παιχνιδιού να εκτελούν ενέργειες όταν λαμβάνουν χώρα αυτά τα γεγονότα. GeoMakeIt! Designers, μια αυτόματη παραγωγή για διαδραστικές διεπαφές χρήστη, που επιτρέπει στους προγραμματιστές των plugin να εξάγουν προσαρμοσμένο περιβάλλον στον χρήστη απευθείας, μέσω του GeoMakeIt! Studio. Τέλος, το GeoMakeIt! Meta Injection System, που επιτρέπει την αυτόματη εισαγωγή μεταδεδομένων, τη δημιουργία υψηλότερων επιπέδων αφαίρεσης και την απλοποίηση της διαδικασίας ανάπτυξης πρόσθετων.

ABSTRACT

Location based games is a new and primitive gaming genre, combining the physical and digital world. Developing such games is far from being trivial requiring high programming skills, but with the introduction of services such as the GeoMakeIt! Platform there is a significant attempt in lowering the gap between game development and the programming skills required to get started. By creating a platform that uses customizable plugin components that can be synthesized and output diverse game experiences, we attempted to lower the knowledge and skill gab required, attracting the interest of both experienced game developers and amateurs to diversify this genre. Motivated by the above, we continue by introducing three new tools, GeoMakeIt! Alchemist, GeoMakeIt! Designers and GeoMakeIt! Meta Injection System, to further simplify the games creation process. GeoMakeIt! Alchemist, an event-driven system that shares and propagate state changes, allowing game authors to execute actions when events take place; GeoMakeIt! Alchemist, an automatic generation for interactive user interfaces, allowing plugin developers to export custom UI directly on the GeoMakeIt! Studio; and the GeoMakeIt! Meta Injection System, allowing for automatic import of meta data, creating higher levels of abstraction, and simplifying the plugin development process.

Table of Contents

CHAPTER 1: INTRODUCTION

(Games 1.1)

(History of Games 1.1.a)

As early as 3000 BC[1] and inevitably throughout the rest of human history, games have been woven into our lives and etched into all cultures. They are usually carried out for fun and entertainment, while at other times they become a valuable method of education. Games can be distinguished in to two categories, the ones that can be played alone, commonly called single-player games, or the ones played in teams of people, usually referred to as multiplayer games. They represent a form of entertainment that is enjoyable both by being an active or a passive member of the game, a player or in the audience.

Games generally involve mental or physical stimulation and more often than not; both. They usually consist of a set of rules, a goal to achieve, interactions between components of the game or even other players, and regularly must present some form of challenge.

(Mobile Games 1.1.b)

Mobile Games are played on a mobile device, commonly a smartphone, a tablet or a hand-held console. It is also not unheard for games to be developed and played on smartwatches, PDAs, portable media players or even calculators! The earliest example of a mobile game in development would be a variant of Tetris[2] for the MT-2000 device. Mobile games were destined to become one of the major players in gaming platforms as mobiles became an affordable commodity for everyday use and a powerful computing machine.

Proof of this was already made in 1997, when Nokia launched the well-known game Snake[3]. Found in more than 350 million devices, Snake became one of the most played games of all times.

Today, mobile games are usually downloaded through an app store, typically the "Google Play Store" on Android devices and the "App Store" on Apple devices. It is also common for some mobile games to come pre=installed on some devices.

Unlike the traditional revenue model[4] for physical games; such as football, basketball, hockey of paying to watch; or the common pay-to-own model that PC and Console games have adopted; Mobile game developers opted for a different revenue generation model.

There are 5 revenue models for mobile games:

- "Premium"; where the user pays for the game in advance and additional downloadable content may be available either through free updates or microtransactions.

- "Freemium"; where the user tries the game for a limited time and then proceeds purchasing the "Premium" version.

- "Free to play"; where the user can download and play the game for free, but there is usually a slow factor such as limited attempts per day or stamina that slows down the user's progress. These restrictions can usually be circumvented using microtransactions.

- "Ad-supported"; where the user can play the game freely but is occasionally shown ads.

- "Subscription"; where the user must pay a monthly subscription to gain access to the game.

Most games usually combine the above methods to increase income. Such an example would be the commonly used "Free to play" and "Ad-supported" models, to make the game available to a wider audience while monetizing it.

Finally, due to their nature, mobile games have some unique limitations. Unlike consoles and computers, mobile games have limited storage and memory, which limits quality, size, and direct migration from PC or console to mobile development. Being mobile also means that battery life is limited, so playtime could potentially be limited by the available battery.

(Location-based Games 1.1.c)

Mobile games that actively use GPS as a core mechanism can be called location-based games. Unlike other types of games, this genre of mobile games takes the player's coordinates into perspective and uses them and his movements as the main elements of the game.

The genre has had a few commercial successes, while most games and projects of this type are developed as scientific research[5] or educational material[6]. Pokémon Go[7] is one of the widely known games that belongs in this category. Developed by Niantic, and released in 2016, it had over 147 million active users as of May 2018, generating over $3 billion in revenue as of 2019. Another popular game is GeoCaching[8], a global treasure hunt game that allows users to visit locations, hide boxes called "caches" containing a journal and a pen or pencil, while additionally having some gifts left from other people's daily adventures.

These genres of games only make tiny percentage of the global gaming market, which can be explained by the lack of knowledge and skills among people who want to develop such games, the high cost of development and maintenance and the lack of standardized tools. However, this leaves a spot open for adopting, testing, and exploring what can be done with this genre if more specialized development tools existed.

(Game Development 1.2)

(Definition and Process 1.2.a)

Video Game Development is the process of developing a video game. There are two common groups that undertake this effort: companies, and individuals. In the former, the development is usually undertaken by a team of programmers, designers, artists, sound engineers, testers, and publishers. Alternatively, games are designed either by a single individual, or by a small independent team. This independent effort is formally called indie development, where the resources are usually limited and generally lack the financial support of a publisher.

The latter has seen an increase recently, with indie game developers using advanced game engines such as Unity and Unreal Engine to create complex games that can be released on existing online distribution platforms.

The development process[9] is organized and can be divided into phases; but it is also labor-intensive. The first phase is called "pre-production", where the idea for the game is formed, the design, core mechanics and materials or art-style are decided. This phase is usually visualized on a Game Engine in the form of a "Game Draft"; or by using the old-school pen and paper method. Then we move on to the "production" stage, where game design, programming, stage production, art production and audio production take place. These are combined and thoroughly tested. Then there is a cycle between "Development" and our next step, "Milestones", where the first playable is released, going from "Alpha" to "Beta" and lastly "Full". Finally, the "post-production" phase, involves keeping the game stable and maintaining it for a long time, with regular updates to fix issues or even improve the gameplay and expand the content.

(Game Engines 1.2.b)

A Game Engine or Game Framework is a software designed to help people create video games[10]. These allow game developers to export games for different platforms such as PC, consoles, mobile devices, etc., and provide many tools for 2D and 3D game development. Common elements that make a functional and powerful game engine are the: graphics engines, physics engines, sound, animation, threading, scripting, and memory management.

There is a plethora of free, proprietary, or even in-house developed game engines. Some of these are designed for specific applications and hardware, while others can be ported to multiple platforms. Some examples of game engines that are commonly used today are Unreal Engine, Unity Unity, Godot Engine, Defold, RPG Maker, and Game Maker Studio[11].

Even if we chose to export only for the Android platform, there are still many options available[12]. One of the most common game engines for Android is Unity, where Pokémon Go, Lara Croft Go and Angry Birds were released. Unreal Engine is also a good option, with well-known games like Heart at Attack and Lineage 2: Revolution. Other engine options available 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[13][14]. Most game engines also contain pre-programmed elements that help in game development.

The big difference between the two is the target audience. Game engines are usually aimed at people with some knowledge of game development and usually some level of skill in some programming or scripting language. Authoring tools are developed usually for non-programmers, usually as educational software, to easily create software with pre-programmed features.

Authoring tools hide programming functions behind buttons, simple text commands, and other tools, so the author doesn't require to have any programming skills. Authoring systems typically provide many of the typical graphics, interactions, and prepackaged scripts that an author may need. In addition, authoring tools can manage tasks such as backend compilation/building, hosting, and delivery to end users. Similar to game engines, authoring tools may contain an authoring language, but they are only created to represent the system with limited commands to avoid overwhelming the author.

(Android Operating System 1.3)

(Introduction 1.3.a)

Android[15][16] is a mobile operating system based on a modified version of the Linux kernel, designed primarily for touchscreen devices such as smartphones and tablets. Additionally, it is used an operating system for smart TVs, smart watches, game consoles, digital cameras, and computers. It is free and open source, licensed under the APACHE license, and developed by a consortium of developers and commercially sponsored by Google.

Software packages are usually distributed through proprietary app stores such as the Google Play Store, Samsung Galaxy Store, Huawei AppGallery, Aptoid or F-Droid using the APK[17] file format directly.

As of 2011, Android is the best-selling mobile operating system for smartphones, with more than 2 billion monthly active users and 3 million apps.

Many phone manufacturers use the Android operating system directly, while some of them choose to modify its appearance and add additional features to differentiate themselves from other competitors.

Apps developed for the Android OS are developed using the Android Software Development Kit, commonly known as the Android SDK, often using Kotlin, which has recently replaced Java as the Android programming language of choice.

(Android Architecture 1.3.b)

As seen in **Figure 1.1**, the Android Architecture[21] can be divided into 4 layers.

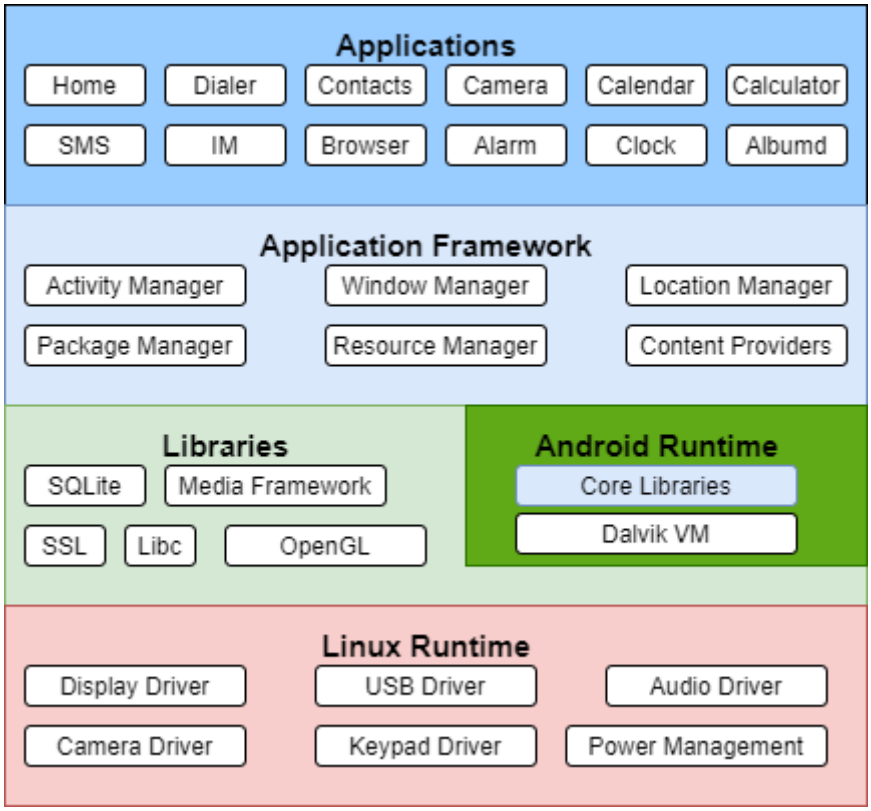

Figure 1.1 Android Architecture

Starting from the bottom layer, the "Linux Kernel" provides a level of abstraction between the device hardware and provides all the necessary drivers such as display, camera, Bluetooth, USB, sound, etc.

The next layer contains common "libraries" used both by Android itself and by most applications developed by third parties, such as WebKit, an open source web browser engine, SQLite for reading and storing data, Media Frameworks for playing and recording sound. and video, Libc shared library, SSL libraries, graphics libraries, etc. In addition, all Java-based Android libraries for third-party applications are stored in this layer, 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 applications developed for Android, and a set of C /C++ based basic libraries.

The "Application Framework" is built on top of the "Library" layer and contains higher-level services that Android developers use to develop their applications. Key services include "Location Manager", "Activity Manager", "Content Providers", "Resource Manager", "Notification Manager" and "View System".

The last layer is the "Application" layer, which contains all the developed applications. These include pre-installed apps like Dialer, SMS, Camera, Clock, Alarm Clock, etc. and third-party apps that can be downloaded from the App Store or installed externally.

CHAPTER 2: GeoMakeIt! PLATFORM

(Concepts 2.1)

(Introduction 2.1.a)

GeoMakeIt! is a platform for developing geo-based applications, typically games, for users without prior knowledge of computer software programming, game development, use of game engines, or development in the Android ecosystem. To achieve this, GeoMakeIt! provides an extendable API, called GeoMakeIt! Plugin Development Kit, where 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 in multiple games and in various ranges of configurability. As such, a single plugin may have different graphical design, content, actions, and results depending on the implementation chosen by the Game Author. Typical examples of plugins are:

- Statistics (such as score, health, energy, etc.)
- Mechanics (such as PvP, PvE combat, quests, quizzes, etc.)
- User Interface elements (such as alert dialogs, pop-up messages, etc.)
- Map elements (such as circular and polygonal zones, markers, etc.)

Thanks to the combination of well-configured plugins, a game developed by GeoMakeIt! can provide unique aesthetics and gameplay.

(Audience 2.1.b)

There are two different audience groups GeoMakeIt! aims to target:

1. Primarily, **people with little or no prior experience** with the Android ecosystem, software development, and game engine, where the expected end users will be children, teachers, or people exploring easy ways to prototyping and release a game.

2. Subsequently, a secondary target audience could be **software developers**, where they can rapidly develop and test different prototypes of geo-applications.

(User roles 2.1.c)

Three user roles are currently recognized and distinguished on the GeoMakeIt! Platform, which are as follows:

1. **Game Authors (aka Game Developers):** 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 on both coding and the Android eco-system, looking to create modular, reusable plugins for the GeoMakeIt! community and to improve GeoMakeIt! PDK.

3. **Players:** Users that download and play the released game.

Users are eligible for more than just one role. For example, a game author can also become a plugin developer, by creating plugins for GeoMakeIt!. Plugin developers are additionally game authors, as they must demo and test their plugins by using the GeoMakeIt! Platform and the GeoMakeIt! Studio. Both game authors and plugin developer become players as soon as they test their game or play with other people on their mobile devices.

(Platform: GeoMakeIt! Studio 2.1.d)

GeoMakeIt! is an authoring tool for quick and easy development of prototype or fully-fledged geo-games. To achieve an easier user interface for the customization and configuration efforts of the game author; the publishing and testing of the plugin developer; and the browsing and exploring different game options for the player; we created a web-interface named GeoMakeIt! Studio, where each user can perform their tasks depending on their role. In more detail:

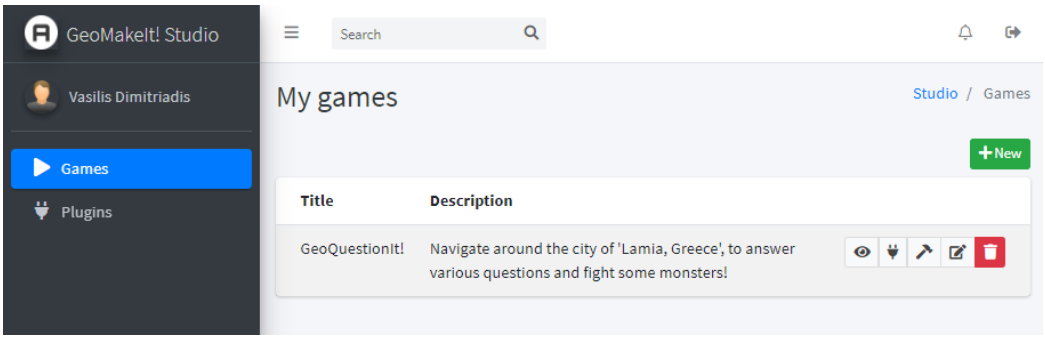

Figure 2.1 A page from the web application "GeoMakeIt! Studio"

Game Authors

When the role of the user is a game author, the user interface is modified to fit a better approach for game development, allowing an author to prototype, build and publish a game in a few easy steps. The author can start from a blank project and select a set of plugins or choose one of the default and common game templates to bootstrap his idea. Following that, the game author can customize the game by configuring said plugins.

Plugin Developers

When the role of the user is plugin developer, the user interface is modified to fit a better approach for plugin development, allowing the developer to create, register, upload and publish their plugins on the platform. Additionally, GeoMakeIt! Studio provides links, tutorials, documentations, and open-source plugins to spark the imagination of the developer. This way, a developer with no knowledge regarding GeoMakeIt! can easily understand on how to develop, implement, publish, and maintain their plugins.

(GeoMakeIt! PDK (Plugin Development Kit) 2.1.e)

GeoMakeIt! PDK is the core component of any plugin and game developed using GeoMakeIt!. It runs concurrently with the Android API and takes part in many processes, some of which are highlighted below:

- Plugin Management[updated] (Enabling/disabling plugins, managing dependencies).
- Database Management[updated] (Providing the ability to communicate with the games database).
- Core Android Features (Providing information about core features, such as current active activities).
- Alchemist Features[new] (Providing an event-based trigger-action model).
- Designer Management[new] (Providing a way to generate UI components for GeoMakeIt! Studio)
- Command Management[deprecated] (Providing the ability to execute plugin features).
- Placeholder Management (Providing the ability to retrieve, reuse and share plugin information between plugins, such as player username, score, health, statistics etc.).
- Permission Management[updated] (Requesting permissions and executing actions if no required permissions are given).
- User Authentication.
- User Location Information.
- Predefined Models (Alert Dialogs, Markers, Quests and Zones).
- Predefined User Interfaces (ex. Login/Register, Permission Request etc.).

Each GeoMakeIt! Plugin also includes an instance of the GeoMakeIt! PDK, serving two purposes; attaching it to any GeoMakeIt! Game and ensuring plugin compliance.

(Technologies 2.2)

(Application development platforms 2.2.a)

The GeoMakeIt! Platform is split between two core components: The GeoMakeIt! PDK, used to develop plugins, and the GeoMakeIt! Studio, a web interface allowing the creation, configuration, management and publishing of games and plugins developed with GeoMakeIt!. Each one of these components use a set of technologies that we found most fitting for the development purposes of the platform.

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

The GeoMakeIt! PDK was developed using a mix of Kotlin and Java, using Android Studio as its development platform. Any GeoMakeIt! Plugin is similarly recommended to use and follow the same tools and technics to provide consistency. While it is possible to develop plugins using solely Java and any other compatible development platform, e.g., Eclipse, it is not covered by any Chapter in this thesis nor included currently in any of GeoMakeIt!'s documentation.

Since GeoMakeIt! PDK was developed from the ground up, more details regarding its in-depth design of it will be discussed on Chapter 3.

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

GeoMakeIt! Studio was developed primarily with PHP; using the Laravel v7.0[19] framework, and JavaScript, using Angular JS[20] as it's front-end.

Laravel was chosen as it was simple and easy to learn, favored by the web-development community, with promising features, and it is widely known. This would ensure that GeoMakeIt!, as an open-source project, could be easily improved in time both by the original developers as well as by third-party contributions. It is a PHP Framework that has lately gained popularity, supposedly due to the PHP 7 improvements.

Designed for the Artisans web developers and with the latest trendy design choices in mind, Laravel is an MVC Framework feature rich and easy to develop on. It is considered the ideal framework for PHP developers with the learning curve of this framework being a lot easier than other competitors, thus making even people with little prior knowledge in PHP to use it. It comes with detailed documentation, describing everything from its ORM, to help in abstraction and automation, all the way to the Blade Template Engine. Additionally, it allows easy testing and automation, and on top of that, it provides a powerful database migration system. It is typically used for creating 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. While Laravel's primary design is to create powerful CMS, that's not its only intent. This allowed us to use some of its powerful features directly on this project.

Angular JS was chosen for the front-end development of GeoMakeIt! Studio, as it is widely known and provides an important set of features for any front-end application,

such as Data Binding, Model View Controller, Dependency Injection and since it is JavaScript based it also cross-platform. As any JS framework, it is built to work on a single thread, meaning there is no multithreading, but its performance is still one of the top ones. It provides many Asynchronous IO APIs, works great with our existing API and it is quite easy to handle concurrent requests.

(Maps API Choices 2.2.d)

There is a plethora of solutions when it comes to mapping APIs, but eventually we chose to use the most widely known solution regarding implementing a map system to the Android API; Google Maps[21]. This 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[22];

"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 each game, there could be significant price-differences per game developed, as it is request dependent. This restriction is slightly lifted in this project instance, as most of the features GeoMakeIt! is using are free when implemented in the Android environment.

(Database Choices 2.2.e)

Lastly, there are two types of databases used with GeoMakeIt!; SQL and NOSQL databases.

SQL is used for GeoMakeIt! Studio, as it has some pre-defined structures, and it benefits using a relational database with table-based design.

NOSQL is used by the GeoMakeIt! PDK since it is easier for the game developers to interact with other plugins using its document-based design with the key-value system. Additionally, by their nature plugins are using JSON and will contain a lot of unstructured data so choosing NOSQL was the more appropriate option.

GeoMakeIt! Studio

For GeoMakeIt! Studio we picked MySQL[23]. This decision was primarily made as MySQL is an open source, free and widely known relational database management system, with it dominating the database marker of SQL databases. Additionally, we can use MySQL to create server-side triggers when necessary and use multiple transactions to make sure all data are written safely. Moreover, it is cheaper to maintain since it is widely used as a general-purpose database and there is substantial support for it.

GeoMakeIt! PDK

For GeoMakeIt! Studio we picked Google's Firebase[24]. 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 MySQL though, 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.

We will mention some of the advantages and disadvantages of choosing Firebase below:

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

It is never easy to choose which database to use in a project as the developers must stick with this option for all the development's timeline. Furthermore, this choice will affect the final design of the product.

We decided to use both and SQL design and a NOSQL design, where the first is applied on GeoMakeIt! Studio as the data is structured and predictable, while the second one is used for GeoMakeIt! PDK where the plugin developers can store arbitrary data, schema-less and do it for real-time applications.

While GeoMakeIt! Studio's database design is final, GeoMakeIt! PDK could implement a feature such as a database wrapper where the game author can choose which product he is going to use. This way, we can avoid the required use of Google's Firebase and include a wider range of alternative options. If such a wrapper is developed we can also allow the game author to choose a pricing scheme appropriate for the game he develops.

CHAPTER 3: GeoMakeIt! PDK

(Introduction to the PDK 3.1)

(Overview 3.1.a)

This chapter will serve as an introduction to the GeoMakeIt! PDK as presented in 2020-21[25], aka GeoMakeIt! API prior of 2021, explaining the main purpose of the PDK, how it works in conjunction with Android OS, the workflow, its functionality and how plugin developers can leverage it. Components that are unchanged will get a quick introduction without extended use-cases, were as new features will be discussed in more details. Specifically, **the GeoMakeIt! Alchemist, GeoMakeIt! Designers and GeoMakeIt! Meta Injection System** have their own separate chapters; Chapter 4: GeoMakeIt! Alchemist, Chapter 5: GeoMakeIt! Designers and Chapter 6: GeoMakeIt! Meta Injection System, respectively. Furthermore, we will discuss about features that were deprecated, either due to the introduction of the aforementioned features, or if they were deemed unnecessary.

The GeoMakeIt! PDK provides implementations for the management of Android API features. Instead of the classic top-to bottom approach of the Android API, GeoMakeIt! PDK uses a hybrid bottom-up and top-to-bottom system. This grants GeoMakeIt! plugins the power to handle some critical decisions regarding how a game should run, while allowing the game author to react to the run-time events by filtering them and executing actions when necessary.

To achieve that, GeoMakeIt! PDK 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, startup, and execution.
- Provides familiar UI for common features.
- Provides common utilities for handling plugin configuration.
- Manages placeholders for player feedback such as player name, score etc.
- Broadcasts and handles game events.
- Generates and exports intuitive UI for game authors to configure.

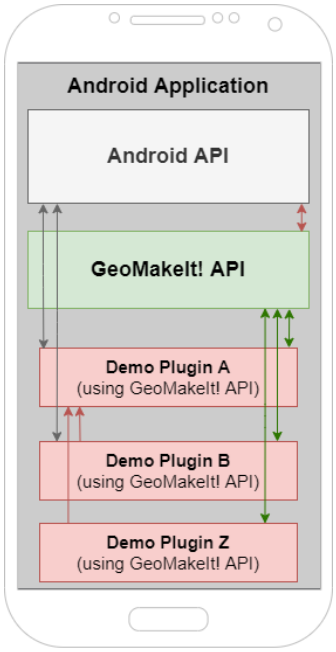

Figure 3.1 GeoMakeIt! PDK Communications and Dependencies

It should be noted that GeoMakeIt! is under continuous development, with its core features being updated, improved and many more features added down the road.

(PDK workflow 3.1.b)

The GeoMakeIt! PDK is an intermediary between the Android API and the game developed with GeoMakeIt!. Some libraries of the Android API are integrated with the GeoMakeIt! PDK to provide a common ground for plugins to implement common tasks, such as configurations, models, activities, databases etc.

Figure 3.1 shows all the possible ways of communication paths between essential components of a GeoMakeIt! Game, such as the `Android Application`, `Android API`, `GeoMakeIt! PDK` and a set of `GeoMakeIt! plugins`.

The GeoMakeIt! QuickStart Application serves only as the front-end and provides very basic functionality such as displaying the preload screen "*Made with GeoMakeIt!*" and initializing the GeoMakeIt! PDK.

The essential components are loaded when GeoMakeIt! PDK is initialized, such as registering events, authenticating the user, and loading all implemented plugins, in this case `Plugin A`, `Plugin B` and `Plugin Z`. It is worth noting that the loading order of each plugin is calculated based on their dependencies, starting with the ones that are stand-alone and require no other plugins to function, and finishing with the ones that are dependent on most of them. For example, `Plugin A` requires both `Plugin B` and `Plugin Z`, so it will be loaded last.

With all the plugins loaded, the application/game can start, and all the plugins perform their functions. Plugins can initiate activities, register events, perform actions, create placeholders, or even access classic Android function with no limitations and communicate with the Android API directly. When an event is triggered or actions need to be executed, such information is forwarded to the GeoMakeIt! PDK first, which in turn pushes the information to the appropriate plugin for processing. Furthermore, plugins can directly communicate with each other if they have verified their existence inside a built game.

(API Components 3.2)

(The "App" component 3.2.a)

The `App` component is the first thing that is registered by the game and includes vital information about the game's run-time features and data, including the database initialization and the current activity pointer. Moreover, it is a direct child of the Android's "`MultiDex Application`" and shares all its available methods. It allows a

plugin to get the game's current `context`, the `current activity`, and additionally it throws events such as when the activity has been `started`, `paused`, `resumed`, `destroyed` etc.

(The "Game" component 3.2.b)

The `Game` component has details regarding the currently built game. It is a straightforward way for any plugin to grab the `title`, `description`, `version` and `built type` of the game.

(The "Plugins" package 3.2.c)

This package includes all the needed tools to bootstrap a GeoMakeIt! Plugin. Plugin developers can create, instantiate and manager their plugins and dependencies. It contains 3 key components, the `Plugin`, `Plugin Manager` and `Plugin Logger` component.

The first one, `Plugin` component, is the core of any plugin, managing all common plugin flow such as enabling, disabling and registering each plugin and its sub-components. Additionally, with the introduction `DBPopulators`, `Meta Injection` and GeoMakeIt! Alchemist, which will be discussed later on, it serves as the common ground to grab such information.

The second one, `Plugin Manager`, manages all the included game plugins. Its main purpose is to register, enable and keep track of installed plugins in the running GeoMakeIt! Game.

Finally, the `Plugin Logger` component handles all the logging and debugging data produced by GeoMakeIt! Plugins.

Plugin component

All GeoMakeIt! Plugins extend the `Plugin` component. Information about the current state of the plugin; whether it is initialized, enabled, disabled; its current name, metadata, and assets it may contain; current version, hard and soft dependencies with other plugins; all of that is stored in this component. Furthermore it also manages the flow of each plugin by running the `onEnable()` and `onDisable()` methods which declare the actions to be performed by the plugin once it is enabled or disabled. Lastly, it contains shortcuts for accessing the plugin's `logger`, `configuration`, `placeholders`, `models`, `dbpolulators`, `actions`, `triggers` etc.

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! PDK 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

This is a JSON file generated by GeoMakeIt! Studio while a game is being built. It provides the paths for finding and loading every plugin included in the game.

A sample '`plugins.json`' file is provided below:

```
[
    {
        "package": "com.geomakeit.plugins.GeoQuiz",
        "main": "GeoQuiz"
    },
    {
        "package": "com.example.PluginB",
        "main": "Main"
    },
            . . . . .
]
```

Loading order

The plugin's loading order is decided based on which dependencies each plugin has. It is all handled through the `registerPlugins()` method. The first plugin to be loaded is always GeoMakeIt!. Following that, all plugins installed are resolved from the '`plugins.json`' file and their existence is verified. Following that, the registration algorithm can initiate, which is as follows:

1. Create a list for registered and unregistered plugins. Fill the unregistered plugins list with all the resolved plugins.
2. For each unregistered plugin:
    a. Check if plugin has no dependencies. If so, skip to `step d`.
    b. Check whether all hard dependencies are met. If so, skip to `step d`.
    c. Keep this plugin in the unregistered plugins list for next iteration and `continue` to next plugin (back to `step a`).
    d. Register plugin, store it in registered plugins list and remove it from unregistered plugins list.
3. If all plugins have been registered successfully, the registration process has been complete. The next process is to populate the database with the data contained in the JSON files of each plugin.

DBPopulators

The `DBPopulator` interface provides two public methods, one for populating and one for destroying a population of a database, named `populate()` and `destroyPopulation()` respectively. The code for these two methods is provided below, but first we will discuss regarding what a `DBPopulator` is.

Since plugin developers don't have a direct access to the database, and game authors lack the expertise to populate the database by themselves, we have created a system for automatically populating the database. When changes are detected on any data files that are marked with the interface DBPopulator, the new data is automatically uploaded on the game author's database. This way, upon compilation of the game, the data can be synchronized immediately.

```
interface DBPopulator {
    val populatorAsset: Asset

    suspend fun destroyPopulation() = withContext(Dispatchers.IO)
    {
        if (this@DBPopulator is ModelDBFactory<*> &&
            this@DBPopulator.populatorAsset is DataAsset)
        {
            val collection = getCollection(
                                this@DBPopulator.TABLE
                            )

            return@withContext deleteCollection(
                                    collection
                                ).await()
        } else {
            return@withContext null
        }
    }

    suspend fun populate() = withContext(Dispatchers.IO)
    {
        val tasks = mutableListOf<Task<Void>>()

        destroyPopulation()

        if (this@DBPopulator is ModelDBFactory<*> &&
            this@DBPopulator.populatorAsset is DataAsset)
        {
            val populator = populatorAsset as DataAsset
            val json = populator.asJSONArray()

            for (i in 0 until json.length()) {
                val data = json.getJSONObject(i)

                if (!data.has(Model.uniqueIdName)) {
                    continue
```

```
                }

                val attrs = Gson().fromJson(
                    json.getJSONObject(i).toString(),
                    Map::class.java
                )

                val cls = identifyObjectClass(attrs)
                val obj = cls.getDeclaredConstructor(
                    Map::class.java
                ).newInstance(attrs)

                tasks.add(obj.save())
            }
        }

        return@withContext tasks.map {
                                it.asDeferred()
                            }.awaitAll()
    }

    private fun getCollection(
                    collectionPath: String
    ): CollectionReference =
        App.getDatabase().collection(collectionPath)

    private fun deleteCollection(
                   collection: CollectionReference
                ): Task<Unit>
    {
        return collection
            .get()
            .continueWith {
                val res = it.result
                res.documents.forEach { doc ->
                    doc.reference.delete()
                }
            }
    }
}
```

Checksum

Additionally, to avoid multiple populations of the same file, we have added a checksum to verify if a file has been changed. The list of data file paths alongside with their checksums are uploaded in the game author's database and checked upon compilation of the game for any changes. If a checksum doesn't match, then that specific file is re-uploaded.

Plugin Logger component

The Plugin Logger gives all GeoMakeIt! plugins the capability to uniformly log messages in the Android Logcat. This way, any plugin message can be easily discovered by both the plugin's author as well as any other plugin developer when using dependencies. When calling this component, a developer can choose one of the following priorities: VERBOSE, DEBUG, INFO, WARN, ERROR and ASSERT.

(The "Activities" package 3.2.d)

This package handles connects all Android activities with GeoMakeIt! PDK. It includes two classes: JActivity and Activity Registry. Further future improvements will be added to these classes to cover wider range of Android Activity classes.

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

(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: register(), unregister() and isRegistered(). 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. Location Service which is provided by the GeoMakeIt! PDK is one such example. This service tracks the player's position and coordinates to later represent him on a map.

Location Manager Service

The Location Manager Service handles information about the current player's 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.

(The "Commands" package 3.2.f)

This package is currently in the grounds of being deprecated amidst the addition of GeoMakeIt! Alchemist, as the Alchemist changes the way developers interact with events and provides an easier method for game authors to react and perform actions.

The `Commands` package gives plugins the ability to register commands for the game author to use in the GeoMakeIt! Studio. Additionally, third-party plugins can also dispatch commands when deemed necessary. Three components belong to this package: 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.

(The "Placeholder" package 3.2.g)

Placeholders give plugin developers and game authors the ability to replace and parse certain plugin-registered keywords to present information to players. Placeholders allow for the creation of modular, customizable and personalized messages.

A simple example could be greeting a player. Instead of a generic message such as "Welcome to my amazing game!", the game author can greet players in a more direct manner such as "Welcome **George** to my amazing game!". This is made possible by using a placeholder similar to this: "Welcome **{{username}}** to my amazing game!".

Taking it a step further, a game author could provide more data from other plugins, for instance broadcasting the following popup message: "`Amazing! {{username}} has broken a new record a became the leader with` **{{leaderboard::most_points}}** `points`!".

Placeholders are vital in making a game more enjoyable and personalized.

(The "Asset" package 3.2.h)

Assets are files that are used by plugins and are placed under the '`/assets/<plugin_name>/`' folder. At the time of the writing, all asset files we have to deal with are JSON files, but that could change in the future. There are three type of asset files, `JSON`, `CONFIG` and `DATA`. Of these three, the last two are descendants of `JSON` file types. These files are also correspondently placed under '`/assets/<plugin_name>/`**`configs`**`/..`' and '`/assets/<plugin_name>/`**`data`**`/..`'.

(The "Configuration" component 3.2.i)

There are currently two configuration components. One under "configuration" package, aka "`configuration v1`", and a second one under "configurations", aka "`configuration v2`". The first one is to be deprecated and remains until GeoMakeIt! PDK v3.x.x, to give time for plugin developers for migration.

Both configuration components perform the same functionality but with different implementation. **We will be focusing only in the "`configuration v2`" as the previous component will be deprecated**. Plugin developers use this component to create simple, yet powerful, configurations that are visible on the GeoMakeIt! Studio and easily configurable by the game author. 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 essential details 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! PDK when the plugin is included. The Main is a class that *extends* the GeoMakeIt! PDK'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. Plugins can be version controlled 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! PDK 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.

Configurations V2

The Configuration component is designed to load JSON files, similar to the one discussed previously, 'plugin.json'. In this new version of the component, Java and Kotlin reflection is used in combination with the MetaInjector class (which will be discussed in a different chapter all together) to automatically parse, resolve and associate the JSON data with the implementing Configuration class file.

An example of an existing configuration can be seen below. The first code snippet shows the configuration object "Messages", which extends the Configuration component. There are two parameters required, the plugin and the filename of the configuration which are used to generate the appropriate asset path. Next, the variables contained in the object class are matched with the second snippet which contains the actual data stored. That data is extracted from the JSON file and stored inside the aforementioned variables.

```
object Messages : Configuration(GeoQuiz, "messages"),
                  ConfigDesigner
{
    lateinit var questionOnCorrectAnswer: String
    lateinit var questionOnWrongAnswer: String
    lateinit var timerOutOfTime: String
    lateinit var fillViewAnswer: String
}
```

```
{
  "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!"
}
```

Obviously, common, and more complex types are supported, for example: Boolean, Characters, Float, Lists, Maps, Nested Lists/Maps etc. An in-depth example can be seen below:

```
object DemoConfig: Configuration(GeoMakeIt, "demo"),
                   ConfigDesigner
{
    // Base types
    var demoBoolean by Delegates.notNull<Boolean>()
    var demoChar by Delegates.notNull<Char>()
```

```
        var demoDouble by Delegates.notNull<Double>()
        var demoFloat by Delegates.notNull<Float>()
        var demoInt by Delegates.notNull<Int>()
        lateinit var demoString: String

        // Lists
        lateinit var demoListBoolean: List<Boolean>
        lateinit var demoListChar: List<Char>
        lateinit var demoListDouble: List<Double>
        lateinit var demoListFloat: List<Float>
        lateinit var demoListInt: List<Int>
        lateinit var demoListString: List<String>

        // List of Lists
        lateinit var demoListOfListsStrings: List<List<String>>

        lateinit var demoMapStringToString: Map<String, String>
        lateinit var demoListToMapStringToString: List<Map<String,
String>>
    }
```

```
    {
      "demo_boolean": true,
      "demo_char": "a",
      "demo_double": 3.2832,
      "demo_float": 5.8732,
      "demo_int": 42,
      "demo_string": "Just a string",
      "demo_list_boolean": [true, false, true],
      "demo_list_char": ["a", "b", "c"],
      "demo_list_double": [1.32, 732.82, 11.11],
      "demo_list_float": [1.72, 832.723, 321.312],
      "demo_list_int": [42, 32, 22, 7112],
      "demo_list_string": ["Hello", "World", "From", "List"],
      "demo_list_of_lists_strings": [
          ["hello"], ["world"], ["from"],
          ["list", "of", "lists"]
      ],
      "demo_map_string_to_string": {
          "demo_a": "abc",
          "demo_b": "bcd",
          "demo_c": "cde"
      },
      "demo_list_to_map_string_to_string": [
          {
              "demo_a": "abc",
              "demo_b": "bcd",
              "demo_c": "cde"
          },
          {
              "demo_a": "abc",
              "demo_b": "bcd",
              "demo_c": "cde"
```

```
            }
        ]
    }
```

(The “Model” component 3.2.j)

The `Model` component is one of the most important classes of GeoMakeIt! PDK as it can represent any singular object inside the NoSQL database. It is an abstract class containing the fundamental methods and variables for a database object. There have been numerus improvements in this component but more importantly **support for asynchronous request from the database has been added**, making the whole Model design a task / promise like approach.

The `unique id` variable is the “primary id” / “document id” of the object. This identifier is used to search, store, update, delete the object from the database.

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

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

The `save()` method is used to save the model instance to the database, while the `delete()` method is used to delete it. Furthermore, the `update()` methods exists for updating the object with its changed attributes.

Additionally, there are other methods such as the `fresh()` method which is used to retrieve a fresh version of the model from the database, the `isSameAs()` and `isNotSameAs()` to check if two models are the same and the `collection()` which retrieves the whole collection containing the documents.

Lastly models use some components from the “The Databases package 3.2.k” section, which we will find below; allowing the plugin developer to do query searches such as grabbing all the documents representing a model, finding a specific one etc.

(The “Databases” package 3.2.k)

The `Databases` package contains classes and tools allowing plugin developers to access the shared database easier and without the need of knowing its underlying structure. There are three important components in this package: `ModelDBFactory` class, `DBPopulator` and `Inheritable` interfaces.

The ModelDBFactory class

The `ModelDBFactory` can be attached in any Model class as a companion object and gives that class immediate access to the underlying database structure. There are three key query methods in this class: `insert()`, `all()` and `find()`, all of which are asynchronous task methods.

The `insert(obj)` method inserts / stores a model inside the appropriate document collection.

The `all(source)` method is used to retrieve all the models of the model class from the document collection. Additionally, the source of retrieval can be specified, ex. `DEFAULT`, `SERVER`, `CACHE`.

Lastly, `find(key, source)` can be used to retrieve a specific document from the collection and parse it as a model.

More query methods will be supported in the future, but the plugin developer can still use some lower-level function to create more advanced queries.

Furthermore, if a plugin developer wishes to use the GeoMakeIt!'s custom document to model parser or to identify the object's class, they could use the `toObject()` or `identifyObject()` method's respectively.

The DBPopulator interface

This interface is used by a model's companion object to populate or destroy the population of data from the JSON files to the database. `DBPopulators` are working automatically with the plugin developer having only to add them as an extension of their companion's object class and setting the `populatorAsset`. A simple example can be seen below.

```
companion object: ModelDBFactory<Quiz>(),
                  DBPopulator, ModelDesigner
{
     override val populatorAsset: Asset by lazy {
         GeoQuiz.assets.getData("quizzes")
     }
}
```

The `populatorAsset` is the asset which contains the data to be populated in the database when the plugin is loaded by the game. In this case, it is the data asset "`quizzes`", which is the located in "`/assets/<plugin>/data/quizzes.json`" and has the following structure.

```
[
    {
        "unique_id": "quiz_small_1",
        "questions": [
           "question_1",
           "question_2"
        ],
        "next_question_delay": 1,
        "rewards_on_success": [
```

```
                "geomakeit::zone erase zone_question_1"
            ]
        },
        ...
    ]
```

The aforementioned JSON is automatically added to the correct document collection and uploaded to the NoSQL database as seen in **Figure 3.2**.

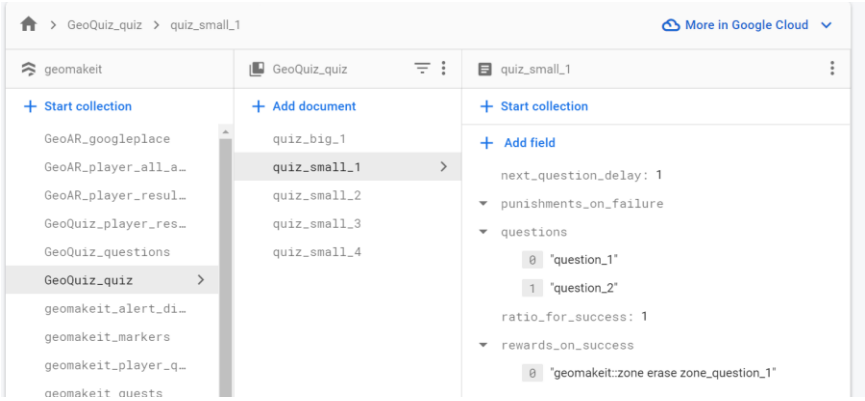

Figure 3.2 JSON converted to document collection in Goggle Firebase.

The Inheritable interface

This interface is used when models need to be inherited by child components. When this is the case, plugin developers append the inheritable interface on the model to let the `DBPopulator` and the `ModelDBFactory` know which methods to use to convert the retrieved document to the appropriate model class.

An example can be seen below:

```
companion object: ModelDBFactory<Zone>(),
                  Inheritable, DBPopulator
{
    override val TABLE = "geomakeit_zones"
    override val cls = Zone::class.java
    ...

    override val subclassField: String = "zone_type"
    override val subclasses: Map<String, Class<out Model>> =
        mapOf(
            "circle" to CircleZone::class.java,
            "polygon" to PolygonZone::class.java
        )

    ...
}
```

(The “Alchemist” package 3.2.l)

GeoMakeIt! Alchemist is an event-driven system to share and propagate state changes throughout plugins. This method allows for easy plugin interoperability and customizability, where plugin developers can create events and actions for game authors to listen and react to.

This is made possible with the concept of `Triggers`, `Actions`, `Filters`, and `Recipes`. When a state changes, an event is broadcasted to a set of recipes. These recipes use filters to further examine these state changes and if all the criteria are matched a set of actions can be executed.

Essentially, `Recipes` combine `triggers`, `actions` (and `filters`) to create the actual game-flow, for example:

- *When a player walks inside the store* (trigger), *the cashier welcomes him* (action).
- *When a player enters a zone* (trigger), *show him a notification* (action).
- *When a player gets a high score* (trigger), *reward him with gold* (action).
- *When a player loses health points* (trigger), *if his hp is equal or less than 0* (filter), *kill him* (action).

More regarding this package and its components will be discussed on a whole separate chapter: “**Chapter 4: GeoMakeIt! Alchemist**”.

(The “Meta” package 3.2.m)

This package contains classes and methods for generating meta-data regarding other GeoMakeIt! components such as the GeoMakeIt! Alchemist components. These meta-data are used to automatically load classes, generate UI components for the GeoMakeIt! Studio, and to allow some reflection methods while live-executing the game! More regarding this package and its components will be discussed on the following chapters, such as “**Chapter 5: GeoMakeIt! Designers**” and “**Chapter 6: Meta Injection System**”.

(Other GeoMakeIt! components 3.2.n)

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 functionallity are usually stored in the “*UI*” package. GeoMakeIt! PDK provides common user interfaces that can 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 etc.

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`.

Exceptions

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

Interfaces

Generic interfaces that can be used on multiple classes, objects, or third-party code in the GeoMakeIt! PDK are stored in the "Interfaces" package.

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

Arrayable, Jsonable, Mapable

These interfaces are single method interfaces containing `toArray()`, `toJson()` and `toMap()` correspondingly.

Expansible, SelfRegisterable

These interfaces are designed to either expand or register functionalities of a class or object.

The `IExpansible` interface for example is used when a class can be expansible. Such a class could be the Placeholders class or the Services class. These classes may be expanded by third-party plugin developers. As such, this interface has three methods `registerExpansion()`, `unregisterExpansion()` and `isExpansionRegistered()` which allow the plugin developer to register their expansions of these classes.

`SelfRegisterable` works in a similar way, but without the need of the plugin-developer registering the class or object.

Drawable

The `Drawable` interface is used when a class need to be drawn on a map. Two methods are included, `draw()` and `erase()`, each one of which contain a single argument, the map.

Overlappable

The `Overlappable` interface can be used when an object needs to be overlapped. Such example can be seen in the GeoMakeIt! Default Plugin[26], specifically 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.

SuspendableObjerver

The `SuspendableObjerver` performs similarly to the standard `objerver` but can be used with the Kotlin's `suspend` keyword.

Deprecated interfaces

Some interfaces have been deprecated as they may no longer be of need, they may have been renamed or repurposed into a different interface.

These interfaces are: `IArrayable`, `IDBLoadable`, `IDrawable`, `IExpansible`, `IFileLoadable`, `IHasAttributes`, `IJsonable`, `ILoadable`, `IMappable`, `IModelFactory`, `IOverlappable`, `ISelfRegisterable`.

CHAPTER 4: GeoMakeIt! ALCHEMIST

(Introducing the Alchemist 4.1)

(Overview 4.1.a)

GeoMakeIt! PDK uses an event-driven system to share and propagate state changes throughout plugins. This method allows for easy plugin interoperability and customizability.

In this chapter, we will explain how the GeoMakeIt! Alchemist components work, how a plugin developer can implement them in their own plugin, and how to give a game author all the tools they may need to customize a plugin.

(Introduction of the system 4.1.b)

First and foremost, just like any great alchemist we have some basic tools that we can use to create magic! These are `Triggers`, `Actions`, `Recipes`, `Filters`, and `Ingredients`.

`Triggers` are the **results of an event happening**, ex. "*John walked inside a store*", or.. "*Player entered the zone*", or.. "*Player got a high-score*", or.. "*Player lost health points*" etc.

`Actions` are **steps that are taken usually after an event was triggered**, such as "*The cashier welcoming the player*", or.. "*A notification popping up*", or.. "*Giving gold to a player*" or.. "*Killing the player*".

`Recipes` combine `triggers`, `actions` (and `filters`) to create the actual game-flow, for example:

- When a player walks inside the store the cashier welcomes him.
- When a player enters a zone, show him a notification.
- When a player gets a high score, reward him with gold.
- When a player loses health points, kill him.

`Filters` allow further customization by evaluating whether certain conditions may apply, for instance:

- When a player walks inside the store, if it is Friday, the cashier welcomes him.
- When a player enters a zone, if the zone is a fighting zone, show the player a notification.

- When a player loses health points, if he has less or equal to 0 hp, kill the player.

Lastly, `Ingredients`, allow the game author to pass arguments directly to the plugin, ex:

- When a player walks inside the store, if it is Friday, the cashier welcomes him by saying "Hello John, long time no see".

- When a player enters a zone, if the zone is a fighting zone, show him a notification with title "Warning" and description "You entered a fighting zone".

By understanding these five components, a plugin developer can take leverage of the whole Alchemist system and create a powerful, customizable, and interoperable plugin!

(Web user interface 4.1.c)

Everything discussed in this chapter is accompanied with a web user interface inside GeoMakeIt! Studio. This is possible through the GeoMakeIt! Designers discussed in "**`Chapter 5: GeoMakeIt! Designers`**".

Essentially, the whole system can be simplified to the following idea:

"*If a significant event is `triggered`, matching the provided `filter` criteria, then execute the following list of `actions`.*"

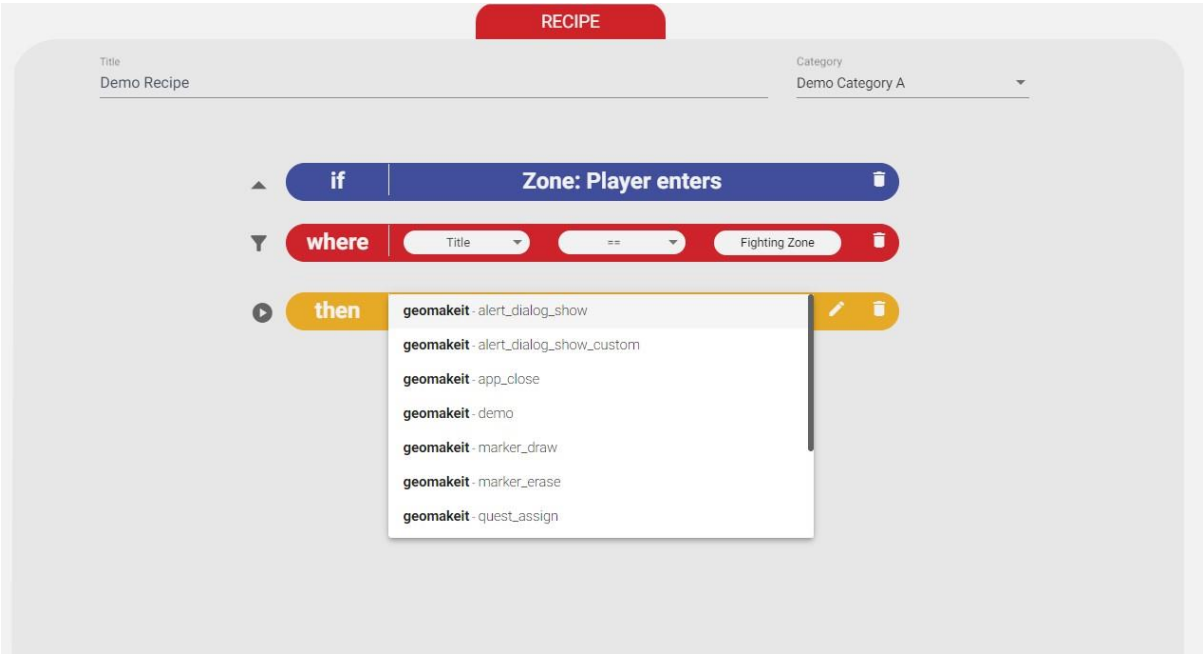

Figure 4.1 Web UI interpretation of GeoMakeIt! Alchemist with an IF <TRIGGER> WHERE <FILTER> THEN <ACTION> representation.

(Triggers 4.2)

(Overview 4.2.a)

Triggers are called when data changes or a note-worthy event happens. Game authors use triggers to detect these changes and perform their desired actions, altering the game-flow.

A simple case would be the following:

```
(Trigger) When a player enters a zone
(Action) Open an alert dialog and say "Hello {{username}}"
```

(Creating a Trigger 4.2.b)

Throughout a plugin's development progress, the plugin developer will create multiple triggers to signify whether a significant event has taken place. Depending on how detailed and customizable the plugin is going to be, a significant event could be something small like "*the player losing some hp*" or a big event such as "*the game has ended*".

At the end of the day though, there is only one question the plugin developer should ask before creating a trigger: "*Is this event significant enough for the Game Author to use it and alter the game-flow?"*

In both of the aforementioned examples, those events can and should alter the game-flow, allowing the game author to take actions when these events happen. For example:

- *When a player loses some hp* (trigger), *if the hp is less or equals to 0* (filter), *kill the player* (action).
- *When the game has ended* (trigger), *show alert dialog with title "Thanks for playing"* (action).

The following code represents the significant event of an alert dialog being shown to the player. In the next part, we are going in more details by dissecting the code.

```
object AlertDialogShownTrigger: Trigger("on_alert_dialog_shown"),
                                TriggerDesigner
{
    override fun getMetadata() = TriggerMetadata(
        pageTitle = "Alert dialog shown",
        pageDescription = "This trigger is executed when an alert
                           dialog is shown to the player."
    )

    override fun getIngredients(): Map<String, String> = mapOf(
        "unique_id" to "string",
```

```
            "title" to "string",
            "message" to "string",
        )
    }
```

By breaking down this code the following details can be distinguished:

1. **Triggers are singleton objects**, represented with the '`object`' keyword in Kotlin, instead of a class.

2. Each trigger **must extend the Trigger class**, providing a unique label per plugin, as seen in `Trigger("on_alert_dialog_shown")`, where `on_alert_dialog_shown` is a unique label chosen by the plugin developer.

3. Each trigger **must implement the** `TriggerDesigner` **interface**, allowing GeoMakeIt! PDK to create the necessary designer, providing the game author a way to view and use the trigger's results. We will be discussing what a designer is in the next chapter: "**Chapter 5: GeoMakeIt! Designers**"; but for now, you can consider it as an automatically generated UI describing the implementing class.

4. It is also very useful if the trigger overrides the `getMetadata()` function to provide a description of its use for the game author.

5. Lastly each trigger most often than not, **must override the** `getIngredients()` **function**, describing the kind of data this trigger contains as payload.

Essentially, the preceding code describes a trigger that is executed when an `Alert Dialog` is shown to the player, with a payload containing the alert dialog's `unique_id`, `title`, and `message`, where these ingredients can be used by the game author in a custom recipe.

The unique `label` variable

Each trigger must be uniquely identifiable per plugin. To accomplish that, the unique label variable is used. Usually, plugin developers chose a name similar to the class name. The most common technic is to use the snake-case version of the trigger's class name.

This unique name is then used to identify the trigger from third-party plugins and when using the web-version of the GeoMakeIt! Studio's, GeoMakeIt! Alchemist.

If a trigger is not unique an error is thrown on the logs while running / compiling the plugin.

The `getIngredients()` method

Each trigger must also declare its expected ingredients. These ingredients are used to inform the game author what data he can certainly expect when an event is triggered.

In the previous example, the data returned are `unique_id`, `title` and `message` of the alert dialog shown. These will be available for the game author to use in the GeoMakeIt! Studio's version of GeoMakeIt! Alchemist.

```
override fun getIngredients(): Map<String, String> = mapOf(
    "unique_id" to "string",
    "title" to "string",
    "message" to "string",
)
```

(Available Trigger Methods 4.2.c)

Usually, plugin developers are only required to `broadcast` their significant event; the trigger; so that the recipes that depend on it can execute their actions. GeoMakeIt! Alchemist does however provide some additional functionality in case it is needed to listen to specific events broadcasted by third-party plugins.

The list of available trigger methods can be seen below:

```
// Class methods
suspend fun broadcast(data: Model)
suspend fun broadcast(data: Mappable)
suspend fun broadcast(data: Map<String, Any?>)
fun launchBroadcast(data: Model)
fun launchBroadcast(data: Mappable)
fun launchBroadcast(data: Map<String, Any?>
fun attach(observer: SuspendableObserver)
fun detach(observer: SuspendableObserver)
fun getTriggerIdentifier(): String
fun isRegistered(): Boolean

// Companion methods
Trigger.get(input: String): Trigger
```

(Using Triggers 4.2.d)

Broadcasting Triggers

When a significant event has taken place, it is time for the plugin developer to send a Trigger signaling that change, alongside a `payload` with details regarding what has changed or other necessary data. For demonstration purposes, we will be using the trigger created in the '**Creating a Trigger**' section, alongside with a plain `Alert`

`Dialog` model, containing just the `title`, `description` as well as a method for showing the alert.

```
class AlertDialog: Model {

    var title: String = ""
    var message: String? = null

    fun show(...) {
        // Logic to show an Android Alert Dialog
        // ...
        // And after the Alert Dialog is shown, broadcast the
        // trigger.
        AlertDialogShownTrigger.launchBroadcast(...)
    }

    // ...
}
```

We can place the trigger just after the logic for showing the alert dialog, signaling that this event has taken place. There are two ways to broadcast a trigger: `broadcast` and `launchBroadcast`.

Both broadcast methods perform the same operation, with the only difference being that the first one is a `suspend` function (coroutine operation), while the latter is a wrapper around the `CoroutineScope(Dispatchers.Default).launch { .. }` operation.

Most of the times, plugin developers will be using the `launchBroadcast` function directly, while they may use the first one when needing to handle the dispatching operation by themselves.

Broadcasting parameters

When broadcasting a trigger, it is a good tactic to add a payload describing the data that were generated by this event. This step is required especially if the trigger contains Ingredients, by using the `getIngredients()` method, that can be later used by the game author with the web-version side of GeoMakeIt! Alchemist. An example is given below:

```
AlertDialogShownTrigger.launchBroadcast(mapOf(
    "unique_id" to this.unique_id,
    "title" to this.title,
    "message" to this.message
))
```

Which would in turn create a broadcast with the final payload of:

```
{
  "meta": {
    "time_sent": 1349333576093,
      ^ time in millis that the event was broadcasted
    "trigger_id": "trigger:::geomakeit::on_alert_dialog_shown",
      ^ unique label per plugin
     "user_id": "Q5JpJUtrKlfd0LlHrEwFizxtMjZ2",
      ^ unique user id
        ...
    },
   "data": {
     "unique_id" to "demo_alert_dialog",
     "title" to "This is a Demo Alert Dialog",
     "description" to "Containing a not-so-exciting but amazing
                       message"
     }
}
```

Broadcasting parameters (Models)

Given that commonly plugin developers require to broadcast the data of a whole model, the broadcast parameters are simplified by using an overloaded method, reducing the whole method to just passing the model itself.

```
suspend fun broadcast(data: Model)
fun launchBroadcast(data: Model)
```

Thus, changing the aforementioned broadcast to this:

```
AlertDialogShownTrigger.launchBroadcast(this)
```

Broadcasting parameters (Mappable)

The same idea applies to classes extending the GeoMakeIt!'s `Mappable` interface.

```
suspend fun broadcast(data: Mappable)
fun launchBroadcast(data: Mappable)
```

(Trigger Designers 4.2.e)

While we have created the trigger, we haven't made it accessible to the game author yet. The game author can't use directly the class that has been made since he doesn't have access to this code. This is where designers come in, and in this particular scenario a `Trigger Designer`.

The plugin developer can export this trigger to the web version of GeoMakeIt! Alchemist in the GeoMakeIt! Studio, creating an interactive user interface, by providing a trigger designer schematic. In this schematic plugin developers can provide information about the variables used, add tooltips, validations and even set how the input will be received.

More information will be discussed and in greater detail in **Chapter 5: GeoMakeIt! Designers**, where there are also some examples of Trigger Designers in use.

Checking whether a trigger exists

When a plugin developer has the unique identifier a third-party plugin's trigger and needs to verify whether it exists or not, the plugin developer can use the `Trigger.get()` function. This method can resolve both writing formats for triggers:

1. `trigger:::<plugin_identifier>::<trigger_identifier>`
2. `<plugin_identifier>::<trigger_identifier>`.

This function throws a `NoSuchElementException` if the trigger is not found. For example:

```
Trigger.get("trigger:::geomakeit::on_alert_dialog_shown")
  ^ Returns AlertDialogShownTrigger

Trigger.get("geomakeit::on_alert_dialog_shown")
  ^ Returns AlertDialogShownTrigger

Trigger.get("trigger:::non_existing::trigger")
  ^ Throws NoSuchElementException
```

Checking if a trigger is registered

Triggers are automatically registered by **GeoMakeIt!'s Meta Injection System**, as soon a plugin is loaded, unless a plugin developer specifies otherwise.

Prior to using a trigger from a third-party plugin, it is a good tactic to verify that this trigger is registered, using the `isRegistered()` method as seen below:

```
if (ThirdPartyPluginTrigger.isRegistered()) {
    // Do stuff...
}
```

(Other supported methods 4.2.f)

Grabbing a trigger's unique identifier

When a plugin developer needs to get the fully qualified name for their or a third-party's trigger, they can use the method `getTriggerIdentifier()` as shown in the examples below.

```
val x = MyCustomTrigger.getTriggerIdentifier()
  ^ trigger:::my_plugin::my_custom_trigger

val y = ThirdPartyTrigger.getTriggerIdentifier()
  ^ trigger:::third_party_plugin::third_party_trigger
```

(Web UI interpretation of a Trigger 4.2.g)

Triggers are collected by the Meta Injection System described on "**`Chapter 6: Meta Injection System`**", and they are exported as an interactable UI for the game author.

The game author can use the "*IF <TRIGGER> WHERE <FILTER> THEN <ACTION>*" as seen below to create a recipe.

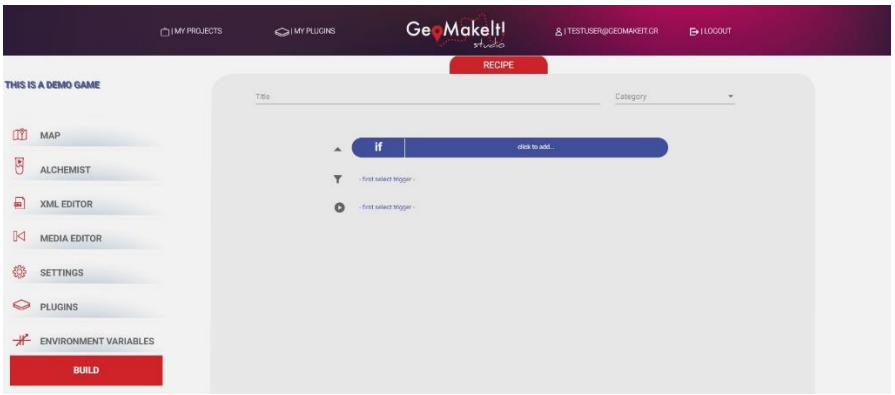

Figure 4.2 Creating a new recipe on GeoMakeIt! Studio

When needing to select a trigger, the game author is presented with a list of available triggers grouped by the plugin they belong to. Additionally, the description is visible to assist the game author for choosing the correct trigger.

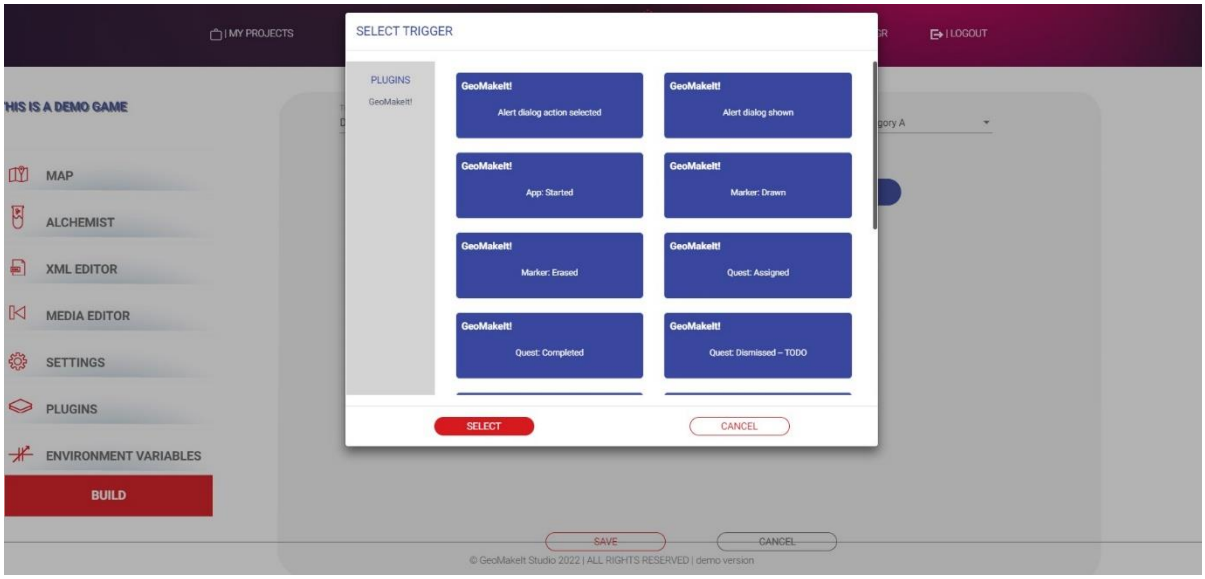

Figure 4.3 Selecting a trigger in GeoMakeIt! Studio

The selected option is then stored, and the game author can complete the recipe by filling the optional filters and executed actions.

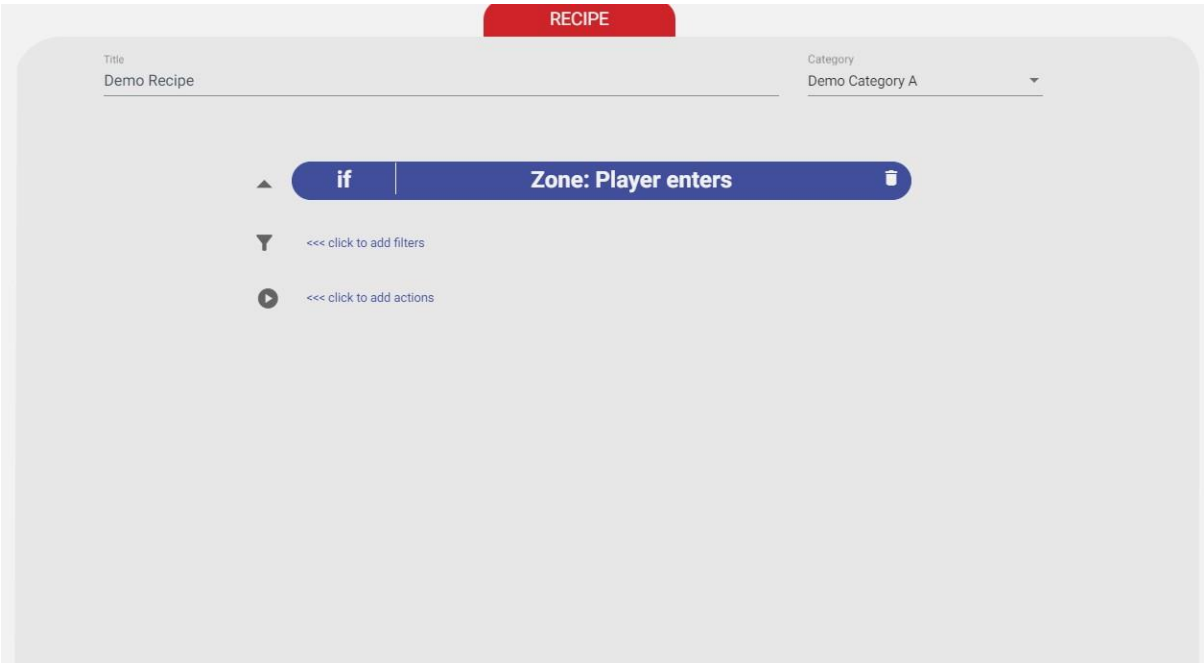

Figure 4.4 Trigger selected in GeoMakeIt! Studio

(Actions 4.3)

(Overview 4.3.a)

Actions are small work-steps that are executed after an event is triggered. Game authors perform actions to achieve their desired game-flow.

A simple case would be the following:

```
(Trigger) When a player enters a zone
(Action) Open an alert dialog and say "Hello {{username}}"
```

(Creating an Action 4.3.b)

Throughout a plugin's development progress, the plugin developer will have to create multiple actions, allowing game authors to interact with the plugin and its functionality. If for instance someone was making a scoring system, they would most likely develop at least these basic functionalities: `IncreaseScoreAction`, `DecreaseScoreAaction`, `SetScoreAction`, `SetHighScoreAction` etc.

At the end of the day though, before creating an action, any plugin developer should ask their selves: "*Should I expose this functionality of my plugin to the game author?*".

In all of the aforementioned examples, the game author should be able to have access to this functionality. Afterall, what kind of a scoring plugin wouldn't allow the game author to change the player's score? Some actual use-cases would be:

- *When a player kills a monster* (trigger), *increase score by 100* (action).
- *When a player dies* (trigger), *decrease score by 500* (action).
- *When the player signs in for the first time* (trigger), *set score as 500* (action).
- *When a player breaks a high score* (trigger), *set new high score as that score* (action).

As a quick example, the following code represents the action of showing an alert dialog to the player:

```
class AlertDialogShowAction(var dialogUid: String): Action() {

    override suspend fun execute(): ExecutionResult {
        val alertDialog = AlertDialog.find(dialogUid).await()

        if(alertDialog == null) {
            return ExecutionResult(null)
        }

        App.getCurrentActivity().runOnUiThread {
            alertDialog.show()
        }

        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        override val kcls = AlertDialogShowAction::class
```

```
        override val label: String = "alert_dialog_show"

        override fun getMetadata() = Metadata(
            pageTitle = "Alert Dialog: Show",
            pageDescription = "Display a saved alert dialog from
                               the database to the current player."
        )
    }
}
```

By breaking down this code we can spot the following details:

1. **Actions are classes**, that also use Kotlin's default constructor. This is the preferred option, allowing the Action Designer to get the correct variable order for the designer's data.

2. Each action **must extend the Action class**, which provides access to the `suspend fun execute()` method.

3. The `execute()` method is executed when the action is called. The action's logic goes in here. In this example, the logic presented uses the Alert Dialog Model to find a known alert dialog saved by the game author under the unique id `dialogUid`. If the alert is found, then we can show it inside the game.

4. The `execute()` method returns an `ExecutionResult`. This holds the result of the executing function. If we return `ExecutionResult(null)` then the execution has failed, otherwise we can return a map as payload with the results of the execution. There is a separate section discussing solely the execution result.

5. Each action must have a companion object implementing the `ActionDesigner` interface, allowing GeoMakeIt-PDK to create the necessary designer, providing the Game Author with a way to call the action with his desired parameters.

6. The `ActionDesigner` interface implements 2 variables, `kcls` (which stands for Kotlin class) and `label`. The first one is the class that the action designer is constructing, while the second one is the unique identifier per-plugin for this action.

7. It is also very useful if the trigger overrides the `getMetadata()` function to provide a description of its use for the game author.

Essentially, the preceding code describes an action that is executed when the game author decides to show to the player an alert dialog of a specific `dialogUid`, using a recipe.

(Using Actions 4.3.c)

Action parameters

Usually, when creating an action, the plugin developer is going to need some information provided from the game author, something like an input. Think of these parameters just like any parameter in a constructor, function, or method. When declaring these parameters, the `ActionDesigner` will filter out all the public ones and create an interface on GeoMakeIt! Studio, allowing the game author to fill them easily.

Consider the following simple example:

```
class IncreaseScoreAction(
    var playerId: String,
    var amount: Int
): Action() {

    // ..

}
```

This would automatically generate the following UI for the game author, alongside with a JSON encoding for a part of a recipe.

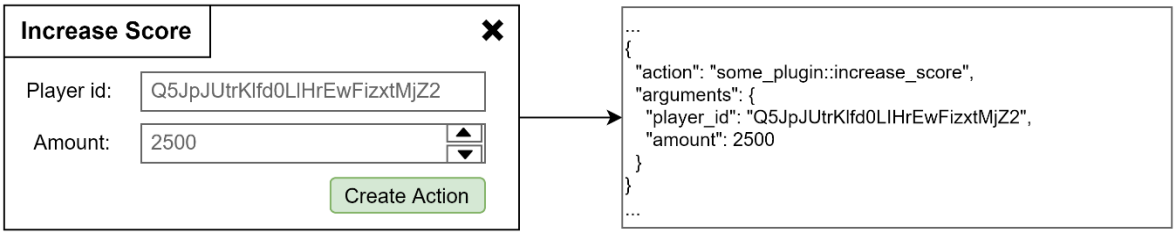

Figure 4.5 Generated UI on the left, JSON encoding on the right.

And this slightly more complex example:

```
class AlertDialogShowCustomAction(
    var title: String,
    var message: String,
    var positiveButton: AlertButton? = null
    var negativeButton: AlertButton? = null
) : Action() { ... }

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

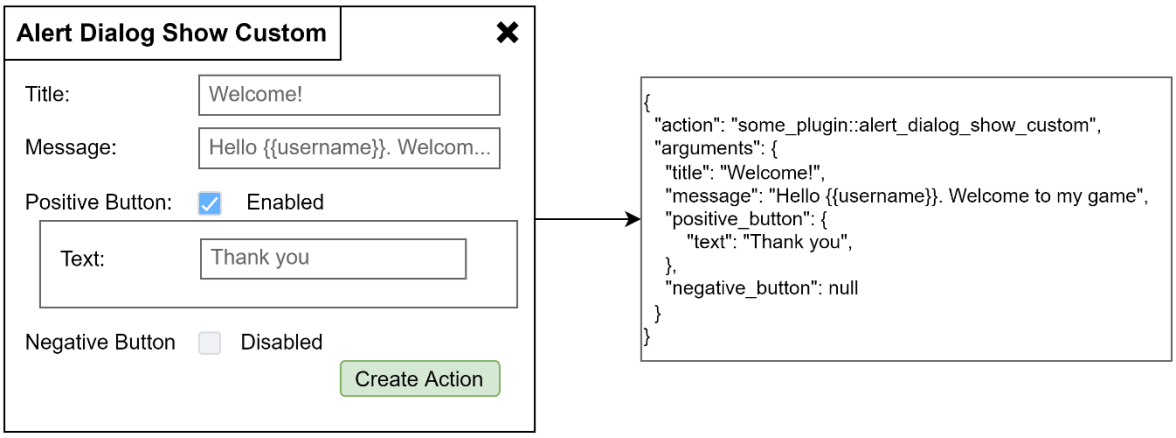

Figure 4.6 Generated UI on the left, JSON encoding on the right

Parameter Order

If an `ActionDesigner` doesn't override the `getDesignerOrder()` method, then the order of the primary constructor will be used.

The `execute()` function

This function contains the logic for the work that has to be 'executed' when the action is called. It can later return an `ExecutionResult`, that can be used by other actions to chain them.

It is a suspended function (Kotlin Coroutine), because most often than not, plugin developers need to access or request database and I/O operations.

You can see some actions and their `execute()` function examples below:

```
// Action that logs the user out
class UserLogoutAction: Action() {
    override suspend fun execute(): ExecutionResult {
        CurrentUser.signOut()
        App.getCurrentActivity().finishAffinity()
        return ExecutionResult(mapOf())
    }
    // ...
}

// Action that assings a quest to the user
class QuestAssignAction(var questId: String) : Action() {
    override suspend fun execute(): ExecutionResult {
        val assigned = PlayerQuest.assign(questId).await()
        if(!assigned) {
            return ExecutionResult(null)
        }
        return ExecutionResult(mapOf())
    }
    // ..
```

```
}
```

Execution Result

Contains the results of an action's execution. Fundamentally it's data wrapper around a map containing the results.

```
data class ExecutionResult(val result: Map<String, Any?>?) {
    fun executed(): Boolean = (result != null)
}
```

When a successful execution has to return a result, it can be appended like the following example:

```
override suspend fun execute(): ExecutionResult {
    // Execution logic...
    return ExecutionResult(mapOf(
        "user_id" to "Q5JpJUtrKlfd0LIHrEwFizxtMjZ2",
        "demo_data" to "some_data_here"
        "demo_nested_data" to mapOf(
            "nested_data1" to "x",
            "nested_data2" to "y"
        )
    ))
}
```

In some successful executions, when no results are needed to return, the aforementioned example can be simplified as follows:

```
override suspend fun execute(): ExecutionResult {
    // Execution logic...
    return ExecutionResult(mapOf())
}
```

While when there is a failed execution, the plugin developer can return null like so:

```
override suspend fun execute(): ExecutionResult {
    // Execution logic...
    return ExecutionResult(null)
}
```

(Action Designers 4.3.d)

While we have created the action, we haven't made it accessible to the game author yet. The game author can't use directly the class that has been made since he doesn't have access to this code. This is where designers come in, and in this particular scenario an `Action Designer`.

The plugin developer can export this action to the web version of GeoMakeIt! Alchemist in the GeoMakeIt! Studio, creating an interactive user interface, by providing an action designer schematic. In this schematic plugin developers can provide information about the variables used, add tooltips, validations and even set how the input will be received.

More information will be discussed and in greater detail in **Chapter 5: GeoMakeIt! Designers**, where there are also some examples of Action Designers in use.

(Web UI interpretation of a Trigger 4.3.e)

Actions are also collected by the Meta Injection System described on "**`Chapter 6: Meta Injection System`**", and they are exported as an interactable UI for the game author, similar to the way triggers are.

When needing to select an action, the game author is presented with a list of available actions. The game author can then select the desired action to be executed.

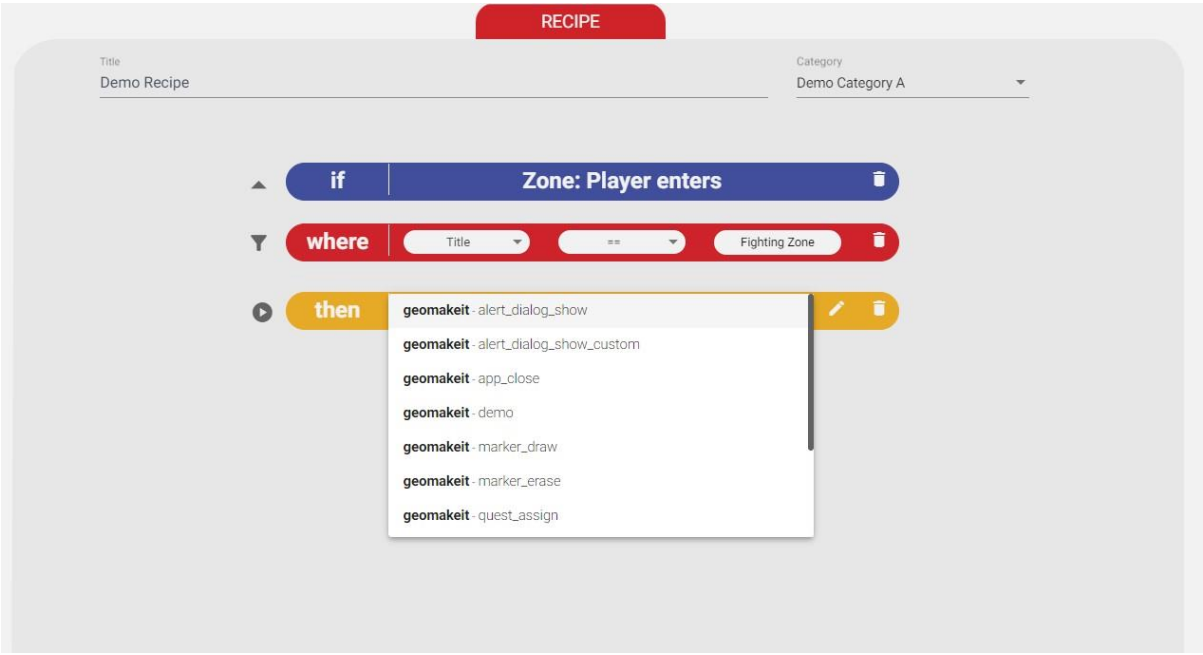

Figure 4.7 Selecting an action inside GeoMakeIt! Studio

Furthermore, the game author can supply arguments to the selected actions which are then passed to the plugin.

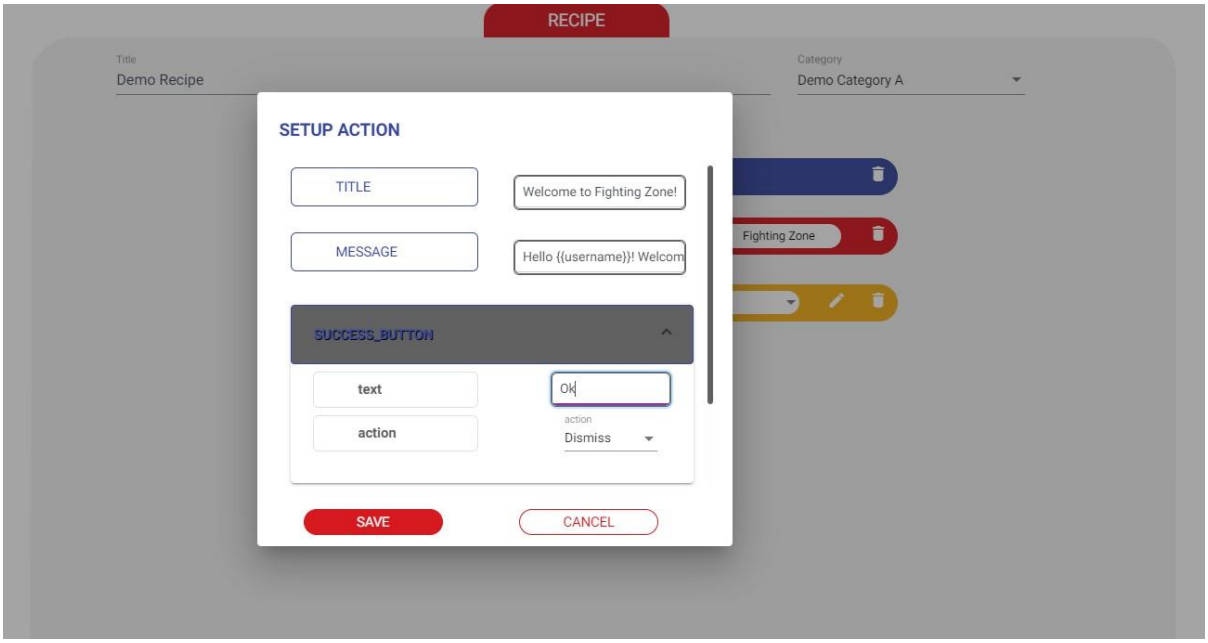

Figure 4.8 Supplying arguments to an action

(Recipes 4.4)

(Overview 4.4.a)

While game authors are not going to be writing custom recipes in JSON format, plugin developers are most likely going to encounter some recipes at-least while they are testing their triggers and actions in a local environment. Consequently, it is essential that they understand how recipes work and even how to write some custom ones by themselves.

The fundamental thing that a plugin developer needs to understand though is: "*what recipes do?*".

Recipes allow game authors to combine `triggers`, `actions`, and `filters` to create a highly customizable game-flow. Additionally, they allow for exceptionally easy plugin interoperability between their triggers and actions. Here are some examples

- *When a player gets a high score* (trigger), *reward him with gold* (action).
- *When John walks inside the store* (trigger), *if it is Friday* (filter), *the cashier welcomes him* (action).
- *When a player enters a zone* (trigger of GeoMakeIt), *if the zone is a fighting zone* (filter), *start a random fight* (action of a fighting plugin)
- *When a player loses health points* (trigger of health plugin), *if he has less or equal to 0 hp* (filter), *kill him and show a notification* (actions of health plugin and GeoMakeIt respectively).

(Creating a Recipe 4.4.b)

Recipes are normally created by the game author, inside GeoMakeIt! Studio, using the GeoMakeIt!'s Alchemist web-version. While the plugin developers are testing their plugins though, there aren't such luxuries yet! This means that plugin developers must understand and manually write the JSON for creating a recipe.

Recipes are stored inside the root project, specifically under this path:

`/app/src/main/`**`assets/alchemist/recipes.json`**

Opening that file for the first time would reveal a very simple recipe that would do the following:

Whenever a player enters a zone (trigger), *show an alert dialog* (action) *with title: "Demo Recipe", message: "This is just a demo recipe. Oh, hello {{username}}."* and *a button saying "Nice!"*.

This would be encoded in a JSON as follows:

```
[
  {
    "trigger": "trigger:::geomakeit::on_zone_enter",
    "filters": [],
    "actions": [
      {
        "action": "geomakeit::alert_dialog_show_custom",
        "arguments": {
          "title": "Demo Recipe",
          "message": "This is just a demo recipe. Oh, hello
{{username}}.",
          "positive_button": {
            "text": "Nice!",
            "action": "CANCEL"
          }
        }
      },
      //... other actions
    ]
  },
  //... other recipes
]
```

When it is the first time creating a recipe, there may not be a '`recipes.json`' file, but rather a '`..recipes.json`', which is prefixed with the '..' notation. The plugin developer must copy that file and rename it to the standard '`recipes.json`'. Every file prefixed with the '..' notation is considered a template to be modified.

Recipe Lifecycle

When a trigger is broadcasted all recipes listening to that trigger check whether their criteria are met and can be executed. Essentially, the recipe lifecycle goes as follows:

- **If the trigger** the recipe was listening to **is broadcasted**.
- And **all filters evaluate to true**.
- Then, **execute all actions** included in the recipe.

A more detailed flow chart of the recipe lifecycle is presented below.

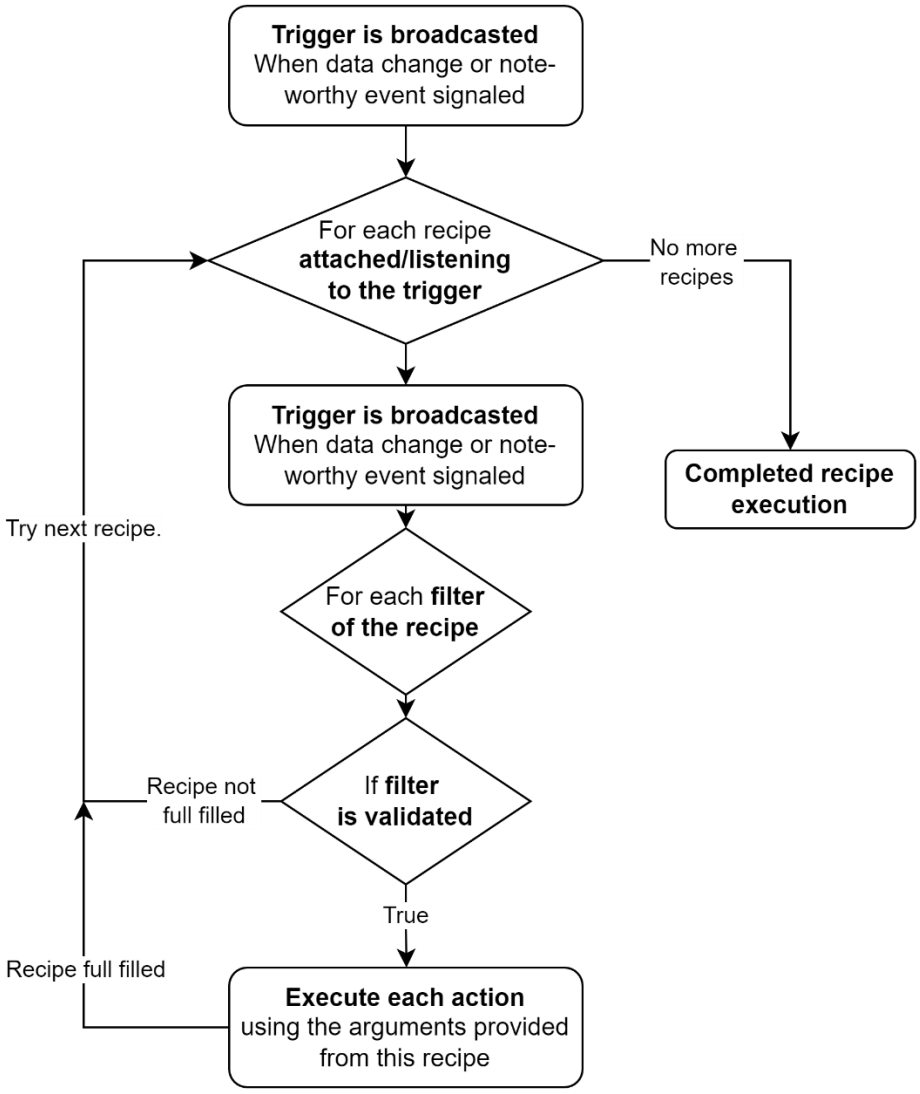

Figure 4.9 Recipe Lifecycle and algorithm

The 'trigger' object

This object contains the string representation of a trigger.

The following formats can be used:

- *trigger:::<plugin_identifier>::<trigger_label>* , or directly:
- *<plugin_identifier>::<trigger_label>* (Known as `trigger identifier`)

Where `plugin_identifier` is the unique plugin's identifier, and `trigger_label` is the unique per-plugin label described on '**The unique trigger label variable**' section of '**Triggers**'.

It is both possible and common for more than one recipe to be listening to a specific trigger. A very simple example is listening for the "`geomakeit::on_zone_enter_trigger`", which will be used multiple times filtered with different unique zones.

The 'filters' object

This contains a list of filter operations that can be performed with data payload received from a trigger. Operations include comparisons, list operators, string operators and more. A comprehensive list of combinations will be discussed later, on the '**Filters**' section.

- **(Case 1)** When **no filter** is used, the moment the trigger is broadcasted, all actions are executed.
- **(Case 2)** If a **single filter** is used, the filter must be evaluated to true for the actions to be executed.
- **(Case 3)** If **more than a one filters** are used, all the filters must be evaluated to true for the actions to be executed.

These cases are represented also in the actual JSON format below:

```
[
  // Case 1
  {
    "trigger": "trigger:::geomakeit::on_zone_enter",
    "filters": [],
    "actions": [...]
  },

  // Case 2
  {
    "trigger": "trigger:::geomakeit::on_zone_enter",
    "filters": [
      ["trigger:::unique_id", "==", "zone_question_1"]
    ],
    "actions": [...]
  },
```

```
    // Case 3
    {
      "trigger": "trigger:::geomakeit::on_zone_enter",
      "filters": [
        ["trigger:::unique_id", "==", "zone_question_1"]
        ["trigger:::visible", "==", true]
      ],
      "actions": [...]
    },
  ]
```

The 'actions' object

This contains a list of actions to be executed as soon as the recipe is validated, and all the filters evaluated to true. A comprehensive list of how to provide the arguments for an action can be found in the aforementioned '**Actions**' section.

For simple actions with no nested components the following example should suffice.

When this recipe is called and all the filters are evaluated to true, then:

1. First, the action `zone_draw` of GeoMakeIt is executed, drawing the zone with unique id of `zone_question_1`.

2. Next, the second action is executed, `app_close`, terminating the application.

These actions are formulated with the following JSON example:

```
{
  "trigger": ...,
  "filters": ...,
  "actions": [
    {
      "action": "geomakeit::zone_draw",
      "arguments": {
        "zone_uid": "zone_question_1"
      }
    },
    {
      "action": "geomakeit:app_close",
      "arguments": {}
    },
    // ...
  ]
}
```

While this example wouldn't be used in an actual game-flow, it is a simple example to show how actions are executed sequentially and how add the necessary arguments.

(Filters 4.4.c)

Filters allow for further customization on when should a recipe be executed. Filters can be compared to an 'if' statement. These filters evaluate whether certain conditions are met, and if that happens then the recipe can proceed into executing its actions.

While plugin developers are likely not going to be using the Filter classes directly, the following is going to provide vital information regarding how filters work for an in-depth understanding of the available operations per filter.

Filter structure

```
abstract class Filter {
    abstract val leftValue: Any?
    abstract val op: FilterOps
    abstract val rightValue: Any?
    // ...
    abstract fun getAllowedOperators(): List<FilterOps>
    protected open fun evaluation(): Boolean
}
```

Each filter `Boolean operation`, similar to operations in any programming language, containing a left operant (`leftValue`), right operant (`rightValue`) and the operator (`op`). Common examples of Boolean operations in a programming language could be the following:

- 15 > 3 (True)
- zone.isVisible == true (True if zone.isVisible is true, false otherwise

Similarly plugin developers and game authors can create such operations on their recipes using filters, where the operant can be either a primitive type (ex. `string`, `int`, `float`, `bool` etc.) or a trigger ingredient (one available from the trigger's `getIngredients()` method).

```
// Example 1: Check if the zone we entered
// has a unique_id of 'zone_question_1'
{
  "trigger": "trigger:::geomakeit::on_zone_enter",
  "filters": [
    ["trigger:::unique_id", "==", "zone_question_1"]
  ],
  "actions": [...],
},
```

```
// Example 2: Check if the zone we entered
// has a unique_id of 'zone_question_1' and
// if it is visible
{
  "trigger": "trigger:::geomakeit::on_zone_enter",
  "filters": [
    ["trigger:::unique_id", "==", "zone_question_1"],
    ["trigger:::visible", "==", true],
  ],
  "actions": [...],
}
```

The `getAllowedOperations()` method

This method lists the allowed operations for each filter. A simple and explanatory case would be the comparison of integers. In this case, the allowed operators would be: ==, !=, <=, >=, <, and >.

The `evaluate()` method

This method performs the evaluation of the `leftValue` and `rightValue` values with the `op` provided.

Web UI interpretation

Filters are also interpreted in the GeoMakeIt! Studio. Filters depend on the ingredients a trigger returns when it is signaled. These ingredients are available for comparison on the GeoMakeIt! Alchemist abroad the GeoMakeIt! Studio.

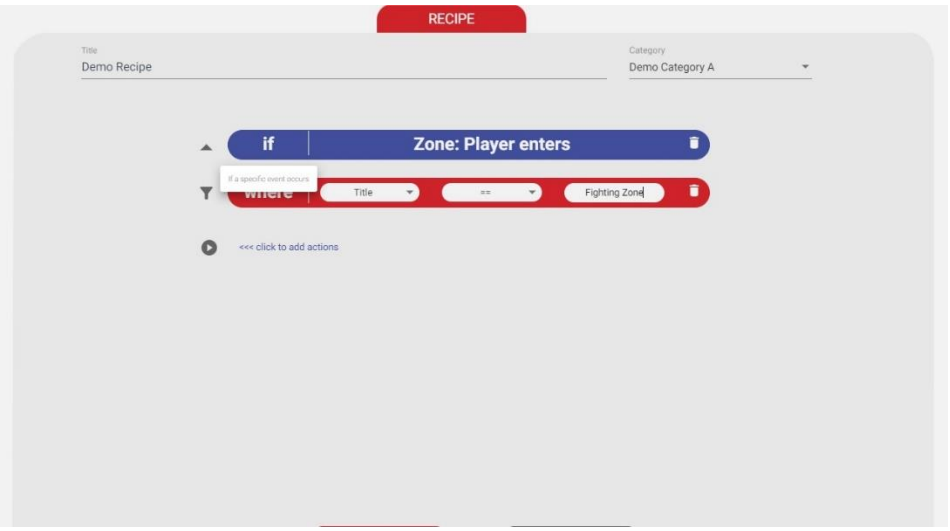

Figure 4.10 Filtering the event "On Zone Enter" of GeoMakeIt! using the zone's title

(Available Filters 4.4.d)

In this section we will be discussing the available filters for this version of GeoMakeIt! Alchemist, keeping in mind that these filters may change in the future adding more functionality or more complex operations. There are currently 9 filters available:

- `Boolean` Filter
- `Character` and `String` Filters
- `Double`, `Float` and `Integer` Filters
- `Model` Filter
- `Geo-point` Filter
- `Proxy` Filter

Boolean Filter

A `Boolean Filter` is used to compare and filter through triggered events by evaluating whether certain `Boolean` conditions apply, ex. true == false, true != false, something == true. The allowed operators are: ==, !=, and, or.

Character and String Filter

A `Character or String Filter` is used to compare and filter through triggered events by evaluating whether certain `Character` or `String` conditions respectively apply, ex. "a" == "b", "a" != "b", something == "a". The allowed operators are: ==, !=, <=, <, >=, >.

Double, Float and Integer Filters

A `Double, Float or Integer Filter` is used to compare and filter through triggered events by evaluating whether certain `Double`, `Float` or `Integer` conditions respectively apply, ex. 5 > 3, 2.5 <= 2.0, something == 8.32. The allowed operators are: ==, !=, <=, <, >=, >.

Model Filter

A `Model Filter` is used to compare and filter through triggered events by evaluating whether certain `Model` conditions respectively apply, ex. "Model A" == "Model B", "Model A" != "Model B", etc. The allowed operators currently are: ==, !=.

Geo-Point Filter

A `Geo-Point Filter` is used to compare and filter through triggered events by evaluating whether certain `Geo-Point` conditions respectively apply, ex. "(LatA, LongA)" == "(LatA, LongB)", "(Lat, Long)" IN [LatLong list], "(Lat, Long)" INSIDE [LatLong list] etc.

Proxy Filter

The `Proxy Filter` is an intermediate filter between the Recipe being executed and the final Filter being applied. It translates the input received from the recipe, the Ingredient Map, to the actual primitive and complex objects that will be executed on filters.

(Examples using triggers 4.4.e)

Simple Example (Single Filter)

```
{
    "trigger": "trigger:::geomakeit::on_zone_enter",
    "filters": [
      ["trigger:::unique_id", "==", "zone_question_1"]
    ],
    "actions": [
      {
        "action": "geomakeit::alert_dialog_show",
        "arguments": {
          "dialog_uid": "alert_demo"
        }
      }
    ]
 },
```

Default and (Filter1 and Filter2)

```
{
    "trigger": "trigger:::geomakeit::on_zone_enter",
    "filters": [
      ["trigger:::unique_id", "==", "zone_question_1"],
      ["trigger:::first_visit", "==", true],
    ],
    "actions": [
      {
        "action": "geomakeit::alert_dialog_show",
        "arguments": {
          "dialog_uid": "alert_demo"
```

```
            }
        }
    ]
},
```

Example with or (Filter1 or Filter2)

```
{
    "trigger": "trigger:::geomakeit::on_zone_enter",
    "filters": [
      [
        ["trigger:::unique_id", "==", "zone_question_1"],
        "OR"
        ["trigger:::unique_id", "==", "zone_question_2"],
      ]
    ],
    "actions": [
      {
        "action": "geomakeit::alert_dialog_show",
        "arguments": {
          "dialog_uid": "alert_demo"
        }
      }
    ]
},
```

Nested Example (Filter1 and (Filter2 or Filter3) and Filter4)

```
{
    "trigger": "trigger:::geomakeit::on_zone_enter",
    "filters": [
      ["trigger:::unique_id", "==", "zone_question_1"],
      [
        ["trigger:::num_visited", ">=", 10],
        "OR"
        ["trigger:::num_visited", "<=", "3"],
      ],
      ["trigger:::visible", "==", true],
    ],
    "actions": [
      {
        "action": "geomakeit::alert_dialog_show",
        "arguments": {
          "dialog_uid": "alert_demo"
        }
      }
    ]
},
```

(Examples of using actions 4.4.f)

We have already read and understood what actions are from the 'Actions' section of this chapter. In this section we will see some examples of actions included inside the GeoMakeIt! Default Plugin.

```
class UserLogoutAction: Action() {

    override suspend fun execute(): ExecutionResult {
        CurrentUser.signOut()
        App.getCurrentActivity().finishAffinity()
        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        override val kcls = UserLogoutAction::class
        override val label: String = "user_logout"

        override fun getMetadata(): Metadata = Metadata(
            pageTitle = "User: Logout",
            pageDescription = "Logout the current user."
        )
    }
}
```

Action Example: Logging out the user

```
class ZoneDrawAction(
    var zoneUid: String,
): Action() {

    override suspend fun execute(): ExecutionResult {
        val zone = Zone.find(zoneUid).await()
        if(zone == null) {
            GeoMakeIt.logger.e("Zone '$zoneUid' doesn't exist.")
            return ExecutionResult(null)
        }

        App.getCurrentActivity().runOnUiThread {
            zone.draw(App.map)
        }

        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        private const val ZONE_UID = "zone_uid"

        override val kcls = ZoneDrawAction::class
```

```
        override val label: String = "zone_draw"

        override fun getMetadata() = Metadata(
            pageTitle = "Zone: Draw",
            pageDescription = "Draws a zone on the map."
        )

        override fun getTooltips(): Map<String, String> = mapOf(
            ZONE_UID to "The unique identifier of the saved zone
                        model."
        )
    }
}
```

Action Example: Drawing a zone model which is saved on the Firebase NoSQL database.

```
class AlertDialogShowCustomAction(
    var title: String,
    var message: String,
    var positiveButton: AlertButton? = null,
    var neutralButton: AlertButton? = null,
    var negativeButton: AlertButton? = null
): Action() {

    override suspend fun execute(): ExecutionResult {
        val alert = AlertDialog(
            Util.autoId(),
            title,
            message,
            false,
            positiveButton,
            neutralButton,
            negativeButton
        )
        App.getCurrentActivity().runOnUiThread {
            if(!App.getCurrentActivity().isFinishing) {
                alert.show()
            }
        }
        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        private const val TITLE = "title"
        private const val MESSAGE = "message"
        private const val POSITIVE_BUTTON = "positive_button"
        private const val NEUTRAL_BUTTON = "neutral_button"
        private const val NEGATIVE_BUTTON = "negative_button"


        override val kcls = AlertDialogShowCustomAction::class
        override val label: String = "alert_dialog_show_custom"

        override fun getMetadata() = Metadata(
```

```
            pageTitle = "Alert Dialog: Show custom",
            pageDescription = "Display a **custom** alert dialog,
                               which is not previously saved as a
                               model, to the current player."
        )

        override fun getTooltips(): Map<String, String> = mapOf(
            TITLE to "The title of the alert dialog to be
                      displayed.",
            MESSAGE to "The message contained inside the alert
                        dialog.",
            POSITIVE_BUTTON to "Button located on the **right**
                                side of the modal.",
            NEUTRAL_BUTTON to "Button located on the **middle**
                               side of the modal.",
            NEGATIVE_BUTTON to "Button located on the **left**
                                side of the modal.",
        )
    }
}
```

Action Example: Drawing a custom alert dialog where every button text and action is customizable

(Other Information 4.4.g)

Initial Loading of recipes

In this section we are going to talk about how all recipes are loaded using the GeoMakeIt! PDK when a game is running.

After GeoMakeIt! PDK has loaded all the installed plugins (as discussed on **Chapter 3: GeoMakeIt! PDK**), all of the plugin's information are available using the Meta Injection System (which will be discussed on **Chapter 6**). The GeoMakeIt! PDK has access to all the available triggers and actions paths of each plugin, so it proceeds to resolve them and associate them with the actual executing class.

With all the triggers, actions, and filters available, the GeoMakeIt! PDK can continue with loading the `recipes.json` file and resolve it to create a list of `Recipe` classes.

The actual algorithm used is shown on **Figure 4.11**, below:

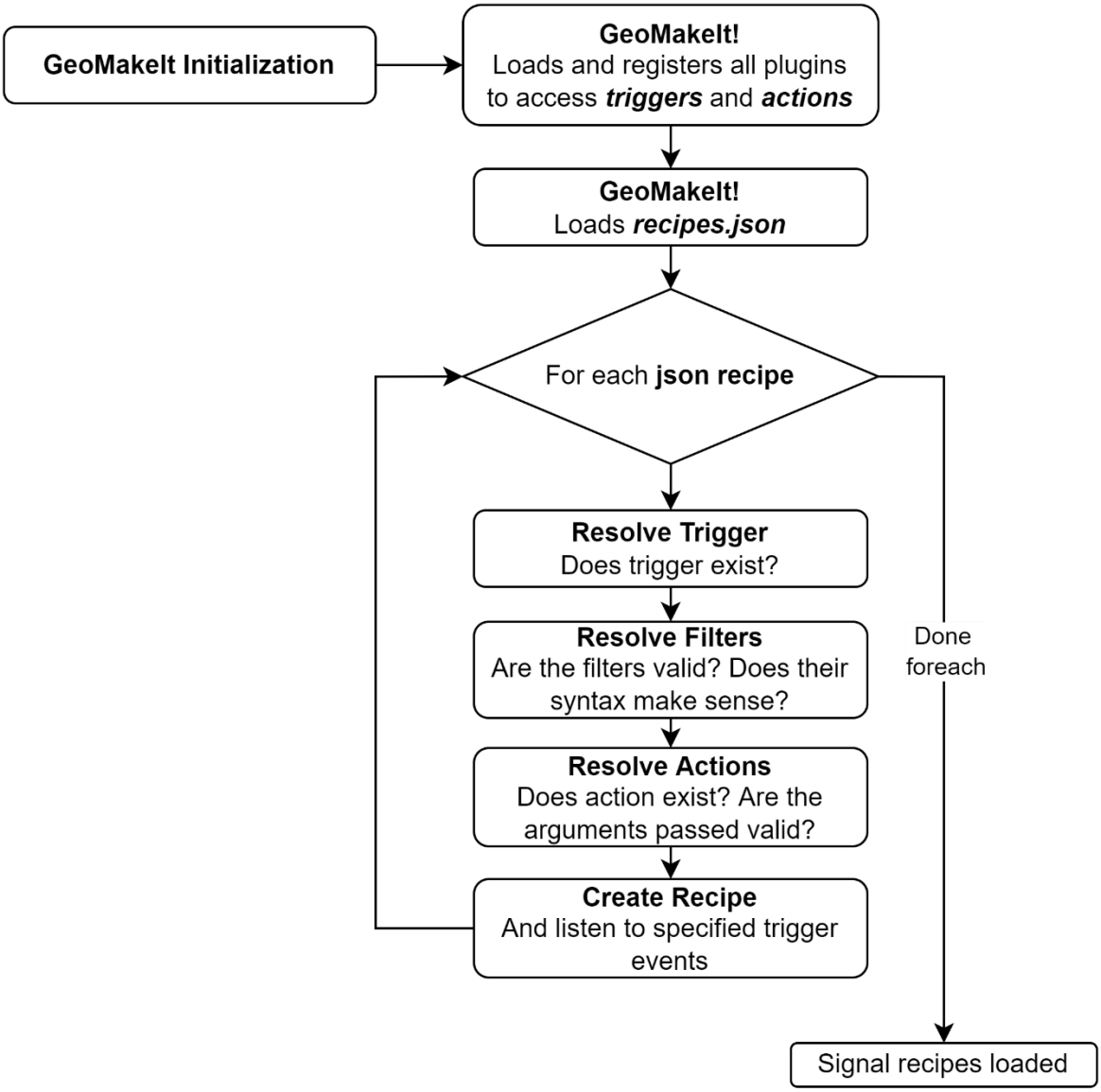

Figure 4.11 Algorithm for resolving recipes.

CHAPTER 5: GeoMakeIt! DESIGNERS

(Introduction of GeoMakeIt! Designers 5.1)

(Overview 5.1.a)

Designers are a GeoMakeIt! feature that allows plugin developers to design customized user interfaces for GeoMakeIt! Studio. These designers are the tools needed for plugin developers allowing them to personalize, modify and control the flow of information between game authors and their plugin.

Designers are generated automatically and by default, whenever a plugin developer creates a GeoMakeIt! Plugin, even if they haven't declared any additional information, thus they may appear to work like "black-magic". In reality, the concept is simple to understand, and throughout this and the next sections we will explain exactly how their inner components work, where plugin developers can implement designers, what types of designers exist, what are `Ingredients`, `Components`, and `Validations` and more!

(The concept 5.1.b)

Imagine we are creating a desktop-application. Our application must receive inputs from our clients through a website. Every time we want to receive that client input, we must:

1. **Create a form on the website**, asking for the clients to give us that input.
2. **Validate** that **the input** given is correct on the website-side.
3. **Pass the input from the website to our application**.
4. **Deserialize / Convert the input** to our final class/object/data.

On a single entity-controlled application, this could be done easy. Whenever we need some new inputs from the user we would follow these steps and create a new form asking for them.

If there are multiple entities though, for example one for controlling the website (Company X), and one for the for the application (us), we can't and wouldn't have direct access to the other entities resources, thus not being able to directly create such forms. We can, however, give them a schematic detailing how to create the form, and they would then assemble it for us.

The exact same scenario takes place here. Designers automatically generate schematics detailing the expected input to be passed from the game author, validating that the input is correct, passing it to the developed plugin and deserializing it, leaving us only with the need to handle what that input means.

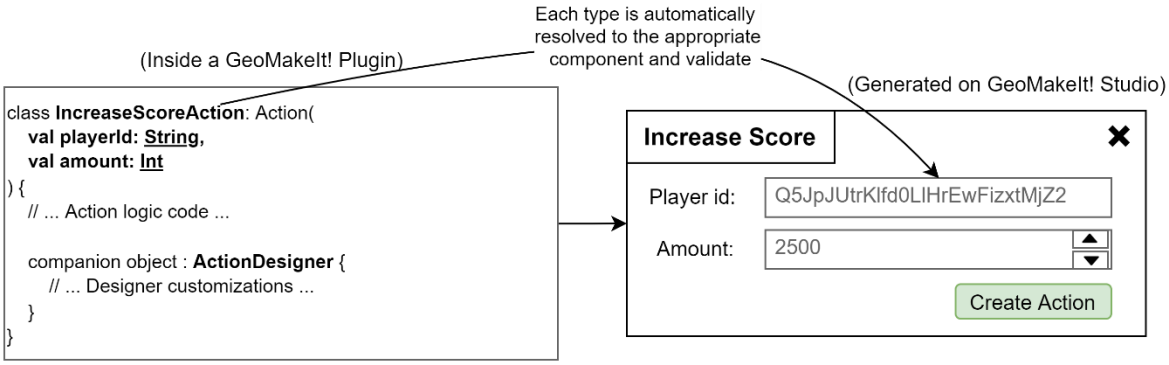

Figure 5.1 Generation of a designer from a plugin for GeoMakeIt! Studio

(Designer types 5.1.c)

There are multiple types of designers, but all with the same goal of generating that schematic. The only difference is that each one performs slightly different operations depending on the object / class it is attached to.

Designers can be sorted out in to two main categories. `Input Designers` and `Output Designers`.

Input Designers

Designers belonging to this category describe the requested input from the game author.

An example of input in GeoMakeIt! would be an Action. When the game author wants to execute a plugin's functionality, they would call an action. The game author must pass some parameters to that action. These parameters would appear similar to submitting an online form, with inputs of style `text box`, `number box`, `text area`, `check box` etc.

Output Designers

Designers belonging to this category describe the potential output that will be generated.

An example of output in GeoMakeIt! would be a Trigger. When a trigger occurs, it generates a set of results, which can be considered the output of a trigger. In order to let the game author, know what variables are outputted, and in turn what variables they can use when this trigger occurs, plugin developers would use a `Trigger Designer`.

Available Designers and when/how to use them

In this section we are presenting a list of the available designers. We will go in detail on how to use each type separately on another section.

- `Output Designers`
    - `Trigger Designer` (As an implementation on a `Trigger` object)
- `Input Designers`
    - `Model Designer` (As a companion object's implementation on a `Model` class)
    - `Config Designer` (As an implementation on a `Config` object)
    - `Action Designer` (As a companion object's implementation on an `Action` class)

(Terminology 5.1.d)

We are going to see the following terminology quite a lot in the following sections as well as in the code we will be presenting. Here, we will provide a short explanation of what everything is.

Ingredients

Ingredients can be considered as any input or output variable the plugin developer creates. In the following code for example, Ingredients would be `playerId` and amount, specifically a `String Ingredient` and an `Integer Ingredient`.

```
class IncreaseScoreAction: Action(
    val playerId: String,
    val amount: Int
) {
    // ... Action logic code ...
    companion object : ActionDesigner {
        // ... Designer customizations ...
    }
}
```

Metadata

Additional data that provide extra information about existing data, for example describing the `title` of a designer, its `description`, `difficulty`, `group` etc. A simple example using metadata to describe a designer is provided below:

```
object MarkerDrawnTrigger: Trigger("on_marker_drawn"),
                           TriggerDesigner {
```

```
        override fun getMetadata() = TriggerMetadata(
            pageTitle = "Marker: Drawn",
            pageDescription = "This trigger is executed when a marker
                               is drawn."
        )
        // Some code
    }
```

Components

Components describe how an `Ingredient` is rendered on GeoMakeIt! Studio by converting it to an `HTML Element`. Each component contains certain specific configurations that are commonly used by these HTML Elements. Some common components would be the `check box`, `text box`, `text area`, `number field` etc.

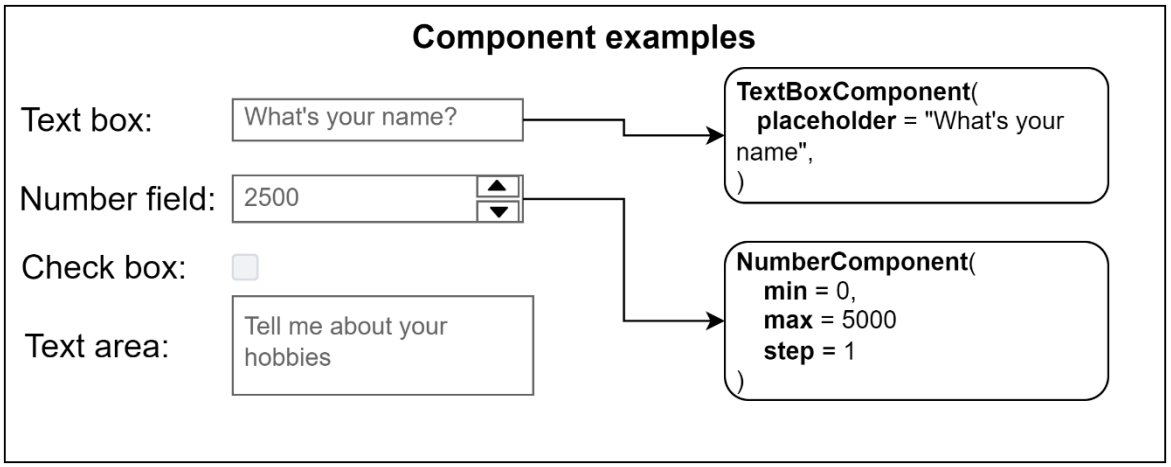

Figure 5.2 Component examples and their options

Tooltips

These are explanatory help / tips that are rendered, assisting the game author inside GeoMakeIt! Studio on how to fill a component.

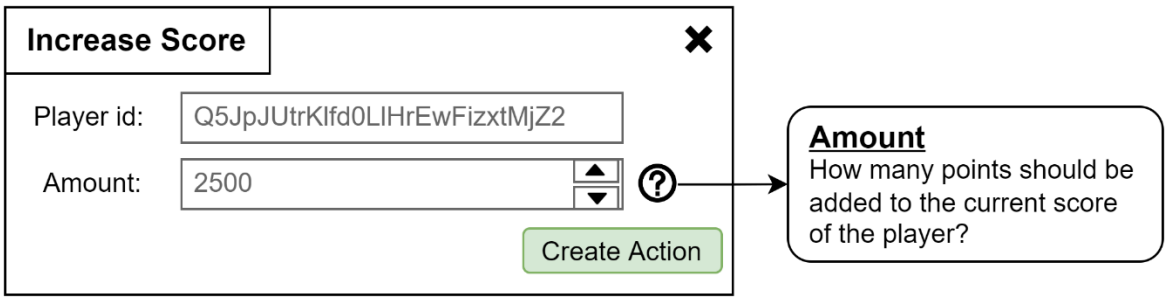

Figure 5.3 Tooltip example

Validations

These are the **restriction added to each component** when the plugin developer wants to control exactly what kind of input must be entered. For example, they could restrict that a number is within a range, that the input is required or optional etc.

(Designer Creation 5.2)

(Overview 5.2.a)

There are multiple types of designers, but all share the same goal of generating a schematic for the interactions between the game author and a plugin inside of GeoMakeIt! Studio. The only difference is that each one performs slightly different operations depending on the object / class it is attached to.

As seen on the introduction, there are 2 generic types of designers, `Input` and `Output designers`. In this section we will not focus on the grouping, but rather on the creation of the designers, thus making a `Trigger`, `Action`, `Model` or `Config Designer`.

(Output Designer Methods 5.2.b)

While we will not dive into the specifics of how an output designer produces the end schematic, we must understand how to modify their result to create a design specific to our needs. Thus, we must understand the methods offered to us by any `Output Designer` and how we can leverage them to generate our final user interface result.

```
interface OutputDesigner : Designable {
    fun getMetadata(): Metadata
}
```

The `getMetadata()` method

Returns basic metadata regarding the designer, such as the `Page Title`, and `Page Description`. Example:

```
override fun getMetadata() = Metadata(
    pageTitle = "Quest: Assigned",
    pageDescription = "This trigger is executed when a quest
                       is assigned to a player."
)
```

If left empty, default resolving of metadata takes place, where:

- `Page Title` is set as the inheriting class's `label` but in title case format. In the following code for example of a trigger, the page's title would be displayed as "*On quest assigned*".

```
object QuestAssignedTrigger: Trigger("on_quest_assigned"),
                             TriggerDesigner { ... }
```

- `Page Description` is not displayed.

(Input Designers 5.2.c)

Similarly, while we will not dive into the specifics of how an input designer produces the end schematic, we must understand how to modify that result to create a design specific to our needs. Thus, we must understand the methods offered to us by any `Input Designer` and how we can leverage them to generate our final user interface.

```
interface InputDesigner : Designable {
    fun getMetadata(): Metadata
    fun getDesignerOrder(): List<String>
    fun getCasts(): Map<String, KClass<Ingredient>>
    fun getComponents(): Map<String, Component>
    fun getTooltips(): Map<String, String> = mapOf()
    fun getValidations(): Map<String, List<Validation>> = mapOf()
    fun getIgnoreList(): List<String>
}
```

The `getMetadata()` method

Functionality stays exactly the same as the one previously described in "***The `getMetadata()` method***" of `Output Designers`.

The `getDesignerOrder()` method

Returns the order of which the designer ingredients are shown.

- **If left empty**, default resolving of the designer order takes place. It is recommended that plugin developers define the exact order of the ingredients, because this might get affected by different JVM versions or even differences between Kotlin and Java.

- **If overridden**, the order defined inside the function is used.

Common usage example

```
class DemoAction(
    var message: String,
    var amount: Int,
    var title: String,
) : Action() {
    // ...
    companion object : ActionDesigner {
        // ...
        override fun getDesignerOrder(): List<String> = listOf(
            "title",
            "message",
            "amount"
        )
    }
}
```

Example including `subclasses`

```
class DemoAction(
    var message: String,
    var title: String,
    var amount: Int
    var subclass: ASubClass
) : Action() {
    // ...
    companion object : ActionDesigner {
        // ...
        override fun getDesignerOrder(): List<String> = listOf(
            "title",
            "message",
            "subclass",
            "subclass.subvariable1",
            "subclass.subvariable2",
            "amount"
        )
    }
}

class ASubClass() {
    val subvariable1: String
    val subvariable2: String
}
```

The `getCasts()` method

Returns per property override regarding the property's ingredient type. If left empty, the `default Ingredient` per variable type is assigned.

The `getComponents()` method

Returns per property override regarding the property's `component`. If left empty, the `default Component per Ingredient` is assigned.

Plugin developers are most likely to use this method to override the UI component that will be rendered. Additionally, they can pass some pre-defined parameters per component to further customize it, such as adding a placeholder, the color of a component etc.

In the following example we change the default component of the variable `message`, which would be a `TextBoxComponent()` since the variable is of type `String`. Additionally, we pass a `placeholder` that will be visible on the UI component.

```
class DemoAction(
    var title: String,
    var message: String,
    var amount: Int,
) : Action() {
    // ...
    companion object : ActionDesigner {
        // ...
        override fun getComponents(): Map<String, Component> =
            mapOf(
                "message" to TextAreaComponent(
                    placeholder="Please write here your message
                                 to display to the player."
                )
            )
    }
}
```

A list of all the available components, and additionally, a list of compatible components per ingredient is also provided, since not all ingredients share the same components! This list can be found a few sections below.

The `getTooltips()` method

Returns per property override regarding the property's explanatory tooltip. If left empty, no tooltip will be displayed.

Plugin developers are often going to use tooltips in order help the game authors fill the correct / expected values for each variable.

```
class DemoAlertAction(
    var title: String,
```

```
        var message: String,
    ) : Action() {
        // ...
        companion object : ActionDesigner {
            // ...
            override fun getTooltips(): Map<String, String> = mapOf(
                "title" to "A title to be displayed on the alert,
                            highlighted with bold and larger font
                            size.",
                "message" to "The message contained inside the
                              alert.",
            )
        }
    }
```

The `getValidations()` method

Returns per property override regarding the property's validation rules. If left empty, the default validations per ingredient and component will be applied.

Each property depending on what `Ingredient` and what `Component` it uses; it applies some default validations restricting the game author to set an appropriate input. You can apply further validations such as `min`, `max`, `between`, `nullable`, `numeric`, `required`, `required if` etc.

Therefor if for example we needed to restrict the user to set a message with length *between 10 and 100 characters*, we could do the following:

```
    class DemoAlertAction(
        var title: String,
        var message: String,
    ) : Action() {
        // ...
        companion object : ActionDesigner {
            // ...
            override fun getValidations(): Map<String,
List<Validation>> = mapOf(
                "message" to BetweenValidation(min=10, max=100),
            )
        }
    }
```

(Trigger Designers 5.2.d)

Below, we can see an actual use-case for a trigger designer implemented in GeoMakeIt! PDK. The following trigger refers to the `AlertDialogShownTrigger`, which is triggered when an alert dialog is shown.

```
object AlertDialogShownTrigger: Trigger("on_alert_dialog_shown"),
                                TriggerDesigner
{
    override fun getMetadata() = TriggerMetadata(
        pageTitle = "Alert dialog shown",
        pageDescription = "This trigger is executed when an alert
                           dialog is shown to the player."
    )

    override fun getIngredients(): Map<String, String> = mapOf(
        "unique_id" to "string",
        "title" to "string",
        "message" to "string",
    )
}
```

Supported ingredients for `getIngredients()` method

Currently, the following ingredients can be used when describing a trigger ingredient:

- Native types: `string`, `char`, `int`, `float`, `double`, `bool`.
- Complex types: `model`
- Collections: `collection`
- Maps: `map`

(Action Designers 5.2.e)

Below, we can see an actual use-case for an action designer implemented in GeoMakeIt! PDK. The following action refers to the `QuestAssignAction`, which is an action that allows the game author to assign quests to players.

```
class QuestAssignAction(var questId: String) : Action() {
    override suspend fun execute(): ExecutionResult {
        val assigned = PlayerQuest.assign(questId).await()
        if(!assigned) {
            GeoMakeIt.logger.e("Quest '$questId' failed to be
                                assigned to player
                                '${CurrentUser.displayName}'.")

             return ExecutionResult(null)
        }

        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        private const val QUEST_UID = "quest_uid"
```

```
        override val kcls = QuestAssignAction::class
        override val label: String = "quest_assign"

        override fun getMetadata() = Metadata(
            pageTitle = "Quest: Assign",
            pageDescription = "Assigns a quest to the player."
        )

        override fun getTooltips(): Map<String, String> = mapOf(
            QUEST_UID to "The unique identifier of the saved
                          quest model."
        )
    }
```

(Config Designers 5.2.f)

Below, we can see an actual use-case for a config designer implemented in GeoQuiz, a GeoMakeIt! authored plugin. The following config refers to the primary configuration of GeoQuiz, named `DefaultsConfigFile`, for the file `config.json`.

```
object DefaultsConfigFile: Configuration(GeoQuiz, "config"),
                           ConfigDesigner {
    lateinit var questions: QuestionsSection
    lateinit var quizzes: QuizzesSection
    lateinit var timer: TimerSection
    lateinit var multipleChoiceView: MultipleChoiceViewSection
    lateinit var fillView: FillViewSection
    lateinit var hinting: HintingSection
}
```

(Ingredients 5.3)

(Overview 5.3.a)

Ingredients can be considered as any `input` or `output` variable that plugin developers create for a class, but more specifically, an `Ingredient` is used to describe how a variable should be presented in GeoMakeIt! Studio. Each ingredient accepts a predefined native/primitive or complex type and using an allowed component it can be generated to an `HTML Element` used in GeoMakeIt! Studio. Additionally, it can provide explanatory tooltips or validations to ensure that a given ingredient will later be safely injected into the receiving variable.

An in-depth example is shown below, where we can see a total of 11 variables, also meaning a total of 11 ingredients that will be automatically generated for the class `AlertDialogShowCustomAction`. Specifically, the following variables and ingredients:

- `title`, which is a `String Ingredient`.
- `message`, which is a `String Ingredient`.
- `positiveButton`, which is a `Class Ingredient`. Additionally positive button contains the following ingredients which are passed from the `AlertButton` class:
    - `text`, which is a `String Ingredient`.
    - `action` which is an `Enum Ingredient`.
- `neutralButton`, which is a `Class Ingredient`, containing again text and action.
- `negativeButton`, which is a `Class Ingredient`, containing again text and action.

```
enum class EAction() { CANCEL, DISMISS, SHUTDOWN, NOTHING; }
data class AlertButton(val text: String, val action: EAction)

class AlertDialogShowCustomAction(
    var title: String,
    var message: String,
    var positiveButton: AlertButton? = null,
    var neutralButton: AlertButton? = null,
    var negativeButton: AlertButton? = null
) : Action() {
  // ..
}
```

(The Ingredient class 5.3.b)

You can see exactly how an `Ingredient` is represented bellow, including a short explanation of the variables and functions included in the ingredient.

```
abstract class Ingredient : Mappable<String, Any?>, Jsonable {
    abstract val type: String
    abstract val value: Any?
    var component: Component
    abstract var tooltip: String?
    var validations: List<Validation>

    abstract fun getAllowedComponents():
        Array<Class<out Component>>
}
```

The `type` variable

Represents the native/primitive or complex type of the ingredient, in string format. For example: a `String Ingredient` would be of type `string`, while an `Integer Ingredient` would be of type `integer`.

The `value` variable

Represents the HTML component that will be used to render the ingredient. Each Ingredient type should use an appropriate component which is listed inside the `getAllowedComponents()` function. A list of components is further below, alongside with the currently supported components per ingredient.

The `tooltip` variable

Represents a helpful tip or explanation regarding the actual ingredient to allow the game author to provide the correct input.

The `validations` variable

Represents the criteria that must be met for an input of the ingredient to be valid. A list of validations is provided further below.

The `getAllowedComponents()` method

Represents an array of components that can be used to render the final HTML representation per ingredient. The list of components that can be used per ingredient is provided in the next section.

(Available Ingredients 5.3.c)

GeoMakeIt! supports a wide range of native and complex types, allowing plugin developers to make amazing creations. The following `Ingredients` are currently supported:

- Native Types:
    - `Boolean Ingredient`
    - `Character Ingredient`
    - `Double Ingredient`
    - `Float Ingredient`

    - Integer Ingredient
    - String Ingredient
- Collections and Maps:
    - Collection Ingredient
    - Map Ingredient
- Complex Types:
    - Class Ingredient
    - Enum Ingredient
    - Model Ingredient

Boolean Ingredient

Representation of a `Boolean` type of variable inside of GeoMakeIt! Studio. The default values alongside with the allowed components are provided below:

- **Default Component**: `Checkbox`
- **Default Value**: `False` (`Unchecked`)
- **Default Validations**: N/A (GeoMakeIt! PDK v2.0.2 doesn't have default validations. Expected in GeoMakeIt! PDK v2.2+)
- **Allowed Components**: `Checkbox`

Character Ingredient

Representation of a `Character` type of variable, inside of GeoMakeIt! Studio. The default values alongside with the allowed components are provided below:

- **Default Component**: `Textbox`
- **Default Value**: `''` (`No character`)
- **Default Validations**: N/A (GeoMakeIt! PDK v2.0.2 doesn't have default validations. Expected in GeoMakeIt! PDK v2.2+)
- **Allowed Components**: `Textbox`

Double Ingredient

Representation of a `Double` type of variable inside of GeoMakeIt! Studio. The default values alongside with the allowed components are provided below:

- **Default Component**: `Number` with `step = 0.1f`.
- **Default Value**: `0.0`

- **Default Validations**: N/A (GeoMakeIt! PDK v2.0.2 doesn't have default validations. Expected in GeoMakeIt! PDK v2.2+)
- **Allowed Components**: `Number`

Float Ingredient

Representation of a `Float` type of variable inside of GeoMakeIt! Studio. The default values alongside with the allowed components are provided below:

- **Default Component**: `Number` with `step = 0.1f`.
- **Default Value**: `0.0`
- **Default Validations**: N/A (GeoMakeIt! PDK v2.0.2 doesn't have default validations. Expected in GeoMakeIt! PDK v2.2+)
- **Allowed Components**: `Number`

Integer Ingredient

Representation of a `Integer` type of variable inside of GeoMakeIt! Studio. The default values alongside with the allowed components are provided below:

- **Default Component**: `Number` with `step = 1f`.
- **Default Value**: `0`
- **Default Validations**: N/A (GeoMakeIt! PDK v2.0.2 doesn't have default validations. Expected in GeoMakeIt! PDK v2.2+)
- **Allowed Components**: `Number`

String Ingredient

Representation of a `String` type of variable inside of GeoMakeIt! Studio. The default values alongside with the allowed components are provided below:

- **Default Component**: `Textbox`
- **Default Value**: `""` (`Empty String`)
- **Default Validations**: N/A (GeoMakeIt! PDK v2.0.2 doesn't have default validations. Expected in GeoMakeIt! PDK v2.2+)
- **Allowed Components**: `Textbox, Textarea`

Collection Ingredient

Representation of a `Collection` type variable inside of GeoMakeIt! Studio. Collections do not have a default component, as their component is inherited from the collection type they represent. The default values alongside with the allowed components are provided below:

- **Default Component**: `NULL`
- **Default Value**: [] (`EMPTY COLLECTION`)

- **Default Validations**: N/A (GeoMakeIt! PDK v2.0.2 doesn't have default validations. Expected in GeoMakeIt! PDK v2.2+)
- **Allowed Components**: `NULL`

Map Ingredient

Representation of a `Map` type variable inside of GeoMakeIt! Studio. Collections do not have a default component, as their component is inherited from the collection type they represent. The default values alongside with the allowed components are provided below:

- **Default Component**: `NULL`
- **Default Value**: {} (`EMPTY MAP/DICTIONARY`)
- **Default Validations**: N/A (GeoMakeIt! PDK v2.0.2 doesn't have default validations. Expected in GeoMakeIt! PDK v2.2+)
- **Allowed Components**: `NULL`

Class Ingredient

Representation of a `Class` type variable inside of GeoMakeIt! Studio. Classes are presented as a group in GeoMakeIt! Studio. Depending on whether the class is required or optional, the actual class must be filled or can be filled. Thus, the `Class Ingredient` is using the `Checkbox` component even though it is represented in GeoMakeIt! Studio as a group. The default values alongside with the allowed components are provided below:

- **Default Component**: `CHECKBOX`
- **Default Value**: `FALSE` (`UNCHECKED`)
- **Default Validations**: N/A (GeoMakeIt! PDK v2.0.2 doesn't have default validations. Expected in GeoMakeIt! PDK v2.2+)
- **Allowed Components**: `CHECKBOX`

Enum Ingredient

Representation of a `Enum` type variable inside of GeoMakeIt! Studio. The default values alongside with the allowed components are provided below:

- **Default Component**: `SELECT`
- **Default Value**: First `enum` from the enumerator list
- **Default Validations**: N/A (GeoMakeIt! PDK v2.0.2 doesn't have default validations. Expected in GeoMakeIt! PDK v2.2+)
- **Allowed Components**: `SELECT, NUMBER`

(Components 5.4)

(Overview 5.4.a)

Components describe how an Ingredient is rendered on GeoMakeIt! Studio by converting it to an HTML Element. Each component contains certain configurations that are commonly used by these HTML Elements.

Technically, these components are almost a '1 to 1' match with their HTML Input Element association, which can be seen in the official mozilla.org documentation.

If a plugin developer wants a different component to be used rather than the default, they must override the `getComponents()` method on their designer. An example can be seen below, were we override the string variable message's component and set it as a `TextAreaComponent()` instead of the default `TextBoxComponent()`. Additionally, we can override the components default configurations by assigning them on the constructor with any order we like.

```
class DemoAction(
    var title: String,
    var message: String,
    var amount: Int,
) : Action() {
    // ...
    companion object : ActionDesigner {
        // ...
        override fun getComponents(): Map<String, Component> =
            mapOf(
                "message" to TextAreaComponent(
                    placeholder="Please write here your message
                                 to display to the player."
                    )
            )
    }
}
```

(Available Components 5.4.b)

The following information is taken as-is from the official mozilla.org documentation. Only the order or grouping is modified to represent the actual GeoMakeIt! PDK Component structure. The generated components on GeoMakeIt! Studio are rendered exactly as the mozilla.org documentation specifies.

Textbox Component

The textbox role is used to identify an element that allows the input of free-form text.

Figure 5.4 Example of a rendered textbox component

The `TextBoxComponent` has the following available configurations:

- `maxLength`: The maximum number of characters (as UTF-16 code units) the user can enter into the text input. This must be an integer value 0 or higher. If no maxlength is specified, or an invalid value is specified, the text input has no maximum length. This value must also be greater than or equal to the value of minlength.

- `minLength`: The minimum number of characters (as UTF-16 code units) the user can enter into the text input. This must be a non-negative integer value smaller than or equal to the value specified by maxlength. If no minlength is specified, or an invalid value is specified, the text input has no minimum length.

- `pattern`: The pattern attribute, when specified, is a regular expression that the input's value must match in order for the value to pass constraint validation.

- `placeholder`: The placeholder attribute is a string that provides a brief hint to the user as to what kind of information is expected in the field. It should be a word or short phrase that demonstrates the expected type of data, rather than an explanatory message. The text must not include carriage returns or line feeds.

- `readOnly`: A Boolean attribute which, if present, means this field cannot be edited by the user. Its value can, however, still be changed by JavaScript code directly setting the `HTMLInputElement` value property.

- `size`: The size attribute is a numeric value indicating how many characters wide the input field should be. The value must be a number greater than zero, and the default value is 20. Since character widths vary, this may or may not be exact and should not be relied upon to be so; the resulting input may be narrower or wider than the specified number of characters, depending on the characters and the font.

This does not set a limit on how many characters the user can enter into the field. It only specifies approximately how many can be seen at a time.

```
data class TextBoxComponent (
    var maxLength: Int = 0,
    var minLength: Int = 0,
```

```
    var pattern: String? = null,
    var placeholder: String? = null,
    var readOnly: Boolean = false,
    var size: Int = 20
): Component()
```

Textarea Component

This component represents a multi-line plain-text editing control, useful when you want to allow users to enter a sizeable amount of free-form text, for example a comment on a review or feedback form.

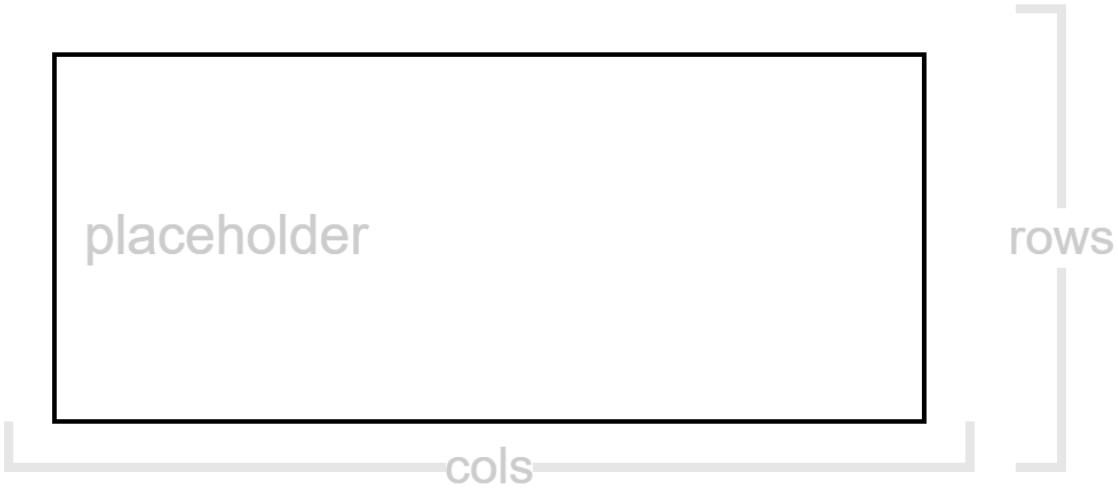

Figure 5.5 Example of a rendered text-area component

The `TextBoxComponent` has the following available configurations:

- `cols`: The visible width of the text control, in average character widths. If it is specified, it must be a positive integer. If it is not specified, the default value is 20.
- `rows`: The number of visible text lines for the control. If it is specified, it must be a positive integer. If it is not specified, the default value is 2.
- `maxLength`: The maximum number of characters (UTF-16 code units) that the user can enter. If this value isn't specified, the user can enter an unlimited number of characters.
- `minLength`: The minimum number of characters (UTF-16 code units) required that the user should enter.
- `placeholder`: A hint to the user of what can be entered in the control. Carriage returns or line-feeds within the placeholder text must be treated as line breaks when rendering the hint.

- `readOnly`: This Boolean attribute indicates that the user cannot modify the value of the control. Unlike the disabled attribute, the read-only attribute does not prevent the user from clicking or selecting in the control. The value of a read-only control is still submitted with the form.

- `required`: This attribute specifies that the user must fill in a value before submitting a form.

```
data class TextAreaComponent (
    var cols: Int = 20,
    var maxLength: Int = 0,
    var minLength: Int = 0,
    var placeholder: String? = null,
    var readOnly: Boolean = false,
    var required: Boolean = false,
): Component()
```

Number Component

Components of this type are used to let the user enter a number. They include built-in validation to reject non-numerical entries.

The browser may opt to provide stepper arrows to let the user increase and decrease the value using their mouse or by tapping with a fingertip.

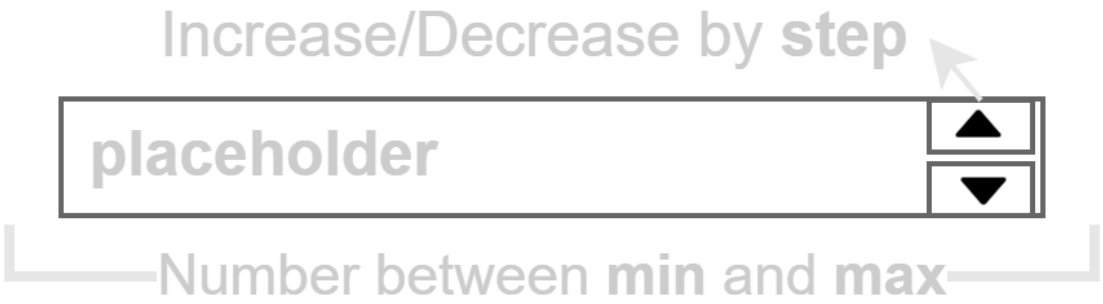

Figure 5.6 Example of a rendered number component

The `NumberComponent` has the following available configurations:

- `max`: The maximum value to accept for this input. If the value entered into the element exceeds this, the element fails constraint validation. If the value of the max attribute isn't a number, then the element has no maximum value.

  This value must be greater than or equal to the value of the `min` attribute.

- `min`: The minimum value to accept for this input. If the value of the element is less than this, the element fails constraint validation. If a value is specified for min that isn't a valid number, the input has no minimum value.

  This value must be less than or equal to the value of the `max` attribute.

- `placeholder`: The placeholder attribute is a string that provides a brief hint to the user as to what kind of information is expected in the field. It should be a word or short phrase that demonstrates the expected type of data, rather than an explanatory message. The text must not include carriage returns or line feeds.

- `readOnly`: A Boolean attribute which, if present, means this field cannot be edited by the user. Its value can, however, still be changed by JavaScript code directly setting the `HTMLInputElement` value property.

- `step`: The step attribute is a number that specifies the granularity that the value must adhere to, or the special value any, which is described below. Only values which are equal to the basis for stepping (`min` if specified, value otherwise, and an appropriate default value if neither of those is provided) are valid.

  A string value of any means that no stepping is implied, and any value is allowed (barring other constraints, such as `min` and `max`).

  The default stepping value for number inputs is 1, allowing only integers to be entered—unless the stepping base is not an integer.

```
data class NumberComponent (
    var max: Int = 0,
    var min: Int = 0,
    var placeholder: String? = null,
    var readOnly: Boolean = false,
    var step: Float = 1f,
): Component()
```

Checkbox Component

This component is rendered by default as a box that is checked (ticked) when activated, like you might see in an official government paper form.

Figure 5.7 Example of a rendered checkbox component

The `CheckboxComponent` has no configurations.

```
data class CheckboxComponent(): Component()
```

Select Component

This component represents a control that provides a menu of options.

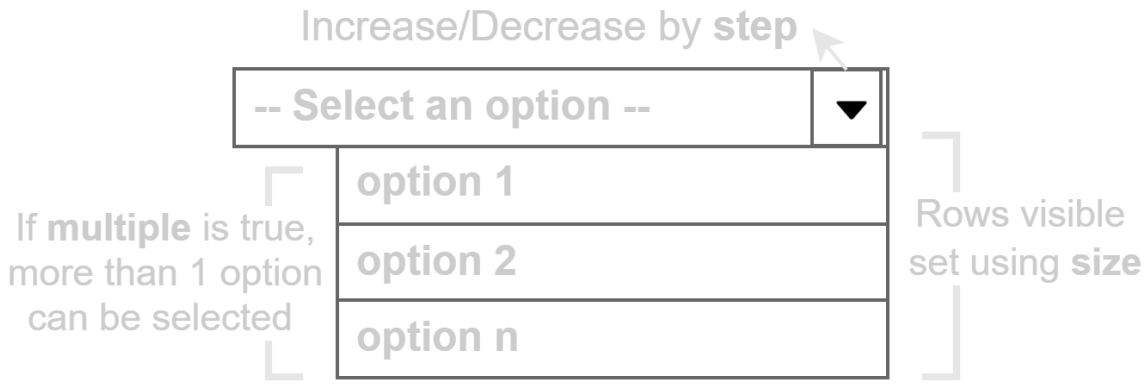

Figure 5.8 Example of a rendered selct component

The `SelectComponent` has the following available configurations:

- `autocomplete`: A string providing a hint for a user agent's autocomplete feature. See The HTML autocomplete attribute for a complete list of values and details on how to use autocomplete.
- `autofocus`: This Boolean attribute lets you specify that a form control should have input focus when the page loads. Only one form element in a document can have the autofocus attribute.
- `disabled`: This Boolean attribute indicates that the user cannot interact with the control. If this attribute is not specified, the control inherits its setting from the containing element, for example `<fieldset>`; if there is no containing element with the disabled attribute set, then the control is enabled.

- `multiple`: This Boolean attribute indicates that multiple options can be selected in the list. If it is not specified, then only one option can be selected at a time.

- `required`: A Boolean attribute indicating that an option with a non-empty string value must be selected.

- `size`: If the control is presented as a scrolling list box (e.g. when multiple is specified), this attribute represents the number of rows in the list that should be visible at one time.

- `options`: A map containing each available option to be selected, where the key is the friendly name, visible on the user while the value is the actual value that is being sent.

```
data class SelectComponent (
    var autocomplete: Boolean = false,
    var autofocus: Boolean = false,
    var disabled: Boolean = false,
    var multiple: Boolean = false,
    var required: Boolean = false,
    var size: Int = 1,
    var options: Map<String, Any> = mapOf()
): Component()
```

NULL Component

Indicates that there is no component to be rendered. This is accepted only in a couple of ingredients where the component itself is inherited by some sub-ingredients.

(Validations 5.5)

(Overview 5.5.a)

Validations are restrictions added on each component when plugin developers need to control what kind of input must be entered. Common uses include for example: keeping an integer input within a range, having an optional input, checking whether an input follows a pattern or not, allowing nullable inputs etc.

If a plugin developer needed their `Ingredient` to behave differently, having different validations, they must override the `getValidations()` method on their designer. An example can be seen below, were we override the string variable message's validation and append an extra `BetweenValidation(min=10, max=300)`. This new validation enforces the game author to add a message between 10 and 300 characters.

```
   class DemoAction(
       var title: String,
       var message: String,
       var amount: Int,
   ) : Action() {
       // ...
       companion object : ActionDesigner {
           // ...
           override fun getValidations(): Map<String,
List<Validation>> {
               return super.getValidations().plus(mapOf(
                   "demo_validations" to listOf(
                       BetweenValidation(min=10, max=100),
                   ),
               ))
           }
       }
   }
```

(Available Validations 5.5.b)

The following validations can be used with most of the ingredients. Since the GeoMakeIt! Studio backend is based on Laravel, plenty of the validations are a one-to-one analogy with their documentation.

Different behavior per Ingredient

Some validations will behave differently, depending on which Ingredient they are assigned.

For instance, the `min` / `max` validations will validate based on the ingredient, where if:

- `String Ingredient`: Checks whether the input has at least min characters or at most max characters.
- `Integer Ingredient`: Check whether the input is bigger or smaller than the specified min or max validations respectively.
- `Collection` or `Map Ingredient`: Check whether at least min or at most max elements exist in the collection or map.

Between Validation

The field under validation must have a size between the given `min` and `max`. Strings, numerics, arrays, and files are evaluated in the same fashion as the size rule.

```
   class BetweenValidation(val min: Int, val max: Int): Validation()
```

Boolean Validation

The field under validation must be able to be cast as a Boolean. Accepted input are `true`, `false`, `1`, `0`, "`1`", and "`0`".

```
class BetweenValidation(val min: Int, val max: Int): Validation()
```

Max Validation

The field under validation must be less than or equal to a maximum value. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

```
class MaxValidation(val max: Int): Validation()
```

Min Validation

The field under validation must have a minimum value. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

```
class MinValidation(val min: Int): Validation()
```

Nullable Validation

The field under validation may be `null`.

Pattern Validation

The field under validation must match the given regular expression.

Internally, this rule uses the PHP `preg_match` function. The pattern specified should obey the same formatting required by `preg_match` and thus also include valid delimiters. For example: `'email' => 'regex:/^.+@.+$/i'`.

```
class PatternValidation(val pattern: String): Validation()
```

Required Validation

The field under validation must be present in the input data and not empty.

A field is considered "empty" if one of the following conditions are true:

- The value is `null`.
- The value is an `empty string`.
- The value is an `empty array` or `empty Countable` object.
- The value is an `uploaded file with no path`.

```
class RequiredValidation(val pattern: String): Validation()
```

Size Validation

The field under validation must have a size matching the given value. For string data, value corresponds to the number of characters. For numeric data, value corresponds to a given integer value. For an array, size corresponds to the count of the array.

```
class SizeValidation(val value: Int): Validation()
```

CHAPTER 6: GeoMakeIt! Meta Injection System

(Introduction of the system 6.1)

(Why such a system is needed 6.1.a)

GeoMakeIt! PDK contains a lot of features that can, and are going to, be used by a plethora of plugins. Most of these features are likely to be structured in a similar way. These features exist to make the development of plugins easier, allowing the plugin developer to focus only on what is needed; Creating his unique, modular, and interoperable plugin, allowing game authors to create dynamic and powerful games.

On the previous chapters we introduced new features such as the `GeoMakeIt! Alchemist` and `GeoMakeIt! Designers`. Moreover, we mentioned features, classes and methods that existed from GeoMakeIt! PDK v1.0.0 that have already proven themselves useful when creating games, such as `GeoMakeIt! Models` and `GeoMakeIt! Placeholders`. Additionally, even some "hidden" classes have been added such as `DBPopulators`, that allow models to be automatically populated and depopulated when changes are detected on the models.

All these features allow plugin developers to bootstrap the development of their plugin by having standard structures and implementations they can depend on. Plugin developers need to be focusing only on the features that want to provide to the game authors and not having to decide on their implementations regarding GeoMakeIt!.

Old implementation example

A very simple and self-explanatory case would be when loading a configuration. All plugins will have some sort of configuration provided by JSON files that are loaded upon the startup of the plugin. On versions prior of the Meta Injection System update, a configuration file would look like this.

```
object ConfigFile {
    val authentication: AuthenticationFileSection

    init {
        val configuration = GeoMakeIt.getConfig("config.json")

        if (!configuration.exists()) {
            // Load Default Configuration File Section
            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, *>
                )

                // Keep loading the next JSON sections ...
            }
        }
    }

    object AuthenticationFileSection {
        // More complex implementation for file section with
        // constructors with different arguments and
        // methods for parsing data
    }
```

Old configuration implementation

The plugin developer had to create their configuration, ex. '`config.json`', populate it and then create an object such as `ConfigFile`, where the process for loading the data from the JSON file would begin. The steps taken are the following:

1. (Optional) Chose between splitting the configuration in file sections or having all the configurations in a single file.

2. Fetch the configuration file using `GeoMakeIt.getConfig("config.json")`. This would return the file object containing the path of the file and its data.

3. Next, check whether this file (or the data) exists by using the `exists()` method.

4. If the file (or data) doesn't exist, load the default values by parsing the default constructor of the file section.

5. If the file (or data) does exist, foreach section: Grab the data (in this case using GSON), convert them to a Map and pass them as the default constructor of the file section.

6. Repeat for every file section.

This would then be repeated for each JSON configuration file included in the plugin.

While this may work on small plugins, on the long run it would present multiple issues and hypothetical questions such as:

- What happens when we want to change implementation of the configuration file?

- How much does the complexity increase by having many more configuration files?

- Is this code readable? Does giving such low-level access to the plugin developer provide him with any actual functionality?

By asking such questions we came into the conclusion that a higher level of abstraction was necessary to allow the plugin developers focus their efforts on other things that matter. This would also allow GeoMakeIt! PDK to work like "magic" hiding the whole loading process behind the scenes.

New implementation example

The same exact code can be now executed but the whole implementation is hidden behind classes leveraging the power of the Meta Injection System, making the code smaller and much more understandable.

```
object ConfigFile: Configuration(GeoMakeIt, "config") {
    lateinit var authentication: AuthenticationFileSection
}

...

class AuthenticationFileSection {
    var enabled by Delegates.notNull<Boolean>()
    lateinit var providers: Set<EAuthProviders>

    fun setProviders(providers: List<String>):
        Set<EAuthProviders>
    {
        return providers
            .mapNotNull {
                try {
                    EAuthProviders.valueOf(it)
                } catch(ex: IllegalArgumentException) {
                    null
                }
            }.toSet()
    }
}
```

New configuration implementation

By making the `ConfigFile` a child of the `Configuration` class we are immediately hiding the whole implementation and loading of the file. The `Configuration` class is leveraging Java's / Kotlin's `reflection` methods to automatically detect and perform the necessary actions for finding the files, resolving the JSON data, and populating the final object file!

Additionally, since JSON only understands primitive data types, we have added support for the plugin developer to create overriding function which alter the final casted data types! Such an example can be seen with the `setProviders()` method.

(Discussing the issue 6.1.b)

By using the reflections package in Java / Kotlin, we can examine and manipulate properties of the application live. For example, we could get or set all the properties of a specific class, or a specific class instance; we can find which classes extends a specific parent class; or we can obtain information about methods and functions included in classes or objects, such as their name, return types, parameters etc.

A small demo example would be the following, were we can filter from a list of classes which ones extend the `ConfigDesigner` class.

```
...
val configDesigners = reflections.getSubTypesOf(
                          ConfigDesigner::class.java
                      )
configDesigners.forEach {
    val designer = Class.forName(it.name).
                         kotlin.objectInstance
                         as? ConfigDesigner

    if(designer != null) {
        File(configsDir, "${designer.label}.json").
            printWriter().use { out ->
                out.print(designer.toJson())
            }
        }
    }
...
```

Let's discuss how this code works:

1. `Reflections.getSubTypesOf()` returns all the class paths that are subtypes of the class provided as first parameter.

2. For each returned:

    a. Convert the path to the actual class using `Class.forName()`

    b. Perform the necessary actions for the algorithm…

While this code would run without any issues in any Java or Kotlin application, running it on the Android environment causes a fundamental roadblock regarding the `reflections` package.

Due to the nature of the `reflection's` package code, it cannot run on any DEX format, as the metadata scanner included in javassist doesn't support the Dalvik system. Thus, it made the integration of the reflection's package an issue.

Without the reflection's package, GeoMakeIt! PDK cannot differentiate and automatically find the subtypes a plugin's classes, meaning it is not able to categorize these classes to models, configs, designers etc.

(The solution 6.1.c)

While we haven't yet decided on a permanent solution, we did find a workaround in order to continue developing the GeoMakeIt! PDK and create levels of abstraction for the plugin developers.

This workaround will be further investigated upon in order to minimize the execution time and to make it more effortless for the plugin developer to integrate it on their plugin.

The solution we propose is generating the metadata required for the reflections package prior to compiling and uploading the actual plugin. Extra steps have been added on the compilation of each plugin where the reflections package can run normally and export the required data for the Java's / Kotlin's reflection to work.

The exact implementation used will be discussed on the next section, '**Implementation 6.2**'.

(Implementation 6.2)

(Meta Injection System design 6.2.a)

To make the Meta Injection System work there are two main classes created, one for exporting meta-data and one for importing them. The first one is called `MetaGenerator` while the second one `MetaInjector`.

Furthermore, there are other classes that take the generated / injected data and use the accordingly. We are going to see such classes in another section.

(The meta generation class 6.2.b)

Initialization of the class

The `MetaGenerator` class exists to create and export all the necessary metadata a plugin may carry. In this section we are going to dissect the whole class and see how it works step by step.

There are 5 global variables that declare objects and paths. The `reflections` variable must take as a parameter the path of the plugin's package, for example `Reflections("com.geomakeit.geoquiz")`. This initiates the reflection's package and allows the GeoMakeIt! PDK to do some investigative operations on any plugin. The

other 4 variables, `rootDir`, `pluginDir`, `assetsDir` and `designersDir` are there to define the directory paths for the next operations.

```
lateinit var reflections: Reflections
lateinit var rootDir: String
lateinit var pluginDir: String
lateinit var assetsDir: String
lateinit var designersDir: String

fun main(args: Array<String>) {
    reflections = Reflections(args[0])

    rootDir = System.getProperty("user.dir") as String
    pluginDir = "$rootDir\\src\\main"
    assetsDir = "$pluginDir\\assets\\${args[1]}"
    designersDir = "$assetsDir\\designers"

    generateDesigners()
    generateMeta()
}
```

Once the directories have been set, we can proceed to the next two steps, generating the designers and generating the meta for the plugin. Essentially, designer variables and methods that are associated with the designer classes are used to generate the final JSON file. This process is done in multiple steps.

The `generateDesigners()` function

We can proceed by generating the designers. Designers are procedurally generated based on the information available by each class a designer is assigned to.

Initializing Designer Generation

On the current version of GeoMakeIt! PDK, there are 4 available designers, the `Trigger Designer`, `Action Designer`, `Model Design` and `Config Designer`. For each designer a similar method of generation is used, with a few differences regarding on if a designer is an object instance or a class.

We will be looking on the `Trigger Designer` generation. Using reflection, we collect the available children of the `TriggerDesigner` class and use their implemented `toJson()` function.

This `toJson()` function takes all the variables included in a designer and converts them to their relative primitive types, saving them with the snake case format.

```
fun generateDesigners() {
    generateTriggerDesigners()
    generateActionDesigners()
    generateModelDesigners()
```

```
    generateConfigDesigners()
}

fun generateTriggerDesigners() {
    val triggersDir = "$designersDir\\triggers"
    deleteDirectoryFiles(File(triggersDir))

    val triggerDesigners = reflections.getSubTypesOf(
                               TriggerDesigner::class.java
                           )

    triggerDesigners.forEach {
        val designer = Class.forName(it.name).
                       kotlin.objectInstance as? TriggerDesigner
        if(designer != null && designer is Trigger) {
            File(triggersDir, "${designer.label}.json")
                .printWriter()
                .use { out ->
                    out.print(designer.toJson())
                }
        }
    }
}

fun generateActionDesigners() { ... }
fun generateModelDesigners() { ... }
fun generateConfigDesigners() { ... }
```

The `generateMeta()` function

Finally, we need to generate the list with the `models`, `placeholders`, `populators`, `triggers` and `actions` that need to be imported later on by the Meta Injector. A similar method for each class is used, where we are using the reflection package to collect available children of each class.

Instead of using the `toJson()` function though, as this is only organizing these classes into groups, we only need to map each class to its appropriate group and later do a simple conversion of the map to JSON format.

```
fun generateMeta() {

    val models = reflections
        .getSubTypesOf(Model::class.java)
        .map { it.name }



    val populators = reflections
        .getSubTypesOf(DBPopulator::class.java)
        .associate { it.name to
                     Class.forName(it.name).
```

```
                        kotlin.objectInstance as? DBPopulator
                    }
            .filter { it.value != null &&
                      it.value is ModelDBFactory<*>}
            .map { it.key }

        val triggers = reflections
            .getSubTypesOf(Trigger::class.java)
            .map { it.name }

        val actions = reflections
            .getSubTypesOf(ActionDesigner::class.java)
            .associate {
                (Class.forName(it.name).
                 kotlin.objectInstance as ActionDesigner).label to
                (Class.forName(it.name).kotlin.objectInstance
                 as ActionDesigner).kcls.qualifiedName!!
            }

        val placeholders = reflections
            .getSubTypesOf(Placeholder::class.java)
            .map { it.name }

        val metaMap = mutableMapOf(
            "models" to models,
            "populators" to populators,
            "triggers" to triggers,
            "actions" to actions,
            "placeholders" to placeholders
        )

        File(assetsDir, "meta.json").printWriter().use { out ->
            out.print(Util.getDefaultGson().toJson(metaMap))
        }
    }
```

This would in turn generate an output like the following:

```
{
  "models": [
    "com.geomakeit.pdk.defaults.models.zones.Zone",
    "com.geomakeit.pdk.defaults.models.quests.PlayerQuest",
    "com.geomakeit.pdk.defaults.models.zones.CircleZone",
    "com.geomakeit.pdk.defaults.models.users.User",
    "com.geomakeit.pdk.defaults.models.quests.Quest",
    "com.geomakeit.pdk.defaults.models.alertdialog.AlertDialog",
    "com.geomakeit.pdk.defaults.models.markers.Marker",
    "com.geomakeit.pdk.defaults.models.users.CurrentUser",
    "com.geomakeit.pdk.defaults.models.zones.ZoneVisit",
    "com.geomakeit.pdk.defaults.models.zones.PolygonZone"
  ],
  "populators": [
    "com.geomakeit.pdk.defaults.models.quests.Quest$Companion",
```

```
      "com.geomakeit.pdk.defaults.models.markers.Marker$Companion",
      "com.geomakeit.pdk.defaults.models.zones.Zone$Companion"
    ],
    "triggers": [
        ...
    ],
    "actions": {
      "zone_draw": "....actions.zones.ZoneDrawAction",
      "app_close": "....actions.app.AppCloseAction",
        ...
    },
    "placeholders": [
        ...
    ]
}
```

(The meta injector class 6.2.c)

The `MetaInjector` class exists to import all the necessary metadata a plugin may carry and have them available on runtime. This class is much simpler than the `MetaGenerator` class as it may only requires fetching the already grouped classes and convert their `classpath` to the actual class.

```
class MetaInjector(plugin: IPlugin) {
    var models: List<KClass<out Model>>
        private set

    var DBPopulators: List<DBPopulator>
        private set

    var triggers: List<Trigger>
        private set

    var actions: Map<String, KClass<out Action>>
        private set

    var placeholders: List<Placeholder>
        private set

    init {
        val meta = plugin.assets.getJSON("meta")
        val json = meta.asJSONObject()

        models = resolveModels(
                    json.getJSONArray("models")
                        .toList() as List<String>
                )

        DBPopulators = resolveDBPopulators(
                        json.getJSONArray("populators")
                            .toList() as List<String>)
```

```
            triggers = resolveTriggers(
                            json.getJSONArray("triggers")
                                .toList() as List<String>)

            actions = resolveActions(
                            json.getJSONObject("actions")
                                .toMap() as Map<String, String>)

            placeholders = resolvePlaceholders(
                            json.getJSONArray("placeholders")
                                .toList() as List<String>, plugin)
        }

        private fun resolveModels(list: List<String>) = list.map {
            Class.forName(it).kotlin as KClass<out Model>
        }

        private fun resolveTriggers(list: List<String>) = list.map {
            Class.forName(it).kotlin.objectInstance as Trigger
        }

        private fun resolveDBPopulators(list: List<String>) =
        list.map {
            Class.forName(it).kotlin.objectInstance as DBPopulator
        }

        private fun resolveActions(map: Map<String, String>) =
        map.mapValues {
            Class.forName(it.value).kotlin as KClass<out Action>
        }

        private fun resolvePlaceholders(
            list: List<String>,
            plugin: IPlugin
        ) = list.map {
           Class.forName(it).kotlin.primaryConstructor!!
                            .call(plugin) as Placeholder
        }
    }
```

(Auto Loading 6.2.d)

Thanks to the metadata provided by the Meta Injection System, and together using the reflections package, we can automatically resolve and load data from all sorts of components. We are going to take a sneak peek on how the `Configuration` component uses reflection to associate the data collected by a configurations data file with the configuration object itself.

The following code resides inside the `Configuration.kt` file. While there is a lot of code here, we are going to split the code in a few logical parts as seen below:

1. Initial call of the configuration component and resolving of data.
2. For the resolving of data:
    a. Fetch all the public variables except `label` and `asset`.
    b. Fetch all `public setters` that alter the set function of a variable.
    c. Foreach `variable`:
        i. If it has a custom `setter`, call that and parse as parameters the incoming data.
        ii. Otherwise:
            1. If data are of `primitive` type, assign them directly.
            2. If data are `nested`, continue with `step 2.c` in the nested variable.

```
open class Configuration(val plugin: IPlugin, val label: String)
{
    private val asset: ConfigAsset by lazy {
        plugin.assets.getConfig(label)
    }

    private val self: Configuration
        get() = this

    init {
        // Step 1.
        if(App.isInitialized()) {
            resolve(self, this::class, asset.asString())
        }
    }

    private fun resolve(
        instance: Any, kcls: KClass<*>, dataStr: String
    ) {
        val gson = Util.getDefaultGson()
        val data = gson.fromJson(dataStr, Map::class.java)

        // Step 2.a.
        @Suppress("UNCHECKED_CAST")
        val properties = kcls.memberProperties
            .filter { it.visibility == KVisibility.PUBLIC }
            .filter { it.name !in listOf("label", "asset") }
            .filter { it is KMutableProperty<*> }
            .associate { it.name to it as KMutableProperty<*> }

        // Step 2.b.
        val customPropertySetter = kcls.declaredFunctions
            .filter { it.name.startsWith("set")}
```

```
            .filter {
                val propertyName = it.name
                    .removePrefix("set")
                    .replaceFirstChar { it.lowercase() }

                propertyName in properties.keys
            }.associateBy {
                val propertyName = it.name
                    .removePrefix("set")
                    .replaceFirstChar { it.lowercase() }

                properties[propertyName]
            }

        // Step 2.c.
        properties.forEach {
            val name = it.key
            val property = it.value
            val type = property.returnType

            if (!data.containsKey(name.toSnakeCase())) {
                return@forEach
            }

            // Step 2.c.i.
            if (property in customPropertySetter) {

                val setter = customPropertySetter[property]!!
                if (setter.parameters.size in listOf(0, 1))
                    return@forEach

                if (setter.parameters.size == 2) {
                    val newJson =
                            gson.toJson(data[name.toSnakeCase()])

                    val propertyInput = gson.fromJson(
                        newJson,
                        setter.parameters[1].type.jvmErasure.java
                    )

                    property.setter.call(
                        instance,
                        setter.call(instance, propertyInput)
                    )
                    return@forEach
                }
            }

            // Step 2.c.ii.

            val lst = listOf(
                Boolean::class, Character::class,
                Double::class, Float::class, Integer::class,
                String::class,
                Collection::class, Map::class, List::class
            )
```

```
                if (lst.any {
                    cls -> type.jvmErasure.isSubclassOf(cls)
                }) {
                    val propertyInput =
                       gson.fromJson(
                           gson.toJson(
                               data[name.toSnakeCase()]
                           ),
                           type.jvmErasure.java
                       )

                    property.setter.call(instance, propertyInput)
                    return@forEach
                }

                val subtypeData = gson.toJson(
                                      data[name.toSnakeCase()]
                )

                val newInstance = type.jvmErasure.createInstance()
                resolve(newInstance, type.jvmErasure, subtypeData)
                property.setter.call(instance, newInstance)
            }
        }
    }
```

(Third-party plugin implementation 6.2.e)

While plugin developer can use immediately most of the functions of GeoMakeIt! PDK, in this case we need to alter quite a few details in the `build.gradle` of our project. Thankfully, this is only done once. We are still actively looking for ways to remove / hide these steps to make it easier for plugin developers.

Why this addition?

For the same reasons we discussed on **section 6.1.b**, it is quite difficult to support pure reflections at this moment on GeoMakeIt! PDK due to the Android's DEX and Dalvik system restrictions. In order to avoid this, we perform a "pre-compiling" step on the plugin's developer system, allowing the whole plugin to run on standard JVM prior to uploading the plugin. This step then permits the Meta Injection System to run fully and generate the necessary metadata paths.

To execute this workaround, we use a system called `FAT JAR shadowing`, essentially downloading all necessary dependencies and running the plugin as a JAR instead of an AAR. This allows the `reflections` package to work, thus allowing also the **Meta Injection System** to perform any action necessary.

The following highlighted with bold code must be added to the `build.gradle` of any third party plugin running GeoMakeIt! PDK version 2.0.1+. This code will not be further

explained as it is a temporary workaround, and we are looking for a better implementation for it.

```
apply plugin: 'com.android.library'
apply plugin: 'kotlin-android'
apply plugin: 'com.github.johnrengelman.shadow'

import com.github.jengelman.gradle.plugins.shadow.tasks.ShadowJar

var identifier = '<PLUGIN IDENTIFIER>'
project.group = '<PLUGIN PACKAGE>'
project.version = '<PLUGIN VERSION>'

configurations {
    shadow // to be included in fat aar with changed package name
    shadow.extendsFrom(implementation)
}

android {
    compileSdkVersion 31

    defaultConfig {
        minSdkVersion 19
        targetSdkVersion 31

        testInstrumentationRunner
                        'androidx.test.runner.AndroidJUnitRunner'
        consumerProguardFiles 'consumer-rules.pro'


        javaCompileOptions {
            annotationProcessorOptions {
                arguments = [
                   "room.schemaLocation":
                                "$projectDir/schemas".toString()
                ]
            }
        }
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile(
                                'proguard-android-optimize.txt'
                            ), 'proguard-rules.pro'
        }
    }


    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
```

```
}

dependencies {
    implementation fileTree(
                        dir: 'libs', include: ['*.jar', '*.aar']
    )


    implementation project(":geomakeit")

    ...

    implementation 'org.reflections:reflections:0.9.12'
    implementation
        "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
    shadow 'org.robolectric:android-all:12.1-robolectric-8229987'
    implementation 'androidx.multidex:multidex:2.0.1'

    testImplementation "junit:junit:$junit_version"
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation
        'androidx.test.espresso:espresso-core:3.4.0'

    implementation 'androidx.gridlayout:gridlayout:1.0.0'
}

apply plugin: 'com.google.gms.google-services'

afterEvaluate {
    def aarDir = "$buildDir\\extracted-aars"

    task extractAars {
        group 'other'

        doLast {
            configurations.shadow.filter {
              it.name.endsWith('.aar')
            }.collect { aar ->
                copy {
                  from zipTree(aar)
                  into "$aarDir/${aar.name.replace('.aar', '')}"
                }
            }
        }
    }

    task shadowJar(type: ShadowJar) {
        dependsOn assembleDebug, extractAars
        group 'group'
        zip64 = true

        configurations = [project.configurations.shadow]

        exclude '*.aar'
        exclude '*.xml'
        exclude 'META-INF/maven/**'
```

```
            exclude 'META-INF/services/**'

            from fileTree(aarDir).filter { it.name == 'classes.jar' }
            from zipTree(
                 "$buildDir/outputs/aar/$archivesBaseName-debug.aar")
                 .filter { it.name == 'classes.jar' }
        }

        task runGenerator(type: JavaExec) {
            dependsOn shadowJar
            group = 'geomakeit'
            classpath = files("build/libs/$archivesBaseName-
                                ${project.version}.jar")
            mainClass = 'com.geomakeit.pdk.meta.MetaGeneratorKt'
            args Arrays.asList(project.group, identifier)
        }
    }
```

CHAPTER 7: GeoMakeIt! Default Plugin Updates

(Introduction of the plugins 7.1)

Continuing from where we left of on GeoMakeIt! PDK v1.0.0 (a.k.a. GeoMakeIt! API), the two initial plugins are still actively developed and updated to meet the requirements and leverage the newly added functionalities of GeoMakeIt! PDK v2.0.0+.

These two plugins are the GeoMakeIt! Default Plugin and GeoQuiz.

The first one, GeoMakeIt! Default Plugin, resides inside GeoMakeIt! PDK. It is the first plugin that it is loaded since all other plugins are most likely depending on it. Its identifier is "`geomakeit`", and it provides some of the main functions that a game author will use in their Games, such as starting specific activities, managing the users (signing in, registration, logging out etc.), using the basic placeholders etc.

The other one, GeoQuiz, is a plugin developed to simulate how would a third-party developer use GeoMakeIt! PDK to develop his own modular and customizable plugin. It allows game authors to create multiple quizzes where players can attempt solving them for rewards.

In this chapter we are going to have a quick overview of the first one, the GeoMakeIt! Default Plugin and focus on the changes from the previous installed version. While there have not been a lot of changes regarding their actual functionality, the methods, code, and implementations of the plugins have changed significantly.

(Common Classes 7.2)

(The "Main" class 7.2.a)

This is the first class to be loaded when GeoMakeIt! Default Plugin is initialized. All main classes are singleton objects, thus when enabling a plugin there is only one active instance of it.

```
object GeoMakeIt: 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, such as the plugin's `identifier`.

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

Similarly, the `onEnable()` method must be overridden, which is the method executed upon enabling the plugin. This is the common place to dispatch initial commands, perform registration of placeholders, activities, services etc.

The `onEnable()` method has been additionally enhanced in version 2.0.0+ of GeoMakeIt! PDK, with the addition of the Meta Injection System which we are going to see on a separate chapter. This allows any GeoMakeIt! Plugin, and consequently the GeoMakeIt! Default Plugin, to automatically find and load `Triggers`, `Actions`, `Services`, `Commands` etc.

```
override fun onEnable() {
    CoroutineScope(Dispatchers.IO).launch {
        val cmds = StartupConfig.startupCommands
        cmds.forEach { cmd ->
            CommandManager.dispatch(cmd)
        }
    }

    ...
}
```

Finally, the `onDisable()` method is used to unload the whole plugin if necessary.

(Placeholders 7.2.b)

Placeholders remain unchanged for this version of GeoMakeIt! Default plugin, with only an addition of an alias regarding the display name of the player.

By opening the `DefaultsPlaceholders.kt`, we can take a look of how a placeholder is constructed.

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

```
            return null
        }
    }
```

This class extends continues with the same format as mentioned in GeoMakeIt! PDK v1.0.0, with the only addition of the `onRequest()` method changing from a simple method to an `suspendable` method. This is common on GeoMakeIt! PDK v2.0.0 as many of the available methods can now run asynchronously, so it is easier for the system to fetch data stored either on the database or perform I/O operations.

Additionally using the **Meta Injection System** mentioned on **Chapter 6**, it is no longer necessary to manually register these placeholders, as it is done automatically.

By creating such a placeholder GeoMakeIt! PDK can now parse personalized messages like "*Hello there {{geomakeit::username}}, how are you?*" and reconstruct them as the following: "*Hello there AwesomePlayer123, how are you?*" where `AwesomePlayer123` is the name of the player.

If nothing is matched, the `null` is returned.

(Configurations 7.2.c)

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`.

The contents of these JSON configurations remain the same, but the implementation has changed from Configurations V1 to Configurations V2 which is included in GeoMakeIt! PDK v2.0.0+.

But first, let's have a look on a file we are familiar with, the `plugin.json` file.

```
{
  "main": "com.geomakeit.api.defaults.GeoMakeIt",
  "title": "GeoMakeIt!",
  "short_description": "The default plugin",
  "description": "GeoMakeIt! is the main plugin integrated in all
                  of GeoMakeIt! games. Every game that is created
                  through our platform must require it to be able
                  to create the final game.",
  "version": "2.0.0",
  "author": "Vasilis Dimitriadis",
  "gradle_implementations": []
}
```

Underneath this text we can see the old configuration included under the package: "*com.geomakeit.api.defaults.configurations"*, and following that we can compare it with the new configuration implementation.

```
   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 ...
           }
       }
   }
```

Old configuration implementation

```
   object ConfigFile: Configuration(GeoMakeIt, "config"),
                      ConfigDesigner
   {
       lateinit var authentication: AuthenticationFileSection

       override fun getTooltips(): Map<String, String> =
           mapOf(
               "authentication.enabled" to "Enable user
                        authentication and require player to sign in
                        prior to accessing the game.",
               "authentication.providers" to "List of available
                        authentication methods/providers."
       )
   }

   class AuthenticationFileSection {
       var enabled by Delegates.notNull<Boolean>()
       lateinit var providers: Set<EAuthProviders>
```

```
}
```

New configuration implementation

With this example it is obvious how far the **Meta Injection System** described in **Chapter 6** has simplified the process of creating configuration, with the plugin developer only needing to just set the name and types of the variables to be imported. Moreover, it is no longer required to register the configuration as this step is also done automatically.

Alike the GeoMakeIt! PDK v1.0.0, 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 aforementioned JSON configuration file looks like this, containing information about if the game is going to contain authentication methods, which one of these it will contain etc.

Additionally, the startup commands have been separated in a file of their own, named `statup.json`. This file has its own configuration file class, named `StartupConfig`, which is presented below.

```
object StartupConfig: Configuration(GeoMakeIt, "startup"),
                      ConfigDesigner
{
    lateinit var startupCommands: List<String>
}
```

(Available Models 7.3)

GeoMakeIt! Default plugin provides 5 initial models that a plugin developer can later use or extend upon them, and the game author can use them to design their game.

These are an `Alert Dialog` Model, a `Marker` Model, a `Quest` Model, a `User` Model and a `Zone` Model.

These 5 models persist from GeoMakeIt! PDK v1.0.0 with a few changes in their implementations, as there have been changes regarding how models are fetched from the database and loaded.

Further down the line, some of the models might be separated to plugins of their own, while new ones may also emerge. Until then, let's discuss the ones currently implemented in this version of the GeoMakeIt! Default Plugin.

(Alert Dialog Model 7.3.a)

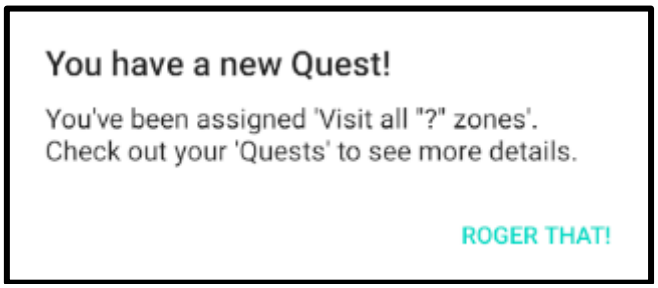

Figure 7.1 Alert Dialog example inside a GeoMakeIt! game

The `Alert Dialog` is an Android based model for a small alert / popup window that allows the player to decide regarding a set of actions 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 is presented to the player, 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 four available constructors for calling this class, two of which are the default constructors provided by the GeoMakeIt! PDK specifications.

There is also the method `show()` which takes as an argument the current context of the application. By default, this is the current activity's context. This method makes the alert dialog visible to the player.

```
fun show(context: Context = App.getCurrentActivity())
```

There are some additional private methods for setting up the alert dialogs.

A method called `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.

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.

Finally, we are going to compare the old style of fetching the model data from the database, using the `ModelFactory` and `IFileLoadable`, versus the new one which is automatically integrated through the GeoMakeIt's **Meta Injection System** presented in **Chapter 6.** By comparing the two we can see how this new system simplifies all of the database operations by hiding them behind the `ModelDBFactory<CLS>()` object.

```
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
       }
    }
}
```

Old method for loading the model from the database using the ModelFactory interface and IFileLoadable interface

```
companion object : ModelDBFactory<AlertDialog>(),
                   DBPopulator,
                   ModelDesigner
{
        override val TABLE = "geomakeit_alert_dialogs"
        override val cls = AlertDialog::class.java

        override val kcls = AlertDialog::class

        override val label = "alert_dialogs"

        override val populatorAsset: Asset by lazy {
            GeoMakeIt.assets.getData(label)
```

```
        }

        const val FIELD_TITLE = "title"
        const val FIELD_MESSAGE = "message"
        const val FIELD_CANCELLABLE = "cancellable"
        const val FIELD_POSITIVE_BUTTON = "positive_button"
        const val FIELD_NEUTRAL_BUTTON = "neutral_button"
        const val FIELD_NEGATIVE_BUTTON = "negative_button"
    }
```

New method for loading the model from the database using the ModelDBFactory class

Alert Button class

The alert button has evolved from its simple data class structure to one containing more structured data. Both actions and button types have been refactored to use enumerators.

Alert Dialog Command class

This class is now deprecated with the addition of GeoMakeIt! Alchemist.

Data 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 author can create an alert dialog through the GeoMakeIt! Studio, using a designer generated with the designers we discussed on **Chapter 5: GeoMakeIt! Designers**. The results outputted from the designers will be similar to the one above.

(Users Model 7.3.b)

The `Users` Model contains information about the player, such as his username and location. It is structured by 3 classes, the `Auth`, `User` and `CurrentUser`.

These classes are refactored to fit the needs of GeoMakeIt! v2.0.0+. Most notably the `ModelDBFactory` is used to create the database abstraction. Additionally, methods for managing and deleting the user have been added.

(Markers Model 7.3.c)

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 set of actions/triggers which we will see separately, allowing the game author to listen and perform actions using markers.

Marker class

The `Marker` class includes the following configurable variables through the `marker.json`.

```
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

The `Marker Command` used to allow a game author to draw or erase specific markers, depending on their needs. This has since been deprecated and a similar action has been developed instead, using GeoMakeIt! Alchemist.

(Zones Model 7.3.d)

Similar to 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 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.

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.

Finally, an important thing to mention regarding the Zone model is the fact that it is inheritable. We will see the two children's classes on the next section in more detail. The reason this is mentioned is to showcase the `Inheritable` interface in composition with `ModelDBFactory`. As seen below, by introducing the `Inheritable` interface we only had to declare which subclasses of this class exist, allowing the data to be parsed to the correct class when fetched from the database.

```
 companion object: ModelDBFactory<Zone>(),
                   Inheritable, DBPopulator {

     override val TABLE = "geomakeit_zones"
     override val cls = Zone::class.java
     override val populatorAsset: Asset by lazy {
         GeoMakeIt.assets.getData("zones")
     }

     override val subclassField: String = "zone_type"
     override val subclasses: Map<String, Class<out Model>> =
         mapOf(
             "circle" to CircleZone::class.java,
             "polygon" to PolygonZone::class.java
         )
     ...
}
```

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

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.

This class has been updated to fit better the new implementations introduced from GeoMakeIt! v2.0.0.

Zone Command class

The zone commands that existed and allowed a game author to verify whether a user has visited a specific zone, or to draw / erase zones from the map, have been deprecated. Instead of using these commands, a game author should now use the available actions and triggers provided with the GeoMakeIt! Default Plugin.

(Quests Model 7.3.e)

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! PDK.

The `Quests` Model contains a few classes and UI Elements. The UI elements are stored in the `res` folder of the GeoMakeIt! PDK and are prefixed with "`geomakeit`".

Let's dive immediately onto the classes.

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 author changes to decide how a quest behaves, more importantly:

EQuestStatus Enumarator

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

This enumerator 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.

The following two commands have been replaced with the use of GeoMakeIt! Alchemist Actions but remain to provide backwards compatibility. Specifically:

The "`assign`" command assigns a specific quest to a player. To execute this command the game author could execute "`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 author wants to execute this command they may call "`geomakeit::quest geomakeit_activity_quests`".

The same functionality could be also accomplished using "`geomakeit::activity geomakeit::quests`", which is also recommended.

Quest Placeholder class

These placeholders remain with the same exact functionallity. Their implementation has slightly changed to accommodate the asynchronous calls from the database.

The placeholders registered regarding the quests are the following:

- `{{ 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.

(Alchemist Updates 7.4)

GeoMakeIt! Default plugin provides 5 initial models that a plugin developer can

(Precursor commands 7.4.a)

With the introduction of GeoMakeIt! Alchemist, commands are no longer necessary, but we do keep them for backwards compatibility. The available commands on version 1.0.0 thus remain unchanged.

These commands are:

- `Sign Out` Command
- `Activity` Command
- `Alert Dialog` Command
- `Quest` Command
- `Zone` Command

We are not going to discuss their specific implementation as it is not of essence and they remain unchanged, but we are going to see a quick demonstration of a `Command Map` and a `Command` to see how GeoMakeIt! PDK used to function.

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

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

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

    private fun setDefaultCommands() {
        registerCommand(SignOutCommand())
        registerCommand(ActivityCommand())
                    ...
    }
}
```

(GeoMakeIt! Alchemist replacing commands 7.4.b)

As previously mentioned, with the introduction of GeoMakeIt! Alchemist, commands are no longer necessary. These commands are replaced by Alchemist components, more specifically `Actions`.

Actions; analogous to commands; are small work-steps that are executed to perform changes in the actual game-flow.

On the GeoMakeIt! Default Plugin we have grouped these alchemists components under the package "`com.geomakeit.pdk.default.alchemists`". This package contains all the available `Actions` and `Triggers` created by this default plugin which are made available for the game author to use.

(Actions 7.4.c)

The following actions are available in GeoMakeIt! PDK v2.0.0+, but as the PDK is under continuous development those actions may be further upgraded, modified, or removed in the future.

Alert Dialog Show Action

This action's main purpose is to present the player with a dialog model that already available and saved in the database. The dialog is fetched from the database and if it contains any placeholders they are resolved. Next, the dialog is presented to the player.

```
class AlertDialogShowAction(
    var dialogUid: String,
) : Action() {

    override suspend fun execute(): ExecutionResult {
        val alertDialog = AlertDialog.find(dialogUid).await()

        if(alertDialog == null) {
            GeoMakeIt.logger.e("Alert Dialog '$alertDialog'
                                doesn't exist.")
            return ExecutionResult(null)
        }

        App.getCurrentActivity().runOnUiThread {
            alertDialog.show()
        }

        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        private const val DIALOG_UID = "dialog_uid"
```

```
        override val kcls = AlertDialogShowAction::class
        override val label: String = "alert_dialog_show"

        override fun getMetadata() = Metadata(
            pageTitle = "Alert Dialog: Show",
            pageDescription = "Display a saved alert dialog from
                               the database to the current
                              player."
        )

        override fun getTooltips(): Map<String, String> = mapOf(
            DIALOG_UID to "The unique identifier of the saved
                           alert dialog model."
        )
    }
}
```

Custom Alert Dialog Show Action

When the game author wants to present the player with a dialog that isn't already saved and available in the database, he can request it dynamically by using this action. The dialog is based on the input parameters provided by the game author and any placeholders contained are resolved. Next, the dialog is presented to the player.

```
class AlertDialogShowCustomAction(
    var title: String,
    var message: String,
    var positiveButton: AlertButton? = null,
    var neutralButton: AlertButton? = null,
    var negativeButton: AlertButton? = null
) : Action() {

    override suspend fun execute(): ExecutionResult {
        val alert = AlertDialog(
            Util.autoId(),
            title,
            message,
            false,
            positiveButton,
            neutralButton,
            negativeButton
        )
        App.getCurrentActivity().runOnUiThread {
            if(!App.getCurrentActivity().isFinishing) {
                alert.show()
            }
        }
        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
```

```
        private const val TITLE = "title"
        private const val MESSAGE = "message"
        private const val POSITIVE_BUTTON = "positive_button"
        private const val NEUTRAL_BUTTON = "neutral_button"
        private const val NEGATIVE_BUTTON = "negative_button"


        override val kcls = AlertDialogShowCustomAction::class
        override val label: String = "alert_dialog_show_custom"

        override fun getMetadata() = Metadata(
            pageTitle = "Alert Dialog: Show custom",
            pageDescription = "Display a **custom** alert dialog,
                               which is not previously saved as a
                               model, to the current player."
        )

        override fun getTooltips(): Map<String, String> = mapOf(
            TITLE to "The title of the alert dialog to be
                      displayed.",
            MESSAGE to "The message contained inside the alert
                        dialog.",

            POSITIVE_BUTTON to "Button located on the **right**
                                side of the modal.",
            NEUTRAL_BUTTON to "Button located on the **middle**
                               side of the modal.",
            NEGATIVE_BUTTON to "Button located on the **left**
                                side of the modal.",
        )
    }
}
```

App Close / Terminate Action

There are times when the game author may want to terminate the game, such as the game has ended, or a crash may have occurred, or even when the user has decided to exit the game.

This action can be also executed by using the `App Close Action`, which forcefully terminates the application, closing all active activities and windows.

```
class AppCloseAction: Action() {

    override suspend fun execute(): ExecutionResult {
        App.getCurrentActivity().finishAffinity()
        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        override val kcls = AppCloseAction::class
```

```
        override val label: String = "app_close"

        override fun getMetadata() = Metadata(
            pageTitle = "App: Close",
            pageDescription = "Terminate the app by closing it
                               and all its windows."
        )
    }
}
```

Marker Draw/Erase Action

Instead of using the command for drawing or erasing a marker, the game author can now use an action. There are two different action classes, one performing each action, but for the sake of simplicity we will merge them into one code here were both functionalities can be presented.

```
class MarkerDraw/EraseAction(
    var markerUid: String,
) : Action() {

    override suspend fun execute(): ExecutionResult {
        val marker = Marker.find(markerUid).await()
        if(marker == null) {
            GeoMakeIt.logger.e("Marker '$markerUid' doesn't
                                exist.")
            return ExecutionResult(null)
        }

        App.getCurrentActivity().runOnUiThread {
            marker.draw(App.map)
                  (or)
            marker.erase(App.map)
        }

        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        private const val MARKER_UID = "marker_uid"

        override val kcls = MarkerDraw/EraseAction::class
        override val label: String = "marker_draw/erase"

        override fun getMetadata() = Metadata(
            pageTitle = "Marker: Draw/Erase",
            pageDescription = "Draws/Erases a saved marker from
                               the database on the map."
        )

        override fun getTooltips(): Map<String, String> = mapOf(
```

```
                MARKER_UID to "The unique identifier of the saved
                              marker model."
            )
        }
    }
```

Zone Draw/Erase Action

Similarly, to the `Marker Draw` and `Marker Erase` actions, a zone can be drawn or erased from the map using actions. The game author has to provide the zone identifier as an input and the action does the rest.

```
class Zone Draw/Erase Action(
    var zoneUid: String,
) : Action() {

    override suspend fun execute(): ExecutionResult {
        val zone = Zone.find(zoneUid).await()
        if(zone == null) {
            GeoMakeIt.logger.e("Zone '$zoneUid' doesn't exist.")
            return ExecutionResult(null)
        }

        App.getCurrentActivity().runOnUiThread {
            zone.draw(App.map)
            zone.erase(App.map)
        }

        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        private const val ZONE_UID = "zone_uid"

        override val kcls = ZoneDraw/EraseAction::class
        override val label: String = "zone_draw/erase"

        override fun getMetadata() = Metadata(
            pageTitle = "Zone: Draw/Erase",
            pageDescription = "Draws/Erases a zone on the map."
        )

        override fun getTooltips(): Map<String, String> = mapOf(
            ZONE_UID to "The unique identifier of the saved zone
                         model."
        )
    }
}
```

User Logout Action

If the game author wants to logout a user or provide a UI where the user can press a button to logout, this action can be performed using the `UserLogoutAction`.

```
class UserLogoutAction: Action() {

    override suspend fun execute(): ExecutionResult {
        CurrentUser.signOut()
        App.getCurrentActivity().finishAffinity()
        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        override val kcls = UserLogoutAction::class
        override val label: String = "user_logout"

        override fun getMetadata(): Metadata = Metadata(
            pageTitle = "User: Logout",
            pageDescription = "Logout the current user."
        )
    }
}
```

User Delete Action

There are times when the game author may want to permanently delete a user, such as when the user no longer wants to be part of the game and needs to delete all his data.

This action can be also executed by using the `User Delete Action`.

```
class UserDeleteAction: Action() {

    override suspend fun execute(): ExecutionResult {
        CurrentUser.delete()
        App.getCurrentActivity().finishAffinity()
        return ExecutionResult(mapOf())
    }

    companion object : ActionDesigner {
        override val kcls = UserDeleteAction::class
        override val label: String = "user_delete"

        override fun getMetadata(): Metadata = Metadata(
            pageTitle = "User: Delete",
            pageDescription = "Delete the current user."
        )
    }
```

```
}
```

(Triggers 7.4.d)

Additionally, a set of triggers are also available in GeoMakeIt! PDK v2.0.0+. Since the PDK is under continuous development those triggers may be further upgraded, modified, or removed in the future.

These triggers provide points for the game authors to alter the game flow or monitor significant events.

Many times, a lot of triggers belong to some action counterparts. An easy example would be the `Zone Draw Action`. When this action is executed, a trigger is also broadcasted called `Zone Drawn Trigger`, as drawing a zone is considered a significant event. This can be seen in multiple parts of GeoMakeIt! Default Plugin as well as other plugins on the GeoMakeIt! Platform.

We are going to directly list all the available trigger classes below.

Alert Dialog Shown Trigger

This trigger is broadcasted as soon as an alert dialog is shown. It provides the game author with information such as the `unique_id` of the alert dialog, its `title` and `message`.

```
object AlertDialogShownTrigger: Trigger("on_alert_dialog_shown"),
                                TriggerDesigner
{
    override fun getMetadata() = TriggerMetadata(
        pageTitle = "Alert dialog shown",
        pageDescription = "This trigger is executed when an alert
                           dialog is shown to the player."
    )

    override fun getIngredients(): Map<String, String> = mapOf(
        "unique_id" to "string",
        "title" to "string",
        "message" to "string",
    )
}
```

Alert Dialog Action Selected Trigger

This trigger is broadcasted as soon as a player clicks an action on an alert dialog. It provides the game author with information such as the `alert_dialog_id`, the `type` of the button clicked, the `action` it executed and the `text` it contained.

```
object AlertDialogActionSelected:
                    Trigger("on_alert_dialog_action_selected"),
                    TriggerDesigner
{
    override fun getMetadata() = TriggerMetadata(
        pageTitle = "Alert dialog action selected",
        pageDescription = "This trigger is executed when the
                           player clicks a button inside the
                           alert dialog."
    )

    override fun getIngredients(): Map<String, String> = mapOf(
        "alert_dialog_id" to "string",
        "type" to "enum",
        "text" to "string",
        "action" to "enum",
    )
}
```

App Started Trigger

This trigger is broadcasted the moment the application is started.

```
object AppStartedTrigger: Trigger("on_app_started"),
                          TriggerDesigner {

    override fun getMetadata() = TriggerMetadata(
        pageTitle = "App: Application Started",
        pageDescription = "This trigger is executed when the app
                           starts."
    )

    override fun getIngredients(): Map<String, String> = mapOf()
}
```

Game Started Trigger

Similar to the trigger above but executed when the game is started.

```
object GameStartedTrigger: Trigger("on_game_started"),
                           TriggerDesigner {

    override fun getMetadata() = TriggerMetadata(
        pageTitle = "App: Game Started",
        pageDescription = "This trigger is executed when the game
                           starts."
```

```
    )

    override fun getIngredients(): Map<String, String> = mapOf()
}
```

Marker Drawn / Erased Trigger

```
object MarkerDrawn/ErasedTrigger:
                            Trigger("on_marker_drawn/erased"),
                            TriggerDesigner {

    override fun getMetadata() = TriggerMetadata(
        pageTitle = "Marker: Drawn/Erased",
        pageDescription = "This trigger is executed when a marker
                            is drawn/erased."
    )

    override fun getIngredients(): Map<String, String> = mapOf(
        "unique_id" to "string",
        "position" to "geopoint",
        "title" to "string",
        "snippet" to "string",
        "alpha" to "float",
        "flat" to "boolean",
        "draggable" to "boolean",
        "visible" to "boolean",
        "rotation" to "float",
        "z_index" to "float"
    )
}
```

Zone Drawn / Erased Trigger

```
object ZoneDrawn/ErasedTrigger: Trigger("on_zone_drawn/erased"),
                                TriggerDesigner {

    override fun getMetadata() = TriggerMetadata(
        pageTitle = "Zone: Drawn/Erased",
        pageDescription = "This trigger is executed when a zone
                            is drawn/erased."
    )

    override fun getIngredients(): Map<String, String> = mapOf(
        "unique_id" to "string",
        "title" to "string",
        "clickable" to "boolean",
        "visible" to "boolean",
        "z_index" to "float",
```

```
            "fill_color" to "string",
            "stroke_color" to "string",
            "stroke_width" to "float"
        )
    }
```

Zone Enter / Exit Trigger

```
object ZoneEnter/ExitTrigger: Trigger("on_zone_enter/exit"),
                              TriggerDesigner {

    override fun getMetadata() = TriggerMetadata(
        pageTitle = "Zone: Player enters/exits",
        pageDescription = "This trigger is executed when a player
                           enters/exits any zone."
    )

    override fun getIngredients(): Map<String, String> = mapOf(
        "unique_id" to "string",
        "title" to "string",
        "clickable" to "boolean",
        "visible" to "boolean",
        "z_index" to "float",
        "fill_color" to "string",
        "stroke_color" to "string",
        "stroke_width" to "float"
    )
}
```

Quest Assigned/Completed/Dismissed/Failed Trigger

```
object QuestAssigned/Completed/Dismissed/FailedTrigger:
        Trigger("on_quest_assigned/completed/dismissed/failed"),
        TriggerDesigner {

    override fun getMetadata() = TriggerMetadata(
        pageTitle = "Quest: Assigned/Completed/Dismissed/Failed",
        pageDescription = "This trigger is executed when a quest
                           is assigned/completed/dismissed/failed
                           to/by/by/by a player."
    )

    override fun getIngredients(): Map<String, String> = mapOf(
        "unique_id" to "string",
        "title" to "string",
        "description" to "string",
    )
}
```

User Created/Deleted/SignedIn/SignedOut Trigger

```
object UserCreated/Deleted/SignedIn/SignedOutTrigger:
          Trigger("on_user_created/deleted/signed_in/signed_out"),
          TriggerDesigner {

    override fun getMetadata() = TriggerMetadata(
        pageTitle = "User: Created/Deleted/Signed In/Signed Out",
        pageDescription = "This trigger is executed when a user
                           account is created/deleted/signed in/
                           signed out."
    )

    override fun getIngredients(): Map<String, String> = mapOf(
        "unique_id" to "string",
        "username" to "string",
        "email" to "string",
        "anonymous" to "boolean",
    )
}
```

CHAPTER 8: Summarizing and Future Planning

GeoMakeIt! PDK v2.0.0+ has brought a lot of significant changes in the whole concept of designing a modular and interoperable plugin. New levels of abstraction were introduced, making plugin development a lot easier.

While on the older versions of GeoMakeIt! PDK plugin developers required to specify the exact implementation for grabbing files, parsing JSON, fetching database objects, registering classes etc., on the newer versions of GeoMakeIt! PDK this is no longer required. By inheriting some abstract implementations all of these functionalities are done behind the scenes, with the plugin developer needing to only focus on what matters, creating his plugin.

The tools introduced were:

- **GeoMakeIt! Alchemist**: An event-driven system to share and propagate state changes throughout plugins, allowing for easy plugin interoperability and customizability. We further discussed `Actions`, `Triggers`, `Filters`, and `Recipes`; and showed multiple examples of their implementations.

- **GeoMakeIt! Designers**: An automatic generation for interactive user interface from GeoMakeIt! PDK to the GeoMakeIt! Studio. This feature additionally allows plugin developers to export custom UI on GeoMakeIt! Studio. Furthermore, we discussed `Input / Output Designers`, `Trigger / Action / Config Designers`, `Ingredients`, `Components` and `Validation` Tools.

- **GeoMakeIt! Meta Injection Tool**: Automatic generation of meta data, automatic import of meta data for creating higher levels of abstraction using Java / Kotlin reflection tools.

There were additional modifications for further improving the GeoMakeIt! PDK, such as `asynchronous tasks`, `asynchronous IO / DB Operations`, `automatic database population` etc., that couldn't be grouped in a specific chapter.

With all these and more to come in the future, we hope, we strive, for a better and easier experience for game authors and plugin developers alike. Some of the features that the GeoMakeIt! Platform is currently working on are:

Graphically rich environment

We believe that the most powerful tool over the GeoMakeIt! Platform should be the Studio. GeoMakeIt! Studio is already having a major update, providing a more user-friendly environment that can be easily grasped by non-experts and feel natural. JSON files and hard-coded commands have been modified to using a dynamic system, with GeoMakeIt! Alchemist, where all plugins make easily accessible to the GeoMakeIt!

Studio their available triggers, actions and filters can be uniformly displayed to the game author.

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, triggers, actions, 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 are already being in the works for GeoMakeIt! Studio, allowing a game author to create a prototype with different combinations of plugins instantly based on common game templates 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 authors 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 authors and plugin developers to earn a generous income out of their creations. Game authors 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.

Database and Map Wrappers

The current state of GeoMakeIt! provides hard-coded database and map choices, specifically Google's Firebase and Google Maps. Game authors 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 game authors easily. As such a game created with GeoMakeIt! can be used by different people in multiple countries without language restrictions.

REFERENCES

[1] "History of Games", https://en.wikipedia.org/wiki/History_of_games, Accessed 2022-11-01

[2] Tetris, Andreas Elmenthaler (Elmi), "Hagenuk MT-2000 with Tetris", http://handy-sammler.de, Archived from the original on June 17, 2013, Accessed 2022-11-01

[3] "Snake is born: A mobile gaming classic" (in Dutch), Nokia, https://web.archive.org/web/20090209232201/http://www.nokia.com/A4303014, Accessed 2022-11-01

[4] "Revenue Model", https://en.wikipedia.org/wiki/Video_game_monetization#Retail, Accessed 2022-11-01

[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 2022-11-01

[9] Bates, Bob (2004), p. 204. Game Design (2nd ed.). Thomson Course Technology. ISBN 1-59200-493-8.

[10] "Game Engines", https://en.wikipedia.org/wiki/Game_engine, Accessed 2022-11-01

[11] "List of Game Engines", https://en.wikipedia.org/wiki/List_of_game_engines, Accessed 2022-11-01

[12] "Using a game engine on Android", https://developer.android.com/games/engines/engines-overview, Accessed 2022-11-01

[13] Martinez, Juan (27 October 2016). "The Best eLearning Tools". PC Mag.

[14] "Authoring Systems", https://en.wikipedia.org/wiki/Authoring_system, Accessed 2022-11-01

[15] "Android", https://www.android.com, Accessed 2022-11-01

[16] "Android OS", https://en.wikipedia.org/wiki/Android_(operating_system), Accessed 2022-11-01

[17] "APK (File format)", https://en.wikipedia.org/wiki/Apk_(file_format), Accessed 2022-11-01

[18] "Android Architecture, https://source.android.com/devices/architecture, Accessed 2022-11-01

[19] “Laravel Framework”, https://laravel.com, Accessed 2022-11-01

[20] “Angular JS”, https://angularjs.org/, Accessed 2022-11-01

[21] “Google Maps”, https://www.google.com/maps, Accessed 2022-11-01

[22] “Google Maps SDK Intro”, https://developers.google.com/maps/documentation/android-sdk/intro, Accessed 2022-11-01

[23] “MySQL”, https://www.mysql.com/, Accessed 2022-11-01

[24] “Google Firebase”, https://firebase.google.com/, Accessed 2022-11-01

[25] V. Dimitriadis, A. N. Dadaliaris, M. Koziri, T. Loukopoulos, “Development of a Web portal for authoring and deploying position-based games using abstract domain specific programming language structures”, Thesis, Computer Science Department, University of Thessaly, Oct. 2020

[26] 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