Maliyo Games
Technical Framework Document

About

Maliyo Games is an African game studio that creates games for mobile devices. Our mission is to **revolutionize** the gaming industry in Africa by being one of the first studios to bring homegrown indigenous content to the world of games.

This technical paper is here to **outline several guidelines** that the team must agree on and adhere to in order to maintain **quality**, **consistency**, and **extensibility** throughout the development of our various games.

This document comes as a first proposal, it will evolve as we progress through the technical implementation of our games. Any member of the development team is **welcome to suggest additions or changes**.

However, each suggestion will need to be discussed and approved by the majority in a team meeting.

We must strive to communicate effectively, listen to and comprehend other team members, and do so with **love and respect**.

Project management

The technical team will have project-based **weekly scheduled meetings** over Google Meet. Days and time for these meetings are to be defined during the project's planning phase and might be adjusted during the project if a need is identified. **Additional meetings might be scheduled** to conduct specific discussions if needs be.

To manage each project and task, we will use the **SCRUM methodology**. A **Trello board** will be created, all tasks identified will be added to a **Backlog List**. Each task will be assigned to a member of the team and will be moved to the **Doing List.**
When each task has been executed, it will be added to the **Review List**. Your Project manager will check the execution of each task and move it to the **Done List** when it passes.
(**Note:** You cannot move your task to the **Review List** until you commit it.)

We'll start with a sprint duration of one week, but this could be re-adjusted eventually if the team feels it's relevant.

A **sprint planning** will take place at the beginning of each sprint, and a **sprint review** on the last day of the sprint. Every day, we will have a short **SCRUM meeting**, where everyone should say:

- what they've been working on the previous working day
- what they will be doing today
- and whether a problem was encountered or something is slowing them down

If someone is **unable to attend a meeting or cannot arrive on time**, the rest of the team should be **notified in advance via Slack**. In such a case, the meeting may be postponed or held without the team member.

Other Communication channels

A **Slack server** with a **dedicated channel** will be created for the team to communicate with each other during the whole duration of the project. Do ensure that you have slack installed on your devices 💻 📱 so that you can be easily reached.

We expect everyone on the project to **communicate** with the rest of the team on a regular basis. For example, one can update the team about progress (git push), issues encountered, good/bad practices, and so forth. You are also expected to notify the rest of the team if you are not going to be able to attend a meeting or make a deadline.

Daily Scrum meetings are compulsory and should be atttended for both the technical 💻 and non-technical 👩‍🎨 teams.

Tasks assignment

Each team member will be assigned tasks during the sprint planning meeting. We will try to **estimate the time required** to achieve the task together and move it to the sprint To-Do section. We will **take personal preferences into account** as much as possible in task assignments, but you might not get "*the task you wanted*" every time, some boring things sometimes need to be done!

It is critical that whoever takes responsibility for a task feels at ease and secure in their abilities. If you are not, please notify the team lead that it may be difficult and that assistance may be needed.

Coding Conventions

Folder Structure

All similar files should be grouped in the root folders or subfolders, for instance:

Assets

Animations	Animation clips
Editor	Editor specified scripts, prefabs, etc
External	For external plugins or solutions
Fonts	Fonts used in-game
Materials	Texture materials
Plugin	Platform specific libraries
Prefabs	In-game prefabs
Resources	Unity resources assets
Scenes	Scenes
Scripts	Code scripts, grouped in sub-folders
ScriptableObjects	ScriptableObjects Data assets and managers
Sounds	Music, sounds
Textures	Image textures
UI	Texture used for UI
Icons	App icons

Scripts Naming

- XxxPanel, XxxSlot, XxxButton, etc for UI
  MenuPanel, AchievementSlot, CoinShopButton

- XxxManager, for main systems scripts(one instance per scene)
  DailyMissionManager, *AchievementManager*

- XxxController, for scripts that control one specific kind of GameObject (one or more instances per scene)
  PlayerController, *BossController*, *BackgroundController*

- XxxData, data loading scripts (ScriptableObject, JSON, CSV...)
  WeaponData, *ShopItemData*

- XxxEditor, Editor extension scripts
  TutorialTaskEditor, *AchievementSettingsEditor*

File Naming

File naming is simple, always use Pascal Case except for images:

- Folders - PascalCase
  FolderName/
- Files names separator is underscored:
  image_name_64x64.jpg
- The rest - PascalCase
  ScriptName.cs, PrefabName.prefab, SceneName.unity

Use icons for different scripts to easily identify some objects in the scene

Variable Naming

Variable naming like file naming, allows us to know what the variable is used for without having to read the entire code. A variable name should **clearly** describe what it contains, thus be verbose and utilize long variable names based on word construction. The compiler will optimize your script and your IDE will offer auto-complete as you type.

Examples : playerInventoryItemList / nextDeliveryDelay / ...

Please **do not use any abbreviation for words**, like `pInvItList` instead of `playerInventoryItemList`.

- Use camelCase for **local variables**

```
hasRewarded, currentPosition, isInitialized
```

- Prefix with “m_” for class members

```
m_itemCount, m_controller, m_titleText
```

- Prefix with an underscore for **methods parameters**

```
_itemCount, _controller, _titleText
```

- All capital letters for const

```
    //MAX_HP_COUNT, BASE_DAMAGE

private const float INVENTORY_REFRESH_RATE = 0.5f;
```

- You can use a const for and prefix with "PREFS_" for *PlayerPref* keys

```
private const string PREFS_LAST_FREE_DRAW = "LastFreeDraw";
PlayerPrefs.GetInt(PREFS_LAST_FREE_DRAW)
```

- Use xxxGO for scene GameObject variables

```
    public GameObject m_optionButtonGO;
```

- Use xxxPrefab for scene Prefabs variables

```
    public GameObject m_itemSlotPrefab
```

- Use xxxComponent for all other components, abbreviate if needed

```
    itemSpriteRenderer, walkAnimation, itemSoldAudioClip
```

- Use xxxxList for List and xxxxDict for Dictionary

```
    weaponTransformList = new List()
    achievementProgressDict = new Dictionary()
```

- Use xxxx(s) for arrays (prefer Lists)

```
    achievementIds = new int [4]
```

Method Naming

Methods use PascalCase naming convention. For Example:

```
    public void InputMethod()
```

Comments

If code stays organized and naming conventions are well respected, we shouldn't need to write a lot of comments. Though there are a few use cases where comments can come up in handy:

- Doing something for a non-obvious reason

```
// Sorting this List because ...
```

- Leaving a note that some of the code will need to be improved in the future:

```
// TODO: support this very specific use case
```

- Leaving a note that some sections of code might be optimized in the future (it might more relevant to have a non-optimal feature quickly):

```
// OPTIMIZATION: use an object pool instead of instantiating
prefabs
```

Please don't leave whole chunks of commented code, the SCM will remember it for you in case it needs to be recovered!

Namespaces

For classes that are unique to this game, there is no need to use namespaces. However, whenever there is a chance to reuse a script in a subsequent project, it should be placed under the **MaliyoGames** namespace. For instance:

```
namespace MaliyoGames
{
    public class Localization: Singleton<Localization>
{
...
```

Eventually, we could build a library that Maliyo developers are familiar with, and that can widely be reused across projects.

Assembly Definitions

To better organize your scripts in the unity game engine and also boost compile times, make use of assembly definitions
For better understanding, please refer to the following resources:

- Unity Documentation: Assembly definitions
- Speed Up Compile Times Using Assembly Definitions

Code Architecture

For various project requirements, we employ several code architectures. The design and organization of your code are critical to ensuring that your project does not become **tedious** and **difficult to debug** during long development durations. We should strive for code that is **clean**, **reusable**, and **loosely coupled.** Architectures and workflows evolve in response to new technology or tools, as we strive for faster and more efficient workflows.

Singleton Design Pattern

Singletons are powerful but also become hard to manage with the increasing scope of the project with systems being tightly coupled. Singletons are to be used in quick prototypes and small projects and are to be used mostly for manager classes. A script to quickly create a singleton would be added to the library.

C# Events (Subscriber / Observer Pattern)

If the project is a complex game with many systems interacting, and to **keep systems loosely coupled, events will be used extensively**.

Here is an example of how to declare events:

```
public class EventManager : Singleton<EventManager>
{
     // Dialog Start Event
     public static event Action<string> onDialogEnd;

     public void FireOnDialogEnd(string _sequenceId)
     {
          onDialogEnd?.Invoke(_sequenceId);
     }
...
```

Now here is how to subscribe/unsubscribe to events:

```
private void OnEnable()
{
     EventManager.onDialogEnd += OnDialogEnd;
}
private void OnDisable()
{
     EventManager.onDialogEnd -= OnDialogEnd;
}
```

Note that event callback methods are named *OnNounVerbed* like the `OnDialogEnd` method above.

ScriptableObjects Based Architecture

This is the recommended architecture for large projects as it incorporates the use of scriptable objects to make the project loosely coupled and not scene-dependent.
This architecture is best used in combination with singletons to make **singleton scriptable objects** and **scriptable object event systems**. The Architecture is available on the library repo and would evolve.

Entity Component System (ECS) and DOTS

This architecture is best used when the project demands a large number of objects in the scene and performance is critical, as in 3D mobile games. It is still in its early stages, and the workflow is not yet fully structured; additional information will be added as time goes on.

Source Code Management (SCM)

We use a **BitBucket or Github git repository** for source code management. It is recommended that every team member uses SourceTree (BitBucket) / GitHub to interact with the git repository.

A "*.gitignore*" file has already been placed on the repository, but developers should be cautious about not submitting locally generated files (eg. Visual Studio files, sprite atlases, etc...).

Some Unity files generally don't merge well, like the *.scene* files. To avoid conflicts on that kind of file, developers are encouraged to:

- work with **Unity prefabs** as much as possible
- use a multi-scene workflow
- let the team know that they want to take **ownership of a scene** on a Slack channel
- **Temporarily work on a copy of the scene** that may or not be pushed to the repository.

Other SCM tools like Plastic SCM(For projects with large files) and Unity Collaborate may be used based on the use case.

Unity Version

Every member of the team must use the same version of Unity. This will prevent incompatibilities and conflicts in the configuration files.

The most recent **LTS version** of Unity is recommended. We may upgrade this version later in a project if we believe it is necessary, but we will strive to keep the version as stable as possible.

Code Reviews

Each time a developer pushes on the GIT repository, a **code review should take place**. At the beginning of the project, the code review will be conducted by the project's *lead developer*. But the responsibility might shift to other team members later on, if relevant.

Standard Process:

- Developers simply need to push to the master branch, and the code review would occur post-commit. Every push on the GIT repository should be notified to the other team members on the dedicated Slack channel (through **webhooks** or **email notifications**).

Alternatively:

- We could use **branches** and integrate after the code review for larger items that need to be done in isolation until they are stable, however, this could add a significant overhead or timespan. In that case, the code will be reviewed before being pushed to the master branch.

In any case, any spotted issue will be notified to the developer and discussed (if necessary). Sometimes, issues will be notified to the team on the tech channel of Slack and good practice will be added to this document.

Debug log messages

It's okay to apply Debug.Log during development, but every developer should be aware of how it's used. If we all contribute our own debug messages throughout development, the console will eventually be bombarded with debug messages that may be difficult to filter and may not make sense to everyone. As a result, submitting Debug.Log messages to the repository is strongly discouraged.

However, *Debug.LogWarning* and *Debug.Error* messages should be used in the right situations:

- A **warning message** is here to inform programmers that something is unexpected and might cause issues in the future if it stays as is. There is no emergency to fix it, but it should be considered a potential issue.
- An **error message** informs programmers that something wrong has happened and that it should be fixed as soon as possible. It is the responsibility of every programmer on the project to ensure that we fix those errors as quickly as possible.

You can use the script below to disable debug log in the game build:

```
#if UNITY_EDITOR
Debug.logger.logEnabled = true;
#else
Debug.logger.logEnabled = false;
#endif
```

Game Configuration and Balancing

While working on a project with complex systems interacting together to support the global game experience. To balance the game efficiently, it is vital to **offer ways for the game design team to edit any single value** that might have an impact on the game balancing.

Several solutions are being investigated:

1. **Unity Scriptable Objects**
2. JSON data files
3. public class members that can be edited within Unity
4. Spreadsheets
5. Creation of a balancing menu, available at runtime

The **Unity Scriptable Objects** is currently the preferred method, but it might not be used exclusively. Should Scriptable Objects does not appear like the ideal solution, other solutions can be investigated for some features for example using **Google Sheets** or **CSV** files. It'll be discussed and decided in collaboration with the Lead Developer, case by case.

Mobile First

Maliyo Games is a mobile-first game company, which means our games primary target platform is **Android** and **iOS**. Everyone in the team should be conscious that the UX should be optimized for those platforms, and proactively test the game on smartphones regularly.

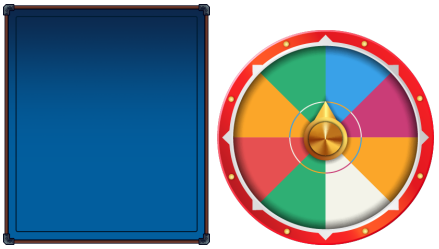
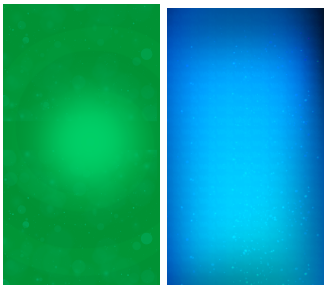
Special attention should be paid to the visual and UI elements that may require presentation adjustments depending on the screen resolution.

Image/Texture Import Settings and Compression

To keep the size of games small while maintaining their quality, the following guidelines are to be followed for good results. Using Whot King as a study case:

Whot King Image Compression Analysis

Category	Suitable max size	Examples
Small Icons and buttons	128	Announcement icon, social media icons 
Items (Images used to represent important objects, valuables)	256	Coins, heart, keys, diamond 

Large UI Images	512	**Large** buttons, pop-up panel header, chests 
Logos	1024	Game logo 
Pop-up Panels	1024	Kingdoms' panels, spin wheel 
Plain Backgrounds	512	Menu background, gameplay background 
Detailed Backgrounds	1024	Kingdom map backgrounds 

Cheats

As we progress through a game implementation, whether via scriptable objects or a custom editor window, we'll need to include cheats to assist us to proceed faster or even skip to a specific place in the game. Here are a few examples:

- add XXX money
- unlock fire powerup
- move on to chapter 3

Don’t hesitate to suggest your cheats to your game!

Use of Unity packages or plugins from Asset Store

The addition of a new Unity package for the package manager might be necessary and beneficial to the project. **Any programmer can suggest the addition of a new package** to the common communication channels. If there is no reaction to this request, and the developer cannot wait to get an answer, it's acceptable to add a package proactively. In such a scenario, the rest of the team shall be informed that the package was added to the project.

Use of external libraries/resources

Similarly, anyone can suggest the integration of an external resource at any point, as long as it’s relevant. However, because of possible integration or licence-related issues, the use of an external library or resource must always be discussed with the *Lead Developer*, ahead of integration.
There would be a dedicated library for MaliyoGames personal scripts and tools with recommendations for libraries and assets to import when creating a new project

Delivery Workflow

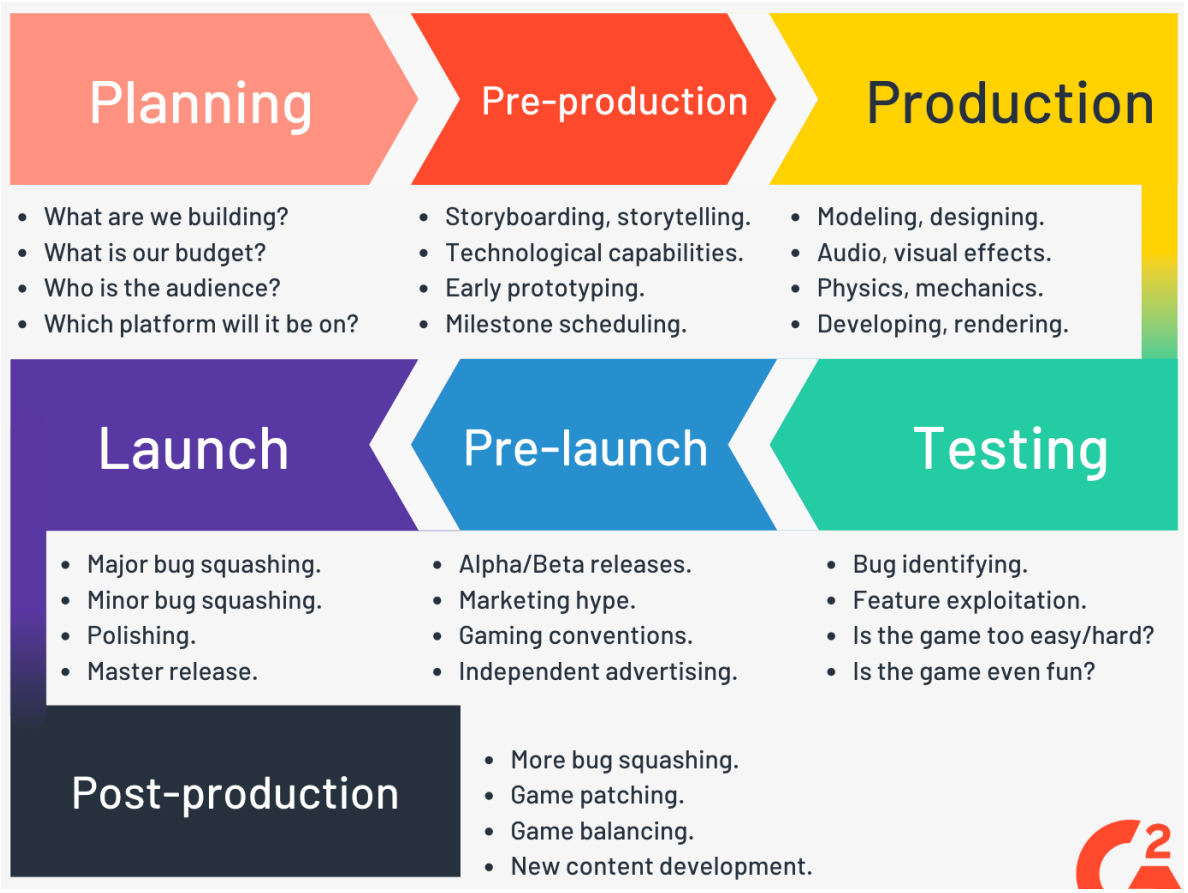
We need a stable delivery workflow in order to continuously deliver high-quality games.

We will release playable builds of the game in the form of APKs or/and executables as soon as we reach a stage when the fundamental gameplay experience of a game is implemented, which will correlate to the **Milestone.**

This process will be manual at first (one build a week), but we might automate this process through CI/CD eventually if it turns out to be time-consuming.

A feedback form would be associated with huge updates either in **Pre-Alpha**, **Alpha**, **Beta** or **Early Access** production stages to get bug reports and a feedback section would be added to the **final production** build.

Bug reports are to be added to the chosen software for tracking.



CI/CD

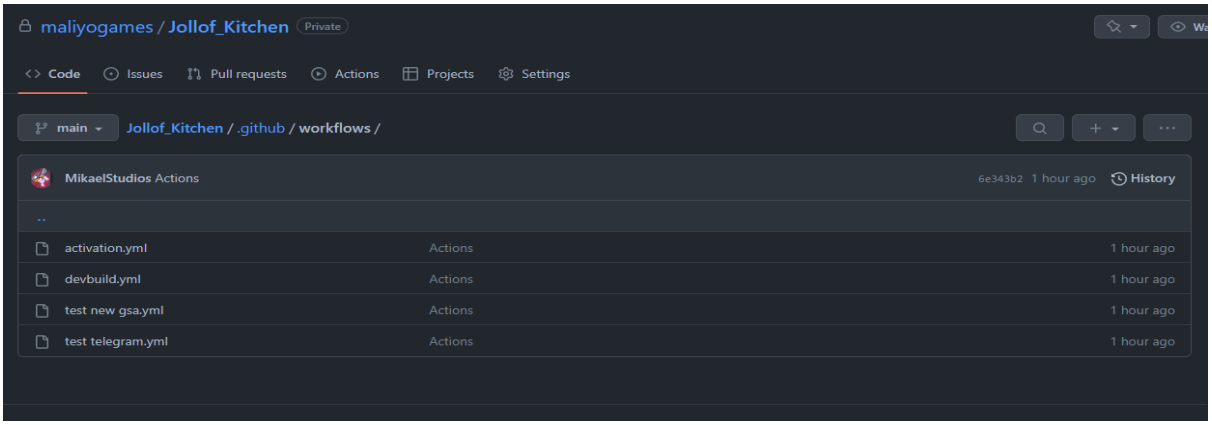
Create Builds with GitHub

To make builds using GitHub, follow these steps automatically:

Android Builds

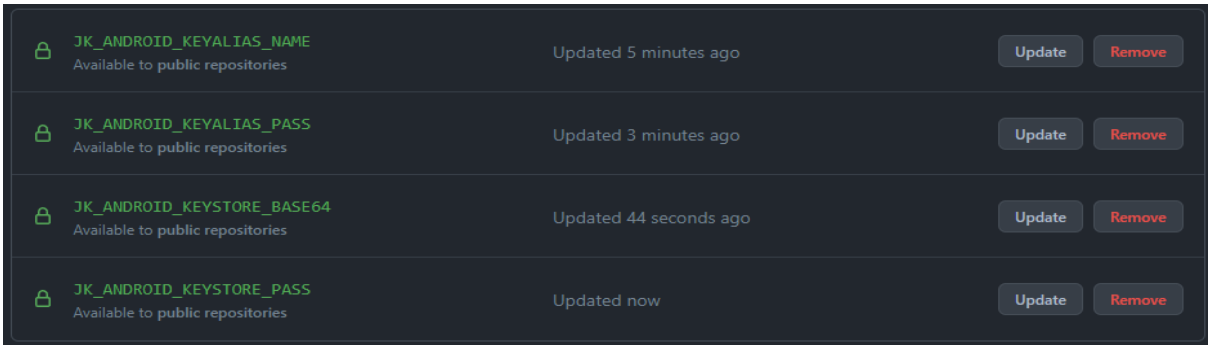
(**Note:** This process only supports apk builds easily, aab builds will need extra setup)

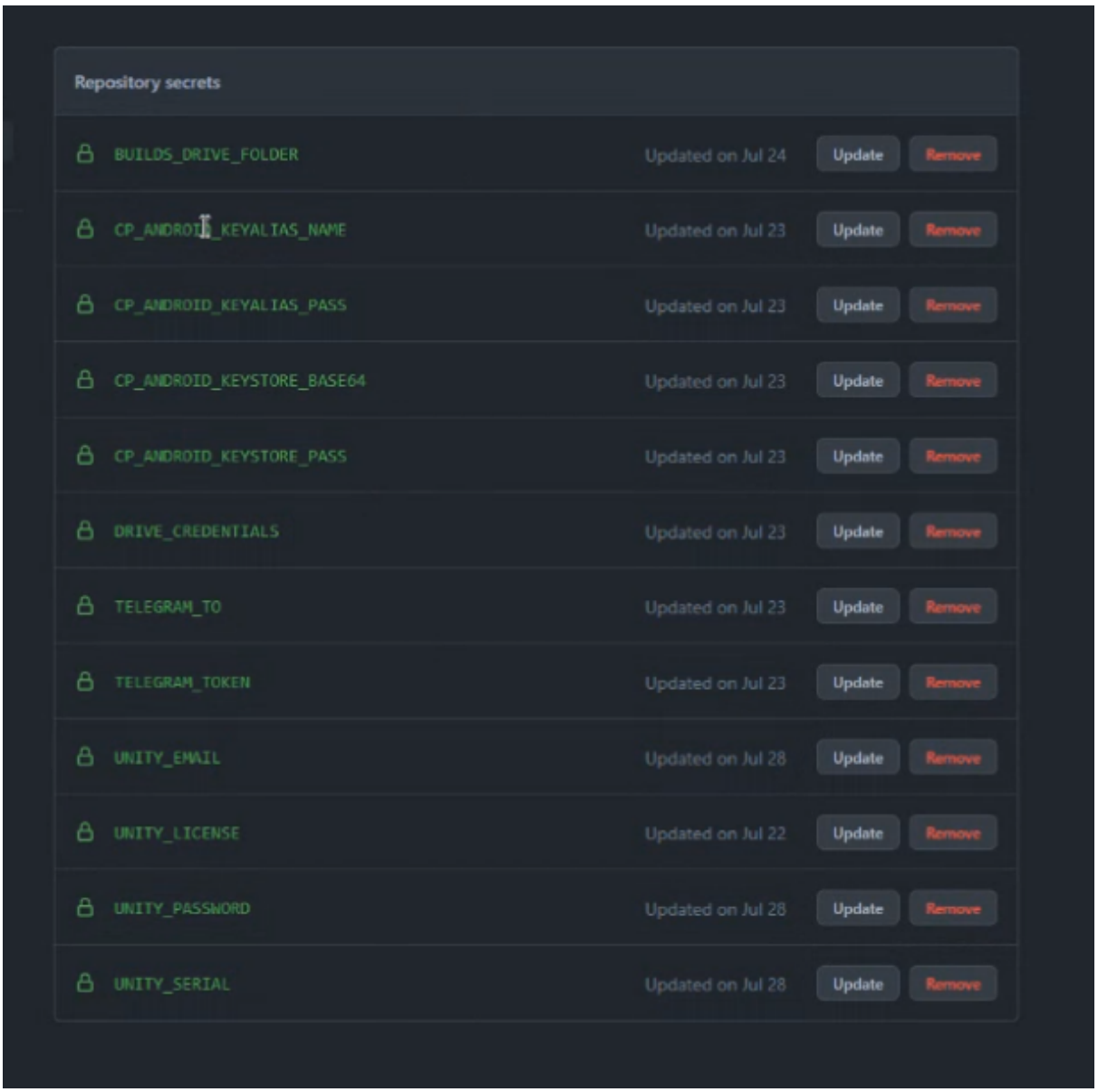
1. Import the .github folder from the Jollof Kitchen Repository, to the root of your project folder repository.



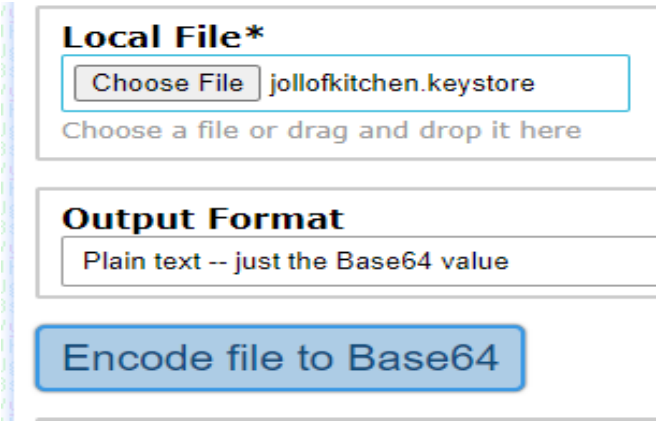
2. Setup **App Secrets** on the GitHub Repository.

3. Add the:
    - Name_Android_Alias
    - Name_Android_AliasPass

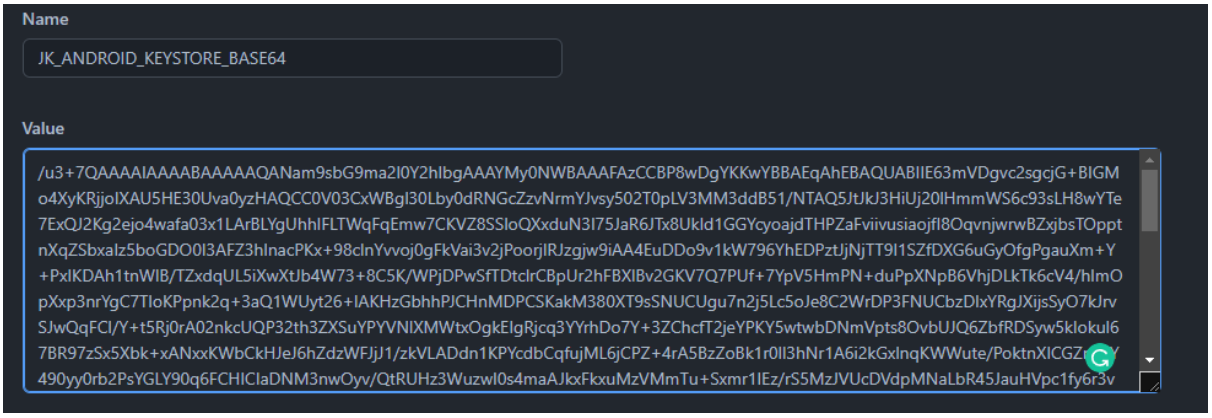


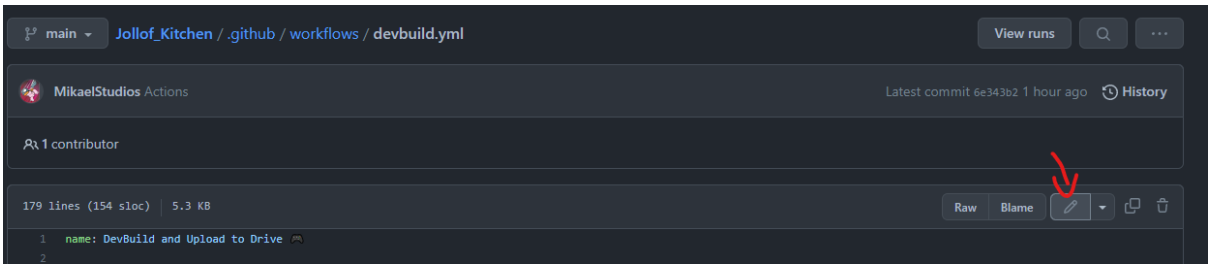

Repository secrets
BUILDS_DRIVE_FOLDER
Updated on Jul 24
Update
Remove
CP_ANDROID_KEYALIAS_NAME
Updated on Jul 23
Update
Remove
CP_ANDROID_KEYALIAS_PASS
Updated on Jul 23
Update
Remove
CP_ANDROID_KEYSTORE_BASE64
Updated on Jul 23
Update
Remove
CP_ANDROID_KEYSTORE_PASS
Updated on Jul 23
Update
Remove
DRIVE_CREDENTIALS
Updated on Jul 23
Update
Remove
TELEGRAM_TO
Updated on Jul 23
Update
Remove
TELEGRAM_TOKEN
Updated on Jul 23
Update
Remove
UNITY_EMAIL
Updated on Jul 28
Update
Remove
UNITY_LICENSE
Updated on Jul 22
Update
Remove
UNITY_PASSWORD
Updated on Jul 28
Update
Remove
UNITY_SERIAL
Updated on Jul 28
Update
Remove

4. Convert the .keystore to Base64 using [Base64 Converter]



5. Save the Base64 converted file to GitHub

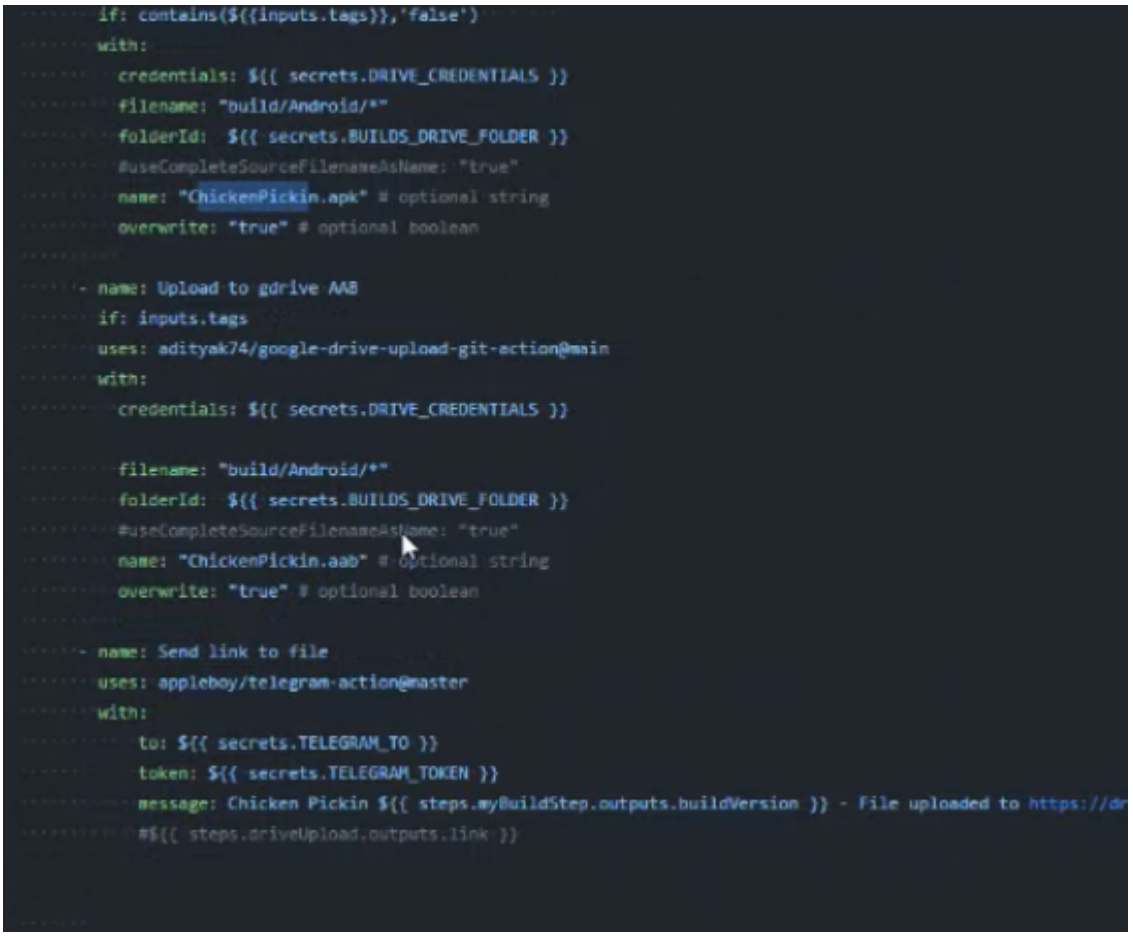




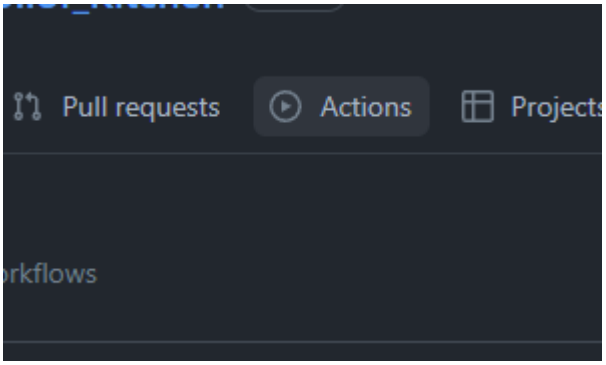
6. Adjust the Game Name, for both aab and apk versions in the .yml file

```
folderId:  ${{ secrets.BUILDS_DRIVE_FOLDER }}
#useCompleteSourceFilenameAsName: "true"
name: "JollofKitchen.apk" # optional string
overwrite: "true" # optional boolean
```

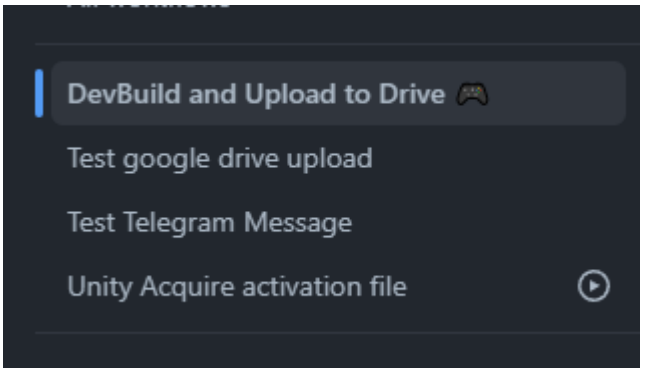
```
#useCompleteSourceFilenameAsName: "true"
name: "JollofKitchen.aab" # optional string
overwrite: "true" # optional boolean
```



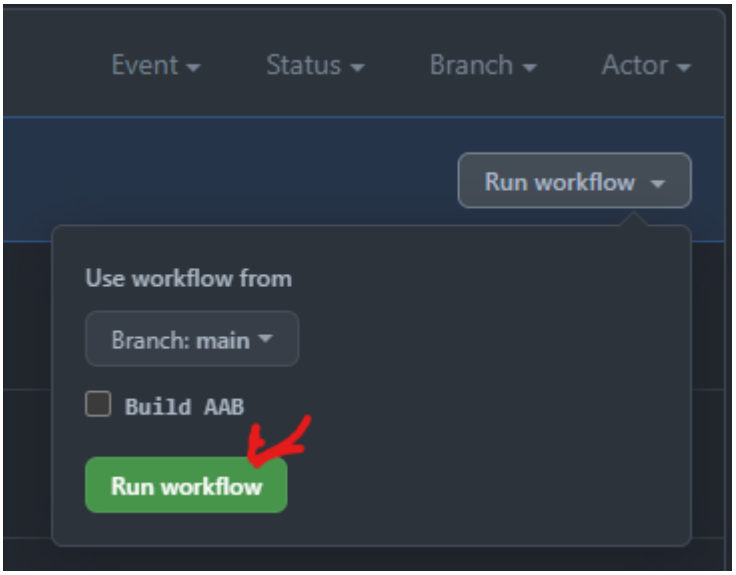
7. To automatically make a build, go to actions in the repo



8. Go to **DevBuild and Upload to Drive**



9. Click on Run Workflow (toggle on AAB to build and AAB)



Further Links

- GameCI Documentation
- Video Documentation