# Documentation

# Table of contents

# Home

# User Guide



## Overview

GameDB is a Unity Plugin that provides an easy to use and powerful game and meta data editor. With this asset your scripts no longer have to use hard-coded references and instead can refer to your database for lists of values, files, prefabs, and other data types. GameDB is designed to produce JSON and associated classes to load and access this data easily within a game. It supports many data types, relationships between tables and hot-reloading to make working with game data easy and intuitive.

Some examples of usages would be for data relating to things like Loot tables, NPC balancing data, Prefab lookups for procedural assets, game configuration, and many many more.

## Getting Started

The GameDB Editor window is opened via navigating to *Window* > *GameDB* menu and clicking on Open Editor menu item.

To get started creating and editing a gameDB click on the *Create GameDB* button. This will ask you to select a location to save your gameDB. When you click OK this will create two files a .json and .schema.json file in the directory chosen (recommended location is within a Resources folder as using Resources.Load is one of the easier ways to access the data).

The gameDB will then be loaded and ready to edit.

The first thing to do with your new gameDB is set a Scope Name for the gameDB. This name will be appended to the namespace of the exported C# classes your gameDB creates. For example a scope name of “MyData” would produce the namespace GameDBMyData. This scope name is also used as an identifier for various other things explained in more detail later.

Next you can create tables by entering a *Table Name*, select a key type and clicking on *Create Table*. This will create a table below that is collapsed by default. Click on the table name and it will expand to show your table. By default this will be a table with no keys or fields.

To create a field click on the *Edit Table* button, then enter a *Field Name* under the *Modify Schema* section. Select a data Type, check the tick-box for whether you would like it to be an array of data and then click *Create Field*.

This will then add a field to your table that can be removed with the [x] button (only visible after clicking *Edit Table*) if required.

To add some data enter a Key in the field at the top of the table then click *Create Key*. This will create a entry in the table below where you can now enter data for each of your fields.

Once you are happy with the data you have entered you can save the gameDB by clicking on *Save GameDB*, this will serialise out the data and the schema to the location selected earlier.

Next to use the data in your game you need to export the C# classes that will allow you to access the data. To do this click *Generate Classes* and select where to export the classes. This will generate C# classes that will provide easy and strongly typed access to your data.

# Usage In Game

You are free to use the json data however you like but the GameDB plugin generates code based on your data schema that provides an easy way to integrate the data with your game.

When you click *Generate Classes* this will generate code in the selected location that provide access to the data. Each of the classes will be namespaced with a name like GameDB{SCOPE_NAME_HERE} for example: GameDBMain. These are auto-generated classes and hand editing of the files is generally not recommended.

The main class that represents your game data is the GameDB class. This class will load the json, deserialize it and provide access to each of the loaded tables. The GameDB class also has a OnDBLoaded event that is triggered when the data is loaded.

An example of loading the game data:

```
using GameDBMain;
using UnityEngine;

public class Test : MonoBehaviour
{
    void Start ()
    {
        //A name is given when instantiating to identify the gameDB when editing during play
        GameDB gameDb = new GameDB("MyGameDB");
        gameDb.OnDBLoaded = delegate() {
            //Access game data here once the DB is loaded
        };

        //Load the json from the resources folder
        gameDb.Load("GameDBs/gameDB");
    }
}
```

A few classes are then created for each of the tables defined in your game data. There is a Table, Schema and Row class. Normally prefixed with the table name for example MyDataTable.cs, MyDataSchema.cs and MyData.cs. The Table class provides accessors for getting a particular row by Key or getting all the rows as a dictionary. The Schema class provides static accessors for the table name, field names and keys in the table. Lastly the Row class represents a particular row in your table and provides accessors for each field in the data.

An example of accessing data:

```
gameDb.OnDBLoaded = delegate() {
    //Access game data here once the DB is loaded
    var shipPart = gameDb.ShipPartsTable.GetByKey(ShipPartsSchema.KeyWing);

    Debug.Log(shipPart.CostVal);

    foreach (var loot in gameDb.LootTable.GetRows())
    {
        Debug.Log(loot.Value.NameVal);
    }
};
```

# Editing in-game

The gameDB can be edited while in game and hot-reloaded so you can balance while playing the game. If you have the GameDB Editor window opened when you are playing the game you can load one of the gameDBs you have loaded in the game by selecting the gameDB by its name in the *Runtime GameDB* dropdown, then selecting the base gameDB it was loaded from in the *Base GameDB* dropdown then clicking *Load GameDB*.

This will load the gameDB that is in memory in the game and allow you to edit it and then cause the data to reload in-game by clicking the *Reload In-Game* button. This will cause the OnDBLoaded event to be triggered on the gameDB in the game and when this happens your game code can react to the change in anyway it likes (see advanced topics for working with hot reloading).

The game data you edit while playing the game won't be saved back to the json until you click the *Save GameDB* button, meaning you can experiment with changes while the game is running then if you stop playing the game without saving the gameDB those changes will be discarded.

# Data Types

Currently the GameDB plugin supports the following data types:

- String
- Int32
- Float
- Bool
- UnityEngine.Color
- UnityEngine.Vector2
- UnityEngine.Vector3
- UnityEngine.Vector4
- Prefabs
  - Prefabs can be imported from the project and when settings data a drop down of imported prefabs will be available.
  - Prefabs must be imported from a resources folder as they are loaded via Resource.Load when accessed in game.
  - There are two accessors added into the generated Row class for each prefab field; one returning the path of the prefab and another returning the UnityEngine.Object representing the prefab.
- Enums (Pro version only)
  - Enums are imported from game code and can be selected as a type. Then when setting data for the field a drop down is given of the enum members that can be selected.
- Table References (Pro version only)
  - Fields can be a reference to another table within the gameDB. Then when setting data for the field a drop down is given with the keys of the referenced table
  - There are two accessors added into the generated Row class for each Table Reference field; one returns the string key of the entry and the other an instance of the row referenced.

*Arrays*

All the above data types can be stored as arrays. To modify the values of an array click on the [E] button next to the field and a popup will show allowing you to modify the size of the array and enter values for each index.

# Advanced Usage

## Add an existing gameDB

If you have a gameDB from another project or being migrated from somewhere else (or the settings have been reset) you can load this by clicking on the *Add Existing GameDB* button. Make sure that both the .json and .schema.json files are in the same location together.

## Importing Prefabs & Enums

To import prefabs & enums you will need to click on the *Configuration* Tab at the top of the editor window and expand the *Imported Prefabs* or *Imported Enums* section. From here you can import and remove any prefab/enum you want.

## Hot Reloading

To support editing the data while playing or support updates of the game data from a server while the client is running. The GameDB has a OnDBLoaded event that is triggered when data has be Imported or Reimported.

Generally data shouldn't be directly referenced from the gameDB due to overhead with runtime conversions of types (and loading prefabs for the prefab type). Best practice is to cache the values from the gameDB into your game classes for further access.

This means that when the OnDBLoaded event is triggered again you will need to likely re-cache any data you accessed as well as reloading any systems that rely on this data.

This is quite different to how serialised fields work but it allows consistent and deliberate work flows when dealing with data edited at run-time or live updating from a server for example.

## Settings

The GameDB editor creates a GameDBEditor.settings file in the same location as the plugin under the Editor folder. This file is a JSON file representing the settings that represent your workspace. This can be edited by hand in the event of settings causing any issue. This file contains the paths of the loaded gameDBs, the export path for C# classes and paths to any imported items, plus more.

# API Reference



# Generated Classes

## GameDB Class

The GameDB class is the top level class providing access to the data within your gameDB. It provides accessors for each table defined as well as various convenience methods for loading the data at runtime.

### Properties

> The GameDB class has public properties for each of the tables within your gameDB. The accessors will provide an instance of the table class (see below for more info) for you table. This table class then provides the ability to get a row or the entire table to iterate over
> ***@return {{TableType}}*** - Returns an instance of a table of type *TableType* ie. LootTable

***{{TableType}} {{TableName}}Table*** *(ie. public LootTable LootTable)*

> OnDBLoaded is a callback triggered whenever the gameDB has been loaded or imported. This includes from initial load/import, from reloading via the GameDB Editor Window or OTA* (Over The Air) updates
> * Pro version only

***Action OnDBLoaded***

> ScopeName is a string representing the Scope Name set when creating the gameDB
> ***@return string***

***string ScopeName***

> Name is a string representing the name given to the instantiated instance of this gameDB on construction
> ***@return string***

***string Name***

### Methods

> GameDB constructor
> ***@param string name*** - A string representing the name of this particular instance of the gameDB. Used to identify the gameDB in the GameDBEditor window during play mode.

***GameDB(string name)***

> Load (Free version) will take a path to a TextAsset in a Unity Resources folder and load the gameDB from that.
> ***(Free version only)***
> ***@param string path*** - A path to a TextAsset in a Unity Resources folder.
> ***@param bool notify*** - true/false for whether to trigger the OnDBLoaded callback. (defaults to true)
> ***@return Exception*** - returns an exception if the gameDB failed to load. null if successful.

***Exception Load(string path, bool notify = true)***

> Load (Pro version) will take a path to a TextAsset saved as .bytes in a Unity Resources folder and load the gameDB from that.

> ***(Pro version only)***
> ***@param string path*** - A path to a Binary TextAsset in a Unity Resources folder.
> ***@param bool notify*** - true/false for whether to trigger the OnDBLoaded callback. (defaults to true)
> ***@param bool binary*** - true/false for whether the gameDB to load is binary serialized. (defaults to false)
> ***@param string key*** - A string representing a strong key to use for decryption, it should be the same used to create the binary. gameDB. (defaults to null, needs to be set if binary == true)
> ***@param string salt*** - A string representing a strong salt to use for decryption, it should be the same used to create the binary gameDB. (defaults to null, needs to be set if binary == true)
> ***@return Exception*** - returns an exception if the gameDB failed to load. null if successful.

***Exception Load(string path, bool notify = true, bool binary = false, string key = null, string salt = null)***

> Import will take json string a load the gameDB from it.
> ***@param string jsonData*** - A string in json format.
> ***@param bool notify*** - true/false for whether to trigger the OnDBLoaded callback. (defaults to true)
> ***@return bool*** - true/false for whether the gameDB was loaded successfully.

***bool Import(string jsonData, bool notify = true)***

> ImportFromServer will check the gameDB server for an update to the gameDB and download and load that otherwise it will try to load a cached version of an earlier download gameDB or one packaged with the app. (More info here) TODO
> ***(Pro version only)***
> ***@param string serverHost*** - A string representing a URL to a gameDB server.
> ***@param string downloadHost*** - A string representing a URL to the download host to use.
> ***@param string gameDBPath*** - A path to a TextAsset in a Unity Resources folder.
> ***@param bool binary*** - true/false for whether the TextAsset is a binary gameDB.
> ***@param string tag*** - A string representing a tag to identify a particular iteration of a gameDB. Recommended value is a version string for the app or similar.
> ***@param string key*** - A string representing a strong key to use for decryption, it should be the same used to create the blob.
> ***@param string salt*** - A string representing a strong salt to use for decryption, it should be the same used to create the blob.
> ***@param string userID*** - (not in use) A string representing a userID used for A/B testing.
> ***@param Action onImport*** - a callback on import when the process is complete either returning null or an exception if something went wrong.
> ***@return RequestUpdater*** - returns a RequestUpdater that is used to pump the update loop for the underlying network connection. will return null if an error occurred before the network connection.

***RequestUpdater ImportFromServer(string serverHost, string downloadHost, string gameDBPath, bool binary, string tag, string key, string salt, string userID, Action onImport = null)***

## Table Class

A Table class is generated for every table in your gameDB. These generally are name {{TableName}}Table ie. LootTable. The Table class provides accessors for the rows in the table.

> GetByKey will take a key of the defined type such as a string and return an instances of the row (if found) matching that key.
> ***@param {{KeyType}} key*** - A key of the configured type, generally a string representing the key or table reference or an enum.
> ***@return {{RowType}}*** - returns an instance of the row or null if not found.

***{{RowType}} GetByKey({{KeyType}} key)*** (ie. public Loot GetByKey(string key))

> GetRows will return a Dictionary keyed by KeyType and with values representing each of the rows in the table.
> ***@return Dictionary*** - returns a dictionary representing the table

***Dictionary GetRows()*** (ie. public Dictionary GetRows())

## Row Class

A Row class is generated for each table in your gameDB. Named like {{TableName}} (ie. Loot) these provided accessors for every field contained within the table. A single instance of a Row class represents a single row in your table.

> The field val accessor will return a value of a particular field in your table of the type defined when creating the field. Fields of type string, int, bool, float & enum will have an accessor like this.
> ***@return {{Type}}*** - returns the value for a particular field in the row

***{{Type}} {{FieldName}}Val*** (ie. public Days DayVal)

> Field accessors for a Field of type *Table Reference* provide two different accessors to get the raw key value or the linked row instances. This accessor return the key to the linked table.
> ***@return {{KeyType}}*** - returns a value of KeyType allowing you to do GetByKey lookups on the associated table.

***{{KeyType}} {{FieldName}}KeyVal*** (ie. public string LootKeyVal)

> The second accessor for a Field of type *Table Reference* provides the actual instance of the linked row.
> ***@return {{RowType}}*** - returns an instance of the linked row in the associated table.

***{{RowType}} {{FieldName}}Val*** (ie. public Loot LootVal)

> Field accessors for a Field of type *Prefab* provide two different accessors to get the path value or a resource loaded instance of the prefab. This accessor provides the path to the asset in the resources folder.
> ***@return string*** - returns a string representing the path to use when doing Resource.Load().

***string {{FieldName}}PathVal*** (ie. public string AssetPathVal)

> The second accessor for a Field of type *Prefab* provides a Resource.Load()'ed UnityEngine.Object instance to the prefab.
> ***(Only available when generating classes for Unity)***
> ***@return UnityEngine.Object*** - returns an instance of the loaded prefab.

***UnityEngine.Object {{FieldName}}Val*** (ie. public UnityEngine.Object AssetVal)

# GameDBLibrary

## BinaryGameDB class

The BinaryGameDB class is provided as a utility for working with binary blobs (more info here) TODO. This is provided to allow advanced handling of your data, on top of other methods provided.
***(Pro version only)***

> Serialize will take some json and convert it to a secure binary blob.
> ***@param string json*** - A string to be converted to a binary blob.
> ***@param string key*** - A string representing a strong key to use for encryption, recommended length 16 or more characters.
> ***@param string salt*** - A string representing a strong salt to use for encryption, recommended length 16 or more characters.
> ***@return byte[]*** - A byte array of the raw bytes representing the binary blob.

**byte[] Serialize(string json, string key, string salt)**

> Deserialize will take a previously converted binary blob and convert it back to a json string.
> ***@param byte[] data*** - A byte array to be converted to string.
> ***@param string key*** - A string representing a strong key to use for decryption, it should be the same used to create the blob.
> ***@param string salt*** - A string representing a strong salt to use for decryption, it should be the same used to create the blob.
> ***@return string*** - A string converted from the binary blob.

**string Deserialize(byte[] data, string key, string salt)**

## JsonPatch Class

The JsonPath class is provided as a utility for working with json files in the JSON Patch format http://jsonpatch.com/. This will apply a patch to a json file in order to patch one json file in an attempt to transform it into another.
***(Pro version only)***

> Patch will take a json string and a JSON Patch formatted string and return the patched json
> ***@param string originalJson*** - A json string to patch.
> ***@param string patchJson*** - A json string in the JSON Patch format representing a diff between the two json files.
> ***@return string*** - A patched json string.

***string Patch(string originalJson, string patchJson)***

## Utils Class

The Utils class provides various helper methods for working with the gameDBs and servers if you choose not to use the helper methods (such as ImportFromServer) on the GameDB class.
***(Pro version only)***

> GetChecksum will return a checksum string of the supplied bytes. Useful for checking if the json has changed between that stored on the server.
> ***@param byte[] input*** - A byte array containing input data to create a checksum for.
> ***@return string*** - A checksum of the supplied byte array.

***string GetChecksum(byte[] input)***

> UrlCombine will combine to strings representing parts of a full URL, ie. "http://www.google.com" and "/testPath/test.html" which will result in "http://www.google.com/testPath/test.html".
>
> ***@param string url1*** - A string representing the beginning of the URL to join.
> ***@param string url2*** - A string representing the end of the URL to join.
> ***@return string*** - A string representing the two join paths with path separators handled correctly.

***string UrlCombine(string url1, string url2)***

# Google Sheets Import/Export

(Pro version only)

## Overview

The GameDB (Pro) plugin supports importing and exporting from/to google sheets via a self-hosted web app.

This functionality allows mainly for designers who may be used to this workflow to migrate to using the tool or allow analysis of the data after it has been exported, or can allow for migrating data that may already be stored in a spreadsheet into Unity and the plugin,

This is provided as a value added component and is not a recommended way of working with the plugin long term as certain future features may not be supported via Google Sheets Export/Import.

## Setup

Found in *Assets/Plugins/GameDBLibrary/GoogleSheets* is a Google App Script file called *GoogleSheetWebApp.gs*

This file will be imported into a Google App Script project that can be created by visiting https://www.google.com/script/start/ and clicking on the *Start Scripting* button. Copy the contents of the *GoogleSheetWebApp.gs* file into the main window (see screenshot below). Then give the project a name and save it from the file menu.

Before the script is usable it needs to be published as a wep app this can be done by select the *Publish* > *Deploy as a web app...* option from the menu. You will be asked to select who the web app operates under and who has access. You have to have it execute as a particular user as the plugin can't authenticate otherwise and you have to allow access to *Everyone, even anonymous* so be careful not to share your web app URL which you will get in the next dialog after clicking deploy.

You can then use the URL you obtained to enter into the plugin in the *Web App Url:* field when you click on the *Google Sheets* button.

## Usage

With the web app setup you can now export data from Unity into a spreadsheet or import it from one.

To specify the spreadsheet to export to/import from, create a spreadsheet in Google Drive or Google Sheets directly then take note of the spreadsheet id found by getting the long alpha-numeric string similar to the one hightlighted in this example:
https:/docs.google.com/spreadsheets/d/**e3jmJoPNlsumfsYjW7sGOv81taMDe3jmJoPNlsumfsYjW7sGOv81taMD**/edit

This is then entered in the *Sheet ID:* field of the dialog that is shown after clicking on the *Google Sheets* button.

## Notes

- To get the best results with importing make sure to export your table first, this will create the required sheet in your document and give you an idea of the format required for import
- A Sheet will be created for each table in your gameDB
- It is recommended to use a new sheet id for each gameDB you have