

# Table of Contents

<a href="#">Introduction</a>	1.1
<a href="#">Gettting Started</a>	1.2
<a href="#">Excel Howto</a>	1.2.1
<a href="#">Google Howto</a>	1.2.2
<a href="#">TroubleShooting</a>	1.2.3
<a href="#">Excel</a>	1.2.3.1
<a href="#">Google</a>	1.2.3.2
<a href="#">Usage</a>	1.3
<a href="#">Runtime Loading</a>	1.3.1
<a href="#">Using with LINQ</a>	1.3.2
<a href="#">Supported Cell Type</a>	1.4
<a href="#">Enum Type</a>	1.4.1
<a href="#">Array Type</a>	1.4.2
<a href="#">Formula Cell Type</a>	1.4.3
<a href="#">Case Study</a>	1.5
<a href="#">Formula Calculation: Part I</a>	1.5.1
<a href="#">Formula Calculation: Part II</a>	1.5.2
<a href="#">Formula Calculation: Part III</a>	1.5.3
<a href="#">Tips and Known Issues</a>	1.6
<a href="#">FAQ</a>	1.7
<a href="#">Appendix</a>	1.8
<a href="#">Goolge OAuth2 Setting</a>	1.8.1
<a href="#">Code Template</a>	1.8.2
<a href="#">Reference</a>	1.9

# Unity-QuickSheet

Unity-QuickSheet enables you to use google and excel spreadsheet data within Unity editor. With Unity-QuickSheet, you can retrieve data from a spreadsheet and save it as an asset file with a [ScriptableObject](#) format even without writing single line of code.

## Features

- **Easy!** No need to write any single line of code.
- **Convenient!** It can retrieve data from excel file. (both of xls and xlsx format are supported on Windows, only xls on OSX.)
- **Flexible!** It can retrieve data from google spreadsheet.
- **Fast!** No need to write a parser to retrieve data, it automatically serializes retrieved data into Unity3D's [ScriptableObject](#), the binary format and so it is fast than to use XML which is usually ASCII format.

Saying again, you don't need to write even single line of code!

## Getting Started

See the page '[Excel Howto](#)' page if you work with Excel's spreadsheet otherwise '[Google Spreadsheet Howto](#)' page to start with *Unity-Quicksheet*.

Also there is [FAQ](#) page for any other information.

You can also find the Unity forum page found on [here](#). If you have any question or suggestion, post it on the forum or leave it as an issue on the [github project page](#). Any feedbacks are welcome!

## Usage

You can use 'Unity-QuickSheet' anywhere you need to import spreadsheet data into Unity editor without pain. It is easy and fast.

- Use it as a database on client-side. It is easy and fast comparing with json or xml.
- Use it with LINQ. You can much easier handle data.
- Various type supported, not only a primitive type like int, float, but also enum and array.
- Localization.
- Automation of formula calculation. See the [Case Study](#) section.

## License

This code is distributed under the terms and conditions of the **MIT** license.

Some codes are borrowed from [GDataDB](#). The license of the that follow theirs.

Copyright (c)2013 Kim, Hyoun Woo

# How to work with Excel

This post shows and helps you how to set up and use [Unity-QuickSheet](#) with excel.

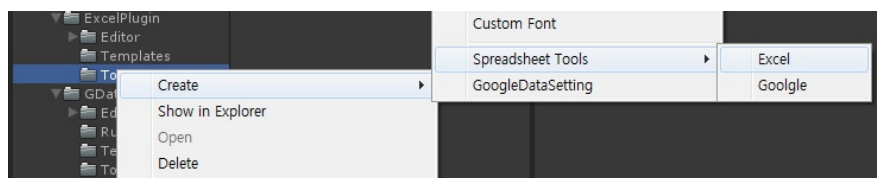
Excel versions 97/2000/XP/2003 are supported for '.xls' (due to that [NPOI](#), an open source project used to read excel spreadsheet for *Unity-QuickSheet* does not support Excel versions 5.0/95 for '.xls') It's doesn't matter with '.xlsx'

	A	B	C	D	E	F	G
1	MeleeSkillLevel	STR	DEX	INTL	TotalDamage	TotalArmor	Health
2	0	10	10	10	8	16	56
3	1	13	11	11	10	20	66.6
4	2	16	12	12	12	24	77.2
5	3	19	13	13	14	28	87.8
6	4	22	14	14	16	32	98.4
7	5	25	15	15	18	36	109
8	6	28	16	16	20	40	119.6
9	7	31	17	17	22	44	130.2
10	8	34	18	18	24	48	140.8
11	9	37	19	19	26	52	151.4
12	10	40	20	20	28	56	162
13	11	43	21	21	30	60	172.6
14	12	46	22	22	32	64	183.2
15	13	49	23	23	34	68	193.8
16	14	52	24	24	36	72	204.4

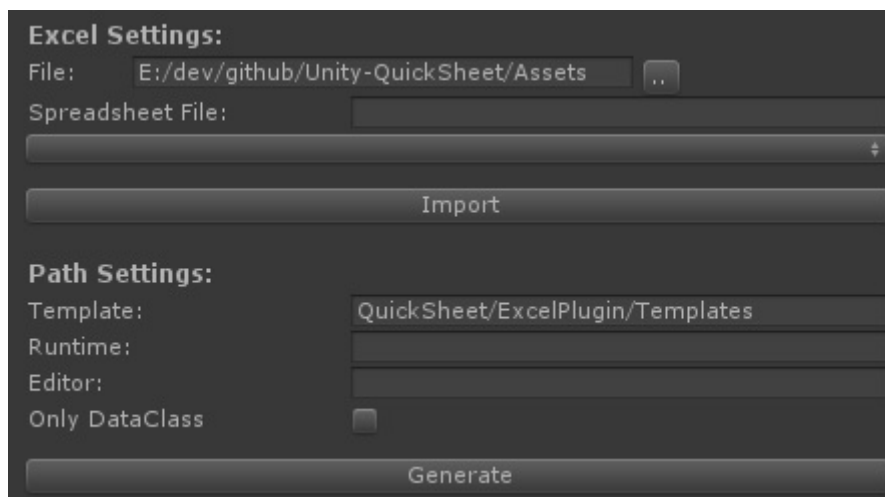
Before starting, check your spreadsheet page again. It should start without an empty row which means the first row should not be an empty one.

## Step 1) Create Excel Setting File

First you need thing to do is creating an excel setting file. Simply right click on the Project view and select '*Create > Spreadsheet Tools > Excel*'. It creates a new file which shows various setting to create script files and get data from the specified excel file.



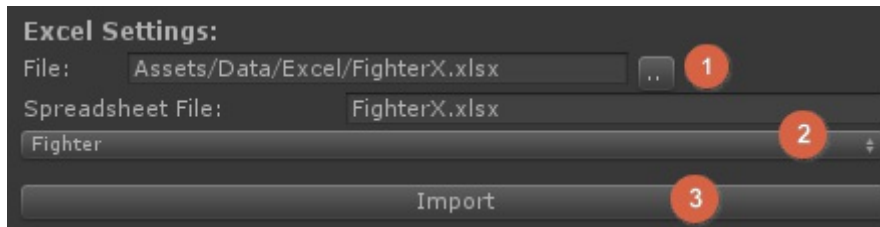
Select **Excel** menu item then it creates setting file. It may be shown like the following:



Let's start to do setting.

## File Setting

File setting is setting for what excel file and its sheet page to import.



First you should specify what excel file to import.

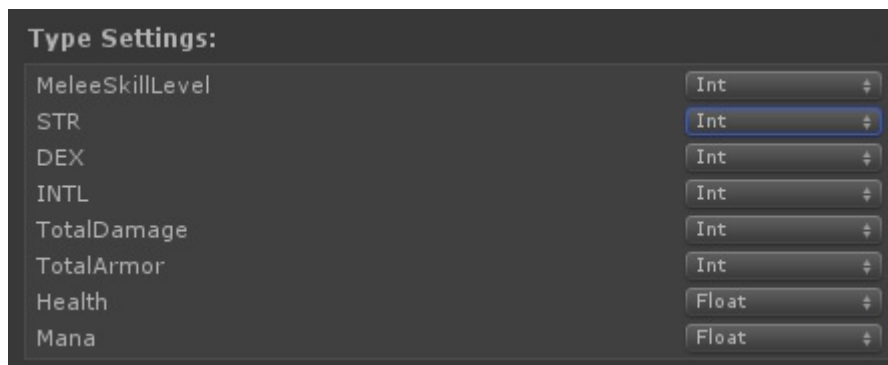
1. Select an excel file to import with **File Open Dialog**.

An excel file can have one or more sheet pages so you need to decide what sheet you select and retrieve data from.

1. Select a sheet page you want to import.
2. Press **Import** button imports the specified excel file and shows all column headers.

## Type Setting

Importing the specified sheet page shows all column headers of the page. That are necessary to let you set the type of the each cells.



Set the proper type of the cells.

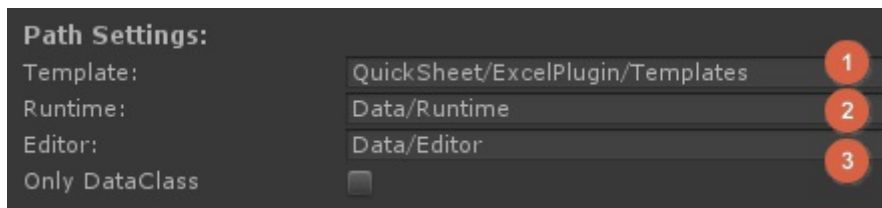
Currently the following types are supported:

- string
- int
- float
- double
- enum
- bool

## Path Setting

Path setting are concerned with specifying paths where the generated script files are put.

**Note:** All paths should be relative without 'Assets/'.



1. **Template** indicates a path where template files which are necessary to generate script files. In most case you don't need to change it.
2. **Runtime** indicates a path where generated script files which are used on runtime will be put.
3. **Editor** indicates a path where generated script files which are used on editor mode will be put.

## Step 2) Generating Script Files

If you've done all necessary setting, it's time to generate some script files which are needed for reading data in from the sheet page of the excel file and to store that within *ScriptableObject* which is being as an asset file in the *Project View*.

Press **Generate** button.

After generating some script files, Unity Editor starts to compile those. Wait till Unity ends doing compile then check the specified *Editor* and *Runtime* paths all necessary script files are correctly generated.

In **Editor** folder should have contain two files:

- *your-sheetpage-nameAssetPostProcessor.cs*
- *your-sheetpage-nameEditor.cs*

In **Runtime** folder should have contain tow files:

- *your-sheetpag-name.cs*
- *your-sheetpage-nameData.cs*

See the *your-sheetpage-nameData.cs* file. The class members of the file represent each cells of the sheet page.

```

using UnityEngine;
using System.Collections;

[System.Serializable]
public class FighterData
{
    [SerializeField]
    int meleeskilllevel;

    [ExposeProperty]
    public int Meleeskilllevel {
        get {return meleeskilllevel; }
        set { meleeskilllevel = value;}
    }

    [SerializeField]
    int str;

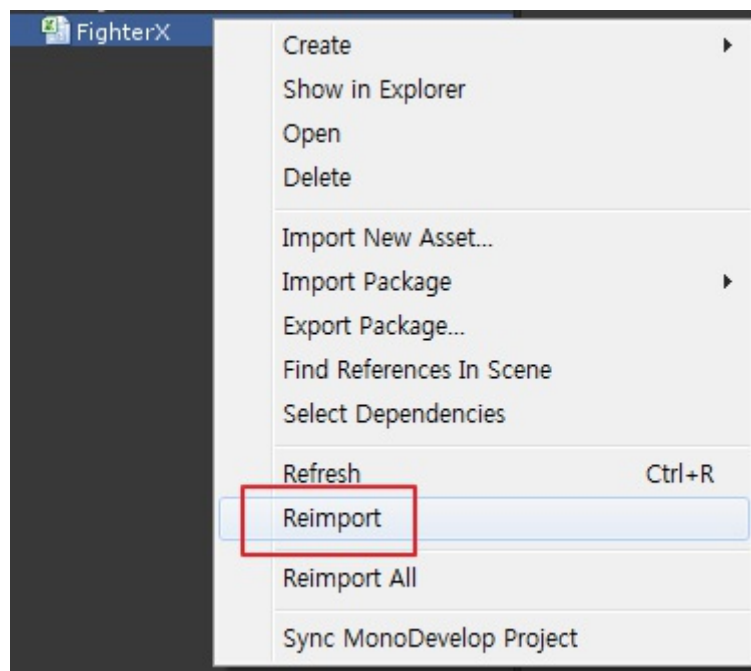
    [ExposeProperty]
    public int STR { get {return str; } set { str = value;} }

    ...
}

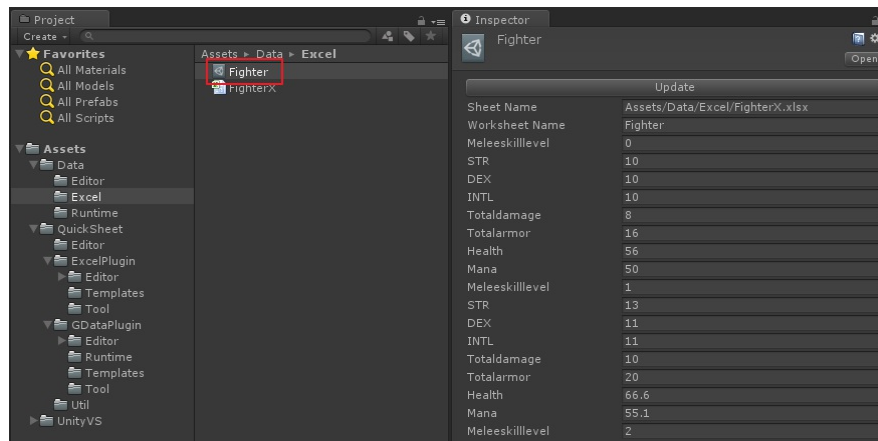
```

### Step 3) Importing Spreadsheet Data

Creating asset file and importing data from the spreadsheet file into that created asset file is done just by simply doing reimport any *xls* or *xlsx* file within Project view.



Reimporting *xls* or *xlsx* file will automatically create an asset file which has same file name as the sheet page name (*not excel file name*) and automatically import data from the sheet page of the excel file into the created asset file.



It's done. Hope you enjoy that!



# How to work with Google Spreadsheet

This page of the document describes and helps you to set up and use [Unity-QuickSheet](#) with 'Google Spreadsheet'.

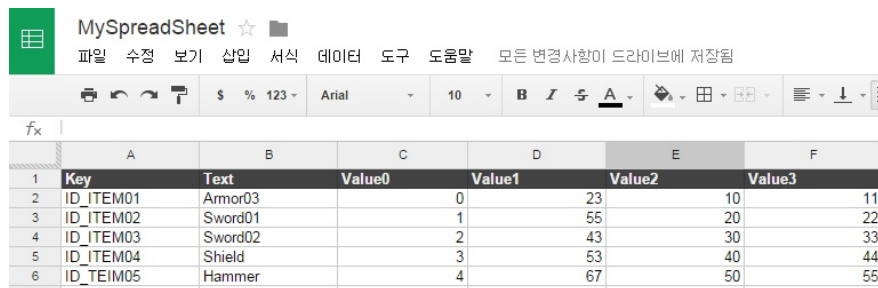
## Setting up OAuth2 for accessing Google Drive

Google has changed the authentication scheme since May 5, 2015. Now, to access a google spreadsheet you need to set up OAuth2. To set this up, visit [Google Developer Console](#) then create a new project, enable the Drive API, create a new client ID of type "service account" and download json file.

See [this page](#) for setting up credentials and getting OAuth2 '`client_ID`' and '`client_secret`' those are needed to set up google spreadsheet setting file.

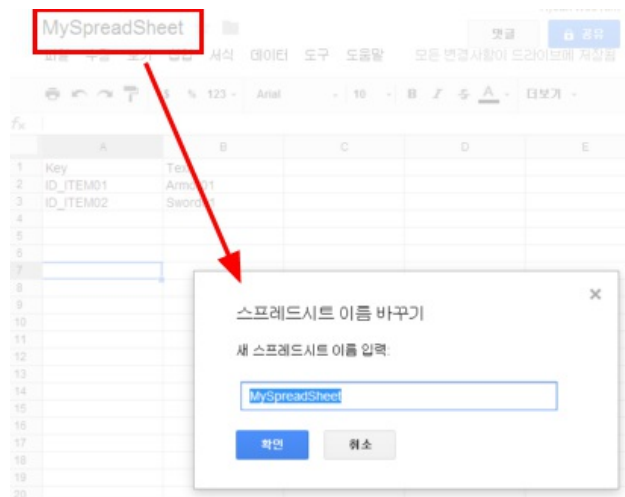
## Create a spreadsheet and worksheet

Create a google spreadsheet on your 'Google Drive' after logging in your 'Google Drive' with your account.

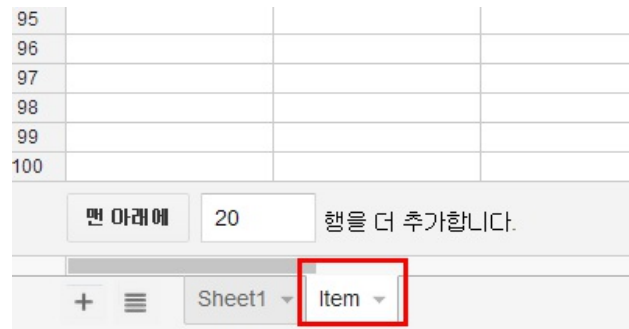


	A	B	C	D	E	F
1	Key	Text	Value0	Value1	Value2	Value3
2	ID_ITEM01	Armor03	0	23	10	11
3	ID_ITEM02	Sword01	1	55	20	22
4	ID_ITEM03	Sword02	2	43	30	33
5	ID_ITEM04	Shield	3	53	40	44
6	ID_ITEM05	Hammer	4	67	50	55

Change the title of the created spreadsheet as 'MySpreadSheet' like the following:



Next, create a new worksheet and rename it to whatever you want to as the following image shows:



Now, it needs to edit cells for spreadsheet. Insert 'Key' and 'Text' at the first row of the created worksheet as like that:

	A	B
1	Key	Text
2	ID_ITEM01	Armor01
3	ID_ITEM02	Sword01
4		
5		

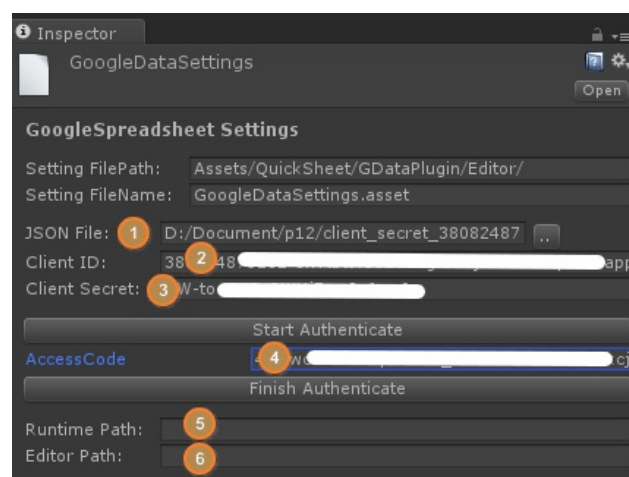
**IMPORTANT** Note that a name of cells on the first row will be used as member field's name of a generated class, you will see later on this page. So, be careful not to use such a name like 'int', 'string', which are keywords of C#.

## Google OAuth2 Service Account

Before further going, you need to create 'Google OAuth2 Account' to verify your account and make Unity available to access on your google spreadsheet.

If you not set 'OAuth2 Service Account' yet, see [OAuth2 setting on Google APIs](#) page found on this page. You must set that up first before importing data into Unity editor.

If you successfully get the json private key then select 'GoogleDataSettings.asset' file which can be found under 'Assets/QuickSheet/GDataPlugin/Editor' folder.

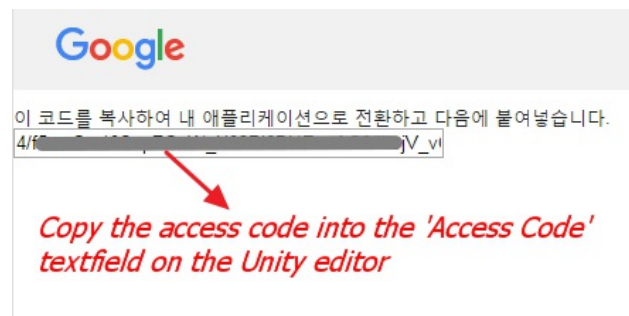


1. First, set the downloaded json private key to the '*JSON File*'.
2. Now, you can see 2)'*Client ID*' and 3)'*Client Secret*' will be automatically sepcified
3. Click '*Start Authentication*' button. It will launch your browser with the following image:



Select 'Allow' button then go to next.

Now you will see the '*access code*'. Copy it and paste to the Unity's 4)'*Access Code*' setting.



The final step is to click 'Finish Authenticate' button to verify your credentials.

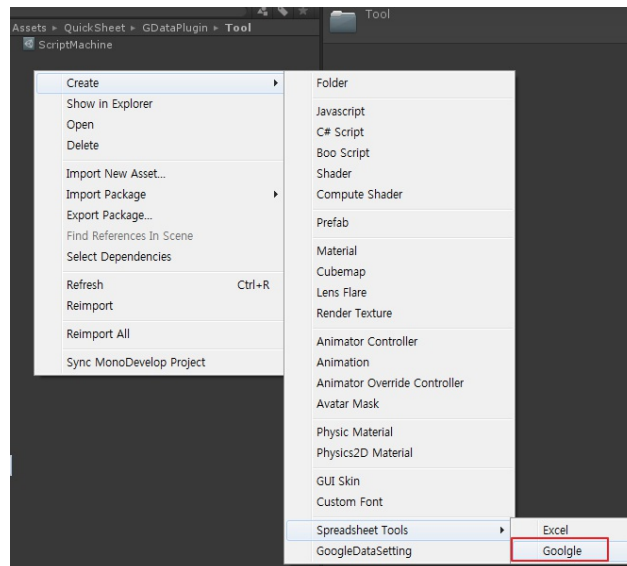
And set other settings like 'Runtime Path' and 'Editor Path' for your project. It may enough to set as the following:

```
Runtime Path: Data/Runtime
Editor Path: Data/Editor
```

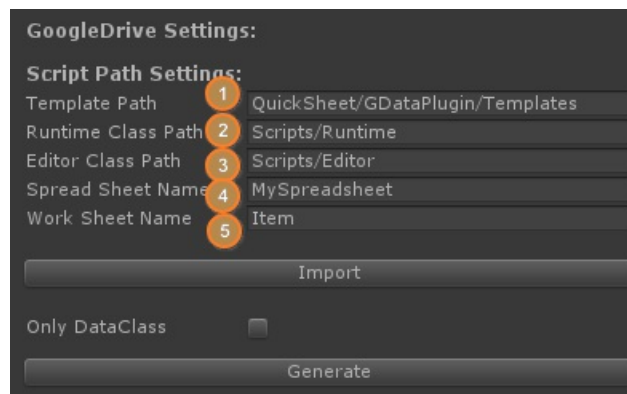
It assumes that the '*Data*' folder is under the '*Assets*' folder.

## Step 1) Creating Google Spreadsheet Setting File

First you need thing to do is creating a google spreadsheet setting file. Simply right click on the Project view and select '*Create > Spreadsheet Tools > Google*'. It creates a new file which shows various setting to create script files and get data from the specified google spreadsheet.



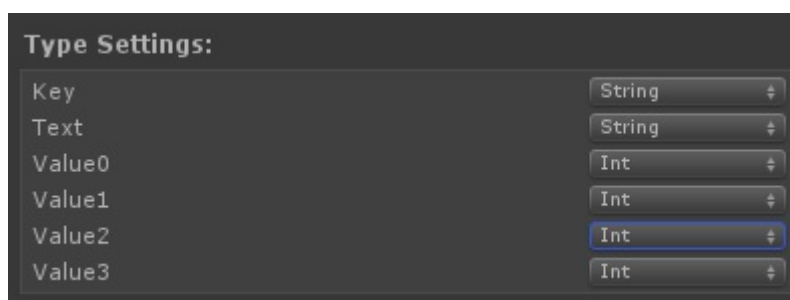
Select **Google** menu item then it creates setting file. It may be shown like the following:



## Script Path Setting

1. **Template** indicates a path where template files which are necessary to generate script files. In most case you don't need to change it.
2. **Runtime** indicates a path where generated script files which are used on runtime will be put.
3. **Editor** indicates a path where generated script files which are used on editor mode will be put.
4. **Spread Sheet Name** is what the name of the spreadsheet which is created on google drive.
5. **Work Sheet Name** is one of the worksheet name of the spreadsheet you want to get data from.

After doing done with all path setting, press **Import** button then it shows all column headers of the page. That are necessary to let you set the type of the each cells.



Set the proper type of the cells.

Currently the following types are supported:

- string
- int
- float
- double
- enum
- bool

## Step 2) Generating Script Files

If you've done all necessary setting, it's time to generate some script files which are needed for reading data in from the sheet page of the excel file and to store that within *ScriptableObject* which is being as an asset file in the *Project View*.

Press **Generate** button.

After generating some script files, Unity Editor starts to compile those. Wait till Unity ends doing compile then check the specified *Editor* and *Runtime* paths all necessary script files are correctly generated.

In **Editor** folder should have contain two files:

- *your-sheetpage-nameAssetCreator.cs*
- *your-sheetpage-nameEditor.cs*

In **Runtime** folder should have contain two files:

- *your-sheetpag-name.cs*
- *your-sheetpage-nameData.cs*

See the *your-sheetpage-nameData.cs* file. The class members of the file represent each cells of the sheet page.

```

using UnityEngine;
using System.Collections;

///
/// !!! Machine generated code !!!
/// !!! DO NOT CHANGE Tabs to Spaces !!!
///
[System.Serializable]
public class PlayerItemData
{
    [SerializeField]
    string key;

    public string Key { get {return key; } set { key = value; } }

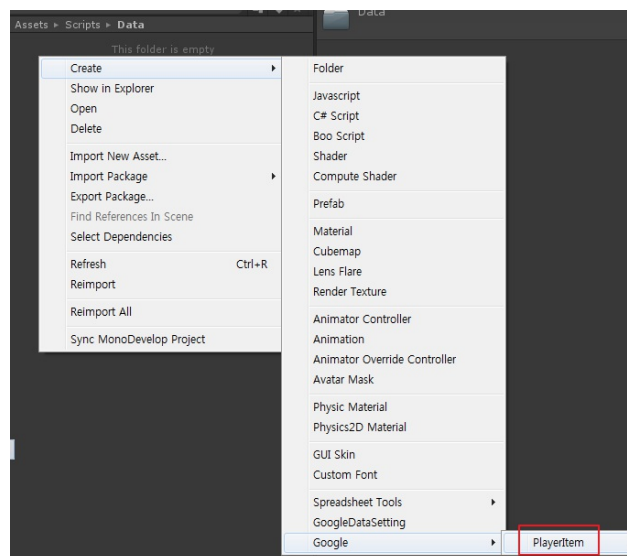
    [SerializeField]
    string text;

    public string Text { get {return text; } set { text = value; } }
}

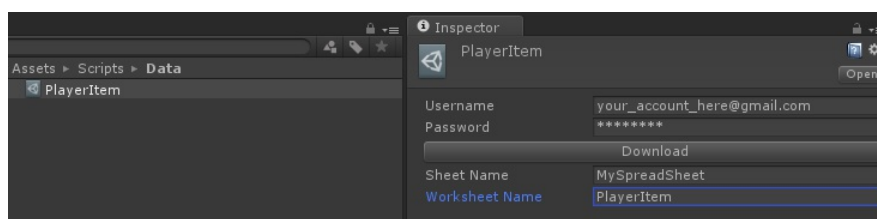
```

## Step 3) Importing Spreadsheet Data

Now you need to create an asset file which will imports and stores all data from google drive. Simply right click on the Project view and select '*Create > Google*'. Now there is a new menu item which has same name with the worksheet name of the google spreadsheet.



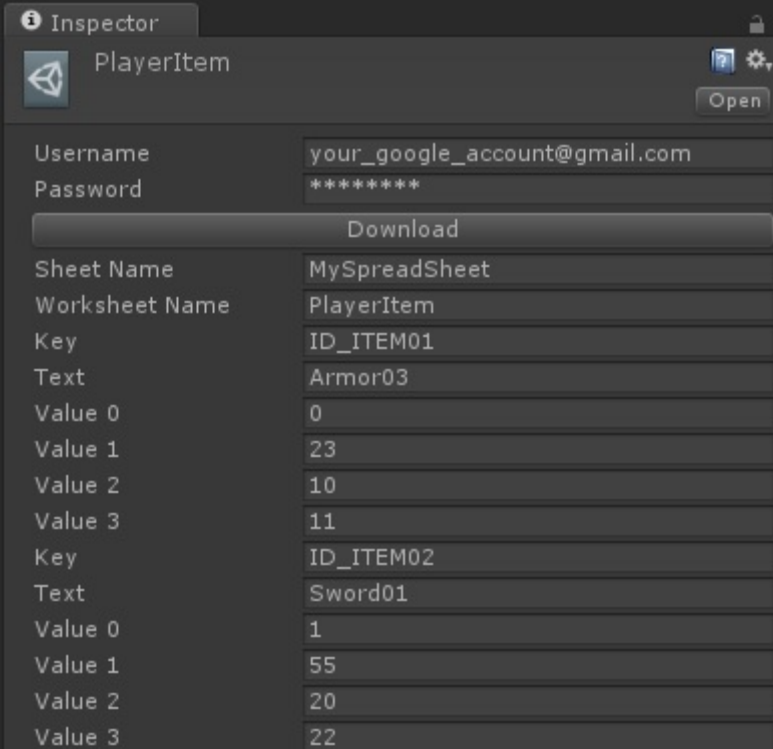
Select the menu item then it creates an asset file. It may be shown like the following:



Note that the created asset file has same file name as the specified worksheet name.

1. Type **Username** and **Password** of your google account.
2. Specify the same Spreadsheet and Worksheet name as the google setting.
3. Press **Download** button then it starts to import data from google drive. (It may takes a few seconds.)

Finished downloading shows the imported data on the Inspector View like the following:



The image shows a software interface titled "Inspector" with a sub-header "PlayerItem". It contains a form for importing data from Google Drive. The form has fields for "Username" (filled with "your\_google\_account@gmail.com") and "Password" (filled with "\*\*\*\*\*"). Below these is a "Download" button. Under the button, there are two sections of data. The first section has fields for "Sheet Name" (MySpreadSheet), "Worksheet Name" (PlayerItem), "Key" (ID\_ITEM01), "Text" (Armor03), and four "Value" fields (0, 23, 10, 11). The second section has fields for "Key" (ID\_ITEM02), "Text" (Sword01), and four "Value" fields (1, 55, 20, 22). There is also an "Open" button in the top right corner.

Field	Value
Username	your_google_account@gmail.com
Password	*****
<b>Download</b>	
Sheet Name	MySpreadSheet
Worksheet Name	PlayerItem
Key	ID_ITEM01
Text	Armor03
Value 0	0
Value 1	23
Value 2	10
Value 3	11
Key	ID_ITEM02
Text	Sword01
Value 0	1
Value 1	55
Value 2	20
Value 3	22

It's done. Hope you enjoy that!

If you encounter any troubles or problems see the [TroubleShoting page](#) on here.

# TroubleShooting

## Excel

### ***'Can't read content types part !' error on Mac***

On a Mac machine, opening .xlsx file cause an error *'Can't read content types part !'*.

Save it as '.xls' then open again. It solves the problem.

### **Supported version of '.xls' file**

Excel versions 97/2000/XP/2003 are supported for '.xls' (due to that [NPOI](#), an open source project used to read excel spreadsheet for *Unity-QuickSheet* does not support Excel versions 5.0/95 for '.xls') But you don't care about it for '.xlsx'.

### ***Excel Deserialize Exception exception error***

There can be an exception error on Unity console such as 'Excel Deserialize Exception: Object reference not set to an instance of an objectRow[5], Cell[6] Is that cell empty?' if a cell of a spreadsheet is empty.

Check the cell with the given row and cell index of the error. Then see the cell is defined as string type and not being empty. If the cell is empty fill a proper data.

## Google

### **Invalid Credential Error**

If you met an error which is shown as an invalid credentials when you try to get data by clicking 'download' button, check that your google account page and you have two-stage verification.

If you have Google two-stage verification on, then it doesn't matter what your Google password is, it won't be accepted. You need to generate (on Google) what is called an Application Specific Password (ASP). Go to [Google Account Page](#) and set up an ASP, enter the password you generate as the password in your code, and you're done.

### **Security Error**

Google Spreadsheet plugin does not work in the Unity web player's security sandbox. You should change the *Platform* to 'Stand Alone' or something else such as 'iOS' or 'Android' platform in the **Build Setting**.



# Runtime Loading

A way to load exported .asset scriptableObject file into a scene on runtime is nothing different as any other things in Unity3D.

Let's consider you have a simple spreadsheet as shown as the below image:

	A	B	C	D	E	F
1	Key	Text	Value0	Value1	Value2	Value3
2	ID_ITEM01	Armor03	0	23	10	11
3	ID_ITEM02	Sword01	1	55	20	22
4	ID_ITEM03	Sword02	2	43	30	33
5	ID_ITEM04	Shield	3	53	40	44
6	ID_ITEM05	Hammer	4	67	50	55

And you've already exported it as .asset scriptableObject.

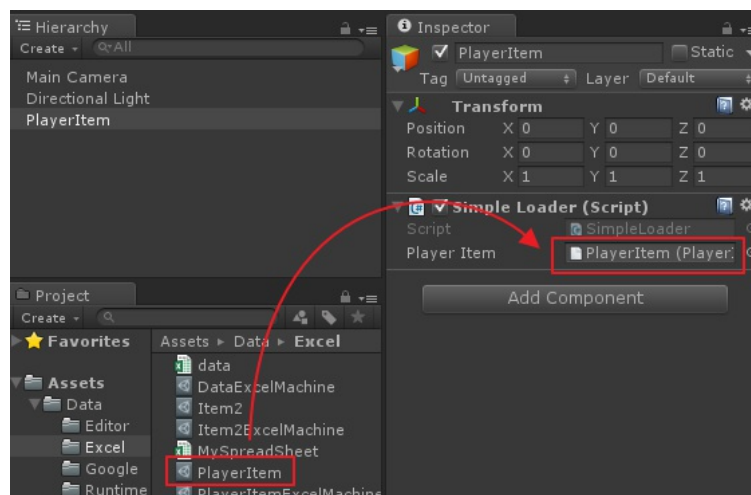
Things to do to load .asset scriptableObject is same as what for a prefab file.

First, create a new MonoBehaviour script file then declare `PlayerItem` class instance as public.

```
public class SimpleLoader : MonoBehaviour
{
    // You've already have PlayerItem class
    public PlayerItem playerItem;

    void Start ()
    {
        // PlayerItem.dataArray array contains all exported data.
        // It will put out Value0 value, 0 onto the console.
        Debug.LogFormat("Value0: {0}", playerItem.dataArray[0].Value0);
    }
}
```

Now, just drag .asset file and drop it into the Inspector view.



That's all what to load .asset file on runtime. Now start to play and the value of 'Valaue0' will be put in the console.

# Using LINQ

This section of the document describes how it can be easily done with **LINQ**(*Language Integrated Query*) to query specific data set which is exported from complex spreadsheet.

Highly recommended to see [101-LINQ Samples](#) if you not familiar with LINQ yet. Also there are plenty of documents for LINQ so just do googling for that.

Ok, let's start.

As you already know spreadsheet like *Excel* is one of the most powerful tool what game designers daily use to make various tables and data set used within game.

The following image shows a table data for all weapon items in a game. Note that the shown columns are not all of columns in the table, there can be much more for usual game especially if the game is RPG genre.

	A	B	C	D	E	F	G	H	I	Q	R	S	T	U	V
1	ID	Name	STR	DEX	INT	FTH	Uni	Ct	Ph	S	Dx			WeaponType	DamageType
2	1000	Battle Axe	12	8	0	0	4	100	250	C	D			Axe	Standard
3	1001	Hand Axe	9	8	0	0	2.5	100	220	C	D			Axe	Standard
4	1002	Thrall Axe	8	8	0	0	1.5	100	208	C	C			Axe	Standard
5	1003	Dragonslayer's Axe	18	14	0	0	4	100	180	D	D			Axe	Standard
6	1004	Butcher Knife	24		0	0	7	100	190	S				Axe	Slash
7	1005	Brigand Axe	14	8	0	0	3	100	248	C	D			Axe	Standard
8	1006	Winged Knight Twinaxes	20	12	0	0	8.5	100	244	C	D			Axe	Standard
9	1007	Elonora	20	8	0	0	6.5	100	284	D	D			Axe	Standard
10	1008	Man Serpent Hatchet	16	13	0	0	4	100	250	C	D			Axe	Standard
11	1009	Short Bow	7	12	0	0	2	100	154	E	D			Bow	Projectile

No matter how complicated the table of the given spreadsheet is, it is easy to get the table into Unity editor via *Unity-Quicksheet*. So assume that importing the table into Unity was successfully done.

Now we have automatically generated *WeaponTable.cs* script file which shows the class as the following:

```
public class WeaponData
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int STR { get; set; }
    ...
}
```

And there is also a *ScriptableObject* derived class which is for *.asset* file.

```
public class WeaponTable : ScriptableObject
{
    ...
    public WeaponData[] dataArray;
    ...
}
```

To make it easy to manage items, all items in the spreadsheet has its unique ID(see *the first column*) which is used to indentify what the item type is with its range.

Type	Range
Weapon	1000 ~ 1999
Armor	2000 ~ 2999
Consumable	3000 ~ 3999

The range can be various depends on its number of item type or each number of item itself.

So just with having integer array of the item ID is enough to describe what items are in the inventory.

```
public class Inventory
{
    ...
    // currently belonging items in the inventory
    int[] items = {
        1000, 1001, 1002, 1003, 2002, 2003,
        2004, 2005, 2006, 2007, 2008, 2009,
        2010, 2011, 2012, 2013, 2014, 2015,
        2016, 2017, 2018, 4014, 4015, 4016,
        4017, 4018, 4019, 4020, 4021, 4022,
        4023, 4024, 4025, 4026, 4027, 4028, 4029,
        4030, 4031, 4032, 4033, 4034, 4035, 4036
    };
    ...
}
```

And then we need to retrieve actual item data we have in the inventory which can be done to query to the item table.

`ItemManager` is a class which provides methods to handle and access various items in the inventory. Note `weapons` member field as an intance which is automatically created from Unity-Quicksheet.

```

public class ItemManager : Singleton<ItemManager>
{
    public WeaponTable weaponTable;
    ...
    // weapon IDN: 1000~1999
    public List<weaponData> GetWeaponList(int[] ids)
    {
        List<weaponData> items = new List<weaponData>();

        var weaponIDs = ids.Where(x => x >= 1000 && x < 2000).ToList();
        var weaponResult = from w in weaponTable.dataArray // all weapon type items from spreadsheet
        .
            from n in weaponIDs // weapons in the inventory.
            where w.ID == n
            select w;
        return weaponResult.ToList();
    }
}

```

As seen from the above code, retrieving items of weapon type can be easily done with using LINQ as seen as `GetWeaponList` method.

Consider our tables in a spreadsheet as a kind of database. Then extracting and importing data into Unity with Unity-Quicksheet at edit-time and prefer to use LINQ to query data from the table at runtime. No matter how complicated the table is, it is a piece of cake.

# Using ENUM Type within Quicksheet

Specify enum type for a data class is easy. Let's say that you want to set enum type for '*RareType*' on a spreadsheet as the following:

	A	B	D	E	F
1	Id	Id	SkillType	RareType	Card
2	1	Smack2	Active	Normal	2
3	2	Strike2	Passive	Rare	2
4	3	Smack3	Active	Legend	3
5	4	Strike3	Passive	Normal	3

The '*RareType*' only can have one of value from three type which are *Normal*, *Rare* and *Legend*.

Because QuickSheet can not generate enum itself, you should first declare an enum type before generating script files.

Create an empty .cs file which usually contains various enums then declare '*RareType*' enum type what for you've set on the spreadsheet.

```
public enum RareType
{
    Normal,
    Rare,
    Legend,
}
```

Now you can generate necessary script files without an error!

# Using Array type with QuickSheet

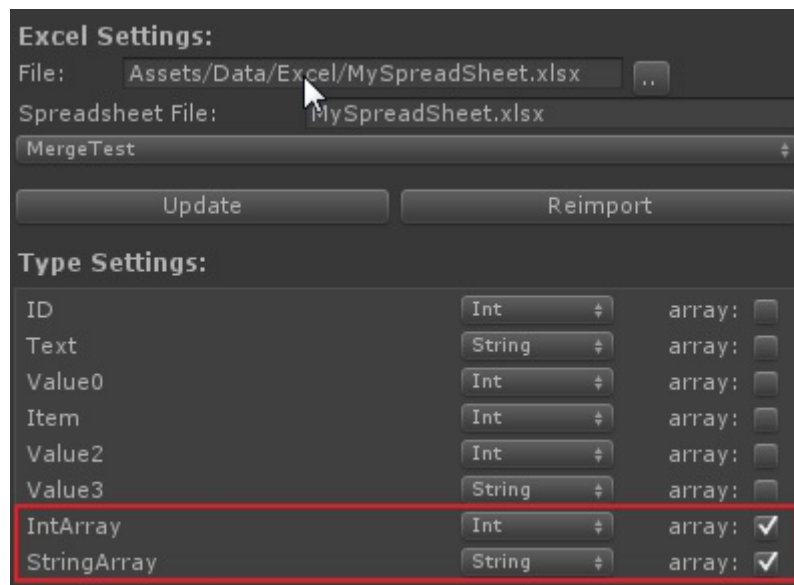
You can use array type with comma sperated values in a cell as the following:

	A	B	C	D	E	F	G	H
1	ID	Text	Value0	Item	Value2	Value3	IntArray	StringArray
2	1	Armor03	0	23	10	a	1,2,3,4,5,	a,b,c,d,e,
3	2	Sword01	1	55	20	b	1,2,3,4,5,	a,b,c,d,e,
4	3	Sword02	2	43	30	c	1,2,3,4,5,	a,b,c,d,e,
5	4	Shield	3	53	40	d	1,2,3,4,5,	a,b,c,d,e,
6	5	Hammer	4	67	50	e	1,2,3,4,5,	a,b,c,d,e,
7								

Note that don't miss the last comma which should be after a last value in a cell. (Changed on v.1.0.0.0)

1,2,3,4,5, -> The comma right after '5' is not needed anymore after v.1.0.0.0.

After importing with the given excel file, specify type of each column-header and check *array* option for an array type.



It will generate array type memeber field of a data class with the specified type as the following code:

```
[SerializeField]
int[] intarray = new int[0];

public int[] Intarray { get {return intarray; } set { intarray = value; } }

[SerializeField]
string[] stringarray = new string[0];

public string[] Stringarray { get {return stringarray; } set { stringarray = value; } }
```

NOTE: enum type array is only supported for google spreadsheet not for excel yet.

# Formula Celltype

You can even formula cell type on your spreadsheet.

C2	:				=AbitiliesBase[@cooldown]/Metadata[Pacing]
	A	B	C	D	
1	AbilityType	castTime	cooldown	power	
2	Heal	0.666666667	6.666666667	5	
3	Poke	0	2.666666667	15	
4	Barr	1.333333333	4	0	
5	Nuke	6.666666667	40	6	
6	Stun	0	6.666666667	15	
7	Swap	0	6.666666667	0	
8	Copy	0	0	0	
9	Taunt	0	13.33333333	0	
10	Buff	0	6.666666667	2	

Use any formula to calculate value on the cells as usual way you do on a spreadsheet. Unity-Quicksheet also serializes that without difference as any other cell types such as numeric or string.

NOTE: only availale on Excel at the moment. (v.1.0.0.1)



# Automation of Formula Calculation

Characters in a RPG normally have various stats such as HP, MP, level etc. and those are usually calculated via various given formulas. For an example, power of a character is increased as they level up and this can be represented with a form of formula.

But there is a problem how the formulas are dealt with. You can write the formulas within code. Easy but it has a problem that it should be rewritten everytime whenever the formula is changed. It is not a surprise thing a game designer changes that again and again over the whole development time.

So we need more flexible way to deal with the formulas as possible as a form of data not a code piece.

This section of the document describes how the formula calculations can be easily done with a spreadsheet and Unity-Quicksheet plugin by fully data-driven way.

Before further reading, if you're not familiar with typical stats and formulas of a RPG, highly recommended to see the article, [The craft of game systems: Practical examples](#) first.

## Formulas on Spreadsheet

First thing to do is defining various stats and formulas on a spreadsheet, so it can make to be used in the game .

You can use either of Excel or Google Spreadsheet whatever you prefer but the description on this document is based on Excel spreadsheet.

Let's consider that a character, in the game you try to make, increases in power as he levels up. He has three basic stats to describe his power. Each of that are strength(STR), dexterity(DEX) and intelligence(ITL). As his stats increase, he get more health(HP) and mana(MP).

The following image shows each of stat and its corresponding formula on the spreadsheet.

	A	B
1	Stat	Formula
2	STR	Round( (5 * (SkillLevel * 0.6)), 0) + 10
3	DEX	Round( (5 * (SkillLevel * 0.2)), 0) + 10
4	ITL	Round( (5 * (SkillLevel * 0.2)), 0) + 10
5	HP	(5 * ((STR * 0.5) + (DEX * 0.39) + (ITL * 0.23)))
6	MP	(5*((STR * 0.01) + (DEX * 0.12)+(ITL * 0.87)))

The data on the spreadsheet will be used for the example on this document.

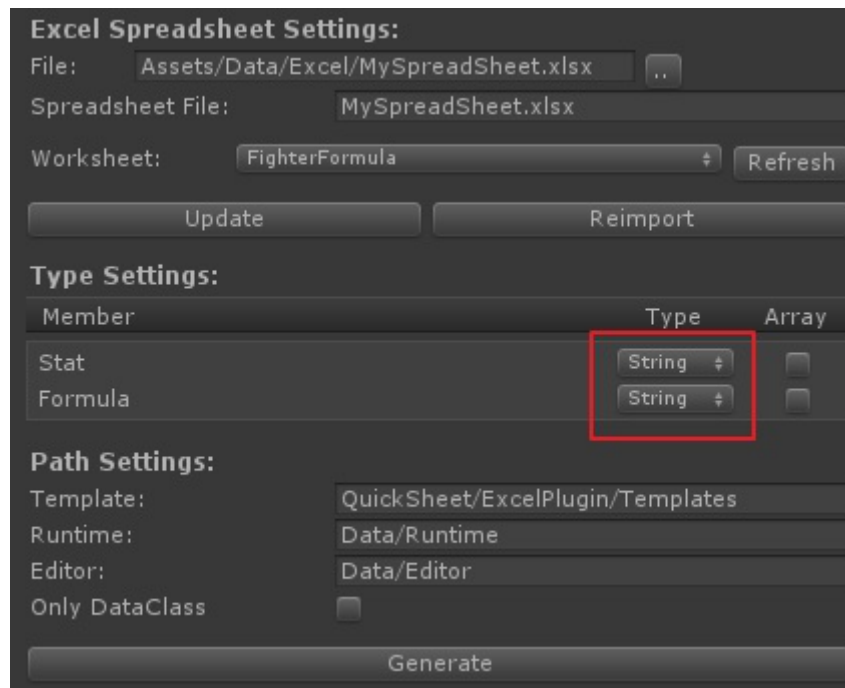
Note that *Stat* and *Formula* of the above image are borrowed from the article, [The craft of game systems: Practical examples](#).

# Data Importing

First, let's import all data in the worksheet into Unity editor.

Create import setting asset file and import the file by specifying the spreadsheet. (See the [How to work with Excel](#) page for the detail steps.)

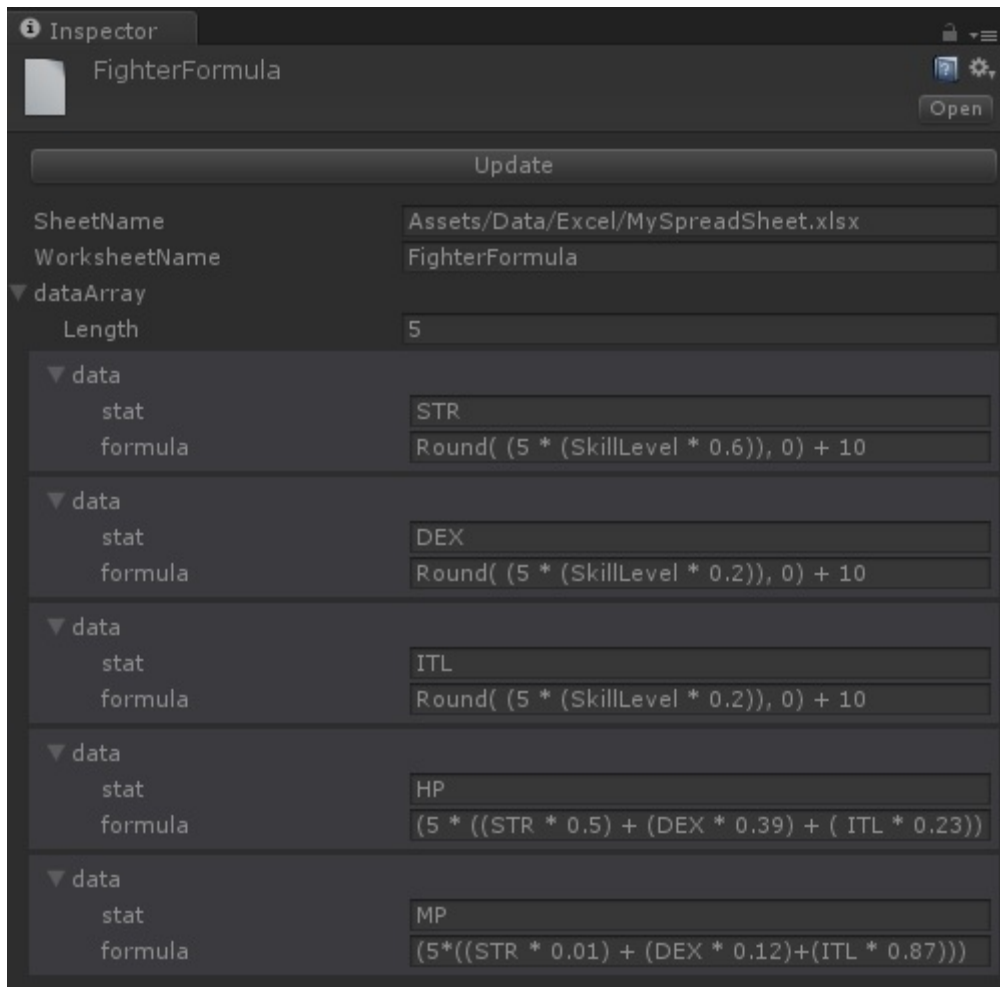
Set both type of `Stat` and `Formula` in the 'Type Settings' as *string* type. `Stat` is just name of each stat which are belong to a player so set its type as string seems to be ok but why `Formula` is also set as string? The answer will be found later. At the moment, just think it again and you will get to know that there is no other proper type for the formula else than string.



Click `Generate` button which will generate all necessary script files under the directories specified on the '*Editor*' and '*Runtime*' textfield.

Then select .xlsx Excel file in the Project window and select `Reimport` menu shown by right-click of the selected excel file.

As reimporting the excel file, it creates '*ScriptableObject*' as an asset file with the name of worksheet and imports all data into it. Select the created asset file and check the imported data is correct. You may see the following image on the Inspector view of Unity editor:



You can see the generated code, especially *FighterFormuladata* class which can be found under the directory specified `Runtime` setting.

```
[System.Serializable]
public class FighterFormulaData
{
    [SerializeField]
    string stat;
    public string Stat { get {return stat; } set { stat = value;} }

    [SerializeField]
    string formula;
    public string Formula { get {return formula; } set { formula = value;} }
}
```

If you're not familiar to import data from a spreadsheet, see [How to work with Excel](#) or [How to work with Google Spreadsheet](#) page depends on the type of your spreadsheet.

## Writing Calculation Code

Now, let's write down code to calculate each of stat with the given formula imported from the spreadsheet.

First of all, we need a class to represent all stats of a fighter type of player in the game. The following simple POCO class, `PlayerStat` can be used for that. It is nothing else but has all stat which are shown on the spreadsheet as properties except `SkillLevel` property.

```
public class PlayerStat
{
    public int SkillLevel { get; set; }
    public float STR { get; set; }
    public float DEX { get; set; }
    public float ITL { get; set; }
    public float HP { get; set; }
    public float MP { get; set; }
}
```

Next thing, we need to do is doing calculation each of stat with its correspond formula and set the result back to each of stat.

Typically, a formula of a RPG is just a polynomial and the imported formulas of ours are string data. So we need a calculator can calculate a polynomial as form of string data.

And so, how do we can calculate the formula as form of string data?

Though it can be done with writing a simple calculator but luckily there are already bunch of calculators written by C# exist. A matter of course, we don't need to reinvent the wheel.

One of notable thing from those is [A Calculation Engine for .NET](#) which is simple to use and already has most functionalities for our formula calculation.

Also there is [Zirpl CalcEngine](#), a port of [A Calculation Engine for .NET](#) to a portable .NET library(PCL) which can be found on [this github project page](#).

Let's see how *CalcEngine* does calculate with the given formula as string type.

```
CalcEngine.CalcEngine calculator = new CalcEngine.CalcEngine();
calculator.Variables["a"] = 1;

// The result ouput 2 as you expect
var result = calculator.Evaluate("a + 1");
```

As shown on the above just specifying any variable of the formula before evaluating then calling *Evaluate* is all of things to do to get the result. Simple enough?

Create a script file as '*Player.cs*' derives *MonoBehaviour* class for a component of Unity's gameobject.

```

public class Player : MonoBehaviour
{
    // ScriptableObject contains imported data from the spreadsheet.
    public FighterFormula fighterFormula;

    // A class instance for persistence data
    private PlayerStatus playerStatus = new PlayerStatus();
    ...
}

```

Now it is time to calculate each of stat with the corresponding formula.

One of the convenient and powerful feature of *CalcEngine* is '*DataContext*', a binding mechanism which connects .NET object to the *CalcEngine*'s evaluation context.

Instead of specifying each of variables to *CalcEngine*, just by setting '*PlayerStatus*' class instance to the *CalcEngine*'s *DataContext*, it is ready to do calculation.

As explained before, calling *CalcEngine*'s '*Evaluate*' function calculates the given formula and returns corresponding stat value. Whatever the formula it is, there is no difference to calculate it and get the result.

```

void Start ()
{
    // Specify skill level
    playerStatus.SkillLevel = 4;

    CalcEngine.CalcEngine calculator = new CalcEngine.CalcEngine();

    // CalcEngine uses Reflection to access the properties of the PlayderData object
    // so they can be used in expressions.
    calculator.DataContext = playerStatus;

    // Calculate each of stat for a player
    playerStatus.STR = Convert.ToSingle(calculator.Evaluate(GetFormula("STR")));
    Debug.LogFormat("STR: {0}", playerStatus.STR);

    playerStatus.DEX = Convert.ToSingle(calculator.Evaluate(GetFormula("DEX")));
    Debug.LogFormat("DEX: {0}", playerStatus.DEX);

    playerStatus.ITL = Convert.ToSingle(calculator.Evaluate(GetFormula("ITL")));
    Debug.LogFormat("ITL: {0}", playerStatus.ITL);

    playerStatus.HP = Convert.ToSingle(calculator.Evaluate(GetFormula("HP")));
    Debug.LogFormat("HP: {0}", playerStatus.HP);

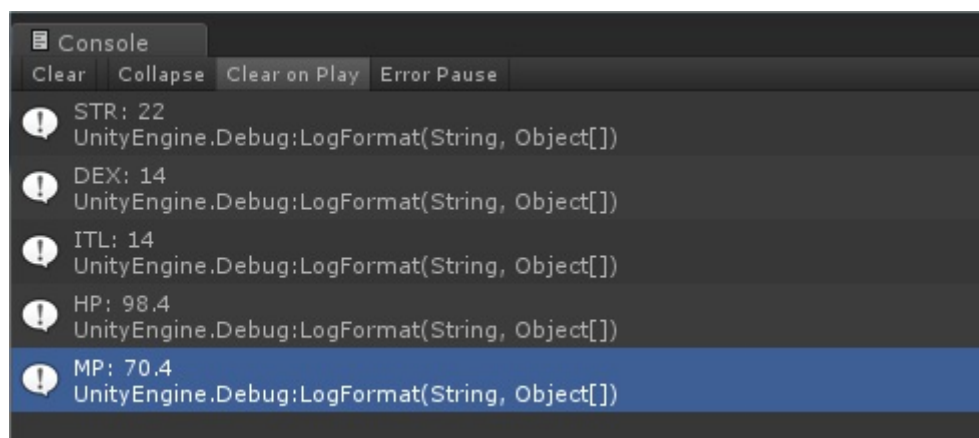
    playerStatus.MP = Convert.ToSingle(calculator.Evaluate(GetFormula("MP")));
    Debug.LogFormat("MP: {0}", playerStatus.MP);
}

// A helper function to retrieve formula data with the given formula name.
string GetFormula(string formulaName)
{
    return fighterFormula.dataArray.Where(e => e.Stat == formulaName)
        .FirstOrDefault().Formula;
}
}

```

That's all. Even you change any formula on the spreadsheet, you don't have to change code. Just by updating imported scriptableObject .asset file, and the last of things are done on run-time without any code side change.

Now, run Unity editor and you get the following result on the console:



Compare the result of calculated stats on the above with the originals on the spreadsheet found at the [Dungeon Siege 2 System Spreadsheet](#) page. As you can see, it has the same result.

	A	B	C	D	E	F
1	<b>FIGHTER STATISTICS</b>					
2	<b>Melee Skill</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
3	STR	10	13	16	19	22
4	DEX	10	11	12	13	14
5	INT	10	11	12	13	14
6	Total Damage	8.0	10.0	12.0	14.0	16.0
7	Total Armor	16	20	24	28	32
8	Health	56.0	66.6	77.2	87.8	98.4
9	Mana	50.0	55.1	60.2	65.3	70.4

In the development cycle, as tuning a game play balance, a designer try to change formulas over several times. Though it is natural process but you do not want to change code each time of formula changes.

As shown on the document, this approach can avoid to change code whenever a designer changes the formulas so it can make not only huge time saving but also error prevention.

# Automation of Formula Calculation: Part II

On the previous document, each of stat is separately calculated. It is tedious work. Even worse, we should calculate and set over again whenever a new stat is added to the PlayerStatus class.

```
playerStatus.SkillLevel = 4;

calculator = new CalcEngine.CalcEngine();
calculator.DataContext = playerStatus;

playerStatus.STR = Convert.ToSingle(calculator.Evaluate(GetFormula("STR")));
playerStatus.DEX = Convert.ToSingle(calculator.Evaluate(GetFormula("DEX")));
playerStatus.ITL = Convert.ToSingle(calculator.Evaluate(GetFormula("ITL")));
playerStatus.HP = Convert.ToSingle(calculator.Evaluate(GetFormula("HP")));
playerStatus.MP = Convert.ToSingle(calculator.Evaluate(GetFormula("MP")));
```

For an example, consider that if we add a new stat, 'LUK' which is used for lucky stat for a character we should write its calculation code again like the following code.

```
playerStatus.MP = Convert.ToSingle(calculator.Evaluate(GetFormula("LUK")));
```

It may not a big deal but, yes, tedious.

If so, how does it look like if we can done all calculations at once like the following code?

FormulaEnhanced.cs

```
playerStatus = new PlayerStatus();
playerStatus.SkillLevel = 4;

// BOOM!
formulaCalculator = new FormulaCalculator();
formulaCalculator.Calculate<PlayerStatus>(playerStatus, fighterFormula.dataArray);
```

Doesn't it look charm? It is not only simple but also even flexible. Ok, let's see how it can be possible.

The first thing to do are extracting formulas and resolving its correspond stat. It can be processed by calling `FormulaCalculator.GetFormulaTable` method. Consider that it creates a dictionary with the given 'FighterFormulaData' array which is imported from the spreadsheet and being in the 'FighterFormula' class, a *ScriptableObject* derived class for its first parameter. The second parameter is 'stat' name and the last is 'formula' itself as a string type.

FormulaCalculator.cs



```

public void Calculate<T>(System.Object data, System.Object[] obj, string key, string value)
{
    // obj: FighterFormulaData array
    // key: Stat property of FighterFormulaData class
    // value: Formula property of FighterFormulaData class
    Dictionary<string, string> formulaTable = GetFormulaTable(obj, key, value);
    Calculate<T>(data, formulaTable);
}

```

By the way, there is a problem that we should calculate only the properties which are needed for the formulas. See 'PlayerStatus' class. Not all of the properties are used as variables for the formula calculation. The `SkillLevel` property is not needed one for that purpose. So we need a way to distinguish the properties which are used for the variables and which are not. But how?

It can be solved by setting a custom attribute on any property which is used formula variable. The `FormulaVariable` attribute which is shown at the following code is used for that purpose.

PlayerStatus.cs

```

public class PlayerStatus
{
    public int SkillLevel { get; set; }

    // A stat member field which has 'FormulaVariable' is automatically recognized
    // as formula variable.
    [FormulaVariable]
    public float STR { get; set; }
    [FormulaVariable]
    public float DEX { get; set; }
    [FormulaVariable]
    public float ITL { get; set; }
    [FormulaVariable]
    public float HP { get; set; }
    [FormulaVariable]
    public float MP { get; set; }
}

```

'FormulaVariableAttribute' is a simple attribute of C#.

FormulaVariableAttribute.cs

```

using System;

[AttributeUsage(AttributeTargets.Property)]
public class FormulaVariableAttribute : Attribute
{
}

```

Before doing calculate the given formula, *GetFormulaProperties* method gathers all properties which have the 'FormulaVariable' attribute from 'PlayerStatus' class. With reflection mechanism, we can easily do that.

## FormulaCalculator.cs

```
private PropertyInfo[] GetFormulaProperties<T>()
{
    var _type = typeof(T);

    List<PropertyInfo> formulaPropList = new List<PropertyInfo>();

    // Reflection. Get all properties of the given class T.
    PropertyInfo[] properties = _type.GetProperties(BindingFlags.Public | BindingFlags.Instance)
;

    foreach (PropertyInfo p in properties)
    {
        if (!(p.CanRead && p.CanWrite))
            continue;

        object[] attributes = p.GetCustomAttributes(true);
        foreach (object o in attributes)
        {
            // We get a property whcih has 'FormulaVariable' custom attribute.
            if (o.GetType() == typeof(FormulaVariableAttribute))
                formulaPropList.Add(p);
        }
    }
    return formulaPropList.ToArray();
}
```

Now, it is time to do actual calculation. We do it by calling *'Calculate'* a generic method, so it can do calculation on any type of class, not just only for *'PlayerStatus'* class.

## FormulaCalculator.cs

```

public void Calculate<T>(System.Object dataInstance, Dictionary<string, string> formulaTable)
{
    var _type = typeof(T);

    // Tell CalcEngine the value of variables.
    calcEngine.DataContext = dataInstance;

    // Evaluate each of properties which has 'FormulaVariable' attribute.
    PropertyInfo[] properties = GetFormulaProperties<T>();
    foreach (PropertyInfo p in properties)
    {
        string formula = null;
        if (formulaTable.TryGetValue(p.Name, out formula))
        {
            if (!string.IsNullOrEmpty(formula))
            {
                var value = calcEngine.Evaluate(formula);
                p.SetValue(dataInstance, Convert.ChangeType(value, p.PropertyType), null);
            }
        }
    }
}

```

Note that the order of declaration of the properties which should be calculated are important.

See the '*PlayerStatus*' class again. To evaluate the variable hold on `HP` property, firstly it should evaluate and set correct value of STR, DEX and ITL.

```
HP = (5 ((STR 0.5) + (DEX 0.39) + (ITL 0.23)))
```

PlayerStatus.cs

```

public class PlayerStatus
{
    ...
    [FormulaVariable]
    public float STR { get; set; }
    [FormulaVariable]
    public float DEX { get; set; }
    [FormulaVariable]
    public float ITL { get; set; }
    [FormulaVariable]
    public float HP { get; set; }
    ...
}

```

That is all.

One thing to keep in mind is that we heavily used reflection to simplify our formula calculations. As you already know, reflection is not a fast way even on runtime. So, when you work with this approach, it should be carefully done where the speed of game loop is important. e.g. Avoid to do calculation in *MonoBehaviour*'s *Update*.

The following shows whole line of FormulaCalculator class code.

FormulaCalculator.cs

```
public class FormulaCalculator
{
    CalcEngine.CalcEngine calcEngine = new CalcEngine.CalcEngine();

    public void Calculate<T>(System.Object data, System.Object[] obj)
    {
        PropertyInfo[] infos = obj[0].GetType().GetProperties(BindingFlags.Public | BindingFlags.Instance);
        string key = infos[0].Name;
        string value = infos[1].Name;

        Calculate<T>(data, obj, key, value);
    }

    public void Calculate<T>(System.Object data, System.Object[] obj, string key, string value)
    {
        Dictionary<string, string> formulaTable = GetFormulaTable(obj, key, value);
        Calculate<T>(data, formulaTable);
    }

    public void Calculate<T>(System.Object dataInstance, Dictionary<string, string> formulaTable)
    {
        var _type = typeof(T);

        calcEngine.DataContext = dataInstance;

        PropertyInfo[] properties = GetFormulaProperties<T>();
        foreach (PropertyInfo p in properties)
        {
            string formula = null;
            if (formulaTable.TryGetValue(p.Name, out formula))
            {
                if (!string.IsNullOrEmpty(formula))
                {
                    var value = calcEngine.Evaluate(formula);
                    p.SetValue(dataInstance, Convert.ChangeType(value, p.PropertyType), null);
                }
            }
        }
    }

    public Dictionary<string, string> GetFormulaTable(System.Object[] obj, string _key, string _value)
    {
        Dictionary<string, string> result = new Dictionary<string, string>();

        foreach (System.Object o in obj)
        {
            PropertyInfo[] infos = o.GetType().GetProperties(BindingFlags.Public | BindingFlags.Instance);

            // key of the dictionary
```

```

        string statName = null;
        var found = infos.Where(e => e.Name == _key).FirstOrDefault();
        if (found != null)
        {
            object value = found.GetValue(o, null);
            statName = value.ToString();
        }

        // value of the dictionary
        string formula = null;
        found = infos.Where(e => e.Name == _value).FirstOrDefault();
        if (found != null)
        {
            object value = found.GetValue(o, null);
            formula = value.ToString();
        }

        if (string.IsNullOrEmpty(statName) == false && string.IsNullOrEmpty(formula) == false)
            result.Add(statName, formula);
        else
        {
            // error
        }
    }

    return result;
}

private PropertyInfo[] GetFormulaProperties<T>()
{
    var _type = typeof(T);

    List<PropertyInfo> formulaPropList = new List<PropertyInfo>();

    PropertyInfo[] properties = _type.GetProperties(BindingFlags.Public | BindingFlags.Instance)
;

    foreach (PropertyInfo p in properties)
    {
        if (!(p.CanRead && p.CanWrite))
            continue;

        object[] attributes = p.GetCustomAttributes(true);
        foreach (object o in attributes)
        {
            if (o.GetType() == typeof(FormulaVariableAttribute))
                formulaPropList.Add(p);
        }
    }

    return formulaPropList.ToArray();
}
}

```

# Automation of Formula Calculation: Part III

*(working in progress...)*

# Tips and Known Issues

## Excel

Keep as small number of sheet in one excel file. Let's consider a case that an excel file which contains over twenty sheet in one excel file and you've generated all *ScriptableObject* asset files for each sheet. And you've created a new sheet in the same excel file then try to generate necessary script files. What happens? Even you've created only one sheet and want to only import data into the new one but Quicksheet try to reimport all data from all sheets because querying data from excel into newly created *ScriptableObject* done by Unity's reimport. So keep in mind that working with an excel file which has too much sheets can be slow.

## Setting up OAuth2 for accessing Google Drive

Google has changed the authentication scheme since May 5, 2015. Now it requires OAuth2. To set this up visit [Google Developer Console](#), create a new project, enable the Drive API, create a new client ID of type "service account" and download json file. See the **OAuth2 Google Service Account** section on the [Google Spreadsheet Howto](#) page for more details.

See [the page](#) for setting up credentials and getting OAuth2 `'client_ID'` and `'client_secret'`.

## Add QuickSheet via subtree

You can add QuickSheet via subtree to your github project like the following:

```
git subtree add --prefix=Assets/QuickSheet https://github.com/your_github_account/your_project.git QuickSheet
```

It creates *QuickSheet* folder under *Assets* unity project folder then put all the necessary files under the *QuickSheet* folder.

Any changes for the remote repository easily can pull with *git subtree pull* as the following:

```
git subtree pull --prefix=Assets/QuickSheet https://github.com/kimsama/Unity-QuickSheet.git QuickSheet
```

# FAQ

**Q: Can I pull data from Google Sheets at runtime?**

**A:** No, *Unity-Quicksheet* is a tool for easy handling of data at edit time. Saying with the technique point of view, all data retrieved from Excel or Google spreadsheet is stored form of an asset file as [ScriptableObject](#) derived class object to make it easy for serializing and deserializing.

Unity's [ScriptableObject](#) is not a one for persistence data which means there is no way to serialize any **runtime** changed data. *Unity-Quicksheet* is not invented for that purpose. So if the data you have can be changed at runtime and should be persistence, you should consider other thing such as json which is being for that purpose.

There are bunch of formats for bidirectional serialization e.g. JSON, BSON or XML etc. The decision entirely depends on your platform and/or requirements.

On the other hand, there is no reason to get data from Excel at run time on most of cases. Not likely Excel, it may useful to connect and get data at runtime from *Google Drive* but to keep the same interfaces with Excel, *Unity-Quicksheet* does not provide that for Google spreadsheet either. (Though it may be changed in the future)

**Q: Does it support all platforms?**

**A:** Yes, it can run on any platform specified as Unity's build target platform except one restriction, Google spreadsheet on Web Player build target setting.

Google Spreadsheet plugin does not work in the Unity web player's security sandbox due to the security restriction on Unity side. So you should change the platform setting to 'Stand Alone' or something else such as 'iOS' or 'Android' platform in the 'Build Setting' to make it work.

**Q: Can I save any changed data from Unity editor back to Excel or Google spreadsheet?**

**A:** No, you can. Keep changes on Excel or Google drive and make one way for the data flow is better for error proof.

**Q: Can I get data from Excel file on OSX of Mac machine?**

**A:** Yes, it runs on Mac machine without any problems. Keep it mind that save your excel file as '.xls' not as '.xlsx' file on Mac machine otherwise you may get the error, 'Can't read content types part!'

**Q: Can I use 'Open Office' instead of Excel?**

**A:** Yes, both of them get same result for '.xls' file.

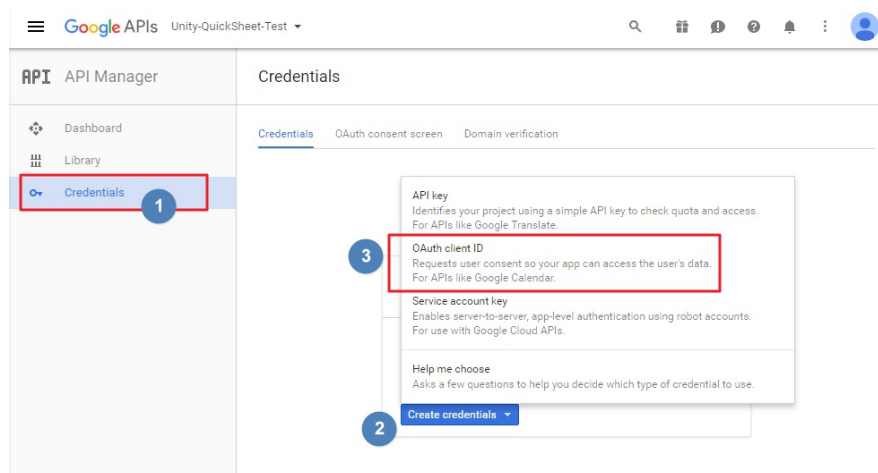


# Setting up OAuth2 for accessing Google Drive

## OAuth2 setting on Google APIs

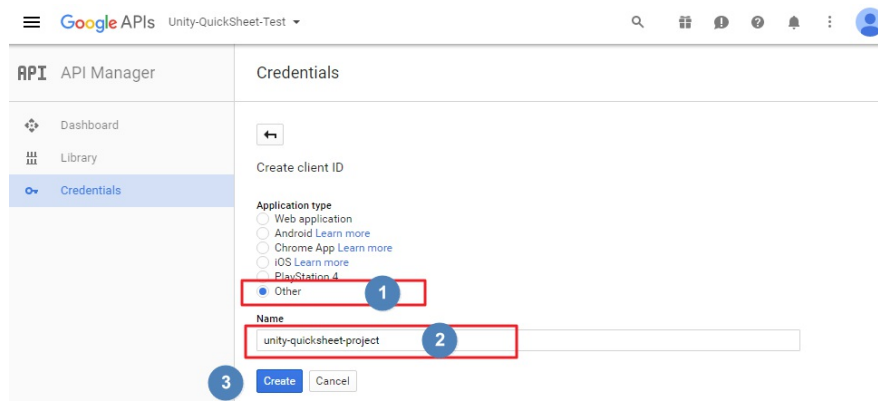
To access google drive, especially google spreadsheet for Unity-Quicksheet, it requires OAuth2 authentication. To set this up visit [Google APIs](#) page and create a new project.

The next describes the way of getting '*client\_ID*' and '*client\_secret*' tokens which are necessary for oauth2 credentials step by step.

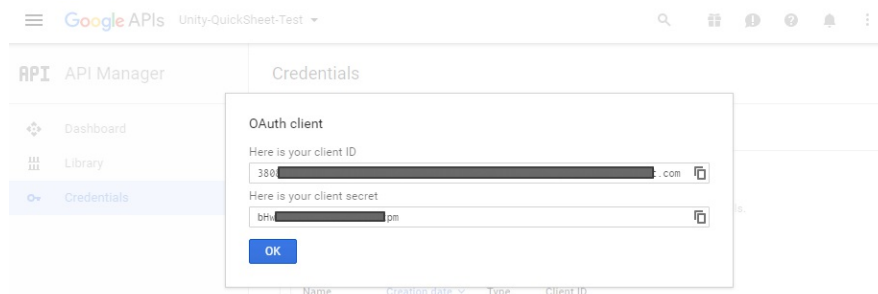


Select 1) **Credentials** on the left panel and **Credential** tab on the top menu then click 2) **Create credentials** button which open a dialogue shows credential types you can create.

Note that you should select 3) **OAuth client ID** otherwise you may have wrong format of json file.

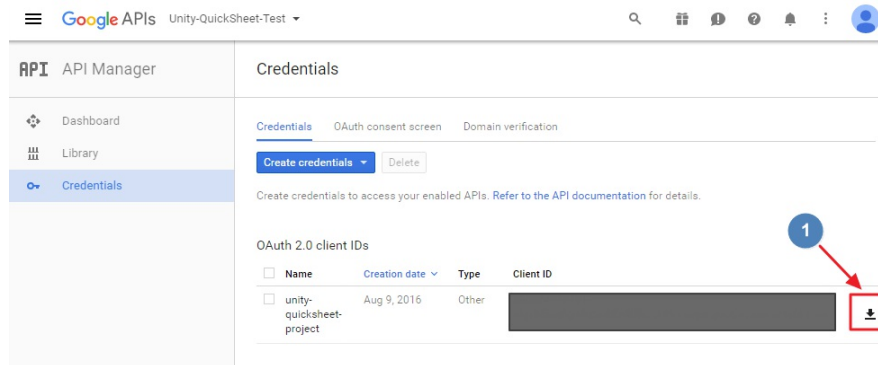


Select 1) **other** for **Application type** and 2) insert a proper name into the **Name** field, then press 3) **'Create'** button will create a credential.



Now you can see '*client\_ID*' and '*client\_secret*' for newly created credential.

At this moment you can directly copy and paste those '*client\_ID*' and '*client\_secret*' into **Unity-Quicksheet's** google setting or download oauth2 json file which contains '*client\_ID*' and '*client\_secret*' are necessary on Unity-Quicksheet setting.



Finally, click 1) the download icon for downloading json file which contains '*client\_ID*' and '*client\_secret*'.

## OAuth2 setting on Unity-Quicksheet

Things for authentication on Google side are done and it is time to setting up remains for accessing google's spreadsheet on Unity side.

See the OAuth2 Google Service Account section on the [Google Spreadsheet Howto](#) page for more details.

# Code Template

Unity-Quicksheet uses various template files to automatically generate '.cs' script files needed to import data from a spreadsheet.

All template files are a simple ascii '.txt' files. So you can easily change if you want it generate other form of script.

The generated scripts are divided into two kind things according to its purpose. One for runtime and the other for editor script.

Runtime	Editor
DataClass.txt	PostProcessor.txt
Property.txt	ScriptableObjectEditorClass.txt
ScriptableObjectClass.txt	N/A

Editor script will be put under the folder, named as '*Editor*', a Unity specific folder.

## Using Custom Template

One of flexible feature of Unity-QuickSheet is that you can change template files to generated script file whatever you want to.

But there is one thing to keep in mind.

A word which has '\$' for its prefix will be replaced when Unity-QuickSheet generates script files. So you should be careful if you change any word has '\$' if not, the generated script files won't be compiled with getting syntax error.

If you familiar with Unity editor script, try to import data from a simple spreadsheet and look at the generated script files. The last of things are self explanation.

## Reference

### DataClass.txt

Keyword	Desc
\$ClassName	A class name of DataClass class. Correspond to name of a spreadsheet. (or 'worksheet' name in case of google spreadsheet)
\$MemberFields	Correspond to name of a ' <i>column-header</i> ' in the spreadsheet.

### Property.txt

Keyword	Desc
\$FieldName	A member field name of the ' <i>DataClass</i> ' class.
\$CapitalFieldName	Same as \$FieldName but capitalized.

## ScriptableObjectClass.txt

Keyword	Desc
\$ClassName	A ' <i>class name</i> ' which derives <i>ScriptableObject</i> class
\$DataClassName	It should same as \$ClassName of ' <i>DataClass.txt</i> ' template file.

## ScriptableObjectEditorClass.txt

Keyword	Desc
\$WorkSheetClassName	Correspond to name of a spreadsheet. (or 'worksheet' name in case of google spreadsheet)
\$ClassName	Same as \$ClassName of ' <i>ScriptableObjectClass</i> ' class.
\$DataClassName	Same as \$ClassName of ' <i>DataClass</i> ' class.

## PostProcessor.txt

Keyword	Desc
\$AssetPostprocessorClass	
\$ClassName	Same as \$ClassName of ' <i>ScriptableObjectClass</i> ' class.
\$DataClassName	Same as \$ClassName of ' <i>DataClass</i> ' class.

# References

- [Unity Serialization](#) on Unity's forum for details of serialization mechanism.
- [GDataDB](#) is used to retrieve data from Google Spreadsheet. Note that [GDataDB](#) is slightly modified to support *enum* type.
- ~~[ExposeProperties](#) is used to easily expose variables of spreadsheet on the Unity3D's inspector view and let [GDataDB](#) access through get/set accessors. (Dropped on v.1.0.0.)~~
- [NPOI](#) is used to read xls and xlsx file.
- All "\*.dll" files of Google Data SDK are available at [Google Data API SDK](#)
- Newtonsoft.Json source code for net 2.0 is available at [here](#)
- [Unity-GoogleData](#), my previous effort to import a spreadsheet data to Unity.