
Getting Started - Website - Asset Store

Please note: This is an offline version of the documentation available here:
http://databox.doorfortyfour.com/documentation/
It's possible that some points might have changed or are not up to date.

Support

If you need any support. Please drop us a line at: mail@doorfortyfour.com

Table of Contents

Getting started

To get started quickly, simply download the Unity package from the Package Manager inside Unity. After installation you should have a new Databox folder under your Assets project folder.

You can move the Databox folder wherever you want inside of your project

Create a Databox object

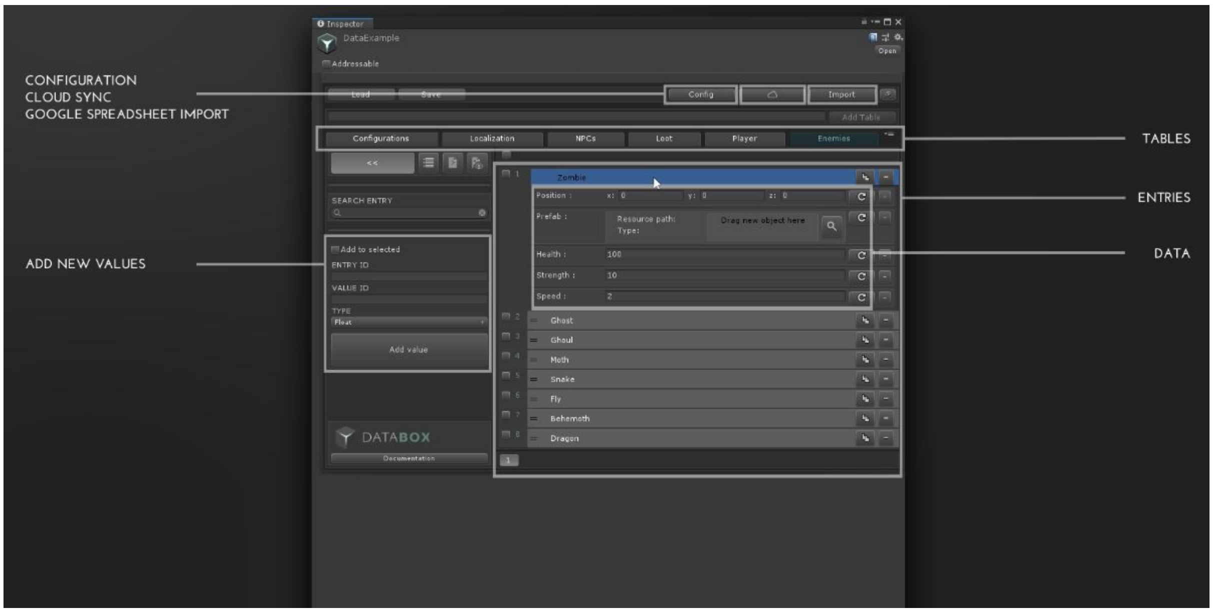
- Right click in the project view and select **Create -> Databox -> New Databox Object**

.NET 4.x

Databox supports Odin Serializer and FullSerializer. To make sure Odin Serializer works properly, you'll have to set the API compatibility level to .NET 4.x in the Unity player settings.

.NET 2.0 - Since version 1.1.1

If you want to use .NET 2.0, then you need to remove the OdinSerializer folder located at: Databox/Core/Serializers.

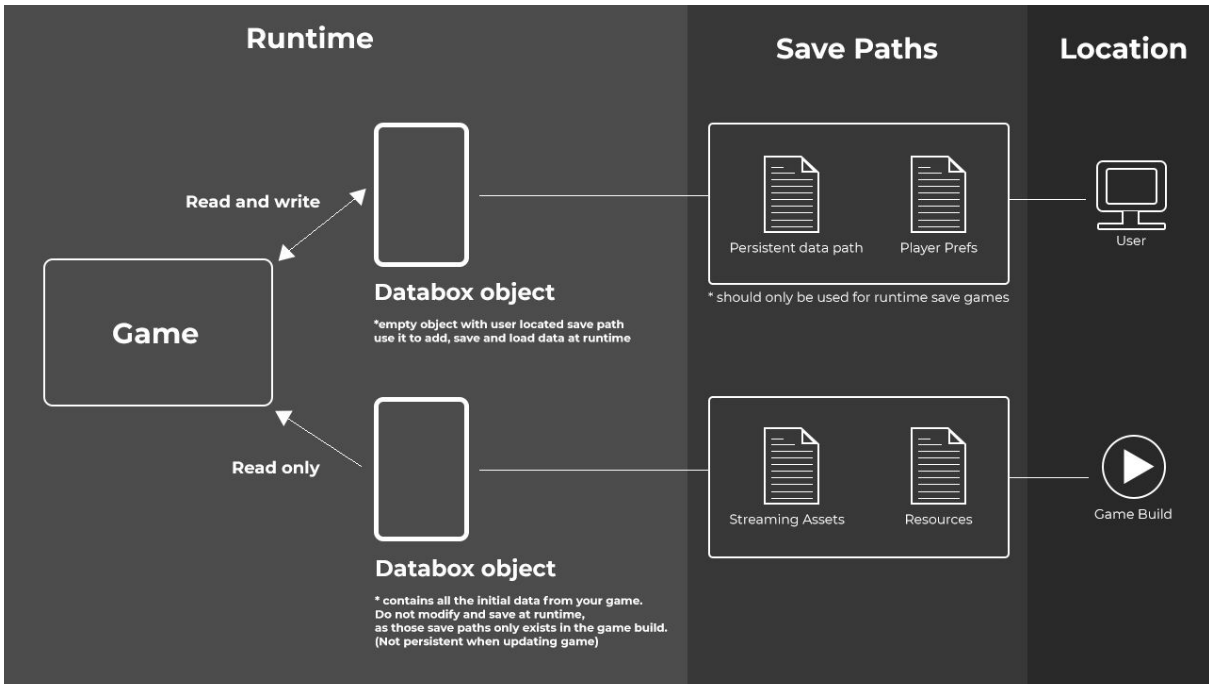


Concept And Workflow

A Databox object (Scriptable object file) serves as a container for your data. All your data is nicely structured in tables, entries and values.

A Databox object saves and loads the data to a file. Depending on the serializer and data format you have chosen this can be a json file or a binary file.

You can setup multiple Databox objects in your project each with different save paths and file names if you like. Please consider the following image in regards to all available save paths. It is important to know the location of a save paths. (User device/User computer or Game build)


Runtime
Save Paths
Location
Read and write
Game
Databox object
*empty object with user located save path use it to add, save and load data at runtime
Persistent data path
Player Prefs
* should only be used for runtime save games
User
Read only
Databox object
* contains all the initial data from your game. Do not modify and save at runtime, as those save paths only exists in the game build. (Not persistent when updating game)
Streaming Assets
Resources
Game Build

Workflow

If you like to use Databox not only as a data container but also as a save system it is best to create two Databox objects. One containing your initial game data which lives in the game build and another empty Databox object which has a persistent save path configured - this one is used for storing runtime data on the users machine.

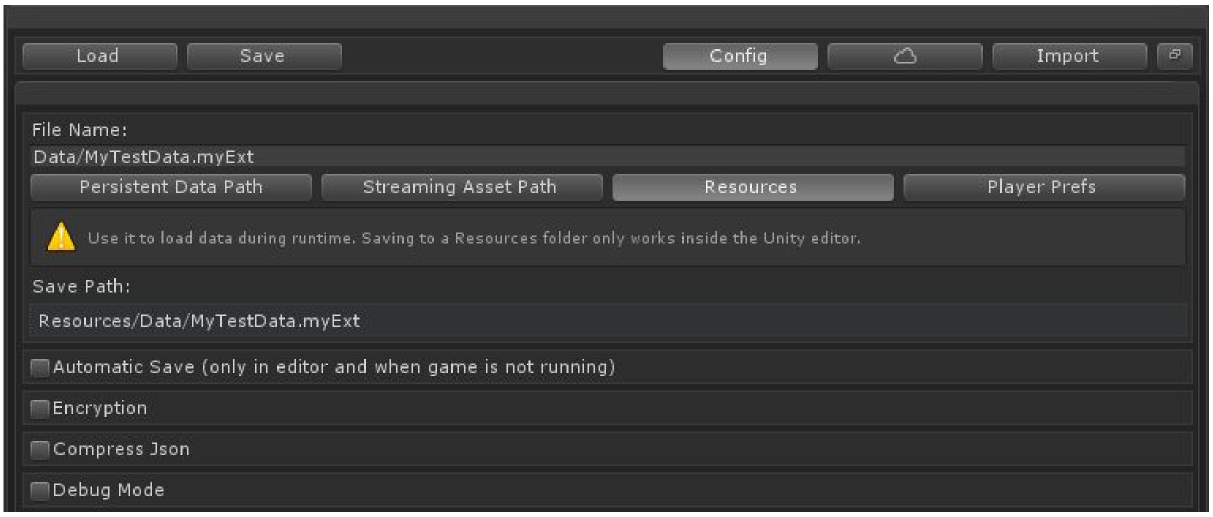
During runtime you can now easily "register" an existing entry from your initial Databox object to the runtime empty Databox object by using following code:

```
    public DataboxObject initialData;
    public DataboxObject runtimeData;


    initialData.RegisterToDatabase(runtimeData, "Initial table name",
"Initial entry name", "New and unique entry name"));
```

By using RegisterToDatabase, Databox populates the runtime Databox object with an initial entry from the initial Databox object. It is important to pass a unique string id for the new entry name. For example when instantiating multiple "enemies" of the same type "Boss1" you should make sure to register them with an unique id, to make sure each enemy has its unique assigned values in the runtime Databox object. For using a unique id you could use:
this.gameObject.GetInstanceID().ToString()

Instead of using RegisterToDatabase you can also simply copy the initial file to the persistent data path. When your game starts for the first time. For more information about this workflow please see: Save and load on Android

Configuration



File Name:
Data/MyTestData.myExt

Persistent Data Path | Streaming Asset Path | Resources | Player Prefs

Use it to load data during runtime. Saving to a Resources folder only works inside the Unity editor.

Save Path:
Resources/Data/MyTestData.myExt

Automatic Save (only in editor and when game is not running)
Encryption
Compress Json
Debug Mode

Before you can start adding data to your databox. You will need to specify a file name and a save path in the config. menu.

File name

- Select the config menu and specify a file name. You can also add a custom extension to your file name and subfolders. Example: "Data/Database.myExt"
  - **Persistent Data Path:** Should be used for save game data only. Do not use it for your initial database data as it will not be included in a build.
  - **Streaming Asset Path:** Uses the "StreamingAssets" folder inside of your project. Great for files that should be loaded and saved upon runtime.
  - **Resources:** Use the resources path for loading the database file from a resources folder. It's not possible to save to a resources folder during runtime.
  - **PlayerPrefs:** Player prefs are a good way to create save games at runtime. Do not use it for your initial database data as it will not be included in a build.

Make sure to check out the official Unity documentation regarding the different save paths
Persistent Path
Streaming Asset Path

Encryption

- Databox supports a simple XOR encryption. When enabled you'll need to provide an encryption key.

Compress Json

- Compressing the json file significantly reduces the file size. On the other hand, the json file is no longer easy to read in a text editing tool.

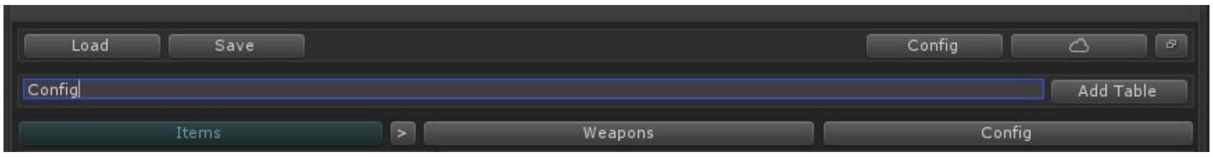
Debug Mode

- By enabling debug mode, Databox will create console logs on several events.

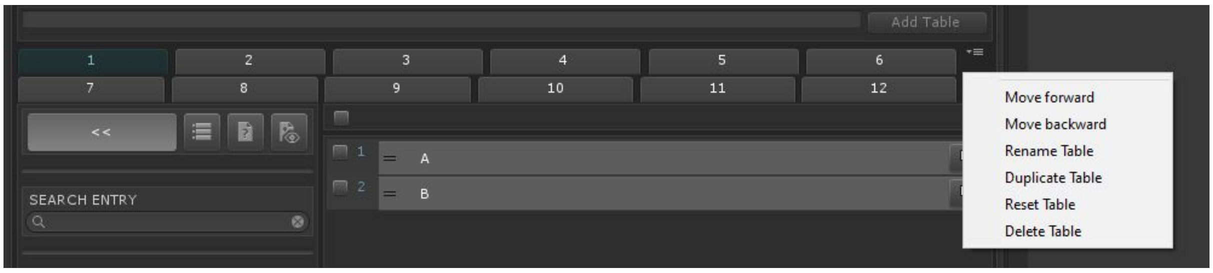
Adding Data

After setting a file name and save path you can start adding data to your Databox object.

Add a table

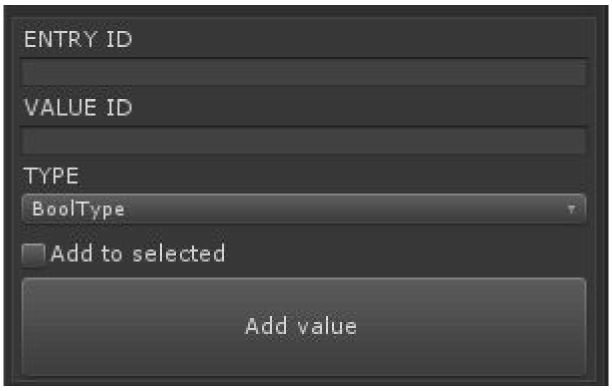


- First you will need to add a table. Simply set a name for a table and click on **Add Table**



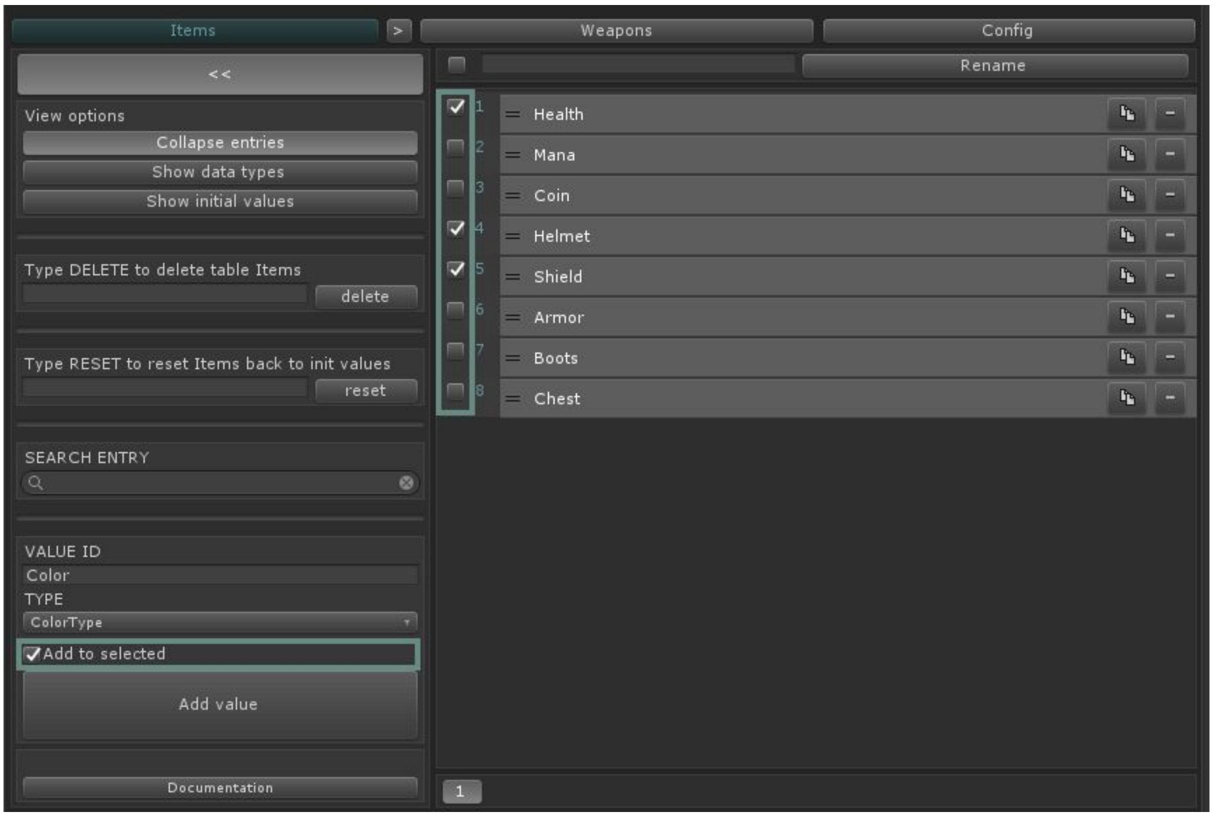
- You can rename, duplicate, move, or delete the tables by clicking on the dropdown menu next to it.
- You can now add your first data entry with some values in it.

Add new entry



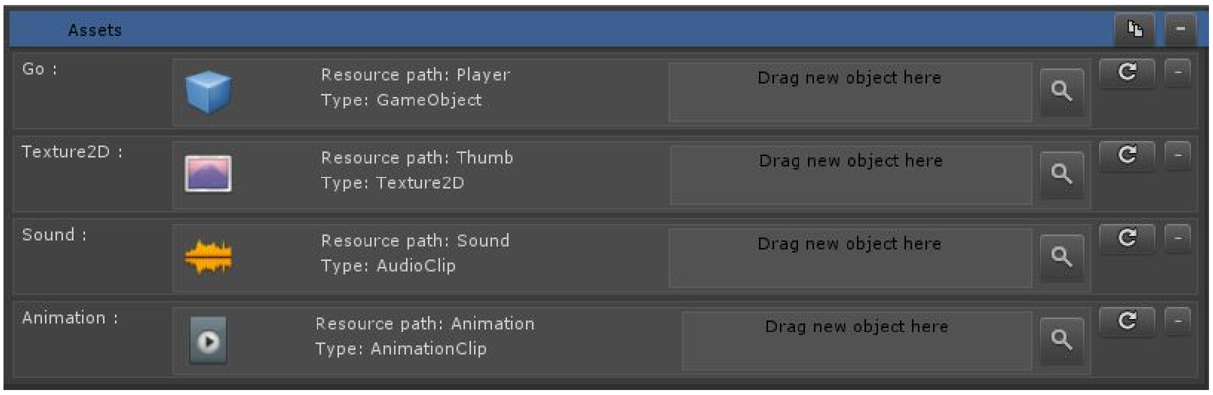
- To add a new data entry set the name of the entry (for example: Player). Then set the name of the value (for example: Health).
- Now select the data type, leave add to selected to false and click on **Add Value**

Add to selected



- Enable add to selected if you want to add data values to existing entries. You will need to select the entries first.

Resource Type



Assets

Go : Resource path: Player Type: GameObject Drag new object here

Texture2D : Resource path: Thumb Type: Texture2D Drag new object here

Sound : Resource path: Sound Type: AudioClip Drag new object here

Animation : Resource path: Animation Type: AnimationClip Drag new object here

Databox supports Unity resources types by storing it's resource path. When calling the GetData on a Databox object with a ResourceType Databox loads the prefab automatically.

Example of loading a GameObject:

```
        // Loads a resource of type GameObject
        var _go = db.GetData<ResourceType>("Assets", "Player", "GameObject").Load() as GameObject;
        Instantiate(_go, Vector3.zero, Quaternion.identity);
```

Example of loading a Texture2D:

```
        // Loads a resource of type Texture2D
        var _thumb = db.GetData<ResourceType>("Assets", "Player", "Thumb").Load() as Texture2D;
```

Addressables

Databox also supports the newly introduced Addressables by Unity. Addressables are still in Beta, therefore it's possible that some features might change or break.

Install

- To use the addressables you need to first download the addressables package with the package manager inside of Unity.
- Make sure to open the addressables window for the first time. Go to: **Window -> Asset Management -> Addressables.** This is important so the addressable system can do some setup.
- Now open the AddressableType.cs script inside of the Databox folder: **Databox -> Types -> AddressableType.cs.**
- Uncomment the script and wait for compilation.
- When adding a new value you can now select the addressable type.
- The addressable type in Databox simply saves the addressable path. You can select all available assets by a dropdown.
- To know how to use addressables (Instantiate, Loading) please refer to the official documentation here: Addressables Documentation

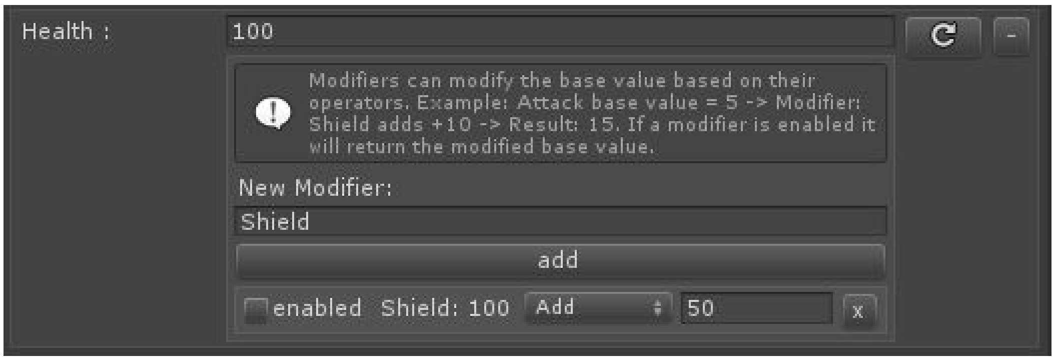
Modifiable Int and Float type

When using the modifiable int and float type you can add additional modifiers to the base value. When getting a value it will be modified by the modifiers which are enabled.

Enable Modifier through code:

```
    var _health = data.GetData<IntWithModifiersType>("Table", "Entry"
"Health");
    _health.Modifiers("Shield", true); // enable shield modifier
```

Example:



```
Health :    100
Modifiers can modify the base value based on their operators. Example: Attack base value = 5 -> Modifier: Shield adds +10 -> Result: 15. If a modifier is enabled it will return the modified base value.
New Modifier:
Shield
add
enabled  Shield: 100  Add  50  x
```

Here we have a health value which is set to 100. Additionally we have added one modifier called "Shield" which adds 50.

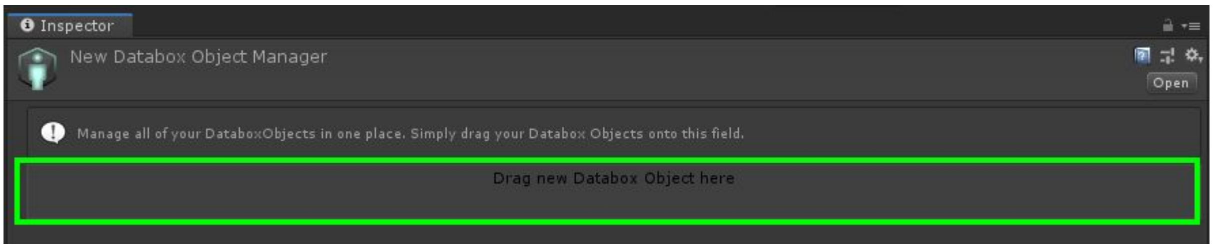
So when returning the health value and modifier is enabled, Databox will return 150 instead of 100.

Databox Manager

Databox comes with a Databox Object Manager which helps you to keep track of all separate Databox objects.

Create a Databox object manager

- Right click in the project view and select Create -> Databox -> New Databox Object Manager
- You can now drag and drop all of your Databox objects in the drag field of the Databox object manager.



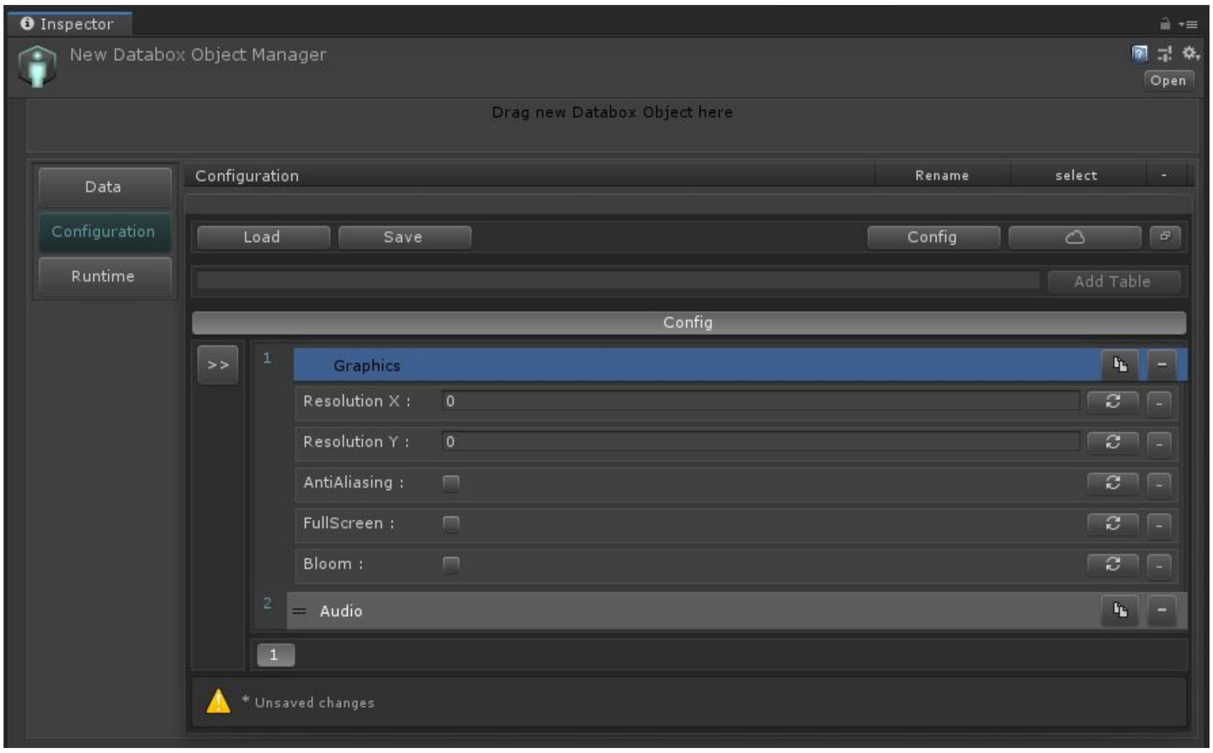
Manage all of your DataboxObjects in one place. Simply drag your Databox Objects onto this field.

Drag new Databox Object here

- Each Databox object can be accessed by it's name from the manager. Simply use:

```
public DataboxObjectManager manager;

DataboxObject databoxObject = manager.GetDatabox("MyData");
```

* The Manager is great if you are using multiple Databox objects in your game. For example: Configuration, GameData, SaveGame



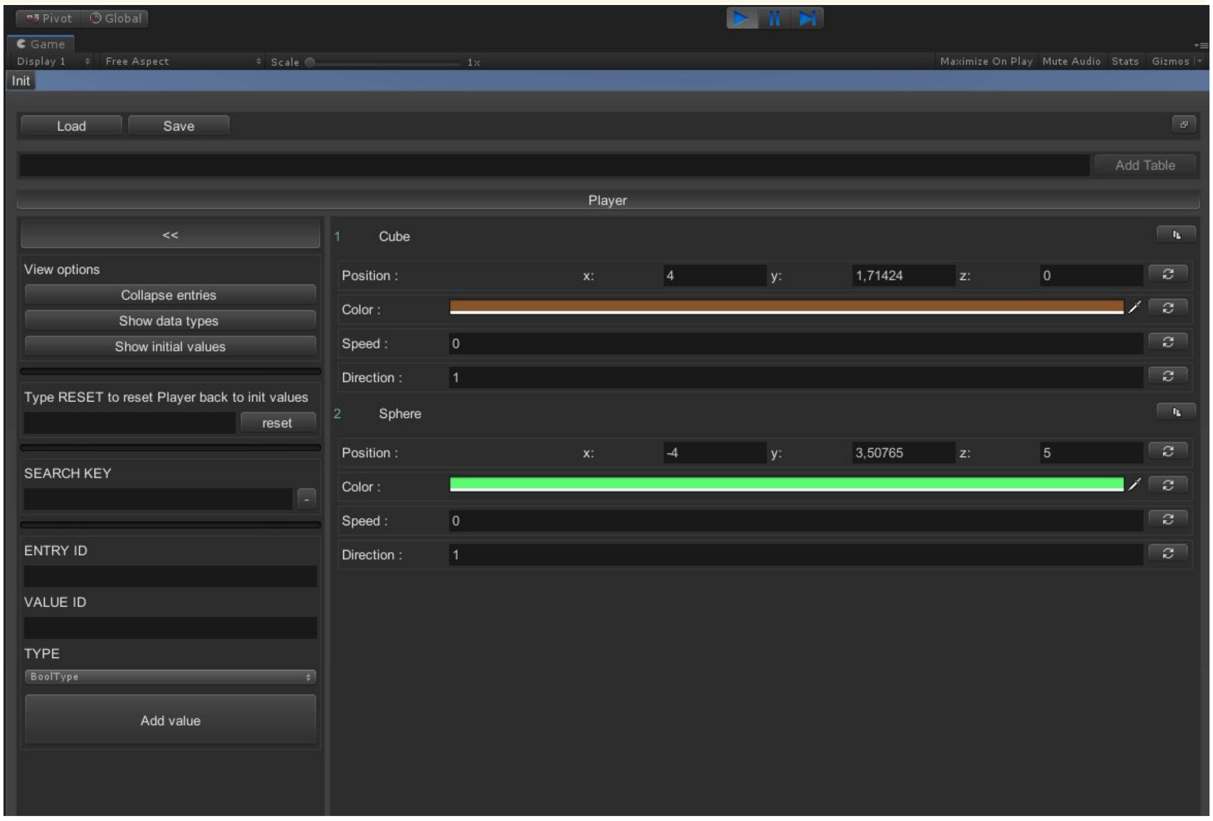
Runtime editor

The runtime editor allows you to access all of your data at runtime as long as your data is using the GUILayout API for drawing it's value. (EditorGUILayout API won't work for runtime builds)

Create the runtime editor

- Simply create an empty game object in your scene and add a new DataboxRuntimeEditor component to it.
- Add a Databox object manager to the runtime editor.

Databox comes with a custom Databox GUI style which you can add to the Editor Skin property.



1 Cube

Position : x: 4 y: 1,71424 z: 0

Color :

Speed : 0

Direction : 1

2 Sphere

Position : x: -4 y: 3,50765 z: 5

Color :

Speed : 0

Direction : 1

Use Databox by script

To use Databox simply make sure to have a reference to a Databox object. It is important to load your data before accessing it. You can register a method to the OnDatabaseLoaded event, which will be called when your database has been loaded.

LoadDatabase

```
using Databox;

public class Example : MonoBehaviour
{
    // The reference to your databox object
    public DataboxObject data;

    void OnEnable()
    {
        data.OnDatabaseLoaded += DataReady;
    }

    void OnDisable()
    {
        data.OnDatabaseLoaded -= DataReady;
    }

    void Start()
    {
        data.LoadDatabase();
    }

    void DataReady()
    {
        // Access data
    }
}
```

GetData

GetData returns a reference of the data in the database. This means that when changing the health value in the example you won't need to put the changed value back to the database as it gets updated automatically.

```
using Databox;

public class Example : MonoBehaviour
{
    // The reference to your databox object
    public DataboxObject data;

    // return data of type float
    public void GetData()
    {
        FloatType health = data.GetData<FloatType>("TableName",
"EntryName", "ValueName");

        // modify value
        health.Value = 10f;

        Debug.Log(health.Value.ToString());
    }
}
```

AddData

To add data by script simply call AddData.

```
using Databox;

public class Example : MonoBehaviour
{
    // The reference to your databox object
    public DataboxObject data;

    public void AddData()
    {
        // create a new float value
        FloatType health = new FloatType();
```

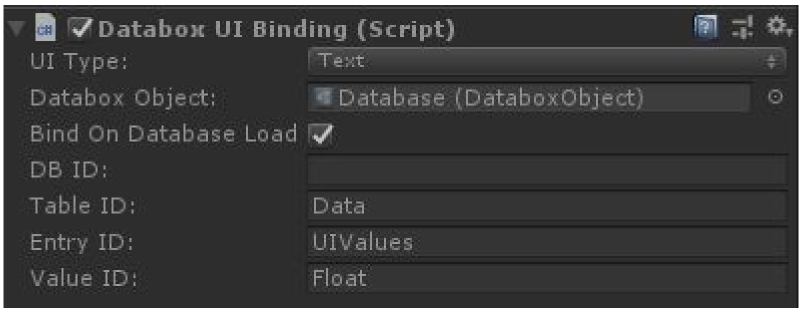
```
            // add value
            health.Value = 100f;

            // Add health to our database
            data.AddData("TableName", "EntryName", "ValueName", health);
        }

    }
```

OnValueChanged

Each data value has a "on value changed" event which gets called if a value has been changed.

```
using Databox;

public class Example : MonoBehaviour
{
    public DataboxObject data;

    FloatType health;

    public void Start()
    {
        // get data
        health = data.GetData<FloatType>("TableName", "EntryName", "ValueName");
        // register event
        health.OnValueChanged += OnHealthChanged;
    }

    void OnHealthChanged(DataboxType _data)
    {
        // instead of using a global variable you can also convert the data back
        // var _health = _data as FloatType;

        Debug.Log("Health has been changed " + health.Value.ToString());
    }
}
```

UI Binding



Databox UI Binding (Script)

UI Type:	Text
Databox Object:	Database (DataboxObject)
Bind On Database Load	✓
DB ID:	
Table ID:	Data
Entry ID:	UIValues
Value ID:	Float

Databox has a very simple but powerful UI Binding component which allows you to bind Unity UI elements - Text, InputFields, Sliders, Toggles... - to a specific Databox object.

- Simply add the DataboxUIBinding component to a UI Element, define the table, entry and value id and you're done.
- Make sure to set the Bind On Database Load to true if you want to bind the ui as soon as the Databox object has been loaded. That requires that you load the database on runtime by script. If you set the On Database Load to false you will need to call the Bind method manually.

Please see the example scene for more information.

Create a custom class

Every supported data type in Databox is derived from DataboxType. Follow these instructions to create a custom databox type class.

You can also have a look at the **ExampleCustomDataType.cs** script to see how you can create your own data types with custom editor ui.

- Create a new C# class and name it like you wish. (In this example: CustomDataClass)
- Add the namespace Databox to the class.

```
using Databox;
```

- Derive from DataboxType.

```
public class CustomDataClass : DataboxType {}
```

- While deriving from DataboxType you will get access to several override methods.

```
public override void DrawEditor(){}
public override void DrawInitValueEditor(){}
public override void Reset(){}
public override string Equal(DataboxType _changedValue){}
```

- Let's add a simple int value called health to our class.

```
    [SerializeField]
    private int _health;
    [SerializeField]
    public int InitHealth; // The initial health value

    // Public property of health. By using get and set we can add an OnValueChanged callback
    public int Health
    {
        get {return _health;}
        set
        {
            if (value == _health){return;}

            _health = value;
            if (OnValueChanged != null){OnValueChanged(this);}
        }
    }
```

- Several things we need to consider.
    1. To make sure the values are being serialized add a **[SerializeField]** attribute to the variable
    2. InitHealth stores the initial default value of our health variable. We will use the **Reset()** override method to set _health back to InitHealth.

- Next we will create a custom editor for our custom class. This will be drawn in the Databox editor.

```
public override void DrawEditor()
{
    var _healthString = Health.ToString();
    _healthString = GUILayout.TextField(_healthString);
    int.TryParse(_healthString, out _health);
}
```

Note you can make use of the GUILayout or EditorGUILayout(not working at runtime) API to create your custom editor gui for your data class. Also check out the example class EnemyType in the project folder.

- Let's add the additional GUI code to draw the initial value.

```
public override void DrawInitValueEditor()
{
    GUI.color = Color.yellow;
    GUILayout.Label ("Init Health:");
    GUI.color = Color.white;

    var _healthString = InitHealth.ToString();
    _healthString = GUILayout.TextField(_healthString);
    int.TryParse(_healthString, out InitHealth);
}
```

- To make sure our reset to initial value functionality works we need to add the Reset() method.

```
// Reset value back to initial value
public override void Reset()
{
    Health = InitHealth;
}
```

- To make sure the cloud sync comparison works we need to add following method. We basically compare each value and if they're different we return a string with the changes.

```
// Important for the cloud sync comparison
public override string Equal(DataboxType _changedValue)
{
    var _v = _changedValue as CustomDataClass;
    if (Health != _v.Health)
    {
        // return original value and changed value
        return Health.ToString() + " : " +
_v.Health.ToString();
```

```
    }
    else
    {
        return "";
    }
}
```

- Finally we have to add the convert method to make sure the CSV import works. The convert method simply converts the string which comes from the CSV file to the appropriate data type. In this example we simply parse the string to a integer value.

```
// Convert the CSV string to an integer value
public override void Convert(string _value)
{
    Health = int.Parse(_value);
    InitHealth = Health;
}
```

- Here's the complete custom class. After saving the file you can add data entries of type "YourCustomDataClass"

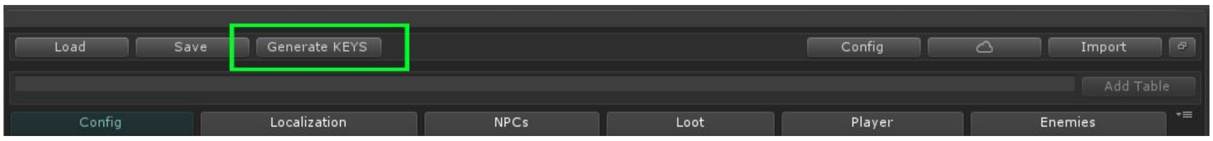
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Databox;

[System.Serializable]
public class CustomDataClass : DataboxType {

    [SerializeField]
    private int _health;
    [SerializeField]
    public int InitHealth;

    public int Health
    {
        get {return _health;}
        set
        {
            if (value == _health){return;}

            _health = value;
            if (OnValueChanged != null){OnValueChanged(this);}
        }
        }

        public override void DrawEditor()
        {
            var _healthString = Health.ToString();
            _healthString = GUILayout.TextField(_healthString);
            int.TryParse(_healthString, out _health);
        }

        public override void DrawInitValueEditor()
        {
            GUI.color = Color.yellow;
            GUILayout.Label ("Init Health:");
            GUI.color = Color.white;

            var _healthString = InitHealth.ToString();
            _healthString = GUILayout.TextField(_healthString);
            int.TryParse(_healthString, out InitHealth);
        }
        // Reset value back to initial value
        public override void Reset()
        {
            Health = InitHealth;
        }

        // Important for the cloud sync comparison
        public override string Equal(DataboxType _changedValue)
        {
            var _v = _changedValue as CustomDataClass;
            if (Health != _v.Health)
            {
                // return original value and changed value
                return Health.ToString() + " : " +
_v.Health.ToString();
            }
            else
            {
                return "";
            }
        }
    }
```

Generate static keys

NEW SINCE UPDATE 1.1.1

Instead of using strings for calling GetData and AddData - which can be error prone - you can generate static keys from your DataboxObject. Another great advantage is, that static keys are structured in classes and sub-classes based on tables and entries.

Simply click on the Generate Keys button in the Databox Object editor.



Select a file name and click on save.

Example:

The keys are now accessible by using the databox object name + "_KEYS". Let's assume we have a DataboxObject called PlayerData then we can use:

```
// Data is the table name - so we use TableName to access the static key
PlayerData_KEYS.Data.TableName

// Player is the entry name inside of the table Data - so we use EntryName to access the static key
PlayerData_KEYS.Data.Player.EntryName

// Health is located inside of the player entry - so we can simply use the _Health key.
PlayerData_KEYS.Data.Player._Health
```

GetData Example:

```
public DataboxObject data;

var health = data.GetData<IntType>(PlayerData_KEYS.Data.TableName, PlayerData_KEYS.Data.Player.EntryName, PlayerData_KEYS.Data.Player._Health);
```

Save and load on Android

Loading and in particular saving works a bit differently on Android / iOS especially when using streaming asset path. From the official Unity documentation:
https://docs.unity3d.com/Manual/StreamingAssets.html

"On many platforms, the streaming assets folder location is read-only; you can not modify or write new files there at runtime."

It's important to consider this when using Databox for Android.

Here you'll find two possible workflows:

Workflow 1:

You can use two Databox objects with different save path configurations. One contains all of your game data which is saved in StreamingAssets folder or Resource folder. This data will be read only during game play.

Then you have the second Databox object which is empty and is configured to be saved in the persistent data path. During runtime you can then put all data which have to be saved in the second Databox object.

Workflow 2:

This is quite a common work-flow on Android and suits you probably better. When the user starts your game for the very first time, you simply copy your database file from the StreamingAssets or Resource folder to the persistent data path. It's important to do this only once, otherwise you would constantly overwrite the existing save file.

Now you simply use the data during runtime (loading and saving) from the persistent data path. In this case you would also use two Databox Objects. One which contains all initial data and is used only in the Unity editor and is being saved in to the streaming asset or resource folder. And a second one which is being used during runtime and loads the data from the persistent data path.

To copy the file to the persistent data path you can use following coded:

From Resources folder: ("Data" -> file name without extension)

```
    var jsonString = Resources.Load<TextAsset>("Data");

File.WriteAllText(System.IO.Path.Combine(Application.persistentDataPath, "Data"), jsonString.text);
```

From Streaming Assets: (Need to use coroutine on android, as files in streaming assets folder are packed)

```
    public IEnumerator Load()
    {
        string _result = "";
        using (UnityEngine.Networking.UnityWebRequest _download =
UnityEngine.Networking.UnityWebRequest.Get(System.IO.Path.Combine(Appl
ication.streamingAssetsPath), "FileName.ext"))
        {
            yield return _download.SendWebRequest();
            while (!_download.isDone)
            {
                if (_download.isNetworkError || _download.isHttpError)
                {
                    break;
                }
                yield return null;
            }

            if (_download.isNetworkError || _download.isHttpError)
            {
                Debug.Log(_download.error);
            }
            else
            {
                _result = _download.downloadHandler.text;
                // Write file to persistent data path

File.WriteAllText(Path.Combine(Application.persistentDataPath,
"FileName.ext"), _result );
            }
        }
    }
```

IMPORT

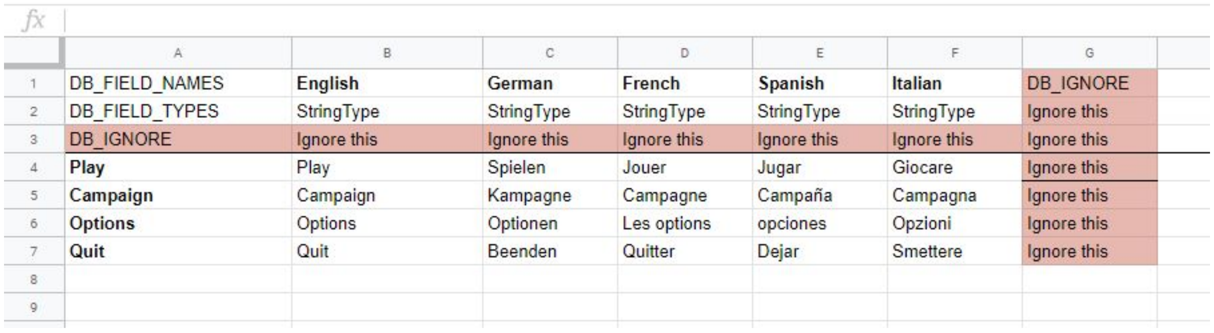
Using spreadsheets

To make sure Databox recognizes the correct data types you will need to setup the spreadsheet correctly.

Use following keys to mark the appropriate fields:

- DB_FIELD_NAMES: The actual name of the value/variable
- DB_FIELD_TYPES: The type of the value. Use the exact name of the data class. (FloatType, IntType...)
- DB_IGNORE: Mark rows and cells you don't want to import with this key.

All basic types are supported. You can add additional support by adding your own convert method to your custom DataboxType class.



	A	B	C	D	E	F	G
1	DB_FIELD_NAMES	**English**	**German**	**French**	**Spanish**	**Italian**	DB_IGNORE
2	DB_FIELD_TYPES	StringType	StringType	StringType	StringType	StringType	Ignore this
3	DB_IGNORE	Ignore this	Ignore this	Ignore this	Ignore this	Ignore this	Ignore this
4	**Play**	Play	Spielen	Jouer	Jugar	Giocare	Ignore this
5	**Campaign**	Campaign	Kampagne	Campagne	Campaña	Campagna	Ignore this
6	**Options**	Options	Optionen	Les options	opciones	Opzioni	Ignore this
7	**Quit**	Quit	Beenden	Quitter	Dejar	Smettere	Ignore this
8							
9							

Formats

Make sure to separate values by comma. For example:

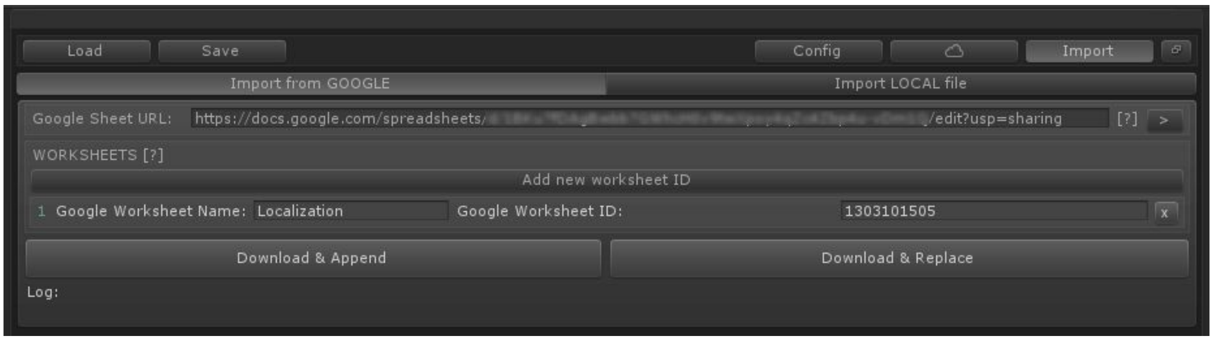
- Vector3: 0.3, 2.0, 4.0
- Color: 255, 220, 120, 255
- String list: A, B, C, D, E, F

IMPORT

Google Spreadsheet

You can import a Google spreadsheet with all worksheets without using any complicated Google authentication. You only need to enable the public share link.

- Open a spreadsheet file in Google.
- In the top right corner, click Share.
- Click "Get shareable link" in the top right of the "Share with others" box.
- To choose whether a person can view, comment, or edit the file, click the Down arrow next to "Anyone with the link."



Google Sheet URL: https://docs.google.com/spreadsheets/[redacted]/edit?usp=sharing [?]

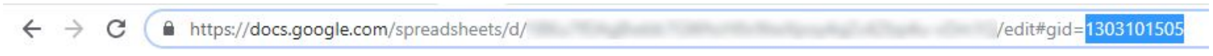
1 Google Worksheet Name: Localization Google Worksheet ID: 1303101505

- Copy the link and paste it in the Google Sheet URL of the Import menu.

Adding Worksheets.

To make sure Databox downloads all worksheets you will need to add them separately by it's unique link ID.

- Open a spreadsheet file in Google.



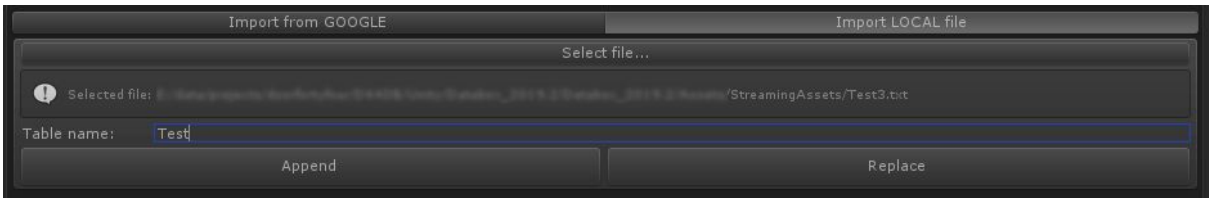
https://docs.google.com/spreadsheets/d/[redacted]/edit#gid=1303101505

- search for the gid number in the browser URL field. Select and copy it.
- Go to the import menu of your Databox object, select import from Google and click on **Add new worksheet ID**.
- Add the table name you want to use and the gid number you have copied previously.
- Click Download & Append or Download & Replace.

IMPORT

Local CSV file

To import a CSV file simply select import from local file in the import menu.



Selected file: [redacted]/StreamingAssets/Test3.txt

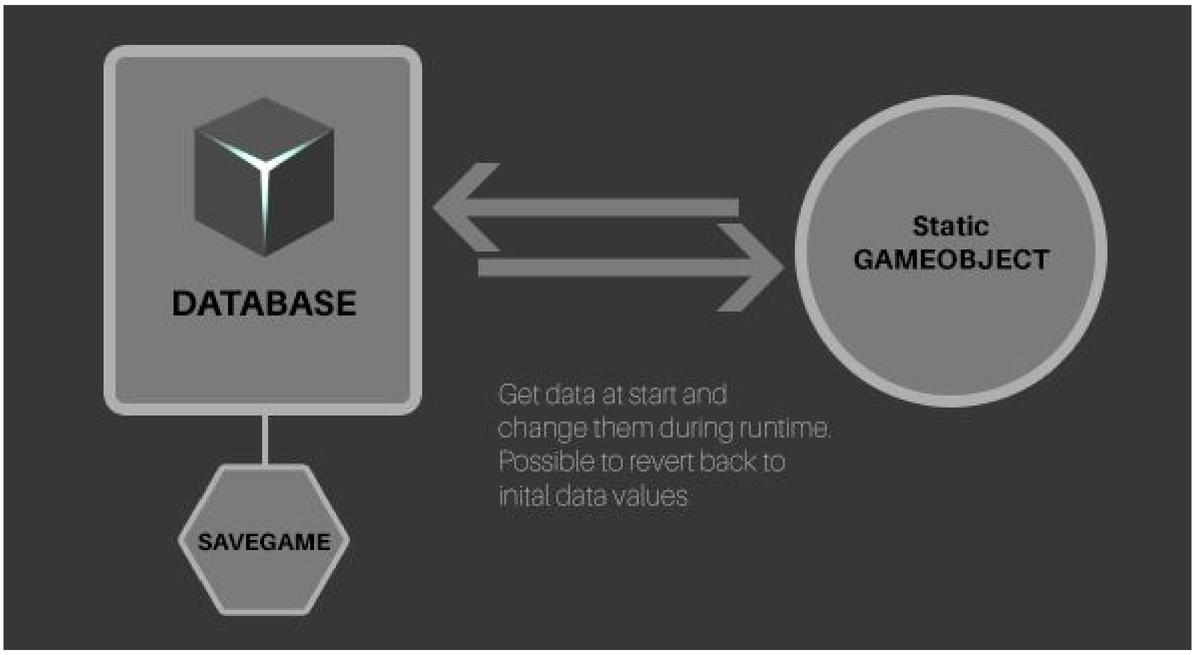
Table name: Test

- Select a text/csv file
- Define the table name
- Click on **Append** to add it to your existing database or **Replace** to replace the current database.

DEMO SCENES

01. Basic Example Scene

The basic scene shows you the basic principals of Databox and how you can load and save existing objects during runtime.


DATABASE
Static GAMEOBJECT
Get data at start and change them during runtime. Possible to revert back to inital data values
SAVEGAME

Scripts

SimpleDataboxLinkking.cs

The cube and the sphere objects have both a script called "SimpleDataboxLinking" This script waits for the database to be loaded. On loaded it takes all values from the database and assigns them to the sphere. (position, color, speed, direction)

DataboxExample.cs

The DataboxExample script has some methods which are being called by the UI.(Save, Load, Reset) It also loads the database on start.

02. OnValueChanged

This examples shows the use of registering an on value change callback and react to it. Simply start the scene and click on the sphere.

03. UI Binding Example

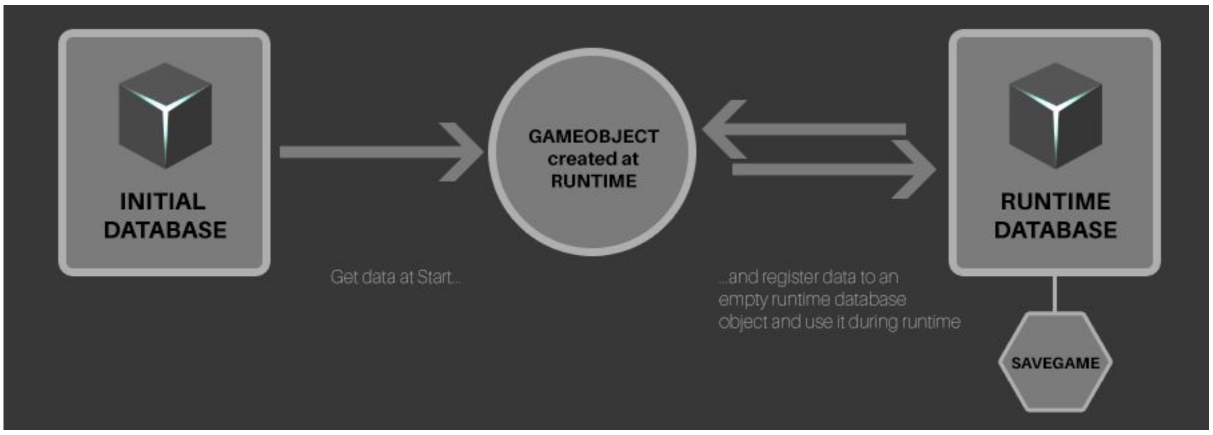
The UI binding example shows you how to bind Unity UI components to values of a Databox object. Each UI component in the scene has a DataboxUIBinding component assigned to it.

04. Advanced Example Scene

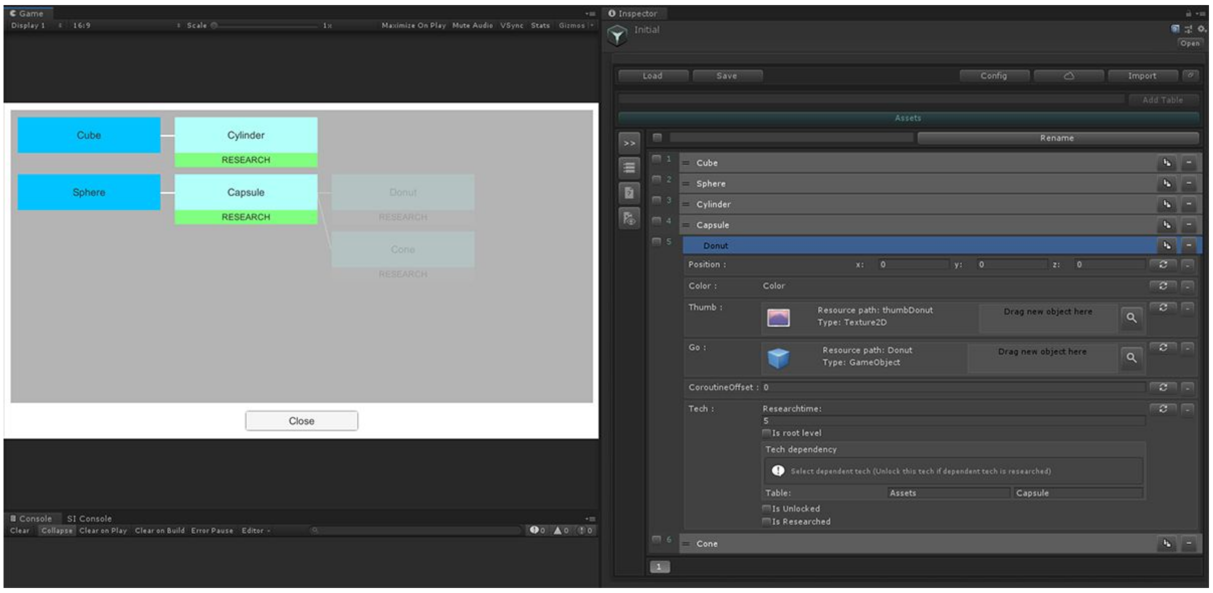
This example shows the use of multiple database objects to support saving runtime generated objects. It also has a simple techtree which demonstrates how you can use a custom data class to store techtree relevant data. The techtree UI dynamically rebuilds itself based on this data.

The general idea is as follow:

- We use the initial databox object for all assets we want to instantiate.
- We use an empty runtime databox object which will maintain all instantiated objects at runtime.
- At runtime a freshly instantiated object gets its data from the initial database and registers them into the runtime database with it's unique instance id.
- We only change the data in the runtime database.
- When loading, we iterate through all entries in the runtime database. As we have saved the type, position and color of each object we can easily re-create the saved state.


INITIAL DATABASE
Get data at Start...
GAMEOBJECT created at RUNTIME
...and register data to an empty runtime database object and use it during runtime
RUNTIME DATABASE
SAVEGAME

Techtree



- The techtree UI dynamically rebuilds itself based on the tech data. This is a great demonstration of a custom data class.
- By changing the dependency in the tech data you can change the arrangement of the techtree UI.

Scripts

AdvancedDataboxLinking.cs

- Registers the game object to the database according to objectID, fromDBId and toDBId variables.

AssignObjectValues.cs

- Assigns all values to the object or the database.

PlaceAsset.cs

- Makes sure the object moves along the mouse cursor and snaps to a grid. Also handles mouse click event and object instantiation. After user has clicked it calls the LinkToNewDatabase method in the AdvancedDataboxLinking script.

SaveManager.cs

- The save manager handles the restoring of saved objects. After the database has been loaded a coroutine is being called which instantiates all saved objects back and registers them to the existing database as a new entry while removing the old entries.

UIManager.cs

- The UI manager handles all ui inputs and dynamic UI creation. It also loads the initial database on Awake and builds the build menu ui after the initial database has been loaded.
- The techtree logic which handles the techtree UI creation and research is also integrated into this script.

05. Cloud Example

Simple cloud upload and download example. Please make sure that the Databox cloud server has been configured correctly. Setup

Using the cloud feature at runtime by script does not check which version is newer. It simply overrides the local or cloud file, depending if you use download or upload. So please use this feature with caution!

Cloud Setup

Please follow these instructions carefully to install the Databox cloud to your webserver.

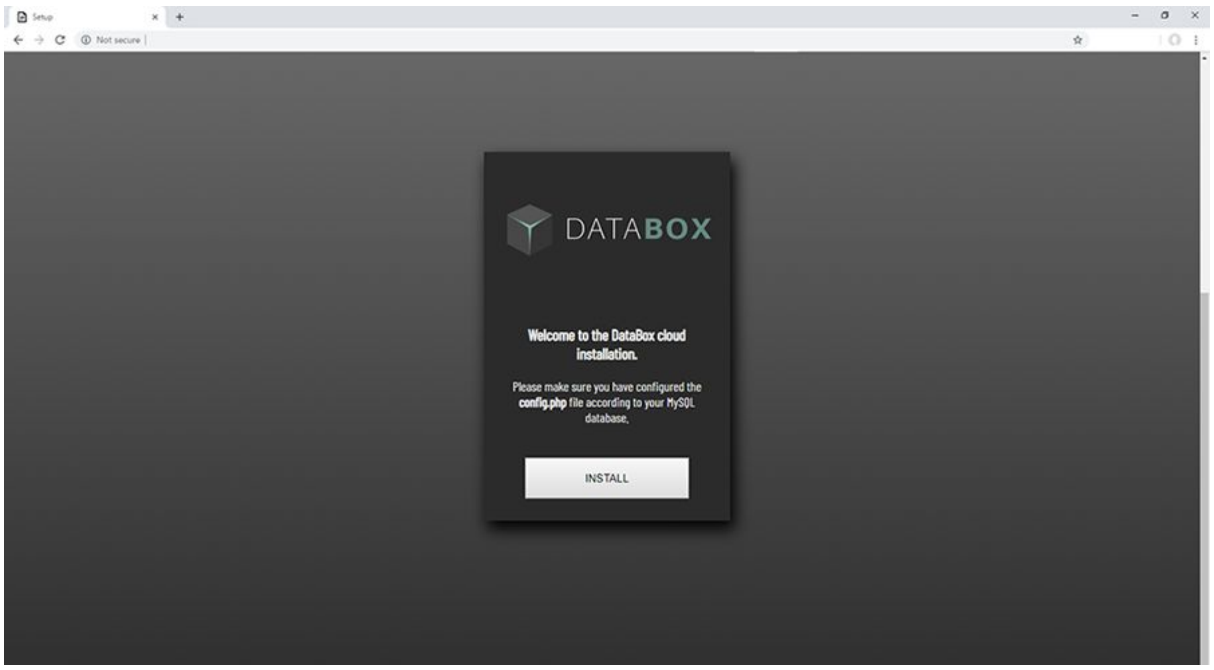
You will need a MySQL database also your server must support PHP. Please refer to your webhosting company if you are not sure how to setup a MySQL database.

Configuration

- Navigate to the CloudSerivce folder inside of the Databox unity project folder.
- Open the config.php file with your default text editing tool.
    - you will see following text:

```
$dbhost = 'localhost';
$dbuser = 'MySQL user';
$dbpass = 'MySQL password';
$dbname = 'MySQL database';
$dbtable = 'Databox';
```

    - Please add the string information according to your MySQL database setup. The dbtable name will be used during the Databox cloud setup you can leave it like this or change it.
- Save the config.php file
- Connect to your webserver via FTP and create a new folder.
- Upload all files located in the CloudService folder to your newly created folder via FTP.
- Open your default web browser and navigate to the setup.php file which is located in the newly created folder. So if you have uploaded all files to a folder called "databox", navigate to: http://mywebserver.com/databox/setup.php

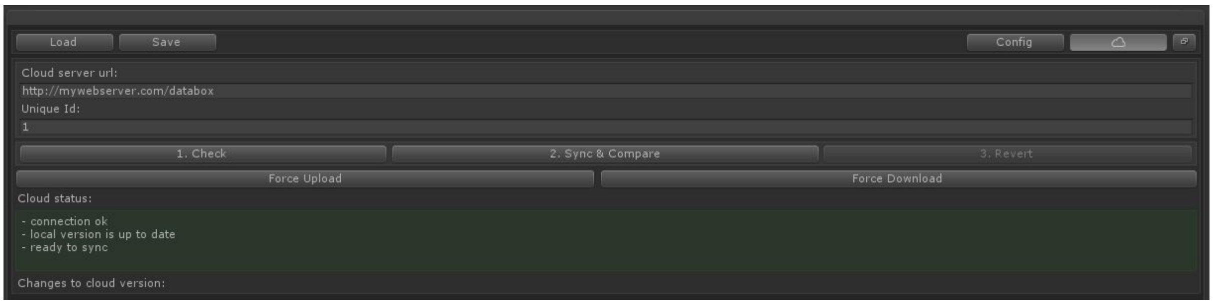


Welcome to the DataBox cloud installation.

Please make sure you have configured the config.php file according to your MySQL database.

- You should now see the databox setup screen.
- Click on install.

- Databox Cloud has been installed on your server.



Cloud server url:
http://mywebserver.com/databox
Unique Id:
1

Cloud status:
```
- connection ok
- local version is up to date
- ready to sync
```
Changes to cloud version:

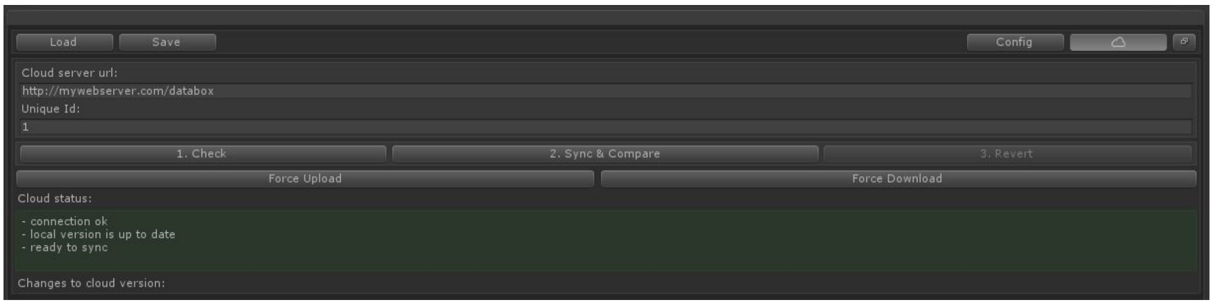
- You can now use the server url in the unity databox object. (the folder where you have uploaded all files) Example: http://mywebserver.com/databox

Unique id

- You can use the Unique id in the configuration to create different upload versions for different Databox Objects or for different team members.

Sync To cloud

It is always wise to save your file before doing a sync.



Cloud server url:
http://mywebserver.com/databox
Unique Id:
1

Cloud status:

```
- connection ok
- local version is up to date
- ready to sync
```

Changes to cloud version:

1. Check

- Before you can sync you'll need to click on the 1. Check button. This will check if the connection to the cloud is ready and which version is newer - local version or cloud version.

2. Sync & Compare

- After the check has been accomplished you can click on 2. Sync & Compare. This will sync to the newest version. You will see all changes in the output.

3. Revert

- If you don't like the changes you can then click on 3. Revert. This will revert the data back to the version before sync.

Force Upload

- Forcing an upload overrides the cloud version with your local version. No matter which version is newer!

Force Download

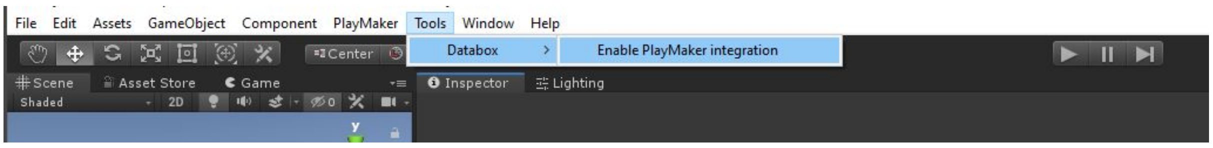
- Forcing a download overrides the local version with the cloud version. No matter which version is newer!

Warning! Only use force upload and download if you know what you do.

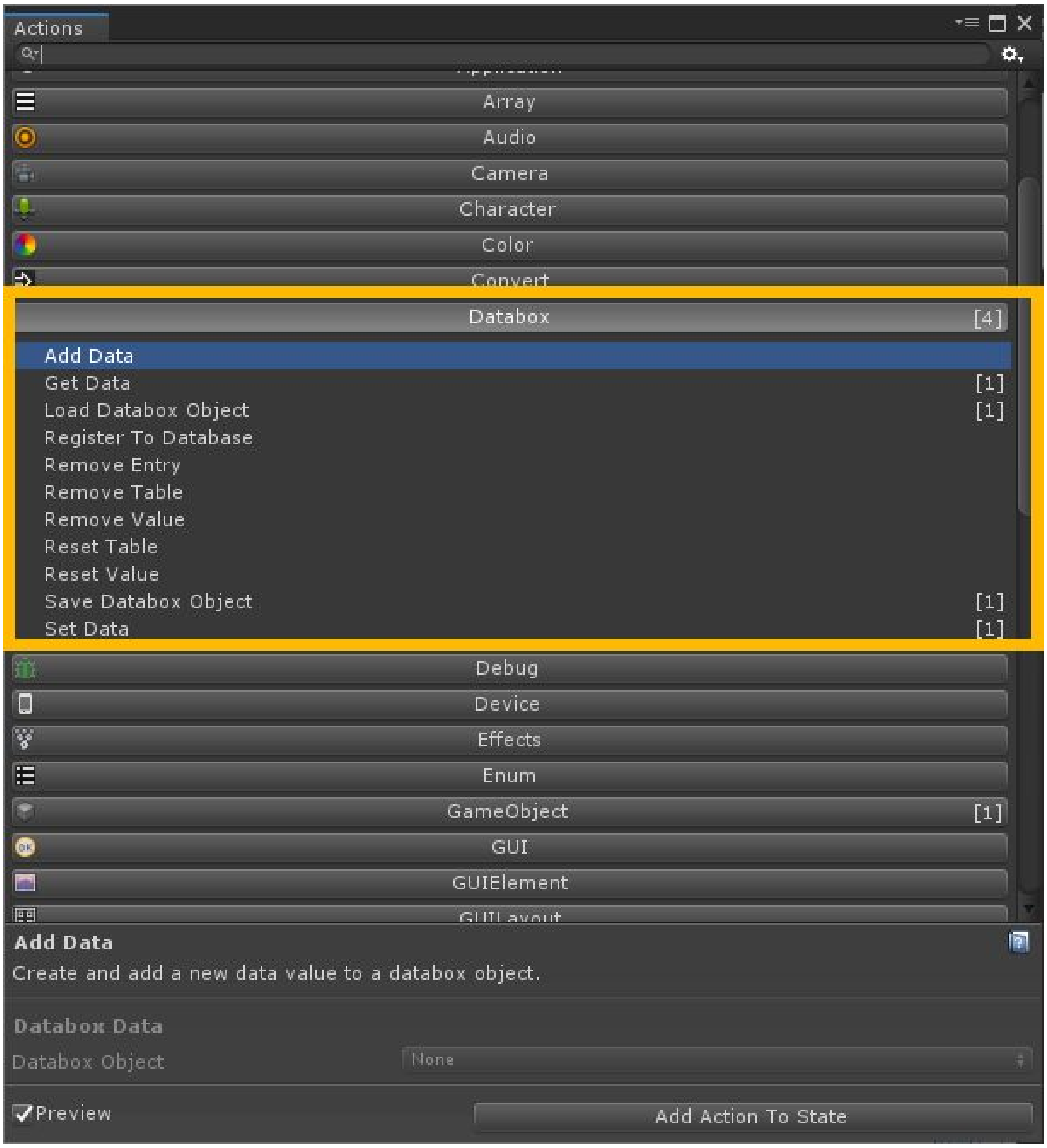
PlayMaker Setup

Since update 1.0.5 Databox officially supports PlayMaker.

To enable the Databox PlayMaker integration select *Tools -> Databox -> Enable PlayMaker integration.*

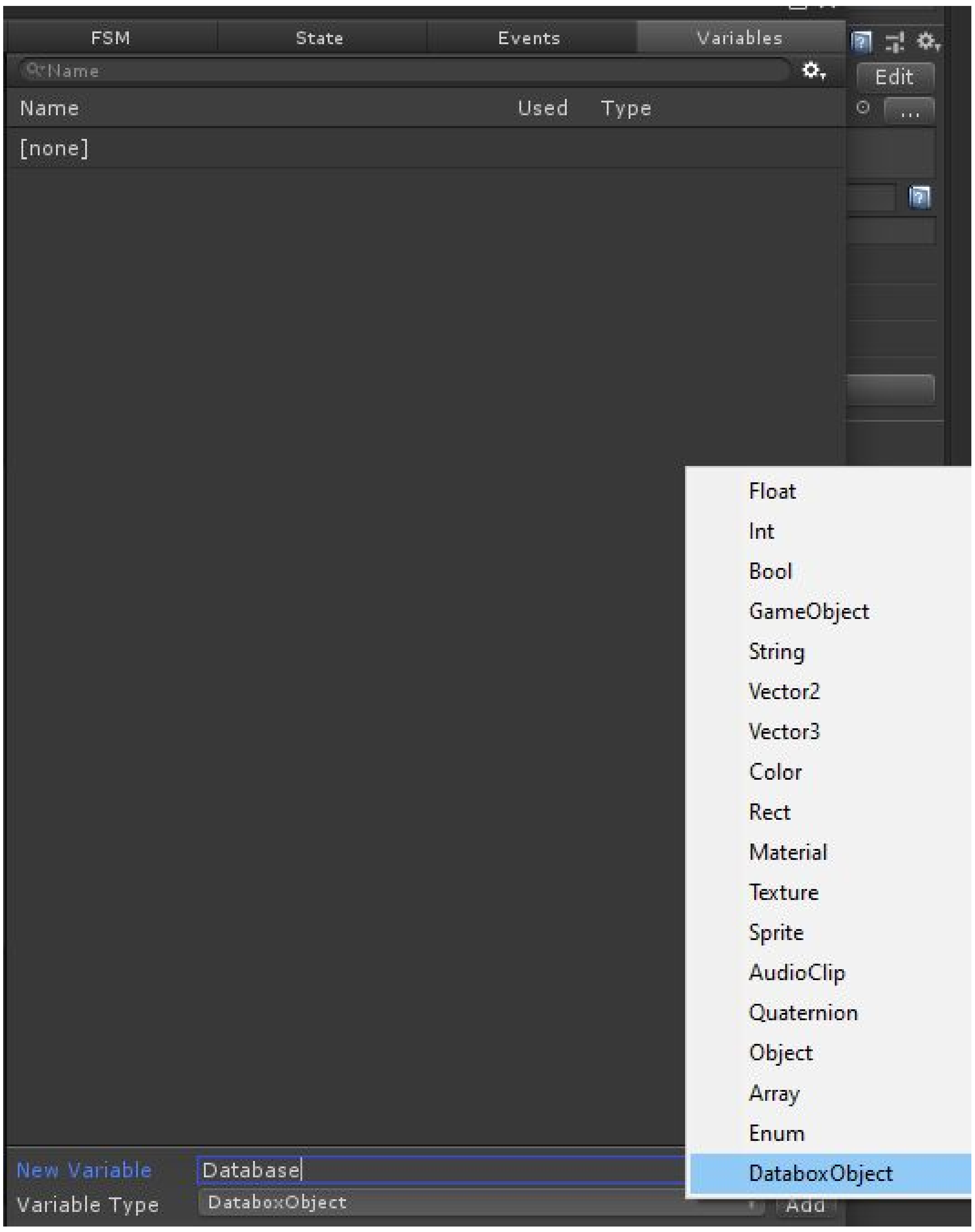


After compilation is done, all Databox-PlayMaker actions are available in the PlayMaker action browser.



Custom Databox Object variable

It is possible to create a DataboxObject variable in PlayMaker which can then be used for the Databox PlayMaker actions.



New Variable: Database
Variable Type: DataboxObject

PlayMaker Actions

- LoadDataboxObject
- SaveDataboxObject
- GetData
- SetData
- AddData
- RemoveTable
- RemoveEntry
- RemoveValue
- ResetTable
- ResetValue
- RegisterToDatabase

API

AddData

Add data to a databox object. If table and/or entry does not exist. Databox will create a new table/entry.

```
public bool AddData(string _tableID, string _entryID, string _valueID, DataboxType _data)
```

- Returns true on success

Example

```
public DataboxObject database;

FloatType _health = new FloatType();
_health.Value = 100f;

database.AddData("TableName", "EntryName", "ValueName", _health);
```

GetData

Returns the data object of type T

```
public T GetData<T>(string _tableID, string _entryID, string _valueID) where T : DataboxType
```

Example

```
public DataboxObject database;

FloatType _floatValue = database.GetData<FloatType>("TableName", "EntryName", "ValueName");
```

TryGetData

Trys to get the data of a specific type, returns true or false if succeeded.

```
public bool TryGetData<T>(string _tableID, string _entryID, string _valueID, out T _data) where T : DataboxType
```

Example

```
public DataboxObject database;
public string tableName;
public string entryName;
public string valueName;

FloatType _float;

if (database.TryGetData<FloatType>(tableName, entryName, valueName, out _float))
{
    // Return the float value
    Debug.Log(_float.Value);
}
```

GetEntriesFromTable

Returns a dictionary with all entries from a table

```
public OrderedDictionary<string, DatabaseEntry> GetEntriesFromTable(string _fromTable)
```

Example

```
public DataboxObject database;
public string tableName;

var _table = database.GetEntriesFromTable(tableName);

Debug.Log(_table.Count + " Entries in " + tableName);

// Iterate through all entries
foreach(var entry in _table.Keys)
{

}
```

GetValuesFromEntry

Returns a dictionary with all values from an entry in a table

```
    public Dictionary<string, Dictionary<Type, DataboxType>>
GetValuesFromEntry(string _fromTable, string _fromEntry)
```

Example

```
    public DataboxObject database;
    public string tableName;
    public string entryName;

    var _values = database.GetValuesFromEntry(tableName, entryName);

    Debug.Log(_values.Count + " Values in " + entryName);

    // Then we iterate through all values
    foreach ( var value in _values.Keys)
    {

    }
```

SetData

Set data to DataboxObject. Returns true if succeeded.

```
        public bool SetData<T>(string _tableID, string _entryID, string
_valueID, DataboxType _value) where T : DataboxType
```

Example

```
public DataboxObject database;

var _ok = database.SetData<BoolType>(tableID, entryID, valueID, new
BoolType(false));
```

EntryExists

Checks if an entry in a databox object exists. Returns true or false

```
        public bool EntryExists(string _tableID, string _entryID)
```

Example

```
public DataboxObject database;

   var _ok = database.EntryExists(tableID, entryID);
```

ValueExists

Checks if a value in a databox object exists. Returns true or false

```
        public bool ValueExists(string _tableID, string _entryID,
string _valueID)
```

Example

```
public DataboxObject database;

   var _ok = database.ValueExists(tableID, entryID, valueID);
```

RegisterToDatabase

Takes an existing entry and registers it to a new Databox object. This can be used if you are creating game objects at runtime. The newly created object takes the initial data information from an initial Databox object and registers itself to a runtime save game Databox object. Please see the advanced demo scene example here.

```
    public bool RegisterToDatabase(DataboxObject _dbToRegister, string
_tableID, string _entryID, string _newEntryID)
```

- **_dbToRegister**: The Databox object where we want to add the entry
- **_tableID**: The original table name
- **_entryID**: The original entry name
- **_newEntryID**: The new entry name. You should usually use a unique id for this.

Example

```
public DataboxObjectManager manager;

// Get DataboxObject from DataboxObjectManager
DataboxObject _fromDB = manager.GetDataboxObject(fromDBId);
DataboxObject _toDB = manager.GetDataboxObject(toDBId);

// Get unique Instance id from object for new entry id
int _objectId = this.gameObject.GetInstanceID().ToString()

_fromDB.RegisterToDatabase(_toDB, "TableName", "EntryName",
_objectId);
```

LoadDatabase

Loads the saved databox object

```
public void LoadDatabase()
```

-

With custom file name.

```
public void LoadDatabase(string _fileName)
```

LoadDatabaseAsync

Loads the saved databox object asynchronously

```
public void LoadDatabaseAsync()
```

-

With custom file name.

```
public void LoadDatabaseAsync(string _fileName)
```

Example:

```
public DataboxObject data;

StartCoroutine(data.LoadDatabaseAsync());
```

SaveDatabase

Saves the databox object to a json file

```
public void SaveDatabase()
```

-

With custom file name.

```
public void SaveDatabase(string _fileName)
```

SaveDatabaseAsync

Saves the databox object to a json file asynchronously

```
public void SaveDatabaseAsync()
```

-

With custom file name.

```
public void SaveDatabaseAsync(string _fileName)
```

Example:

```
public DataboxObject data;

StartCoroutine(data.SaveDatabaseAsync());
```

UploadToCloud

Uploads the database from the appropriate DataboxObject to the cloud.

Uploading overrides the complete file stored in the cloud.

```
public void UploadToCloud()
```

DownloadFromCloud

Downloads the database from the cloud by using the settings of the appropriate Databox object

```
public void DownloadFromCloud()
```

RemoveDatabaseTable

Remove a complete table from the databox object

```
public void RemoveDatabaseTable(string _tableName)
```

RemoveEntry

Remove specific entry from table

```
    public void RemoveEntry(string _tableName, string _entryName)
```

RemoveValue

Remove value from entry

```
    public void RemoveValue(string _tableName, string _entryName,
string _valueName);
```

Events

OnDatabaseLoaded

Is being called after database has been loaded

```
public DataboxEvents OnDatabaseLoaded;
```

OnDatabaseSaving

Is being called as soon as saving the databox object starts

```
public DataboxEvents OnDatabaseSaving;
```

OnDatabaseSaved

Is being called after successfully saving the database

```
public DataboxEvents OnDatabaseSaved;
```

OnDatabaseCloudDownloaded

Is being called after the database has been downloaded

```
public DataboxEvents OnDatabaseCloudDownloaded;
```

OnDatabaseCloudDownloadFailed

Is being called after database download failed

```
public DataboxEvents OnDatabaseCloudDownloadFailed;
```

OnDatabaseCloudUploaded

Is being called after the database has been uploaded

```
public DataboxEvents OnDatabaseCloudUploaded;
```

OnDatabaseCloudUploadFailed

Is being called after database upload failed

```
    public DataboxEvents OnDatabaseCloudUploadFailed;
```

Databox Type

Reset

resets the value back to it's initial value. When creating a custom class you will need to add the reset functionality to it. See the Create a custom class Example here

```
public virtual void Reset(){}
```

OnValueChanged

This event is being called as soon as the appropriate value has changed.

```
public ValueChanged OnValueChanged;
```

Example

```
public void OnEnable()
{
    FloatType health = database.GetData<FloatType>("Table", "Entry", "Value");

    health.OnValueChanged += ValueChanged;
}

public void ValueChanged(DataboxType _value)
{
    Debug.Log("Value has changed")
}
```

Databox object manager

GetDataboxObject

Returns the Databox object according to it's id specified in the DataboxManager.

```
public DataboxObject GetDataboxObject(string _dbName)
```

Databox UI Binding

Bind

Binds a Unity UI component at runtime to a Databox object.

```
public void Bind(DataboxObject _databoxObject, string _tableID, string _entryID, string _valueID)
```

-

```
public void Bind(string _tableID, string _entryID, string _valueID)
```

-

```
//Passing the databaseID to the method will use the DataboxManager script to retrieve the databox object
public void Bind (string _dbID, string _tableID, string _entryID, string _valueID)
```

- **_databoxObject**: The Databox object which should be used
- **_dbID**: The database id assigned to DataboxManager
- **_tableID**: The name of the Databox table
- **_entryID**: The entry name
- **_valueID**: The value name