Welcome to Balancy

Balancy is an essential extension for any game or app. Instead of using ancient solutions, you can spend several minutes learning what Balancy does and how it works. Which in results will save you weeks and months of time in the future.

We've spent more than a decade in Game Dev Industry, creating tons of different games, so we know the problem you are facing right now. Our main goal is to prevent game developers from reinventing the wheel, developing the same technology over and over with each new game. That's why our solution is super easy to use and it does most of the work on your behalf.

Our core features help you to organise and to deliver Game Data, save your players' progress and validate in-app purchases. Many more features are coming.

Our servers are designed to run at massive scale, and scale automatically with your user base. We have your back, so we expect from you to focus on the game itself and make it awesome.

Watch Video

Video Tutorial

Join Our Discord

Out-of-the-box

Integration of Balancy is extremely easy and takes not longer than 10 minutes. Instead of wasting your time on implementing features which any other game already has, focus your energy on creating the best game!

Data Editor

Create and edit your game data in the most convenient way. Balancy will automatically deliver that data to your game.

Cloud Storage

Save and load the progress of your players using our reliable servers. You can forget about any headache in this area.

Payments

Implement InApp purchases with only one simple line of code. We'll take care about all the logic and validation.

How does it work?

Balancy is a social Network for players and game developers. It's available at the address: https://unnynet.com/. When you add Balancy to your game, we create a WebView, which displays a mobile version of our website, so you don't need to create any single window. We provide our Servers and customizable Client UI for developers, making it a plug and play solution.

Adding a Game to Balancy is similar to creating a server in Discord. Any player can find your Game and join it while using the browser version of Balancy. Screenshot When a player connects to Balancy inside of your game, his access is limited to only your game channels. It's obvious that you don't want players to talk in channels of other games while playing yours. Screenshot However players still have access to their friends and they can talk to them no matter which game they are playing. Which is actually good, because your players don't have to switch to WhatsApp/Facebook to reach their friends and they also can invite friends to join your game.

Pricing

Balancy is a SaaS with monthly payments. We are tying to keep our pricing as low as possible without any quality loss. Currently we accept PayPal payments, but we can discuss other options if necessary.

We offer 3 plans for you to choose from: Indie, Startup and Publisher. When you create an account the default plan is **Startup**. If you are eligible for Indie plan, please reach us our by email and send us some information about you to prove that. If your game surpass 1MM MAU, you'll be automatically switched to the Publisher plan. If you decide you need Publisher plan earlier, you can also reach us by email.

We charge per Game Title and each game have it's own payment deadline, depending on it's creation date.

Indie

Only new companies are eligible, who had total in revenue and investment less than $100k. We would love to support new businesses, providing you with 10,000 free MAU (without any time limits). It'll be enough for the whole development cycle and even a small Soft-Launch.

- Free up to 10,000 MAU

Startup

Perfect plan to start your business

- Free Trial Period. It lasts one month or if your reach 10,000 MAU
- $499 per month
- Up to 4h/month of dedicated personal support
- Up to 1MM MAU

Publisher

If you are an experience publisher, who values the time and you are confident with your project.

- Free Trial Period. It lasts one month or if your reach 10,000 MAU
- $999 per month
- Up to 10h/month of dedicated personal support
- Unlimited MAU

Pay as you go

Besides of the flat fee we charge additionally for the bandwidth usage.

- first 10Gb are included in each plan
- $0.2 per additional Gb

All the game balance is split into many json files (one for each Template) and the data is being delivered to your clients when they launch your game. If your make any changes in the balance, only the changed Json files will be delivered. The size of each file multiplied by the time it was downloaded from our servers is the bandwidth we charge for. This fee is not something your should worry about, we just added it to make sure our business is profitable for any kind of games we serve.

Available features

All plans have access to all the features, however the different limits might be applied:

- Data Editor
- Authentication
- Cloud Storage
- Payments

Release Notes

v1.1.0 - April 1, 2021

- Payments added
- Unity Addressables can be synchronized with the Data Editor
- Environments
- Many UX/UI improvements

v1.0.1 - November 7, 2020

- Enumerable types were added
- Many UX/UI improvements

v1.0.0 - October 7, 2020

- First release

Welcome to Balancy

Video Tutorials

I’m planing to create a simple Collection Card Game during my tutorials. I'll guide you through each step required from importing the plugin to using the data in the game.

1. Import Balancy
2. Player Authentication
3. Basic Work with Data Editor
4. Code generation and work with Data in Unity

Authorization

The list of method for authentication will be updated and for some platform we'll try to automate this process.

Email

```
Balancy.Auth.WithEmail(<email>, <password>, authResponse =>
{
    Debug.Log("Authorized " + authResponse.Success);
    if (authResponse.Success)
        Debug.Log("User id: " + authResponse.UserId);
});
```

Name and Password

```
Balancy.Auth.WithName(<username>, <password>, authResponse =>
{
    Debug.Log("Authorized " + authResponse.Success);
    if (authResponse.Success)
        Debug.Log("User id: " + authResponse.UserId);
});
```

As Guest using Device ID

```
Balancy.Auth.AsGuest(authResponse =>
{
    Debug.Log("Authorized " + authResponse.Success);
    if (authResponse.Success)
        Debug.Log("User id: " + authResponse.UserId);
});
```

How to start

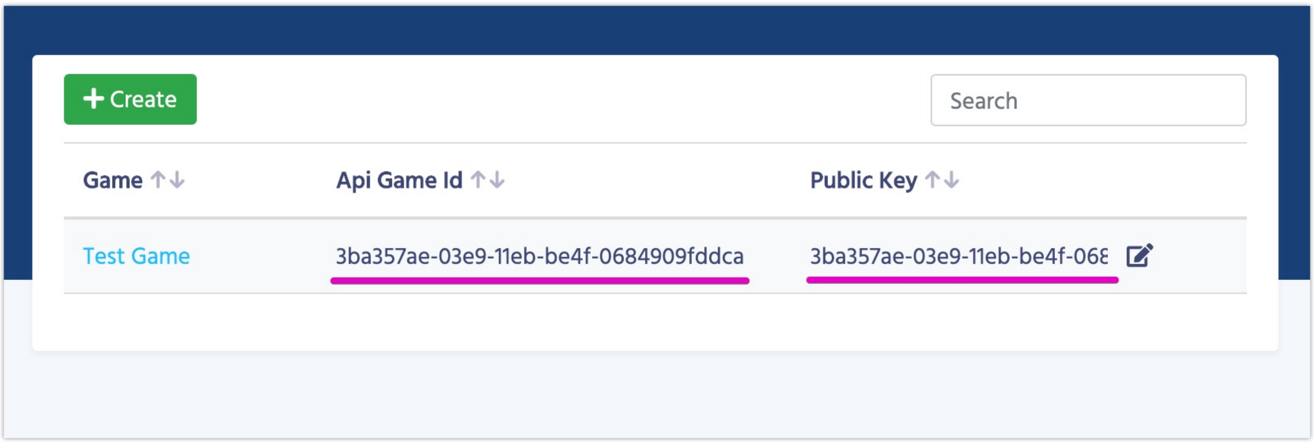
1. Create an account at Balancy
2. Create a new game
3. Select your platform and follow the instructions:
    a. Unity.

If you are looking for any other platform, please contact us to let us know your interest.

Unity3D Integration

For your convenience we've recorded the video of the integration

1. Download the latest version of the plugin from the Asset Store.
2. Import the Balancy plugin.
3. Prepare Game ID and Public Key to use in the code:



Game	Api Game Id	Public Key
Test Game	3ba357ae-03e9-11eb-be4f-0684909fddca	3ba357ae-03e9-11eb-be4f-068

4. Call initialize method at start:

```
Balancy.Main.Init(new Balancy.AppConfig {
    ApiGameId = YOUR_GAME_ID,
    PublicKey = YOUR_PUBLIC_KEY,
    Environment = Balancy.Constants.Environment.Development,
    OnReadyCallback = responseData => { Debug.Log("Balancy Initialized: " +
responseData.Success); }
});
```

Further reading

Balancy consists of several modules for your convenience.

1. **Auth** - authorizations
2. **Data Editor** - a place to work with Data Editor
3. **Storage** - save/load in-game data with the server
4. **Payments** - purchase In-App and get information about them
5. **Localization** - (coming soon) a place to work with localizations

Payments

1. Open **Platforms** section and add all the information about the Platforms, where the game is available. It's required to validate the purchases.
2. Open **Products** section and fill all the information about the Products your game has. In most cases you just need the main table, however if you have different ProductId, Name or Price for different platforms, you might want to use **override** section for each of the platforms.
3. Get List Of Products:

```
Balancy.Payments.GetProducts(productsResponse =>
{
    Debug.Log("Products Received " + productsResponse.Success);
    if (productsResponse.Success)
        Debug.Log("Products Count: " + productsResponse.Products.Length);
});
```

4. Make purchase:

```
Balancy.Payments.PurchaseProduct(<product_id>, doneCallback =>
{
    Debug.Log("Purchase was made " + doneCallback.Success);
});
```

Payments

Balancy offers a one line purchase system for all the platforms. The list of available platforms is available in your Admin Panel and it'll be updated.

Add purchases information to Balancy platform:

//TODO

Prepare Unity

Please follow unity instructions to activate payments. You just need to implement everything before the section **Making a Purchase Script** - we have already did the rest for you. https://learn.unity.com/tutorial/unity-iap#5c7f8528edbc2a002053b46e

Make Purchase

```
Balancy.Payments.PurchaseProduct(<balancy_purchase_id>, doneCallback);
```

Get list of available Purchases

```
Balancy.Payments.GetProducts(doneCallback);
```

This can be used if you want to display the list of available purchases with the prices. But we would recommend to create your own StoreItems system with Data Editor. It'll be more flexible.

Get list of already purchased products:

```
Balancy.Payments.GetPurchases(Balancy.Constants.PurchaseStatusFilter.Claimed, doneCallback);
```

Restore Purchases

Sometimes something can go wrong, for example a player had purchased a product, but the game crashed and he didn't get the item. We have prepared a method which might help in such situations.

```
Balancy.Payments.RestorePurchases(doneCallback);
```

We recommend to call it every time a player opens your Store Window.

Storage

Allows you to Save and Load any data on Balancy servers. Make sure to authorize the player (at least as a guest) before using the Storage, because all records are connected to specific players.

General Information

Collections and Keys

Each player can have a set of Collections, and each collection can have a set of Keys. When you save to the Storage you need to specify both a Collection and a key. However, to Load the data there are two options:

1. Load a specific key from a specific collection
2. Load the whole collection.

You can think of a Collection as a Book, where a Key is a chapter of that book. You can have as many books as you want with many different chapters in each one. It's up to you what you want to write in each chapter.

Both Collection and Key are string values. Just make sure you are loading from the same Collection/Key pair, which you used to save your data.

For example

You might have a collection 'Player' with many keys: 'Inventory', 'Spells', 'Stats', etc.. When one of the data changes, you just need to update only a small portion, which will be stored in one key. But when you start a game you can load the whole profile with all the keys

Versions

Another worth mentioning topic is **versions**. Balancy handle them automatically, but it would be better for you to understand how it works. Each record in database has a version number, which increases every time you change the data. It's used to prevent using an out-dated data.

For example

You launch a game on the **Device1**, play for some time and the last version of your progress will be **5**. Then you switch to the **Device2**, which loads progress with version **5**, saves several changes, making the last version of the progress **10**. Finally you reopen the game on the **Device1**, where the client still has version **5**. The next time it tries to save an update, it'll get an error of a wrong version.

There are two ways to solve such issue:

1. Ignore version (we have such parameter in the Save method). I would not recommend this option, in most cases this is wrong.

2. If you get such error, reload your game progress to get the latest game data and version, apply it to your game and keep playing.

Data types

We support 2 options:

1. Simple **string**
2. Any Class.

We are using Newtonsoft converter to Serialize and Deserialize objects, here is an example of the class we can use to Save/Load in the Storage :

```
private class SaveExample
{
    public int IntValue;
    public string StringValue;

    [JsonIgnore]
    public int IgnoredValue;

    [JsonIgnore]
    public int AlsoIgnoredValue { get { return IntValue; } }
}
```

In the example above **IntValue** and **StringValue** will be saved and loaded, when other 2 field won't. So if you don't want any property or attribute to be synchronized, just mark it with [JsonIgnore].

Save

Example to save a string:

```
Balancy.Storage.Save("MyCollection1", "StringKey", "Hello World!", saveResponse =>
{
    Debug.Log("String was saved result: " + saveResponse.Success);
});
```

Example to save an object:

```
var saveExample = new SaveExample
{
    IntValue = 5,
    StringValue = "Hello World",
    IgnoredValue = 3
};

Balancy.Storage.Save("MyCollection2", "ObjectKey", saveExample, saveResponse =>
{
    Debug.Log("Object was saved result: " + saveResponse.Success);
});
```

Load

Example to load a saved string:

```
Balancy.Storage.Load<string>("MyCollection1", "StringKey", loadResponse =>
{
    Debug.Log("String was loaded result: " + loadResponse.Success);
    if (loadResponse.Success)
        Debug.Log("Value = " + loadResponse.Data.Value);
});
```

Example to load a saved object:

```
Balancy.Storage.Load<SaveExample>("MyCollection2", "ObjectKey", loadResponse =>
{
    Debug.Log("Object was loaded result: " + loadResponse.Success);
    if (loadResponse.Success)
    {
        SaveExample saveExample = loadResponse.Data.Value;
        Debug.Log("String Value = " + saveExample.StringValue);
    }
});
```

Limits

1. You can Save/Load the same combination of Collection & Key not more than once per 20 seconds.
2. The maximum size of a saved value is 100Kb.

It's subject to change in the future.

Data Editor

Data Editor (DE) is an essential part of Balancy. It is used for creating data structure and editing the data. Balancy automatically delivers the newest data to the app and parse it to the convenient auto-generated code, so you could easily access it.

How Does it Work?

1. After adding your game to the platform, you get access to the DE.
2. Inside of DE you can add all types of objects your game has: weapon, item, construction, monster, hero, location, etc...
3. For each type of object you can add as many documents as possible. Each document represents a unique weapon, item, construction, etc...
4. Open your project in Unity and start code generation request. Balancy will automatically generate the code, based on the data you provided.
5. Once the game is launched, all the game data is delivered to the game and already mapped to the generated code.
6. Your programmer has direct access to all the items, weapons and other objects your game has. He doesn't have to write any code for downloading or parsing.
7. All changes in DE will be automatically synchronized with the game on launch. You just need to deploy changes.

We'll explain every step in details in the next articles.

Next: Important Terms

Code Generation

Why do I need it?

It's a good question, because many developers like to rewrite such things in every project.

Here are some reasons for you to consider:

1. Why would you spend your precious time on such a monotone job? Such things should've been automated long time ago.
2. Human mistakes excluded. Very often a game designer or a programmer misspell a word and parsing doesn't work as expected. Such problem might take some time to be found.
3. Whenever a game designer changes a parameter or a Templates, all of that will be instantly reflected in the code after the generation. A programmer won't forget to apply those changes.
4. Code generation is tightly connected with other Balancy awesome features, like: Localization. Using them all together gives your team a huge boost.
5. You can forget about JSONs and how to parse them. Balancy will do that for you, so you could work with convenient Classes only.
6. When document refers to another document, developers usually use some kind of ID to create those links. Balancy will do that for you, it automatically resolves all links and gives you direct access to the Documents you expect.

How to generate code

1. In Unity select Tools->Balancy, input your GameId/PublicKey and start Code Generation.
2. It might take some time depending on your connection and the amount of Templates you have.
3. Generated classes will be placed in Assets/Balancy/AutoGeneratedCode.
4. Please **DO NOT** change anything in this folder, because your changes will be overwritten with the next generation.

How does it work inside?

Balancy server generated JSON files based on your Documents and puts it to the CDN storage. Balancy plugin automatically checks for the updates of those JSON files and downloads only updated files. After that it parses the data from JSON to the generated classes and find all dependencies. The programmer doesn't have to download, parse or understand any of JSONs. There is also no need to find any links if any of the Documents refers to another one - the link will be automatically resolved by Balancy, so you would get a direct access to the Documents you expect.

How to get an access to the documents

1. Let's say you have a Template **ItemModel** and several Documents of this Template. Use **Balancy.DataEditor.ItemModels** to access the list of those Documents.
2. Let's say you have a Template **RecipeModel**, which has a Parameter Item of type **ItemModel**. Once you get an instance of that that RecipeModel as **recipeModel**, you can get an access to it's Item as **recipeModel.Item**. As simple as that!
3. If your Template **GameConfig** is a Singleton, you get access to it by writing **Balancy.DataEditor.GameConfig**.

How to work with the generated code

Every developer has his own style and it's really up to you how to work with the game data. Below we just list you couple of example, which we would use:

1. Extension Method

Let's say you have a generated code for items:

```
public class ItemModel : BaseModel
{
    public string Name;
    public string Description;
    public string IconName;
    public string AssetName;
    public int MaxStack;
}
```

If you want to add a check whether an Item can be stacked in inventory, you can write in a separate file:

```
public static class ItemModelExtension
{
    public static bool CanStack(this ItemModel item)
    {
        return item.MaxStack > 1;
    }
}
```

So later if you have an instance of **ItemModel** as **item**, you can just call the new method:

```
item.CanStack();
```

2. Facade Pattern

Let's say you have a generated code for inventory:

```
public class Inventory : BaseData
{
    public SmartList<ItemSlot> ItemSlots;
}
```

First you create a new class called InventoryController, which has property Inventory in it. Instead of working with Inventory directly, your game has an access only to the InventoryController. So if you want to add a check if there are any empty slots, you can write:

```
public class InventoryController
{
    public Inventory Inventory;

    public bool HasEmptySlot() {
        foreach (var slot in Inventory.ItemSlots)
        {
            if (slot.IsEmpty())
                return true;
        }
        return false;
    }
}
```

This is just a simple example. This very logic can be easily rewritten using the first approach (Extension Method), which would be recommended for this situation.

3. Partial Classes

This feature was deactivated temporary.

If you activate partial checkbox for your Template in the CMS, the generated class will be marked as partial. It means that you can add as many methods and properties for this class in a separate file as you want.

Recommendation:

The first approach is the best one. It definitely suits all simple Templates (which don't have any references as parameters). For more complicated Templates you should use combinations of the first and the seconds approaches. The last approach of Partial classes might look attractive, but it's not recommended to use. Many developers don't like this feature for many reasons. The same can be implemented using the first or the seconds approaches.

Of course there a lot of other ways to implement such logic. If you personally like one, please share it with us, and we might add it to the documentation.

Next: Example

Deploy

Whenever you change a Template or a Document in DE, it's saved on our servers, but not yet available for your end users. You can compare all the changes you make in DE with commits in GIT, with only difference that other DE users can see your commits.

When you are ready to publish (like Push in git) your changes for the end users, you need open **Deploy** section in DE and click on **Deploy** button. The process takes several seconds, and once it's completed, you can launch your game in Unity and get all the updated data.

Bear in mind that your are pushing only for the active Environment. If you want to push the changes to the Stage or Production environments, please use Environment page for Data Migration.

Versions

Before you deploy the new data, you can set a minimum version for your game and data: Screenshot

If your game version (Unity->Player Settings->Version) is lower than any of 2 versions, the new data will not be downloaded.

Min version to Launch Game

This version usually used to let your players know about a new game update available. It's up to the developer to implement all the logic, Balancy will only return an error in Init Method. This is how you can check if the game version is outdated (after you init Balancy):

```
private bool IsVersionOutdated()
{
    var errors = Controller.GetErrors();
    foreach (var err in errors)
    {
        switch (err.Error.Code)
        {
            case Errors.GameVersion:
                return true;
        }
    }

    return false;
}
```

Min version to Update Data

This version is usually used, when Templates in DE were dramatically changed, thus old clients won't be able to parse new structure correctly.

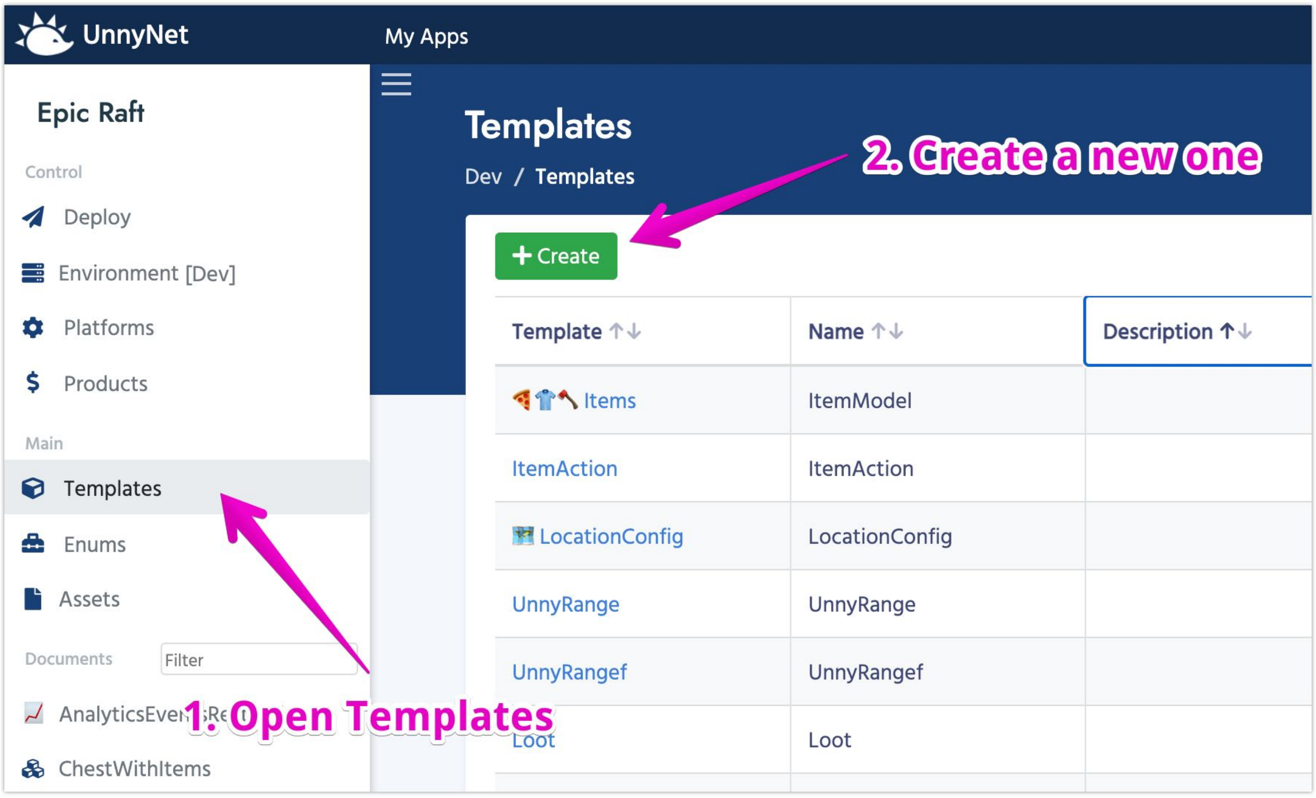
Data Editor usage Example

In this example I'm going to show you how you can create a simple craft logic for your game. We gonna need just several Templates: Item, Recipe and Ingredient. I assume you've read and implemented the basics for your game.

Here is the video of this tutorial.

Templates preparation

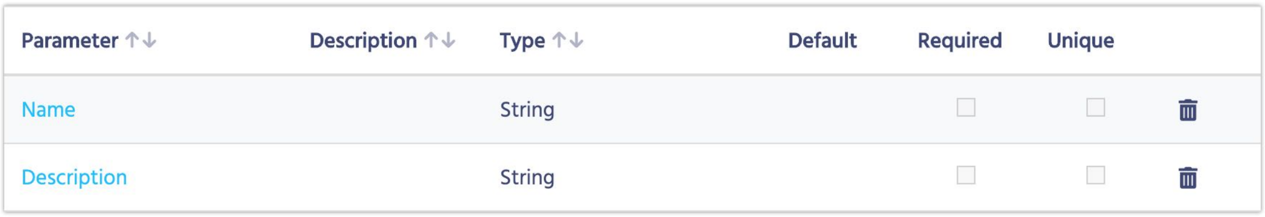
1. Open Data Editor
2. Select Templates Section and click on Create Template



Template	Name	Description
Items	ItemModel	
ItemAction	ItemAction	
LocationConfig	LocationConfig	
UnnyRange	UnnyRange	
UnnyRangef	UnnyRangef	
Loot	Loot	

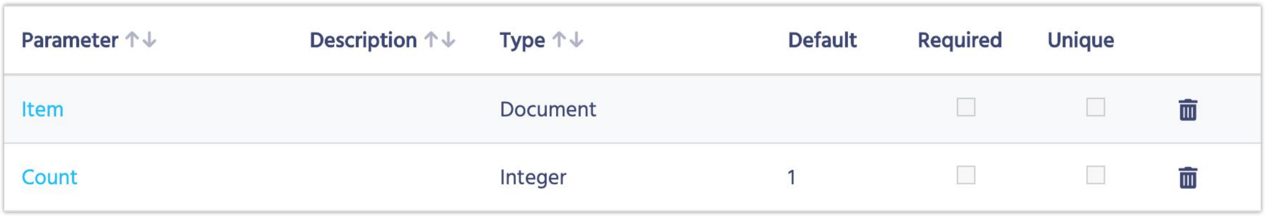
3. Set **Name** as "Item" and leave other fields as default.
4. Add following parameters:
   - **Name**: (Type: String, Use in display name - YES)
   - **Description**: (Type: String)

- When you done, the list of parameters should look like this:



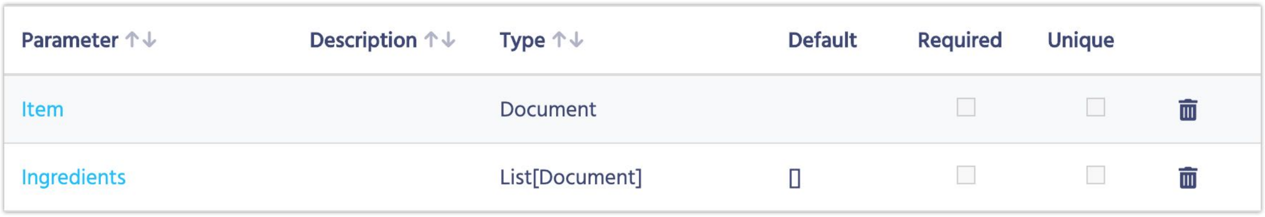
Parameter ↑↓	Description ↑↓	Type ↑↓	Default	Required	Unique	
Name		String		☐	☐	🗑
Description		String		☐	☐	🗑

5. Create another Template for Ingredient. We'll make it Component for the convenience.
   - **Name**: Ingredient
   - **Type**: Component
6. Add following parameters:
   - **Item**: (Type: Document, Reference Template: Item)
   - **Count**: (Type: Integer, Default value: 1)
   - When you done, the list of parameters should look like this:



Parameter ↑↓	Description ↑↓	Type ↑↓	Default	Required	Unique	
Item		Document		☐	☐	🗑
Count		Integer	1	☐	☐	🗑

7. The last template we gonna need is Recipe. Set **Name** as "Recipe" and leave other fields as default.
8. Add following parameters:
   - **Item**: (Type: Document, Reference Template: **Item**)
   - **Ingredients**: (Type: List, List Type: Document, Reference Template: **Ingredient**)
   - When you done, the list of parameters should look like this:



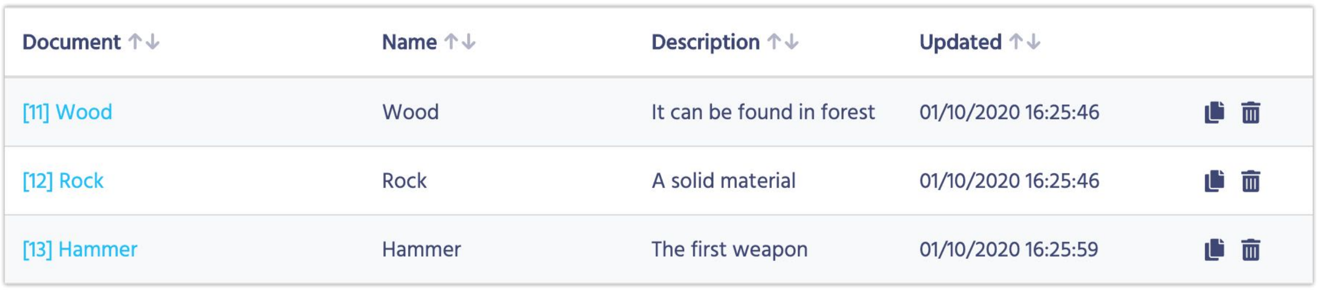
Parameter ↑↓	Description ↑↓	Type ↑↓	Default	Required	Unique	
Item		Document		☐	☐	🗑
Ingredients		List[Document]	[]	☐	☐	🗑

Create documents

Now, as we have our Templates ready, we need to add couple of actual items and recipes. When you added the Templates, you should've noticed that in the navigation (on the left) two new section appeared: Items and Recipe. Ingredient doesn't show up because it's a Component and exists only within another Document.

1. In the Navigation Panel select **Items**
2. Click on the **Create** button to add a new Item

3. Create 3 items as shown below:



Document	Name	Description	Updated	
[11] Wood	Wood	It can be found in forest	01/10/2020 16:25:46	
[12] Rock	Rock	A solid material	01/10/2020 16:25:46	
[13] Hammer	Hammer	The first weapon	01/10/2020 16:25:59	

Now we are going to make a recipe to create a Hammer using a Rock and a Wood:

1. In the Navigation Panel select **Recipe**
2. Click on the **Create** button to add a new Recipe
3. Select Item parameter as Hammer
4. Add 2 ingredients: One Wood and one Rock
5. It means that in order to create a Hammer, you need to spend one Wood and one Rock

Ok, we set all the data we need to the test project. Now we just need to publish our changes.

Deploy

1. Select Deploy section in the Navigation Panel
2. Click **Deploy** and wait couple seconds until it's done

Every time you make any changes in the data, it'll be saved only inside of the Editor. If you want to push the changes to the game, you can to do that in the Deploy Section. As a programmer, you can think of it as commit/push in GIT system.

Code Generation

1. Open Unity Project
2. Import Balancy plugin
3. Open editor window Tools->Balancy
4. Input your GameId and PublicKey
5. Click on **Generate Code**. It might take several seconds
6. Once it's done, you can find a new folder with several scripts created at /Assets/Balancy/AutoGeneratedCode
7. Don't change anything in that folder. The changes will be lost with the next generation

Balancy Initialization

You should read more about Balancy initialization here

1. Create a new script DataEditorExample.cs
2. Create a new scene and add this script to the Main Camera.
3. In the Start method write the following code:

```
Balancy.BalancyNewInit.Init(new Balancy.AppConfig
{
   ApiGameId = YOUR_GAME_ID,
   PublicKey = YOUR_PUBLIC_KEY,
   Environment = Balancy.Constants.Environment.Development,
   OnReadyCallback = responseData =>
   {
       Debug.Log("Balancy Initialized: " + responseData.Success);
   }
});
```

Validate Data

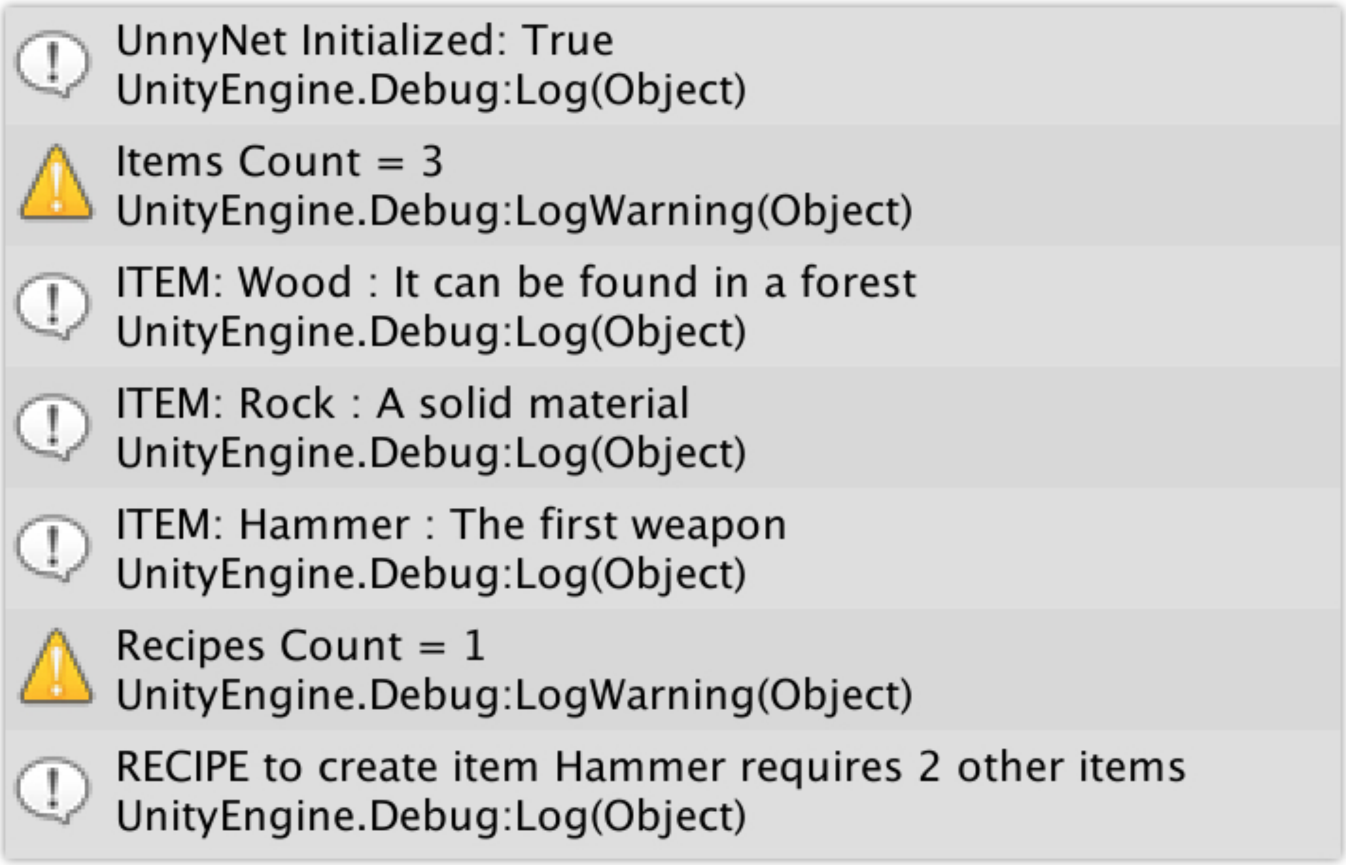
First, lets check if we receive correct data from the Editor. Add those methods to our script:

```
private void PrintAllData()
{
    PrintItems();
    PrintRecipes();
}

private void PrintItems()
{
    var items = Balancy.DataEditor.Items;
    Debug.LogWarning("Items Count = " + items.Count);
    foreach (var item in items)
        Debug.Log("ITEM: " + item.Name + " : " + item.Description);
}

private void PrintRecipes()
{
    var recipes = Balancy.DataEditor.Recipes;
    Debug.LogWarning("Recipes Count = " + recipes.Count);
    foreach (var recipe in recipes)
        Debug.Log("RECIPE to create item " + recipe.Item.Name + " requires " +
recipe.Ingredients.Length + " other items");
}
```

Now we just need to call **PrintAllData** once Balancy was initialized. Call this method after the Log line:

```
Debug.Log("Balancy Initialized: " + responseData.Success);
PrintAllData();
```

Launch the game and you should see the following logs in the console:



```
UnnyNet Initialized: True
UnityEngine.Debug:Log(Object)
Items Count = 3
UnityEngine.Debug:LogWarning(Object)
ITEM: Wood : It can be found in a forest
UnityEngine.Debug:Log(Object)
ITEM: Rock : A solid material
UnityEngine.Debug:Log(Object)
ITEM: Hammer : The first weapon
UnityEngine.Debug:Log(Object)
Recipes Count = 1
UnityEngine.Debug:LogWarning(Object)
RECIPE to create item Hammer requires 2 other items
UnityEngine.Debug:Log(Object)
```

Write Craft Logic

The next step is to write some logic to store the items you have and spend items to make a craft items. I'll provide the whole code listing, so you could investigate it. Keep in mind that this example was made super easy on purpose. For the real project you would definitely want to organize the code better. And don't forget about SOLID principles. Good luck with coding!

DataEditorExample.cs

Offline Games

If your game is offline, you might assume that the game could be launched the first time without an access to the internet. In such situation you can't rely that the game balance will be delivered to the build.

1. Open editor window Tools->Balancy
2. Click on **Download Data**
3. All the latest game data from Editor will be downloaded and put into /Assets/Balancy/Resources
4. If your game is launched without an internet access, it'll use the data from the resources

5. Once internet is available the data will be automatically updated if necessary

Next: What's next

What's next

Currently our Data Editor is becoming the central tool of Balancy. We are planning to improve it dramatically in the future and create awesome connections with other tools we have. Here is a brief plan for our development:

1. **SmartObjects** - they are very similar to Documents, but used mostly to keep track of the player's progress. There will be a simple method like: LoadSmartObject(), which will load or create the player's progress, automatically track all the changes you make and synchronize it with our server.
2. **Localization** - will be integrated into our DE. We'll provide a new type, similar to the **string**, but it'll be hold the localization key. All the values for all the languages are also stored in our DE and automatically delivered to the game as any other game content. The cool feature is that our code will automatically detect any language changes and always give you the localized value. You can forget about writing any code for the localizations. We'll also provide a convenient import/export from DE, so you could share it with companies, who makes actual localization for you.
3. **Data validation** - a lot can change while you edit your game data: the type, the value,.. and it's hard to keep track of everything all the time. For the convenience we'll add additional settings for each parameter: string regex, number range, etc.. Once you decide to Generate new data, Balancy will first validate everything for the errors and only if everything is ok it'll update the game content, otherwise it'll provide you with the information of what data was corrupted. It's a super convenient tool for project of any size, you can be always sure your data is relevant and has no mistakes.

Assets

Assets are all objects which you store in Addressables. If you are not using Addressables in your game you should ignore this page or better start using them.

1. In Unity Open **Balancy->Tools** and click on **Synch Addressables**.
2. The synchronization process starts.
3. Balancy detects which addressables were changed after the last synchronization and uploads images for the new/updated files only. It means that the first synchronization might take some time, while all subsequent will be faster.
4. After the process is done open **Assets** section in DE and you should see the list of all your Assets.

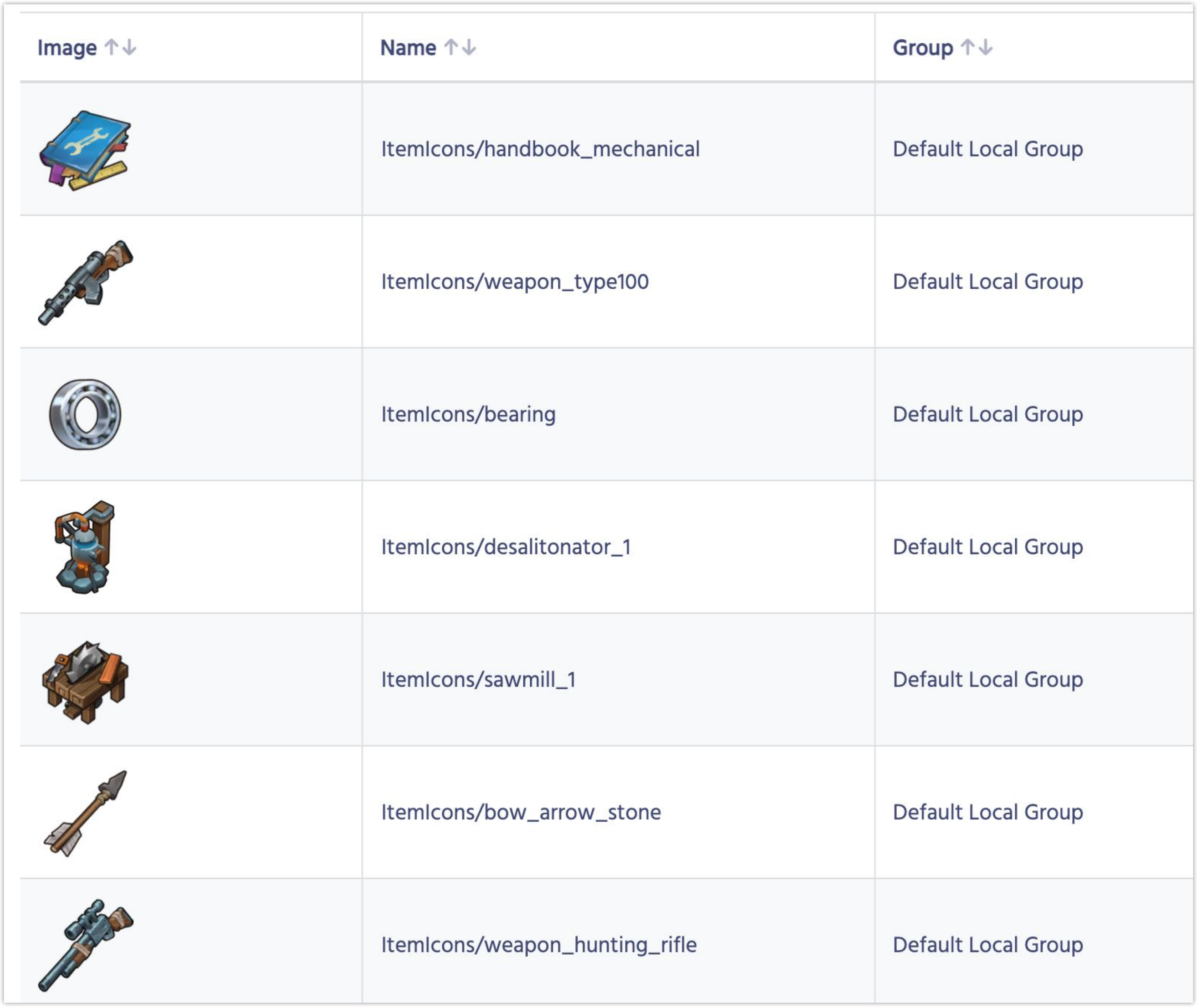
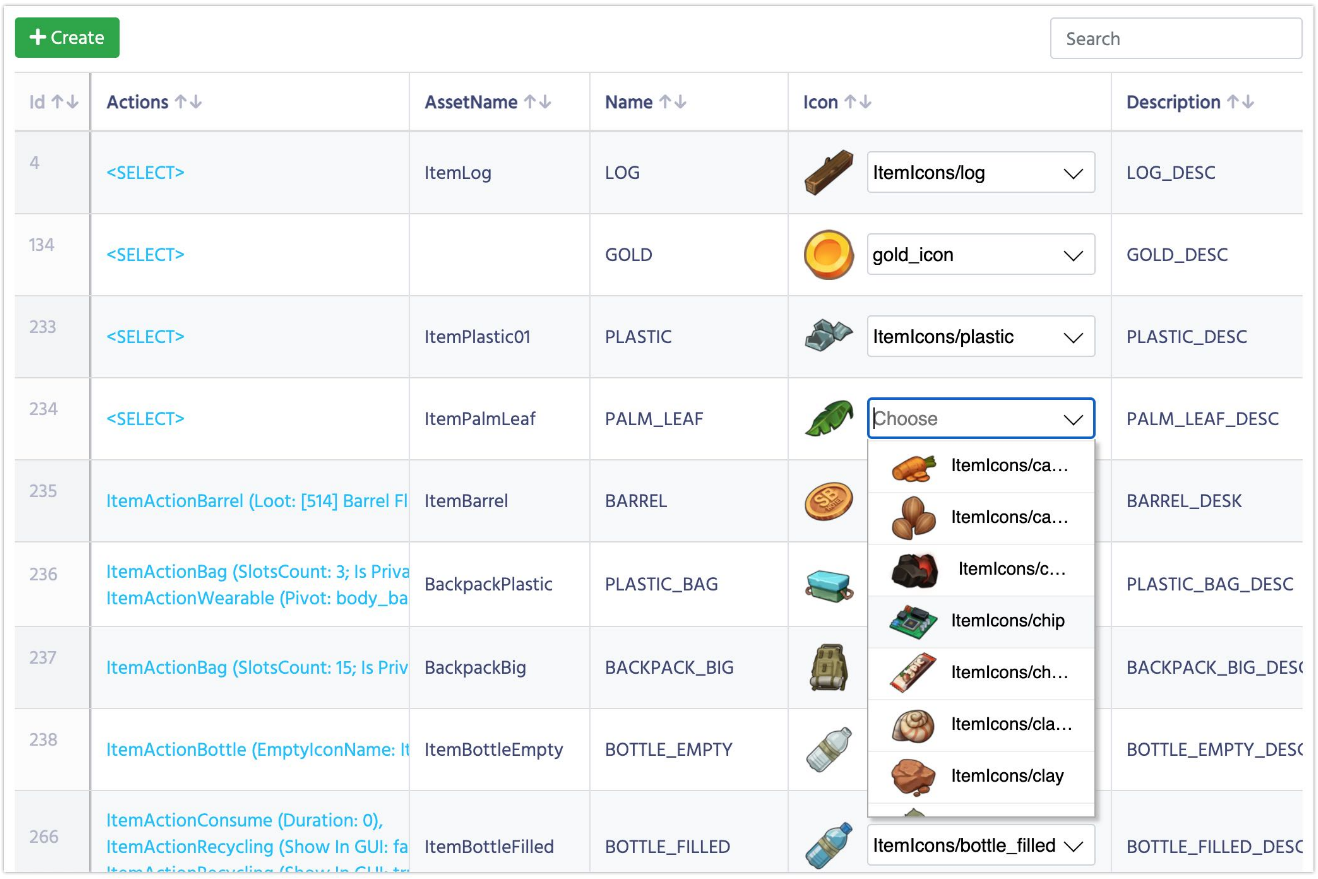


Image ↑↓	Name ↑↓	Group ↑↓
	ItemIcons/handbook_mechanical	Default Local Group
	ItemIcons/weapon_type100	Default Local Group
	ItemIcons/bearing	Default Local Group
	ItemIcons/desalitonator_1	Default Local Group
	ItemIcons/sawmill_1	Default Local Group
	ItemIcons/bow_arrow_stone	Default Local Group
	ItemIcons/weapon_hunting_rifle	Default Local Group

5. If you have any parameter of type **Asset**, you'll be able to pick it from the dropdown menu:



Id ↑↓	Actions ↑↓	AssetName ↑↓	Name ↑↓	Icon ↑↓	Description ↑↓
4	<SELECT>	ItemLog	LOG	ItemIcons/log	LOG_DESC
134	<SELECT>		GOLD	gold_icon	GOLD_DESC
233	<SELECT>	ItemPlastic01	PLASTIC	ItemIcons/plastic	PLASTIC_DESC
234	<SELECT>	ItemPalmLeaf	PALM_LEAF	Choose	PALM_LEAF_DESC
235	ItemActionBarrel (Loot: [514] Barrel Fl	ItemBarrel	BARREL		BARREL_DESK
236	ItemActionBag (SlotsCount: 3; Is Priva ItemActionWearable (Pivot: body_ba	BackpackPlastic	PLASTIC_BAG		PLASTIC_BAG_DESC
237	ItemActionBag (SlotsCount: 15; Is Priv	BackpackBig	BACKPACK_BIG		BACKPACK_BIG_DES(
238	ItemActionBottle (EmptyIconName: It	ItemBottleEmpty	BOTTLE_EMPTY		BOTTLE_EMPTY_DES(
266	ItemActionConsume (Duration: 0), ItemActionRecycling (Show In GUI: fa	ItemBottleFilled	BOTTLE_FILLED	ItemIcons/bottle_filled	BOTTLE_FILLED_DESC

6. In the generated code your parameter will have the type **UnnyAsset**. For now it has only one field **Name**, which is enough for a developer to load the addressable.

Environment

We provide each game with 3 Environments: Development, Stage and Production. It's a standard and commonly used approach.

1. **Development** used during the development process. All new features, bug fixes are created here.
2. **Stage** can be skipped by Indie developers. It's used usually by big companies, which have their own QA department. This environment is used for testing all the features, which were created before. Separate environment for testing is helpful, because it doesn't require the development team to stop.
3. **Production** this is where all your live clients are playing.

Workflow

1. New features are being developed and balanced on the **Development** environment.
2. Once you ready, open Environment section and migrate your **Development** data to the **Stage** environment.
3. It'll erase all the changes you made in **Stage** environment and copy everything from the **Development**.
4. Give the build, connected to the **Stage** environment to your QA.
5. Once QA approves the build, transfer the data from **Stage** to **Production** and publish your game build (connected to the **Production**) in the stores.

Changing of the environment

You should try to avoid changing the environment in DE. In the best case scenario you just need to work in the **Development** and then transfer the data to other environments. Balancy even doesn't support transferring the environment's data in the opposite direction.

However there are situation when you might need to change something in the **Production** environment.

Let's say you have already published your game and your users are playing in the **Production** environment. Then you find some bug in the balance, which you want to fix asap. Your team has already prepared tons of changes in the **Development**, so you can't migrate everything to the **Production** without updating the build. So the solution here is to switch to the **Production** environment in DE, fix your balance and Deploy the changes. That's it! Just don't forget to make the same changes in the **Development**.

Data Migration

In the Environment section you can migrate your data

1. From **Development** to **Stage**

2. From **Stage** to **Production**

When you start a migration process, there are many things are happening. For example when you transfer the data from Dev to Stage, under the hood is happening:

1. Deploy is called for **Development**.
2. All the data is transferred from **Dev** to **Stage**, overriding all the changes made in **Stage** before.
3. Deploy is called for **Stage**.

It means that if you want to send all the data from the **Dev** to **Prod**, you just need to transfer the data from **Dev** to **Stage** and then to **Prod**. You don't need to Deploy anything afterwards. That was already made during the migration.

How to connect to the proper environment

We usually use Define Symbols to deal with different environments:

```
Balancy.Main.Init(new AppConfig
{
    ApiGameId = <API_GAME_ID>,
    PublicKey = <PUBLIC_KEY>,
    OnReadyCallback = response =>
    {
        if (!response.Success)
            Debug.LogError("Couldn't initialize Balancy");
    },
    Environment = GetEnvironment()
});

public static Balancy.Constants.Environment GetEnvironment()
{
#if SERVER_PROD
    return Balancy.Constants.Environment.Production;
#elif SERVER_STAGE
    return Balancy.Constants.Environment.Stage;
#else
    return Balancy.Constants.Environment.Development;
#endif
}
```

Then you just need to switch the define symbols between **SERVER_DEV**, **SERVER_STAGE** and **SERVER_PROD** before you launch your game in Unity or make a build to make it to connect to different environment. Of course you might want to be able to change the environment at runtime for the testing purposes. Just be careful and don't let your end users to be able to connect to an environment different from the **Production**.

Localization

We are going to let you import/export and work with all of your localization right in the Data Editor. This feature is under development right now, but we have a temporary substitution, which you can use and later migrate without any problems.

When you create a new parameter, instead of selecting type **String**, make it **Localized String**. Everytime, when you try to access this parameter in the code, it'll instantly give you already localized value.

Let's say you have a pair "Key": "LocalizedValue":

1. You create a new parameter with type of Localized String.
2. In the document you put the value **Key**.
3. When you access your parameter from the code, you will be getting **LocalizedValue** instead of **Key**.

You are probably using some other solution to store all the keys and values for localization. So in order to make everything work, you need to add those likes of code:

```
Balancy.Localization.Manager.onLocalizationRequested = key =>
{
    return <Value associated with current key for the selected localization>
};
```

Once we ready to store all keys and values in our system, you'll just need to remove that code and everything will be working automatically.

UI

If you are using a lot of UnityEngine.UI.Text with static text, you'll find very helpful our Component: **LocalizationText.cs**. Just add it to your GameObject and set parameter **Localization Key** value as your **Key**. It'll automatically put Localized value as Text when you launch the game.

We've the same Component for Text Mesh Pro. You can download it from here

Documents & Components

Documents

Document is a unique instance of a Template (Specific Item: Hunter's Bow, Gold Bar,...), which has it's own parameter values. Think of it as an **instance** of a **Class** as a programmer.

When you add a new Template (Not Component), a new section in the left navigation appears. If you select any of the Document section, you'll be able to add new Documents in there. Each document can have a unique values for all the parameters of it's Template.

Components

Component is a simple Template, which doesn't have it's own Documents and can exists only inside of another Document.

For example Vector3 component template has parameters: x, y, z. If a Document "Hero" has a Parameter "position" of type Vector3, you'll be able to edit x, y, z values of "position" right inside of "Hero" Document.

Next: Enums

Enumerated Types (enums)

Overview

Enumerated types are widely used in programming. When you have a limited list of possible values, it's often convenient to use **enum** instead of **int** or **string**.

FOR EXAMPLE:

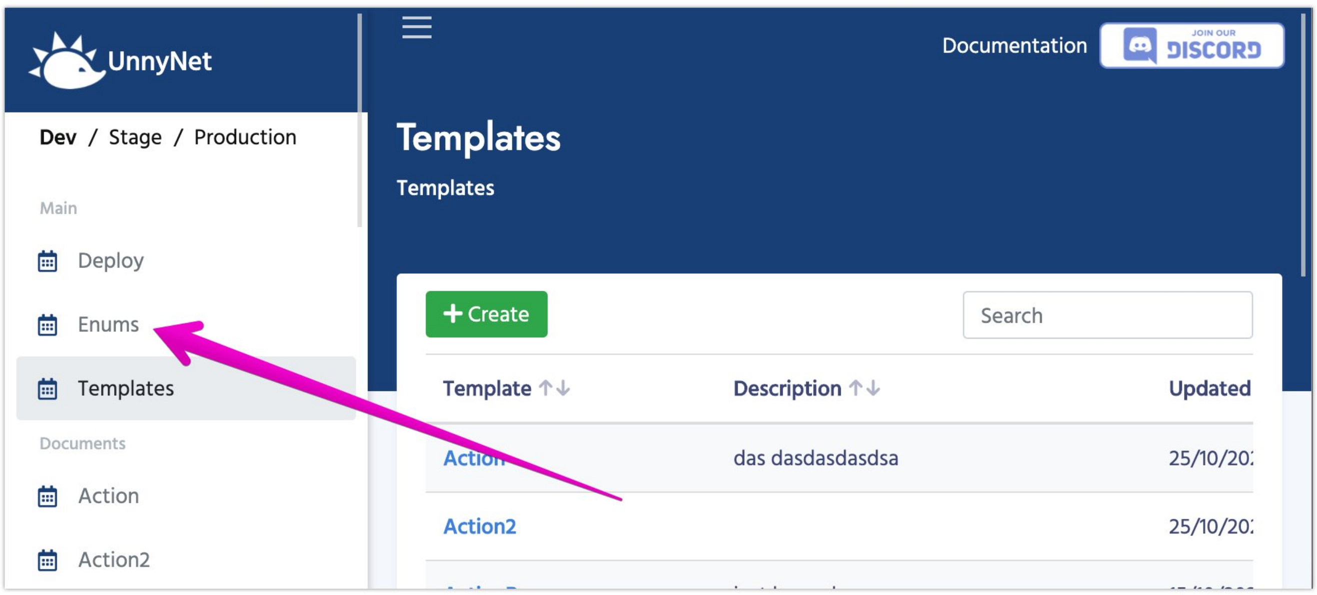
If you have a limited set of Colors to choose in your game, you might want to store the value as integer (1,2,3,4,...) to save memory. However, you can define a new enum, which will make your values more readable and convenient to use.

```
public enum Color
{
    Red = 0,
    Green = 1,
    Blue = 2,
    White = 3,
    Black = 4,
}
```

Now you can use values, like **Color.Blue** in your code instead of **2**.

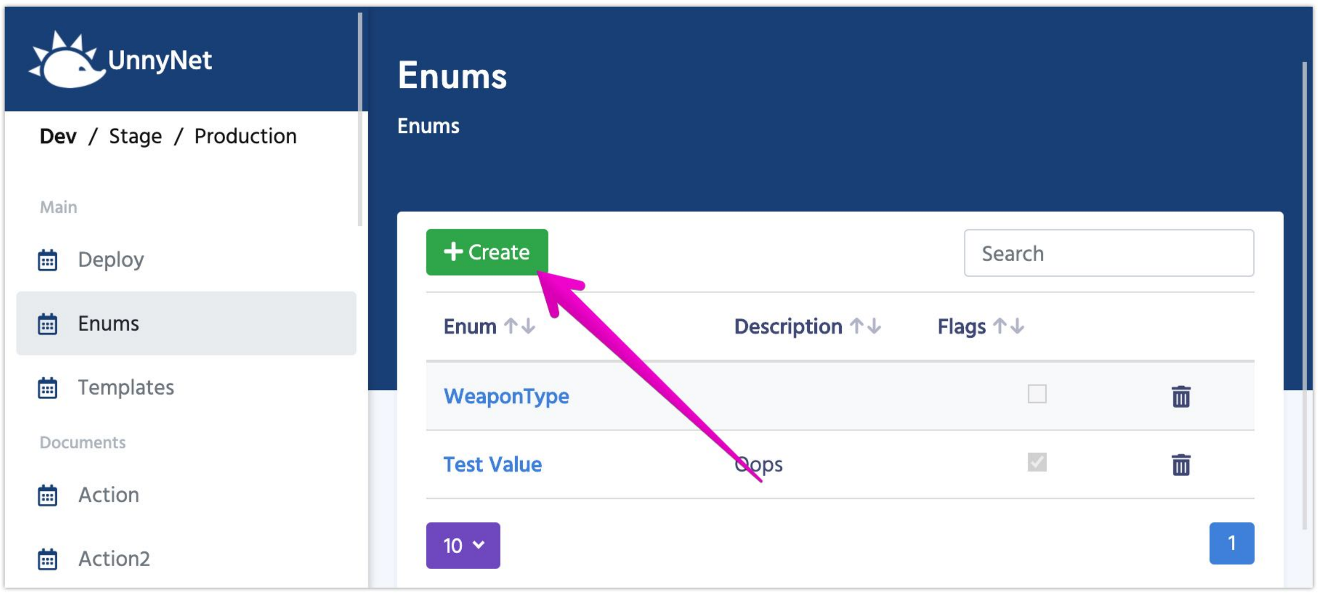
How to create enum

1. Select **Enums** section in Data Editor



Template	Description	Updated
Action	das dasdasdasdsa	25/10/20
Action2		25/10/20

2. Click on **Create** button



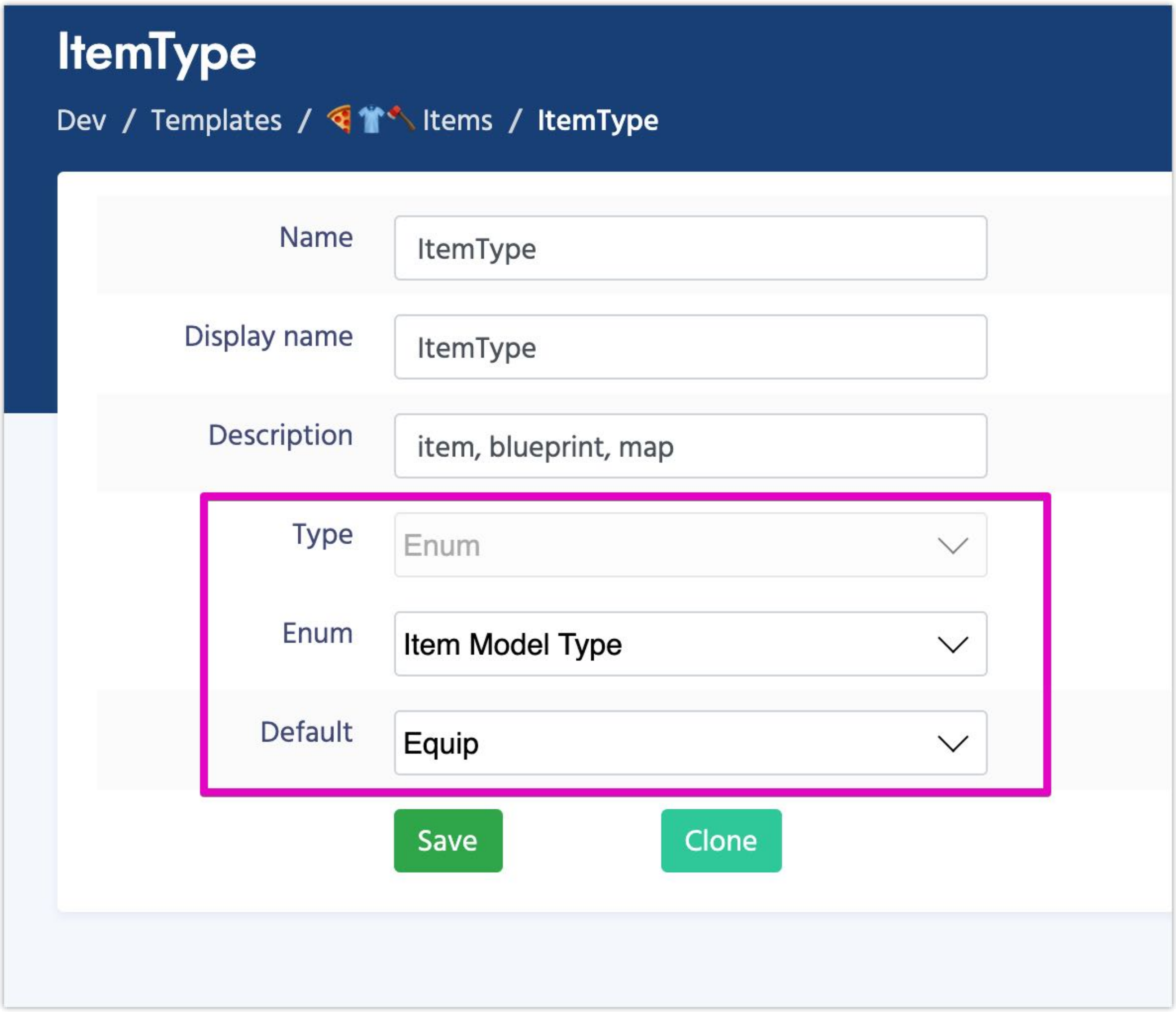
3. Each Enum has several parameters

Name	Description
Name	This name is used when you work with your enum from code.
Display Name	The name which will be displayed in the DE.
Description	Helps other team members to easily understand what this Enum is used for.
Multi-selection	defines if a parameter can contain multiple enum values.

4. Table of Values

Each enum value must have a unique name and a unique value associated with the name. If you are using Multi-selection, all the values must be power of 2 or be equal zero.

5. When you create and save your enum, you can choose type Enum for a parameter.



ItemType

Dev / Templates / 🍕👕🪓 Items / ItemType

Name	ItemType
Display name	ItemType
Description	item, blueprint, map
Type	Enum
Enum	Item Model Type
Default	Equip

Next: Deploy

Important terms:

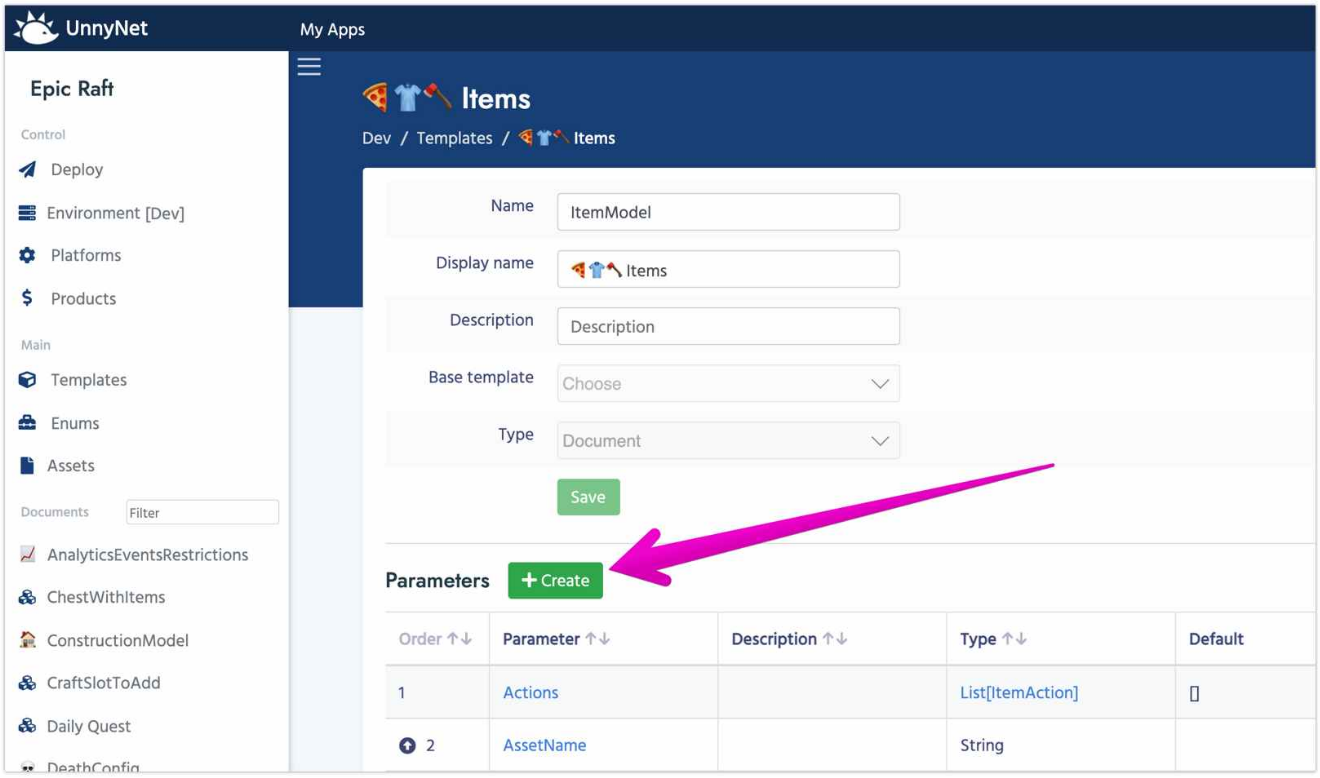
1. **Template** describes the structure and behaviour of your game object (item, monster, construction,...). As a programmer you can think of it as a **Class**. Template has to have a unique name and may contains a set of parameters.
2. **Parameter** describes a part of a Document, storing some value. Each parameter has a name and a type. Type can be simple like string, int, float, bool or a reference to any other Template. As a programmer you can think of parameter as a **Field** or **Property** of the **Class**.
3. **Document** is a unique instance of a Template (Specific Item: Hunter's Bow, Gold Bar,...), which has it's own parameter values. Think of it as an **instance** of a **Class** as a programmer.
4. **Component** is a simple Template, which doesn't have it's own Documents and can exists only inside of another Document.
5. **Enum** is enumerated type, which you can create and use in Data Editor.

Next: Templates

Parameters

Parameter describes a part of a Document, storing some value. Each parameter has a name and a type. Type can be simple like string, int, float, bool or a reference to any other Template. As a programmer you can think of parameter as a **Field** or **Property** of the **Class**.

1. After the creation of a template, you can add parameters to it.



Name	ItemModel
Display name	Items
Description	Description
Base template	Choose
Type	Document

Order	Parameter	Description	Type	Default
1	Actions		List[ItemAction]	[]
2	AssetName		String	

2. On the parameters page you can view/edit all existing parameters and add new one.
3. Each Parameter has several fields:

Name	Description
Name	This very name is used during Class generation. To keep everything in style we advise you to use CamelCase naming. For example: MainTag, ConstructionId, HeroType,...
Display Name	This name will be displayed in the DE for your convenience. For example: Main Tag, Construction Id, Hero Type,...
Description	Helps other team members to easily understand what this Parameter is used for.

Name	Description
Use In Display Name	Means that this parameter will be displayed in search and references for the corresponding Document. Usually **Name** (if any) or any other unique string parameter is selected for this or any other, which will help you instantly understand what instance is that.
Is required	Marks this parameter as a must have value. This flag helps you to make sure you won't forget to add a required reference or value.
Is unique	It's used in case you want all of your Documents of this Template to have different values of this parameter.
Default Value	The value which will be assigned by default upon a new document is created.
Type	A Data type of the parameter. All types are below.

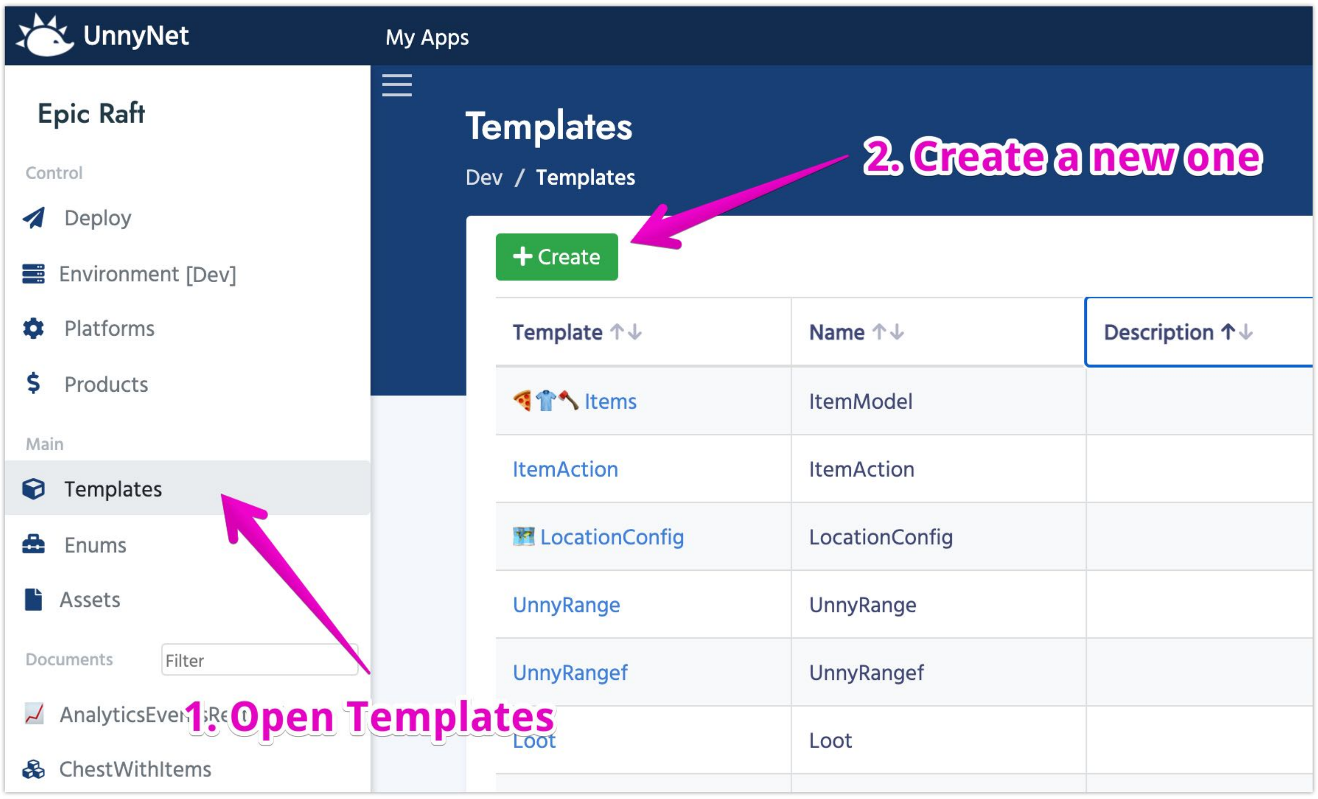
Type	Description
Integer	A Number that can be written without a fractional component. For ex: 1, 2, 999, -200
Float	A Number with a fractional component. For ex: 1.32, -0.7432
Boolean	Logical value: true or false
String	Any Text. For ex: "Hello World", "-+ ta-ta_!! 55"
Enum	Provides a selection from predefined possible values.
Document	A reference to an existing document
List	An Array(list) of other type values
Asset	A reference to an existing Asset. It's usually a prefab, sprite or other Object, which is stored in Unity game as an Addressable.

Next: Documents & Components

Templates

Template describes the structure and behaviour of your game object (item, monster, construction,...). As a programmer you can think of it as a **Class**. Template has to have a unique name and may contains a set of parameters.

1. Open Templates section and click on the Create Button



Template	Name	Description
🍕👕⛏ Items	ItemModel	
ItemAction	ItemAction	
LocationConfig	LocationConfig	
UnnyRange	UnnyRange	
UnnyRangef	UnnyRangef	
Loot	Loot	

2. Each Template has several parameters

Name	Description
Name	This very name is used for Class generation. To keep everything in style we advise you to use CamelCase naming. For example: ItemModel, GameConstruction, MonsterData,...
Display Name	The name which will be displayed in the DE. Usually it's the same as name, but words are separated. For example: Item Model, Game Construction, Monster Data,...

Name	Description
Description	Helps other team members to easily understand what this Template is used for.
Base Template	It's used if your Template inherits from another one.
Type	Can be Document, Component or Singleton: * **Component** Documents of this Template are always embedded into another Documents. For example Vector3 component template has parameters: x, y, z. If a Document "Hero" has a Parameter "position" of type Vector3, you'll be able to edit x, y, z values of "position" right inside of "Hero" Document. * **Singleton** Only one of such Documents will be available from the code. It's usually used for settings and configs.

Next: Parameters

Requests, Response Data and Errors

All Balancy methods are asynchronous with a callback as an argument, which is invoked once the method is complete. The parameter of the callback is inherited from **ResponseData**.

Unity JavaScript

```
public class ResponseData {
    public bool Success;
    public Error Error;
    public object Data;
}

public class Error {
    public int Code;
    public string Message;
}
```

You need to check if **Success** is true to be sure that the request was successful, and the Data is valid. Otherwise, you need to read **Error** to understand what went wrong. Here is the list of error, which might occur:

```
public enum Errors {
    NotInitialized = -1,
    Unknown = 1,

    NoAccessToken = 1000,
    StorageRequestsMadeTooOften = 1001,
    NoSuchProduct = 1002,
    StorageError = 1003,

    UnityPurchasing_PurchasingUnavailable = 1010,
    UnityPurchasing_NoProductsAvailable = 1011,
    UnityPurchasing_AppNotKnown = 1012,
    UnityPurchasing_ProductIsNotAvailable = 1013,
    UnityPurchasing_PurchaseFailed = 1014,

    Nutaku_Error = 1100,
};
```

If everything is ok, you can read **responseData.Data** if needed.