

Welcome to UnnyNet

Page	Description
2	Welcome to UnnyNet
4	How to start
5	Unity3D
7	Authorization
8	Storage
11	Payments
12	Data Editor
13	Important Terms
14	Templates
16	Parameters
19	Documents
20	Code Generation
24	Example
31	What's next
32	Requests, Response Data and Errors
34	Release Notes

Welcome to UnnyNet

UnnyNet is an essential extension for any game or app. Instead of using ancient solutions, you can spend several minutes learning what UnnyNet 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](#)

[Join Our Discord](#)

Out-of-the-box

Integration of UnnyNet 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. UnnyNet 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 to start

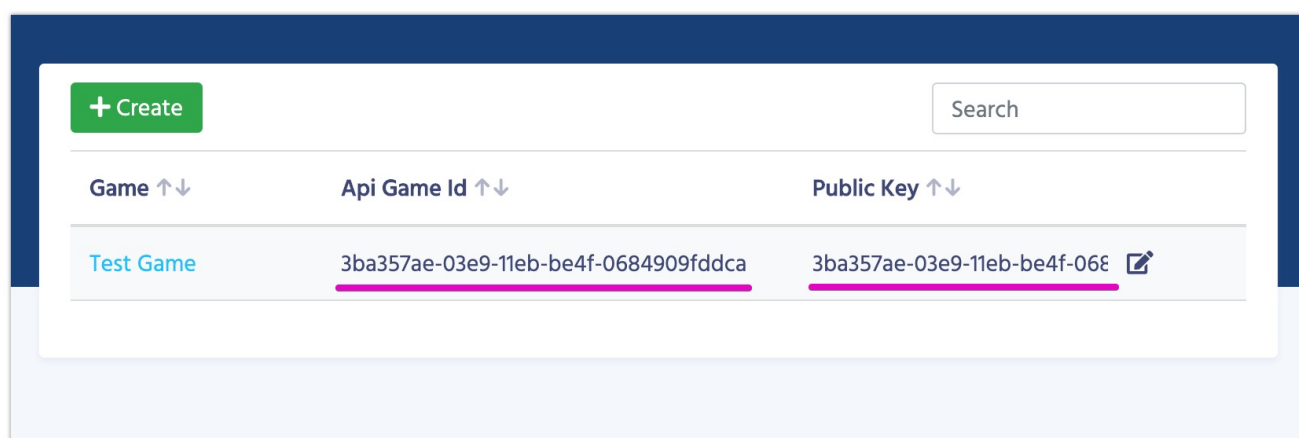
1. Create an account at UnnyNet
2. Create a new game
3. Select your platform and follow the instructions:
 - a. Unity.

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 UnnyNet plugin.
3. Prepare Game ID and Public Key to use in the code:



4. Call initialize method at start:

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

Further reading

UnnyNet 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

Authorization

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

Email

```
UnnyNet.Auth.WithEmail(<email>, <password>, doneCallback);
```

Name and Password

```
UnnyNet.Auth.WithName(<username>, <password>, doneCallback);
```

As Guest using Device ID

```
UnnyNet.Auth.AsGuest(doneCallback);
```

Storage

Allows you to Save and Load any data on UnnyNet 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.

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**. UnnyNet 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 thinks he has a version **5**. The next time it tries to save the update, it'll get 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 actual 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 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

```
UnnyNet.Storage.Save(<Collection Name>, <Key Name>, <Value String or Object>,
doneCallback);
```

Load

```
UnnyNet.Storage.Load<Type of Object or String>(<Collection Name>, doneCallback);  
UnnyNet.Storage.Load<Type of Object or String>(<Collection Name>, <Key Name>,  
doneCallback);
```

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.

Payments

Payments will be revealed in a couple of weeks.

Data Editor

Data Editor (DE) is an essential part of UnnyNet. It is used for creating data structure and editing the data. UnnyNet 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 an 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 Unity with your project and start code generation request. UnnyNet 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 a 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. Whenever you change the game data in DE, it'll be automatically synchronized with the game after the restart.

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

Next: Important Terms

Important terms:

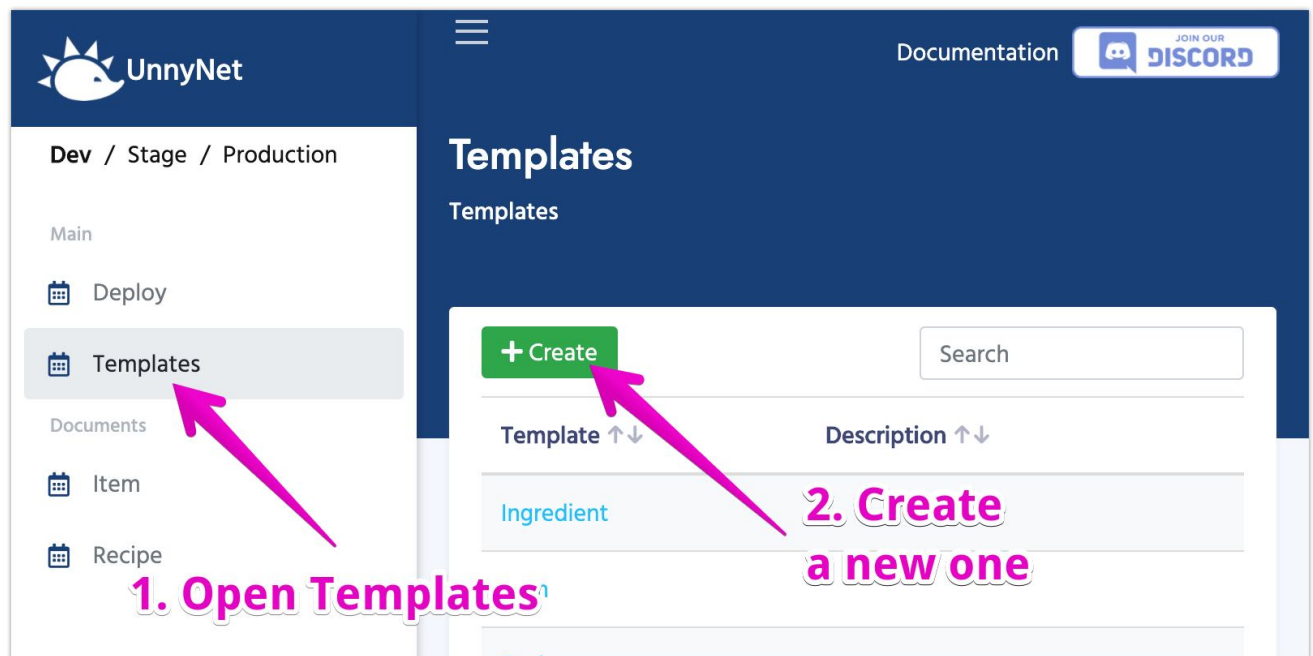
1. **Template** describes the structure and behaviour of your game object (item, monster, construction,...). As a programmer you can think of it as a **Class**. Template has to have a unique name and a set of parameters.
2. **Parameter** describes a part of a Document, storing some value. Each parameter 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.

Next: Templates

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 a set of parameters.

1. Open Templates section and click on the Create Button



2. Each Template has several parameters

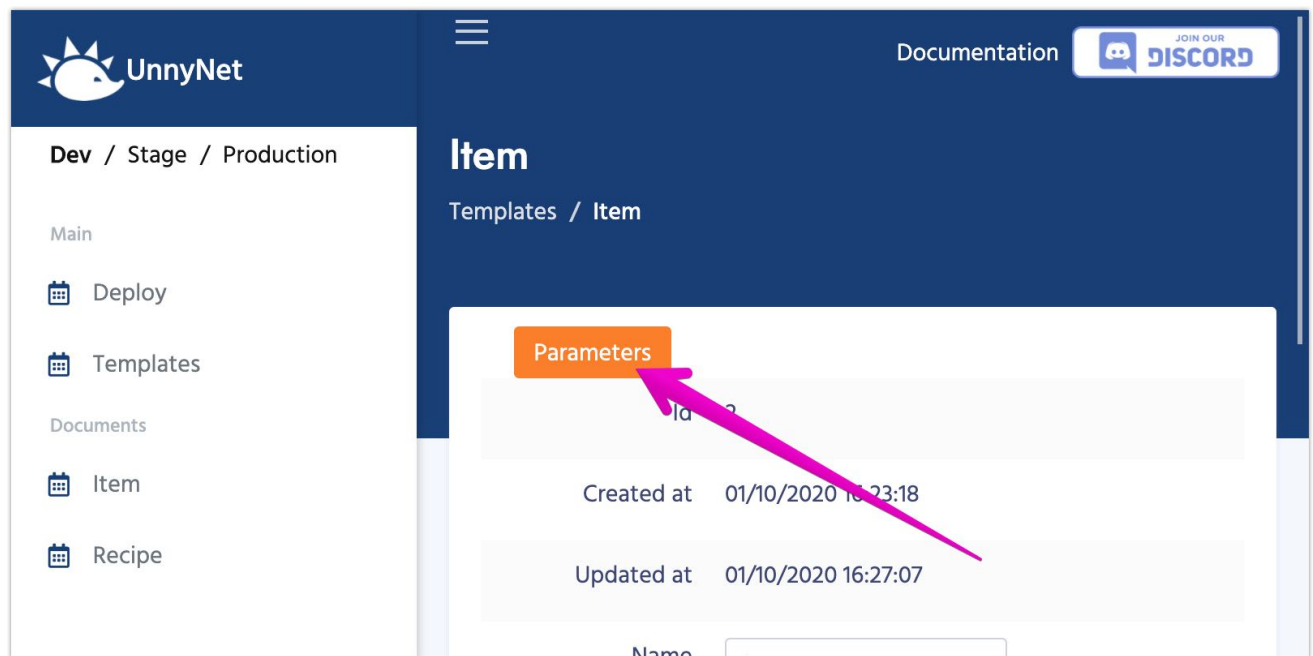
Name	Description
Name	<p>This very name is used for Class generation. To keep everything in style we advise you to use CamelCase naming.</p> <p>For example: ItemModel, GameConstruction, MonsterData,...</p>
Display Name	<p>The name which will be displayed in the DE. Usually it's the same as name, but words are separated.</p> <p>For example: Item Model, Game Construction, Monster Data,...</p>
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. (This feature is under development)
Type	<p>Can be Default, Component or Singleton:</p> <p>* 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.</p> <p>* Singleton Only one of such Documents will be available from the code. It's usually used for settings and configs.</p>

Next: Parameters

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.



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

Name	Description
Name	<p>This very name is used during Class generation. To keep everything in style we advise you to use CamelCase naming.</p> <p>For example: MainTag, ConstructionId, HeroType,...</p>
Display Name	<p>This name will be displayed in the DE for your convenience.</p> <p>For example: Main Tag, Construction Id, Hero Type,...</p>
Description	<p>Helps other team members to easily understand what this Parameter is used for.</p>
Use In Display Name	<p>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.</p>
Is required	<p>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.</p>
Is unique	<p>It's used in case you want all of your Documents of this Template to have different values of this parameter.</p>
Default Value	<p>The value which will be assigned by default upon a new document is created.</p>
Type	<p>A Data type of the parameter. All types are below.</p>

Type	Description
Integer	A Number that can be written without a fractional component. For ex: 1, 2, 999, -200
String	Any Text. For ex: "Hello World", "-+ ta-ta_!! 55"
Float	A Number with a fractional component. For ex: 1.32, -0.7432
Boolean	Logical value: true or false
Document	A reference to an existing document
List	An Array(list) of other type values

Next: Documents & Components

Documents & Components

Documents

Document is a unique instance of a Template (Specific Item: Hunter's Bow, Gold Bar,...), which has its 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 its Template.

Components

Component is a simple Template, which doesn't have its own Documents and can exist 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: Code Generation

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 UnnyNet awesome features, like: SmartObjects and Localization. Using them all together gives your team a huge boost.
5. You can forget about JSONs and how to parse them. UnnyNet 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. UnnyNet 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->UnnyNet, input your Gameld/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/UnnyNet/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?

UnnyNet server generated JSON files based on your Documents and puts it to the CDN storage. UnnyNet 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 UnnyNet, 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 **UnnyNet.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 **UnnyNet.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

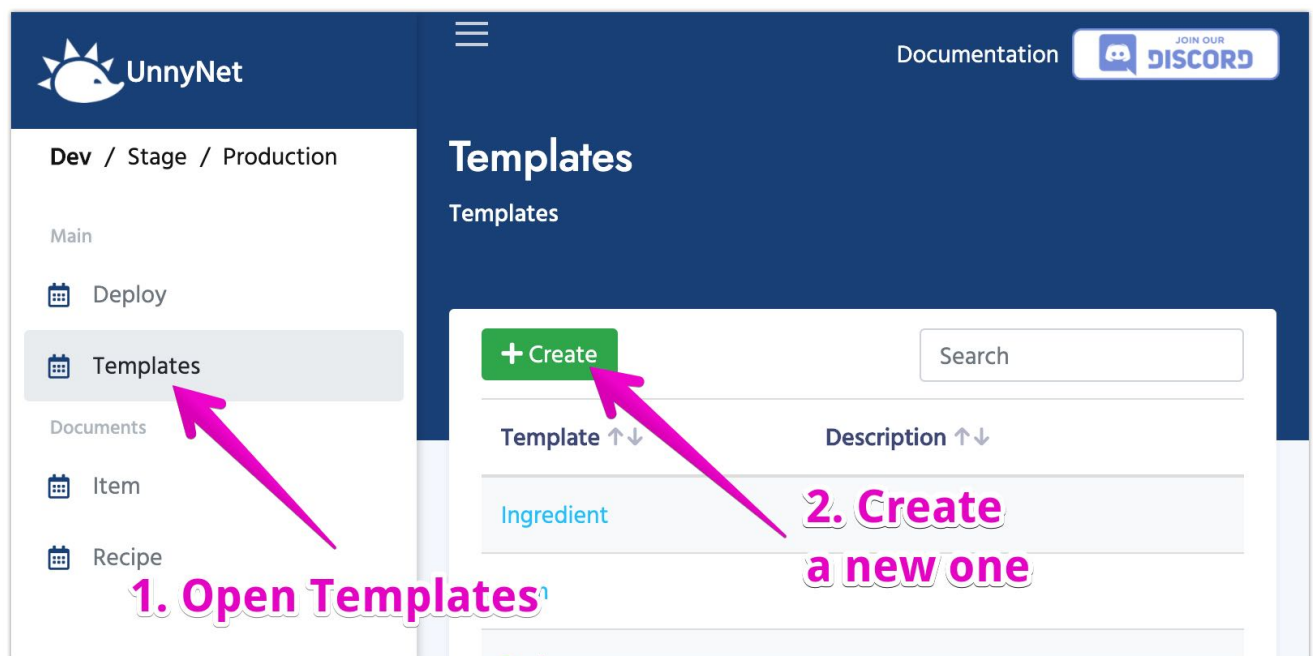
Data Editor usage Example

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



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		<input type="checkbox"/>	<input type="checkbox"/>
Description		String		<input type="checkbox"/>	<input type="checkbox"/>

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		<input type="checkbox"/>	<input type="checkbox"/>
Count		Integer	1	<input type="checkbox"/>	<input type="checkbox"/>

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		<input type="checkbox"/>	<input type="checkbox"/>
Ingredients		List[Document]	[]	<input type="checkbox"/>	<input type="checkbox"/>

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

UnnyNet Initialization

You should read more about UnnyNet 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:

```
UnnyNet.UnnyNetNewInit.Init(new UnnyNet.AppConfig
{
    ApiGameId = YOUR_GAME_ID,
    PublicKey = YOUR_PUBLIC_KEY,
    Environment = UnnyNet.Constants.Environment.Development,
    OnReadyCallback = responseData =>
    {
        Debug.Log("UnnyNet 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 = UnnyNet.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 = UnnyNet.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 UnnyNet was initialized. Call this method after the Log line:

```
Debug.Log("UnnyNet 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->UnnyNet
2. Click on **Download Data**

3. All the latest game data from Editor will be downloaded and put into /Assets/UnnyNet/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 UnnyNet. 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 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. **Dev / Stage / Prod** - multi-servers setup so you could make any changes in the Dev server without affecting the live version of your game. Once you test everything, you can move the Dev data to the Prod server.
4. **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, UnnyNet 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.

Requests, Response Data and Errors

All UnnyNet methods are asynchronous, they have a callback, 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 errors, 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.

Release Notes

v1.0.0 - October 7, 2020

- First release