Remote Config

documentation

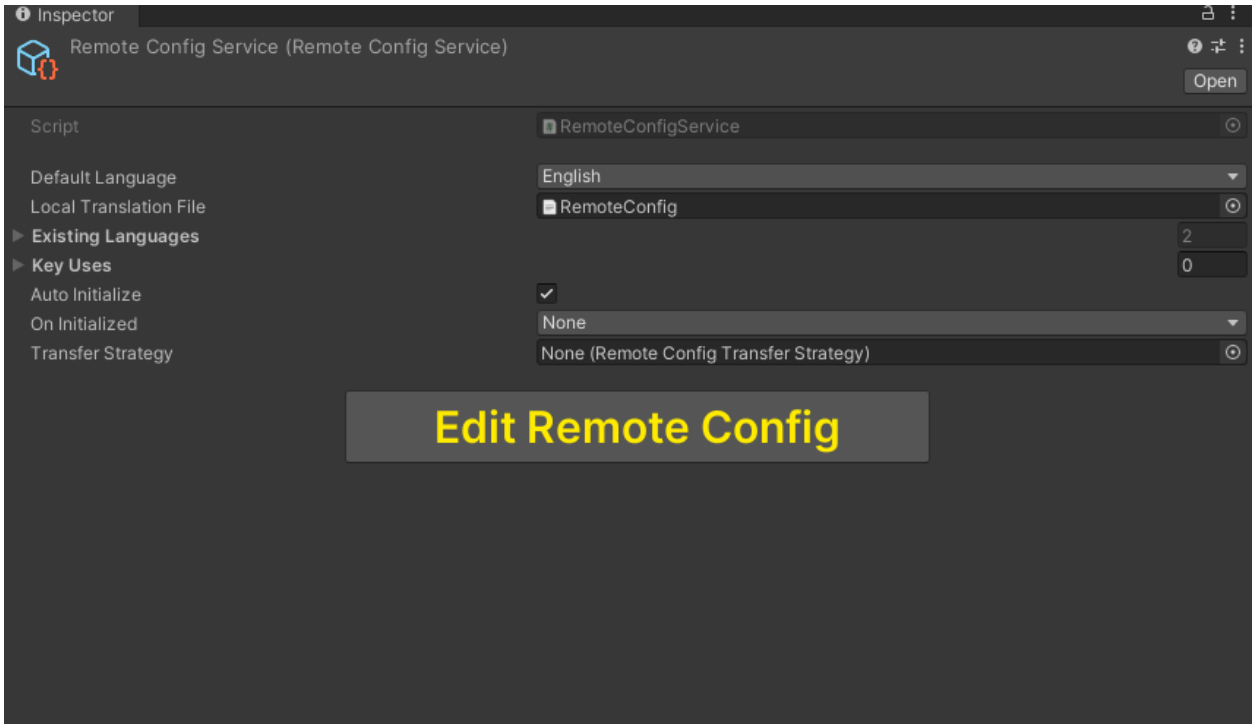
Hello and thank you for your purchase ! This document will cover all the features for this remote config system. This includes :

- Editor data manipulation : this system comes with an optimized editor tool that allows you edit your data. It has many features including uploading and downloading your remote config file, languages, platforms data, entries sorting and editing and usage report.
- Runtime usage : how to use this system when your app is playing.

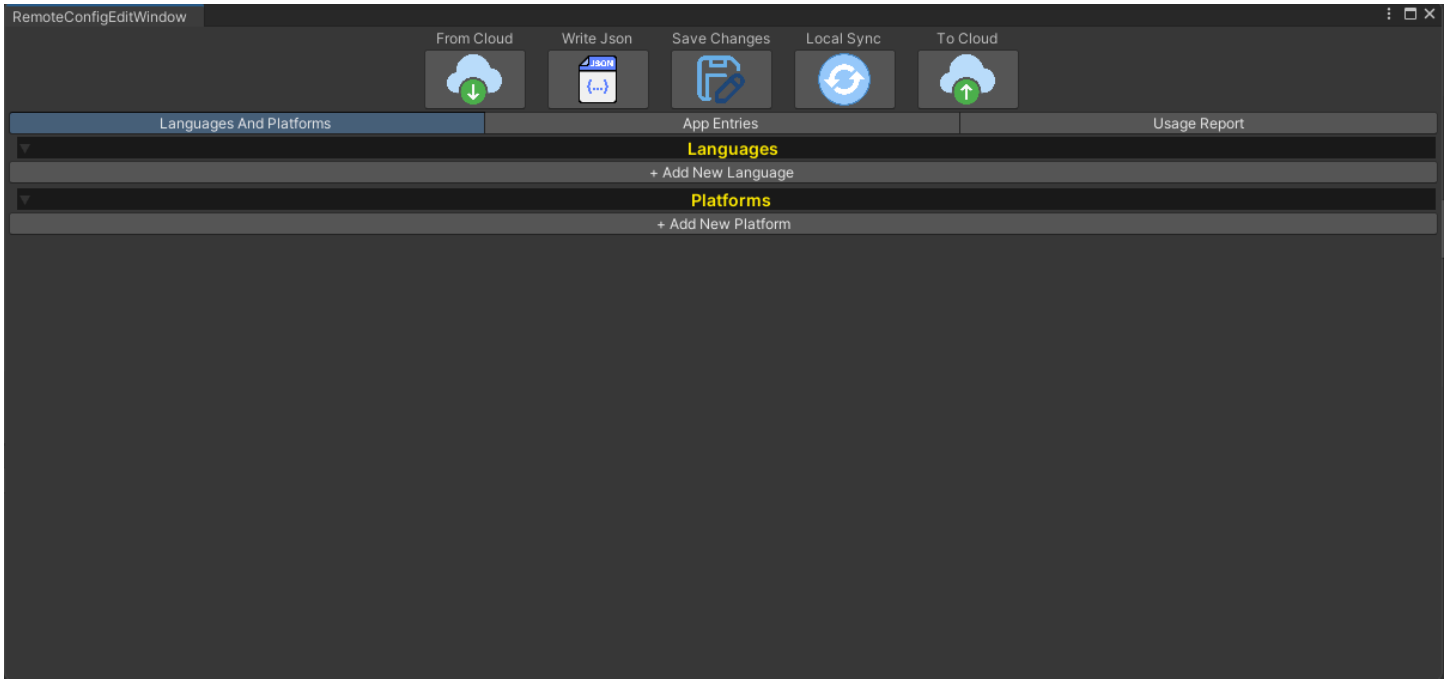
Let’s begin !

Editor Data Manipulation

If you just installed the package, you should have a Scriptable Object named “RemoteConfigService” inside the remote config folder. This is the main script, from which you will be loading and using your remote config settings. If you select it, you should see something like this :



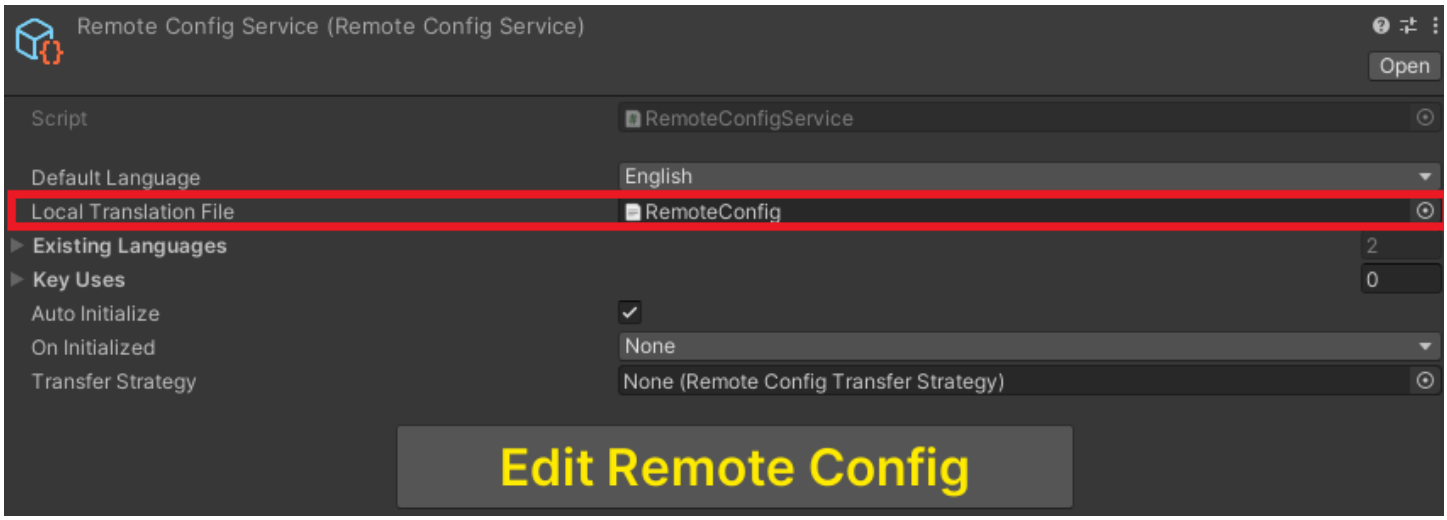
Click the big yellow button to open the file editor :



For the moment everything is mostly empty, but it will quickly grow as you add more and more entries. Before I show you how to do so, let's have a quick overview of this window
First, you can see five buttons on top. Here what they are here for :

 The "From Cloud" button will download and override your data with your remote file. This feature requires a bit of setup, but we will cover that later.

 The "Write Json" button will write all your data to your local json file, which is the text asset binded in the RemoteConfigService file :



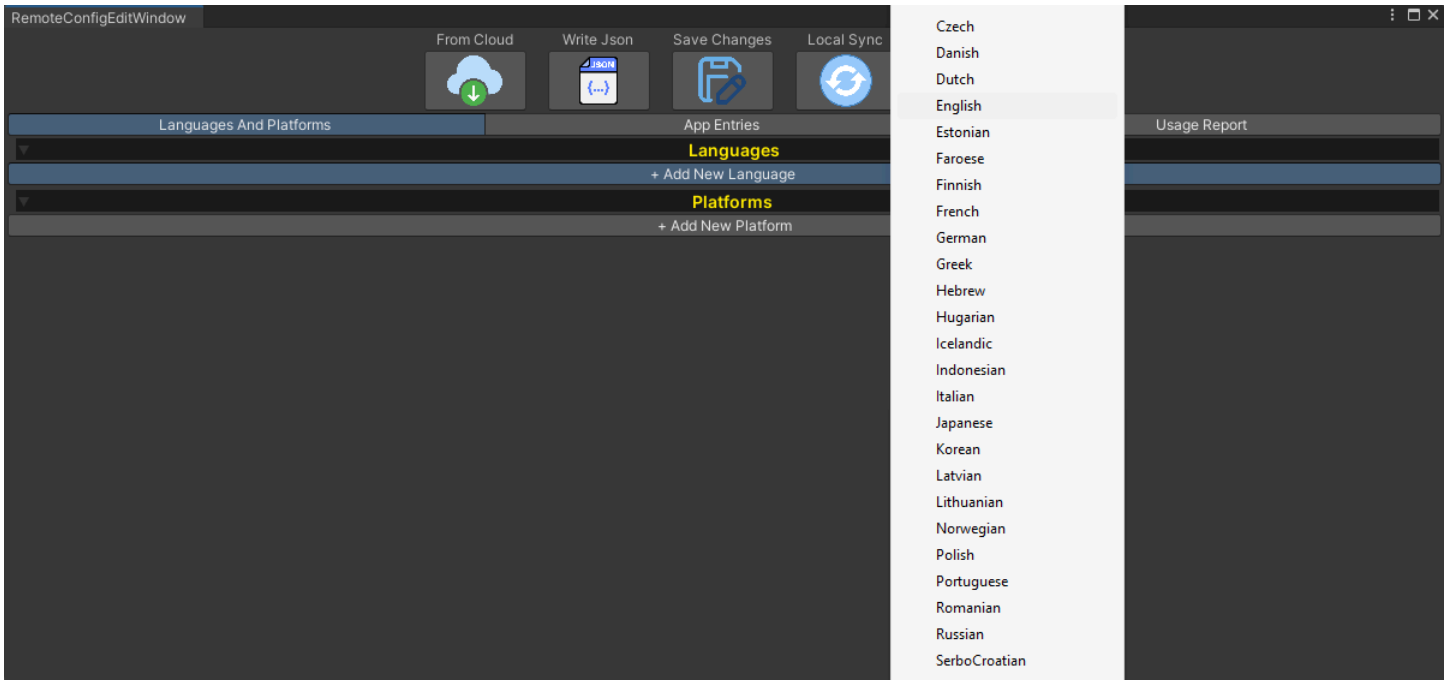
Note that this file will be overwritten when using the "From Cloud" button and is what will be sent when using the "To Cloud" button (see below).

 The "Save Changes" button will save your changes without overwriting your local json file.

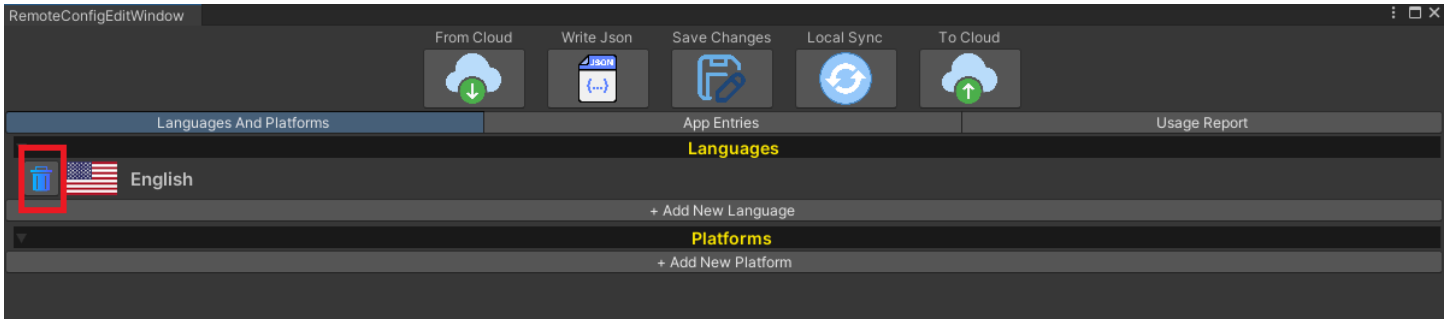
 The "Local Sync" button will force refresh your data with your local file.

 The "To Cloud" will send your local file to your server. As for the "From Cloud" button, it requires a bit of setup.

Below those buttons, you have three sections. The first is “Languages And Platforms”. For the moment it is fully empty, but you can start by adding a new language. Click the “+ Add New Language” button and select the language you want to add :

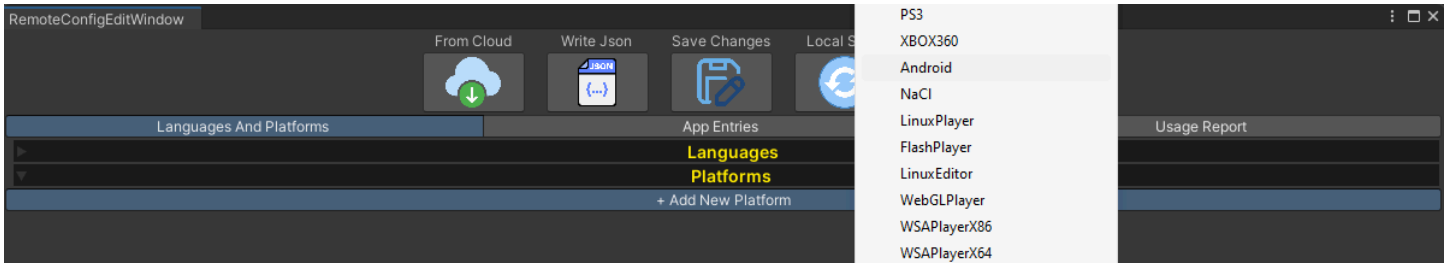


This will add the language of your choice to the list. If you want to delete a language, click the bin button next to the language flag :

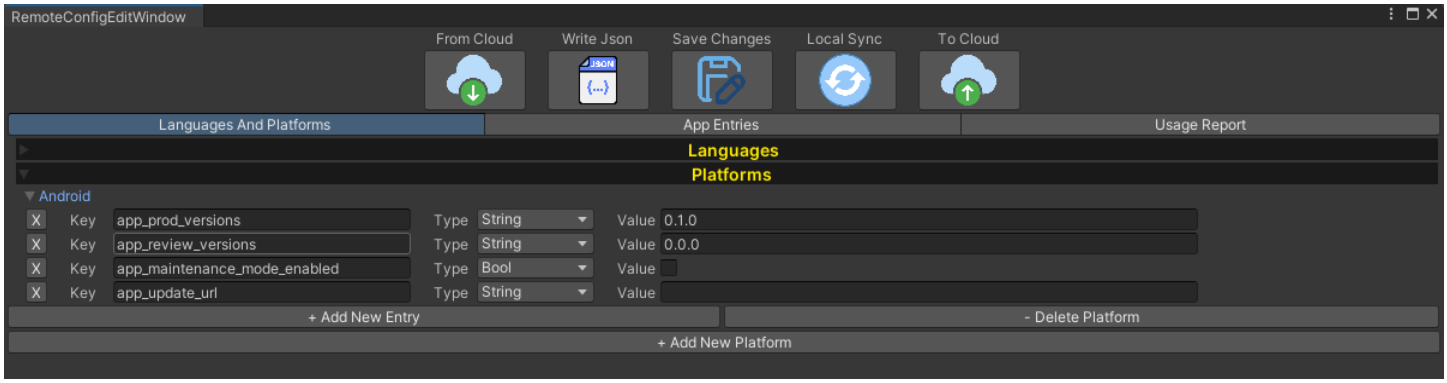


This part is quite straightforward, there is nothing much to say. Note that “Languages” and “Platform” are foldouts that can be folded if needed.

The other element of this section are the platforms. You will often want to have platform specific settings when deploying your application, and this is what this part is all about. When the service script loads a remote config json file, it will automatically look for platform settings that match the current Application.platform and load these (and only these) settings. You can add a new platform by clicking the “+ Add New Platform” button :



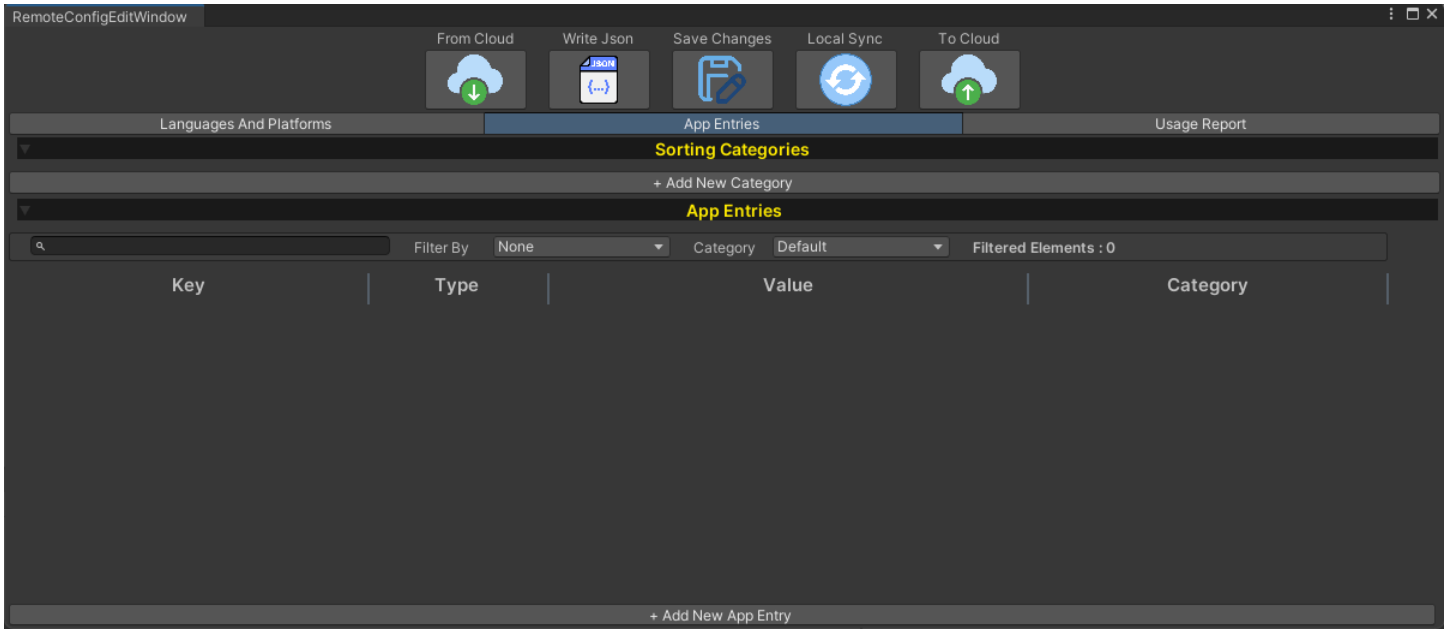
Once you selected your platform, you should see something like this :



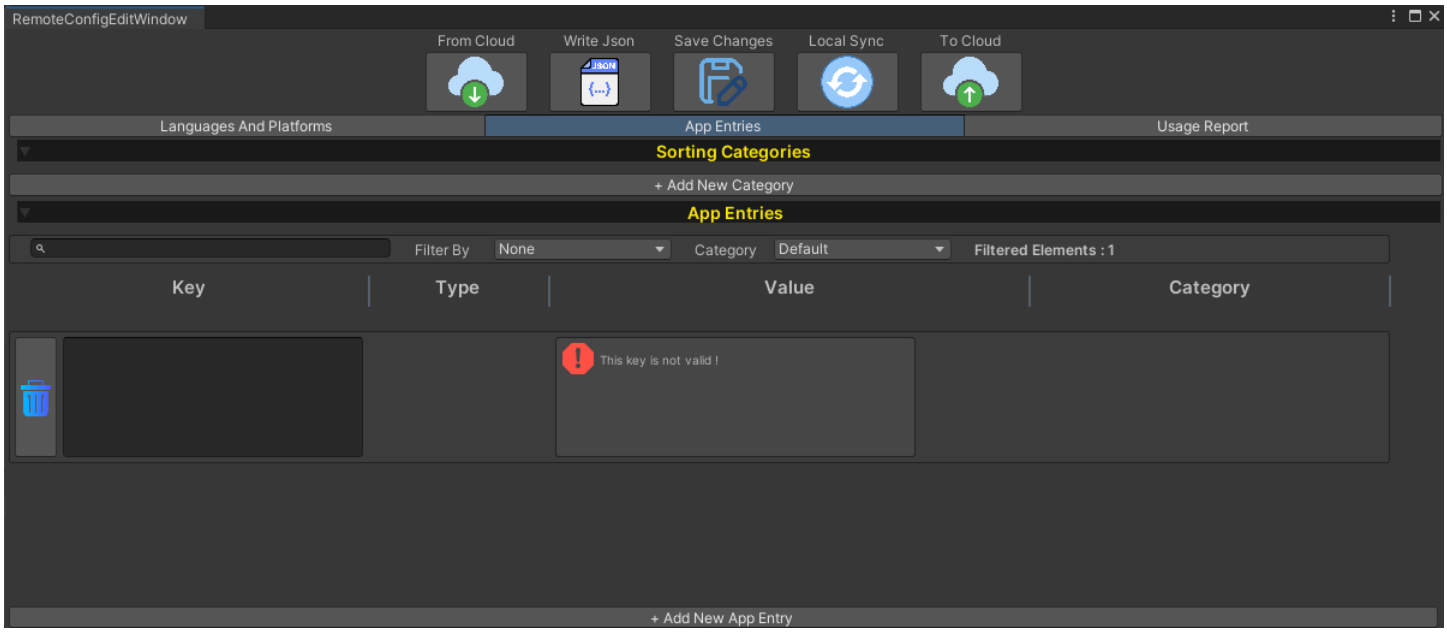
Each line corresponds to an entry, which is a key/value pair. By default four are created when you add a new platform, but you can add or delete as many entries as you want. Add a new entry by clicking the “+ Add New Entry” button and delete one by clicking the small “X” button on the left. Delete a platform by clicking the “- Delete Platform” button.

This concludes the “Language and Platforms” section. The one we will see next is the most important one, the “App Entries” section that will contain almost all your entries.

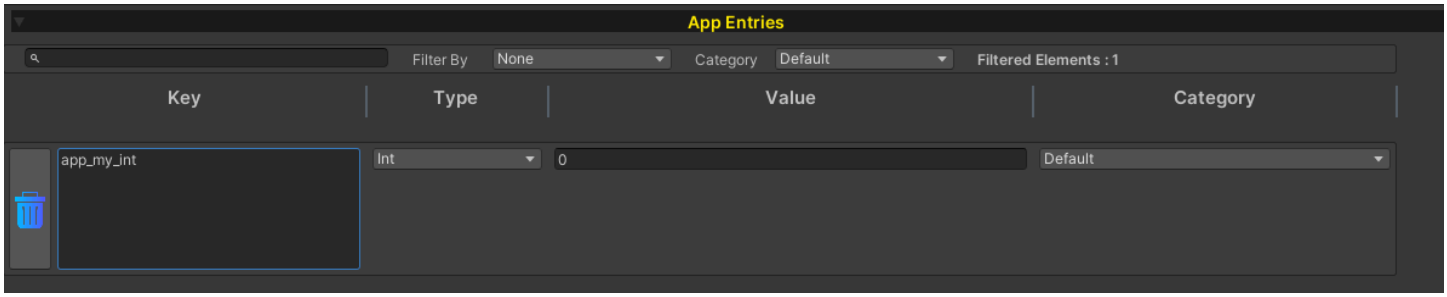
The “App Entries” section is where you will add, edit and remove your entries. At the moment everything should be quite empty like this :



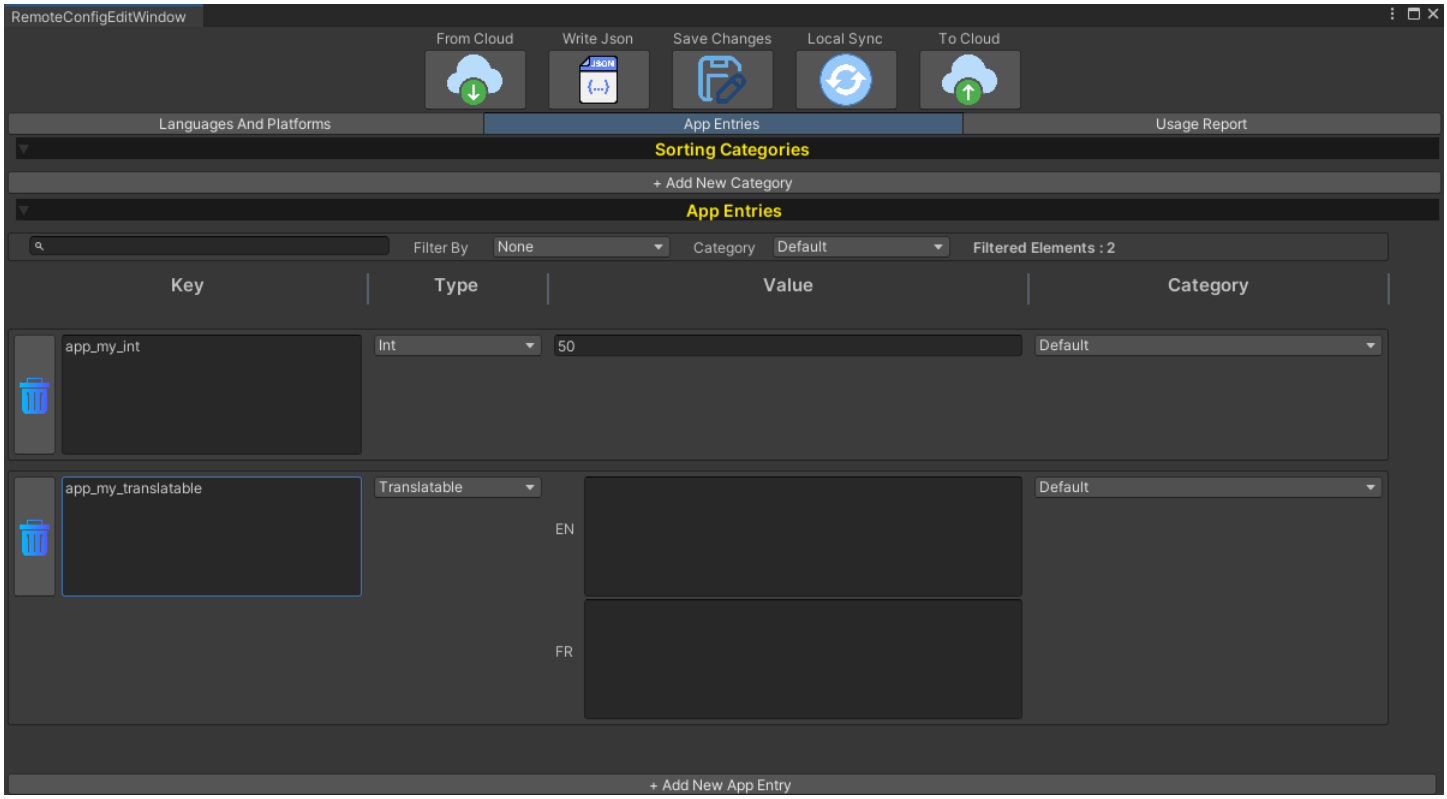
To add a new entry, click the “+ Add New App Entry” at the bottom. It will ask you for the type of entry you want to add (which can be changed at any time), and add it to the list like this :



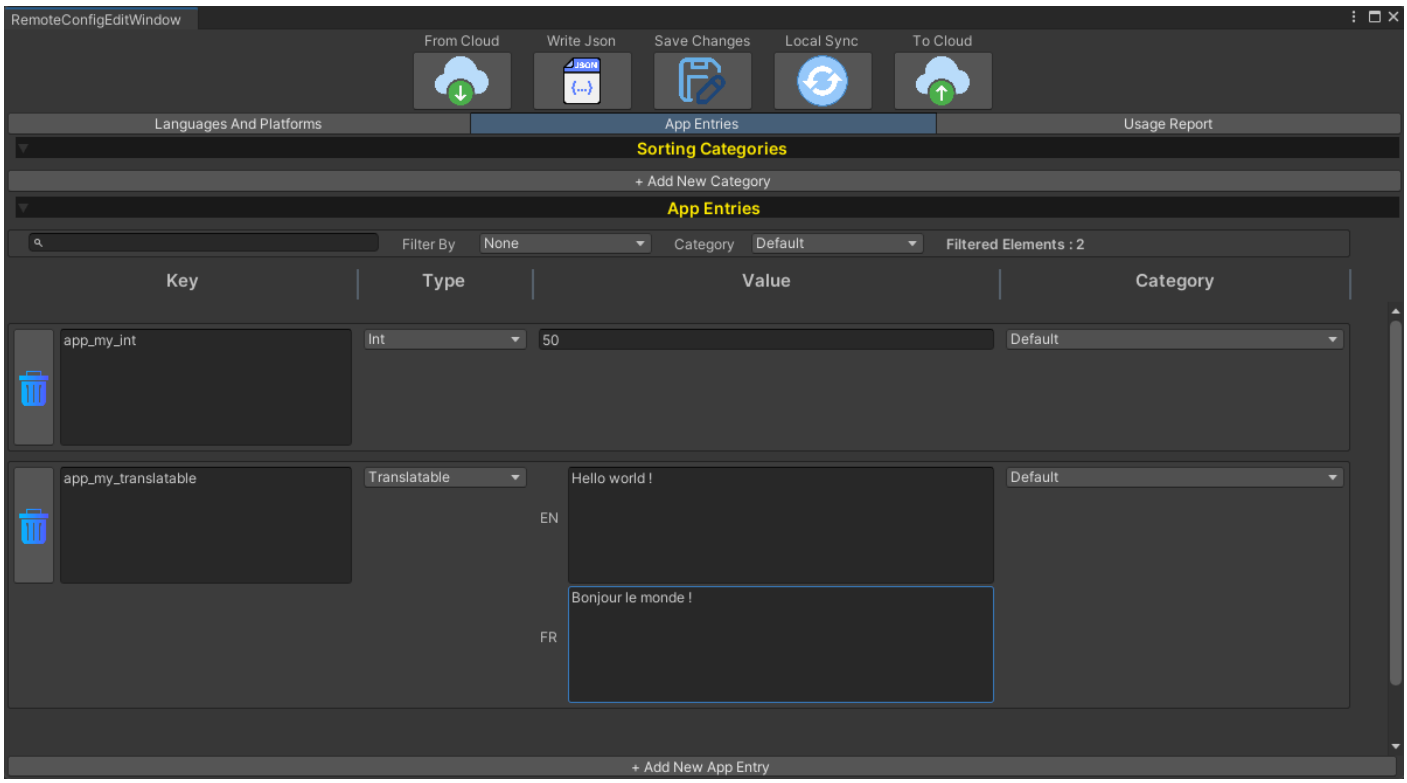
Since the key is empty, the system tells you that the key is invalid. This message will also be displayed if your key is duplicated. Enter something in the Key field to see your actual data :



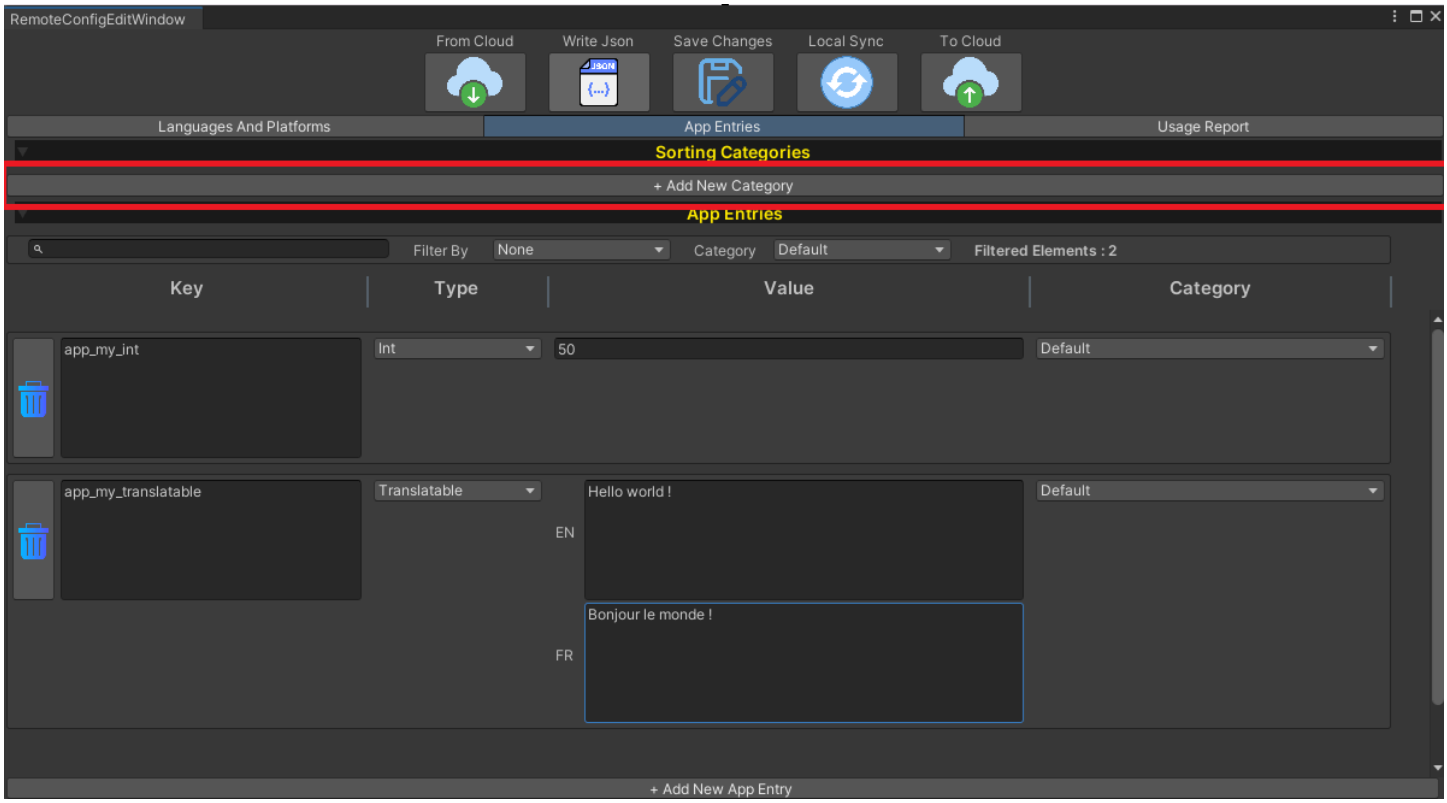
There are five types of entries : int, float, string, bool and translatable. Int, float, string and bool are quite evident so I won't detail what they do. The translatable type however is a bit different; it will have an entry for each language that you have added in the "Languages And Platform" section. Let's say that you have added English and French languages, and that you add a new translatable entry; you should see something like this :



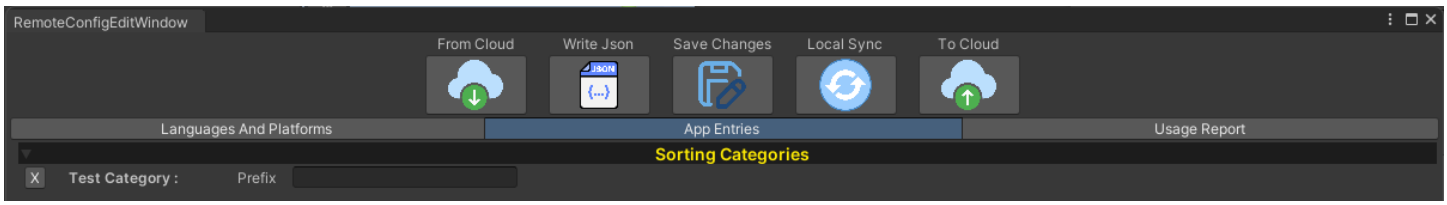
You can now add some text for each of your languages :



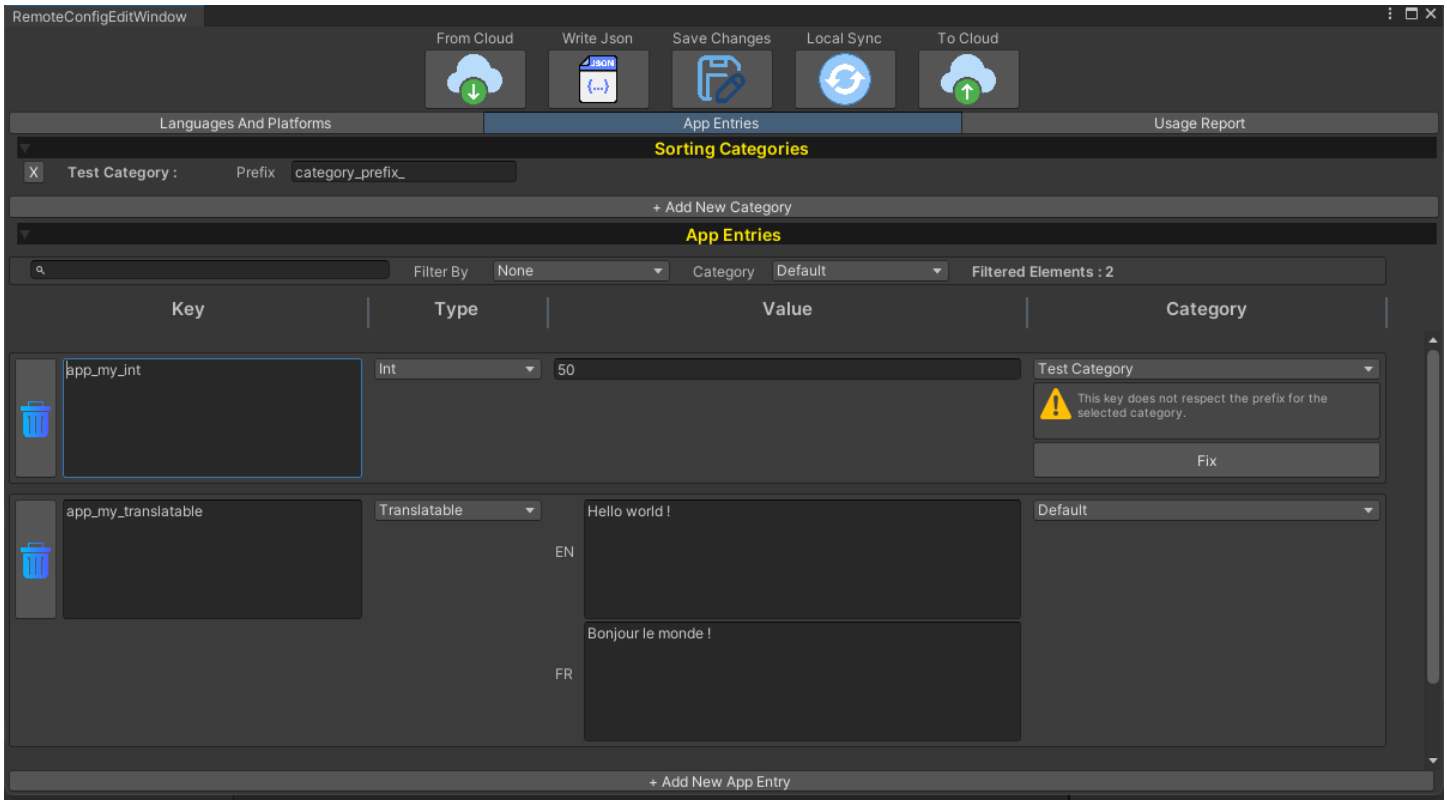
You may have noticed that there is a “Sorting Category” foldout on top of this section, and a “Category” dropdown on the right of each entry. This is something you can use to group and sort your entries. Currently no category has been added, so all entries are assigned the “Default” category. To create a new category, click the “+ Add New Category” button on top :



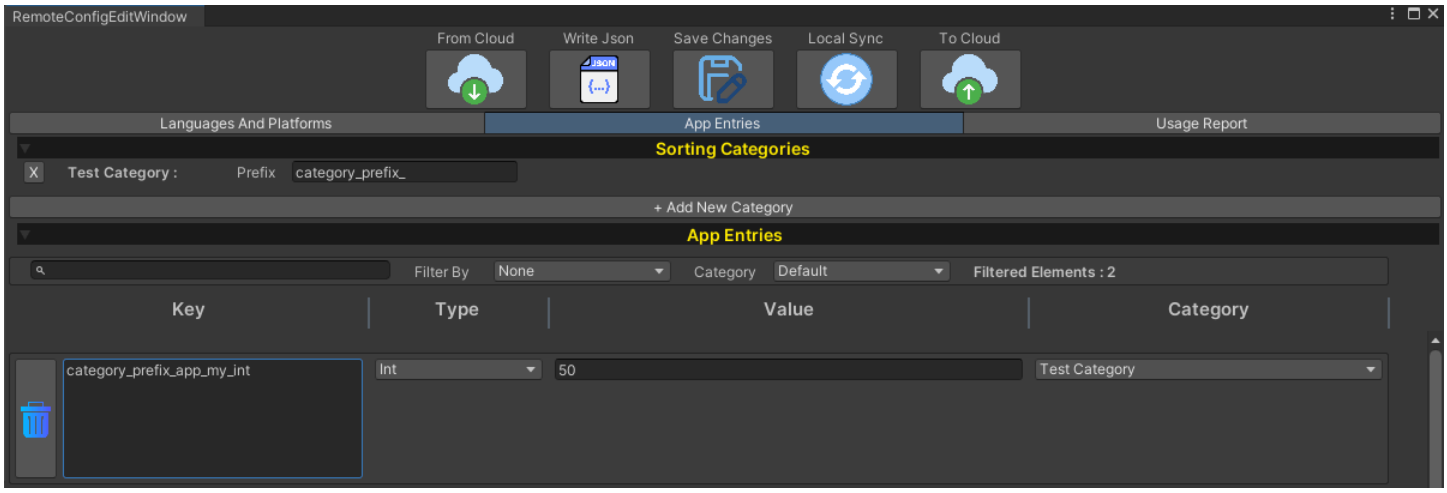
A text field will appear in place of the button, where you can type your category name. Once done, you should see something like this :



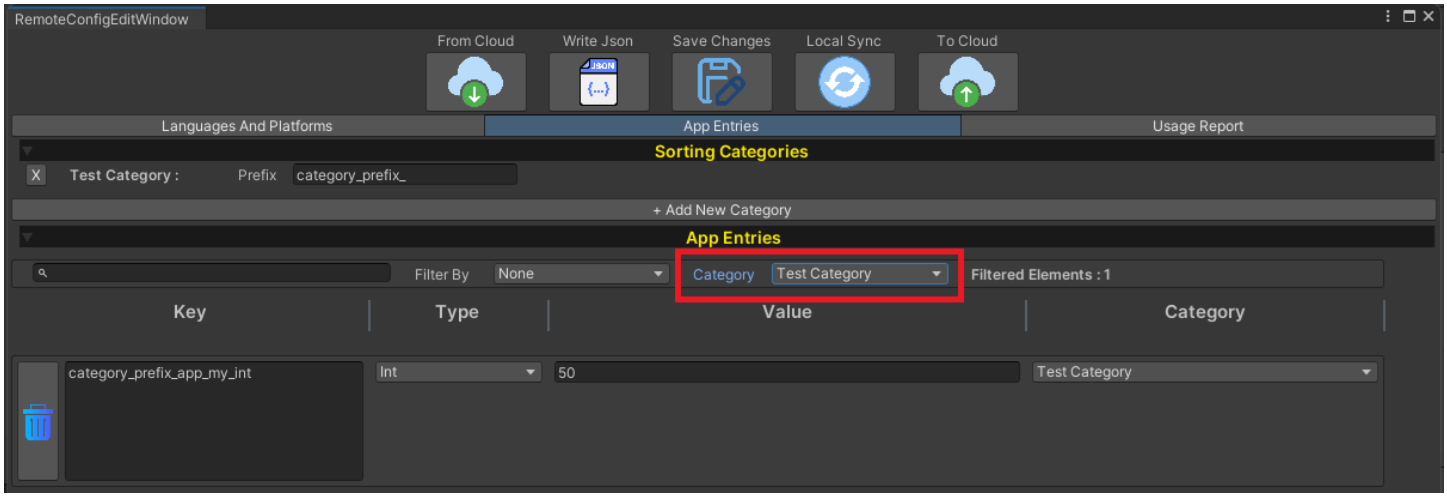
You can delete a category by clicking the small “X” button next to it. You can also specify a prefix by which entries tagged with this category should have their key starting with. Let’s add a prefix and assign an entry to this new category :



In my case I have added the prefix “category_prefix_” to my new category, and I have tagged my first entry with this category. As you can see, a warning is displayed telling me that my key does not respect my prefix. If you click the “Fix” button, it will automatically add the required prefix to your key. You can also manually edit your key to make it start with your prefix :

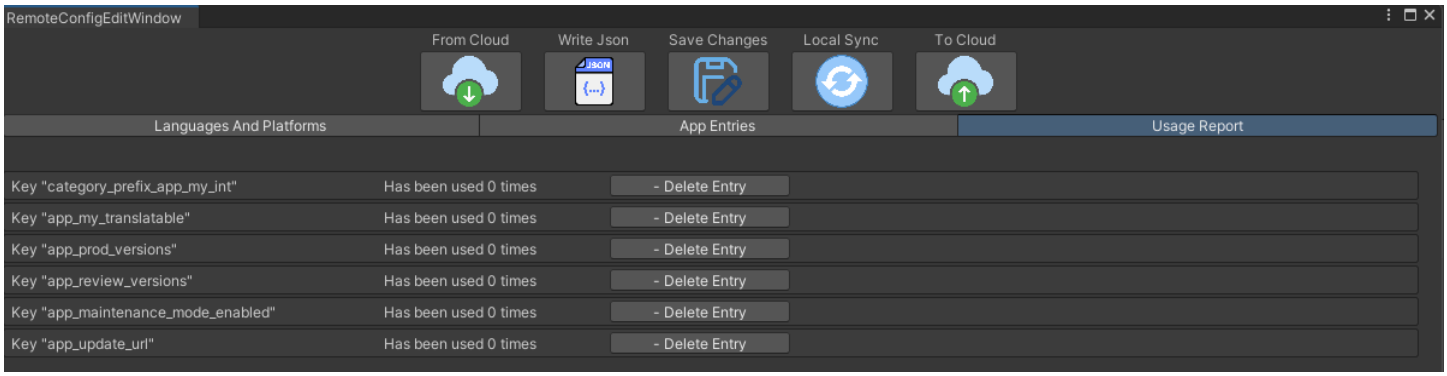


You can now filter your entries by category, to only display the entries of a specific category. To do so simply select your category in the filter header :



Note that you can also filter by type and by key, and all those filters apply together.
This concludes the “App Entries” section.

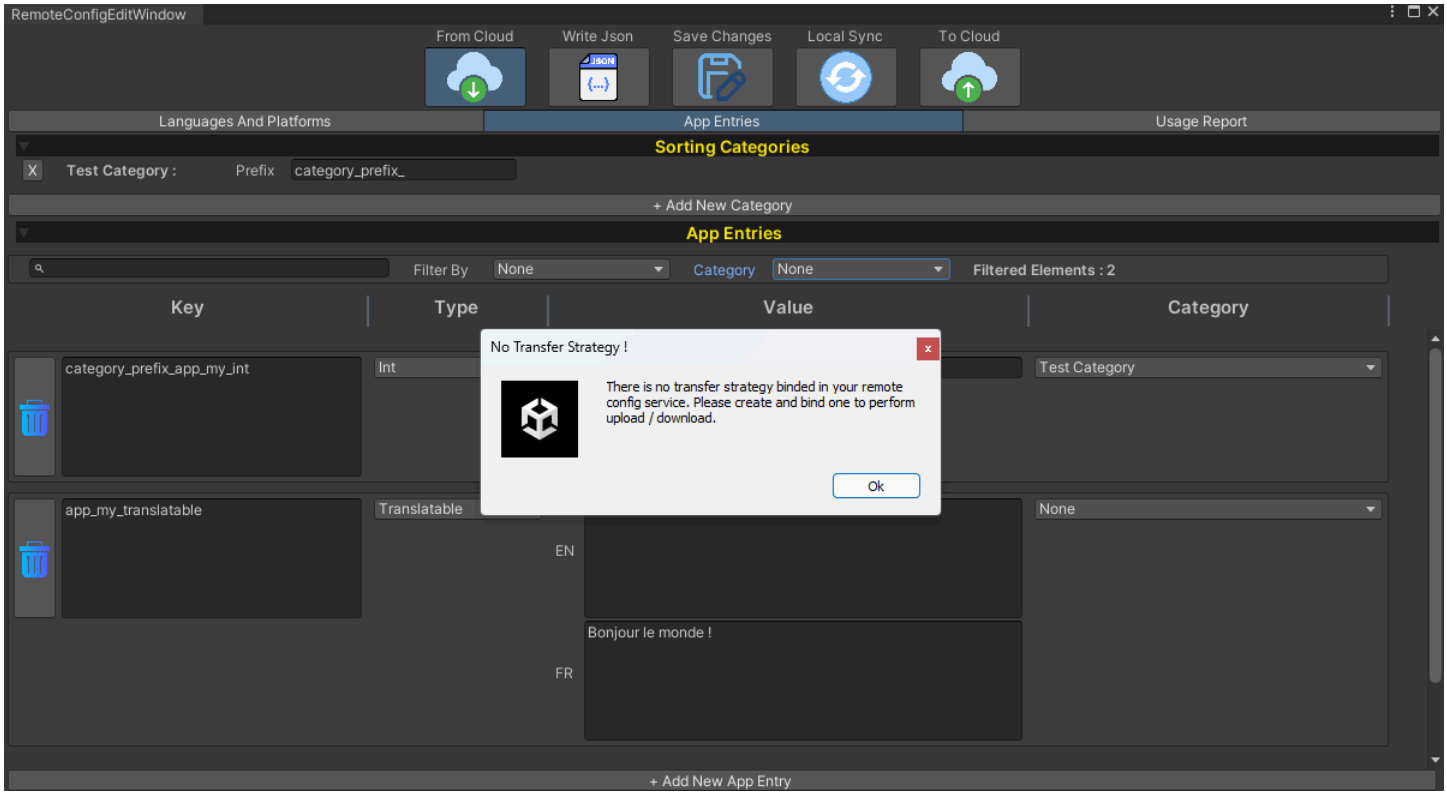
The third and last one is the “Usage Report” section, which gives you information about how many times each entry has been used. For the moment you probably have never accessed an entry during runtime so you should have something like this :



Entries have all been used 0 times. This value will increase every time you access an entry through the RemoteConfigService methods, **in editor only.** This section is here to give you a quick overview of your key usage, and give you a simple way to delete entries that are not being used much.

You now know almost everything to edit your remote config file in the editor. If you remember well there is one last thing we need to discuss, and that’s how to upload and download your file. After all, there is “Remote” in “Remote Config” !

So, if you try to click the upload or the download button you should see an error message popping like this :



It basically tells you that it doesn't know how to perform an upload/download. Since you probably have your own server (Firebase, Drive, private server, etc…) and your own way to communicate with it, you have to implement an adapter for the remote config system to use it. It is what is called a TransferStrategy. This is an abstract ScriptableObject that your custom script will derive from, and that you will link to the RemoteConfigService file. To create your custom transfer strategy, create a new script, make it inherit from the RemoteConfigTransferStrategy class and implement the abstract methods. As an example I will be creating one to download from Google Drive :

```
public class GoogleDriveTransferStrategy : RemoteConfigTransferStrategy
{
    Frequently called  0+1 usages
    public override void UploadJson(string json, Action<float> onUploadProgressed, Action onUploadSucceeded, Action onUploadFailed)
    {
    }

    Frequently called  0+2 usages
    public override void DownloadJson(Action<float> onDownloadProgressed, Action<string> onDownloadSucceeded, Action onDownloadFailed)
    {
    }
}
```

Downloading a public file from Google Drive is quite easy but uploading a file needs way more setup, so I will only implement the download logic.
Let's start by adding some fields and the CreateAssetMenu tag :

```
[CreateAssetMenu(fileName = "GoogleDriveTransferStrategy", menuName = "CCLB Studio/Remote Config/Transfer Strategies/Google Drive Strategy")]
No asset usages
public class GoogleDriveTransferStrategy : RemoteConfigTransferStrategy
{
    [SerializeField] private string fileId;  Unchanged
    private const string BaseUrl = "https://drive.google.com/uc?export=download&id=";
    Frequently called  0+1 usages
    public override void UploadJson(string json, Action<float> onUploadProgressed, Action onUploadSucceeded, Action onUploadFailed)
    {
    }

    Frequently called  0+2 usages
    public override void DownloadJson(Action<float> onDownloadProgressed, Action<string> onDownloadSucceeded, Action onDownloadFailed)
    {
    }
}
```

Then I create a method for my request coroutine :

```
    Frequently called  2 usages
    private IEnumerator DownloadRoutine(Action<float> onDownloadProgressed, Action<string> onDownloadSucceeded, Action onDownloadFailed)
    {
        string url = BaseUrl + fileId;
        using (UnityWebRequest webRequest = UnityWebRequest.Get(url))
        {
            webRequest.SendWebRequest();
            while (!webRequest.isDone)
            {
                Debug.Log( message: $"Download progress : " + webRequest.downloadProgress);
                onDownloadProgressed?.Invoke(webRequest.downloadProgress);
                yield return null;
            }

            if (webRequest.result == UnityWebRequest.Result.Success)
            {
                Debug.Log( message: "Google Drive file downloaded succeeded !");
                string downloadedJson = webRequest.downloadHandler.text;
                onDownloadSucceeded?.Invoke(downloadedJson);
            }
            else
            {
                Debug.LogError( message: "Problem during download : " + webRequest.error);
                onDownloadFailed?.Invoke();
            }
        }
    }
}
```

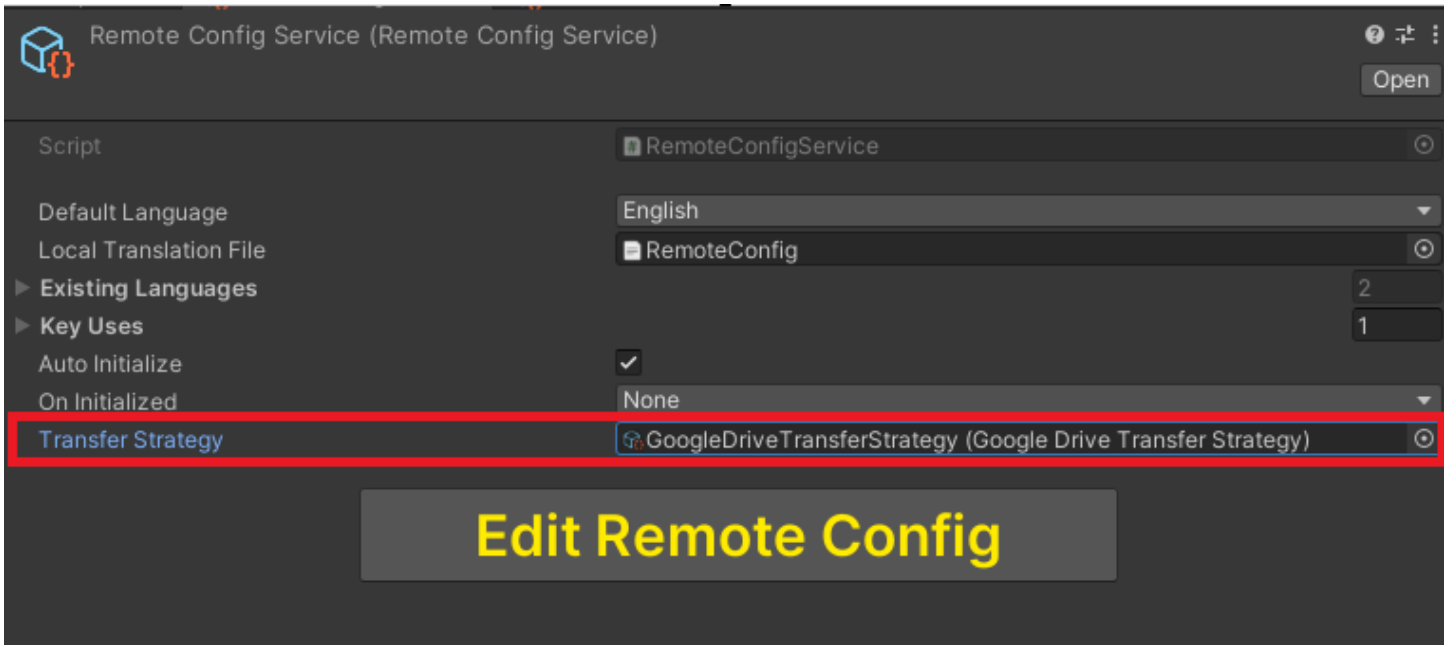
I can now call this coroutine from the DownloadJson method. Note that this method may be called in both editor and runtime scripts, so we need a bit of setup to make it work in both scenarios :

```
Frequently called  0+2 usages
public override void DownloadJson(Action<float> onDownloadProgressed, Action<string> onDownloadSucceeded, Action onDownloadFailed)
{
    if (Application.isPlaying)
    {
        if (!_coroutineRunner)
        {
            _coroutineRunner = new GameObject( name: "CoroutineRunner");
        }
        _coroutineRunner.AddComponent<CoroutineRunner>().StartCoroutine(DownloadRoutine(onDownloadProgressed,
            onDownloadSucceeded: json :string =>
        {
            Destroy(_coroutineRunner);
            onDownloadSucceeded?.Invoke(json);
        }, onDownloadFailed: () =>
        {
            Destroy(_coroutineRunner);
            onDownloadFailed?.Invoke();
        }));
        return;
    }

#if UNITY_EDITOR
    EditorCoroutineUtility.StartCoroutineOwnerless(DownloadRoutine(onDownloadProgressed, onDownloadSucceeded, onDownloadFailed));
#endif
}
```

What happens here is quite simple :
If the Application.isPlaying flag is true, then we create a gameObject that will launch the coroutine for us (since it cannot be launched from a ScriptableObject). When we receive the result callback from the request, we destroy the generated gameObject and we fire the callbacks.
If the Application.isPlaying flag is false, then we use an editor utility class to launch the coroutine. Note that it's important to use the conditional compilation flag #if UNITY_EDITOR since the utility class is part of an editor API that will not be shipped with the project when you build it.

That’s it for this script ! To test it I need to create an instance of my strategy scriptable object and link it in my service file :



Then I just need to put my json file in a public GoogleDrive folder and paste its id in the File Id field of my scriptable object. Et voilà ! We can now download our remote config file, and this both from the editor window and runtimes scripts.
Speaking of runtime scripts, it may be a good time to explain how to get your data while in game.As an example, I created a Demo scene in which I load my remoteConfig file from my GoogleDrive folder and display some text. Let’s see how it works.

First of all, let's have a quick overview of the methods you can use from the Remote Config Service object :

- Initialize() : this method will reset any data and prepare everything for a clean load. It can be automatically called if you toggle the "Auto Initialize" option.
- LoadFromCloud() : This will download your json file using the provided transfer strategy and load its data. It takes three actions as arguments, so you can have callbacks.
- LoadFromLocalFile() : This will load the data from the json file linked.
- SelectLanguage() : This changes the current language you are using. If you already have loaded your data, it will also refresh all the translations stored.
- GetBool(), GetFloat(), GetString(), GetInt() : those are the methods to get a value from a key. Note that all your translatable entries can be accessed through the GetString() method. Those methods return a boolean value telling you if the key has been found or not. The value itself is given through an out parameter.
- AddListener(), RemoveListener() : If you want to have some callbacks when a json file is loaded or when another language is selected, you can implement the IRemoteConfigListener interface in any of your custom script and register/unregister them as listeners through these methods.

Let's see how we can use those methods. The demo scene has a RemoteConfigTester script in it which is a simple MonoBehviour. Its Start() method is defined as is :

```
public class RemoteConfigTester : MonoBehaviour
{
    [SerializeField] private RemoteConfigService service;  RemoteConfigService.asset
    [SerializeField] private SystemLanguage language;  English
    [SerializeField] private TextMeshProUGUI text;  Text (TMP) (TextMeshProUGUI)

    Event function
    void Start()
    {
        service.SelectLanguage(language);
        service.LoadFromCloud( onProgress: null, onSuccess: DownloadSucceeded, onFail: DownloadFailed);
    }
```

It simply selects a language and launches the json download, with no progress callback and two methods for the success and fail callbacks : DownloadSucceeded and DownloadFailed. These callbacks are quite simple :

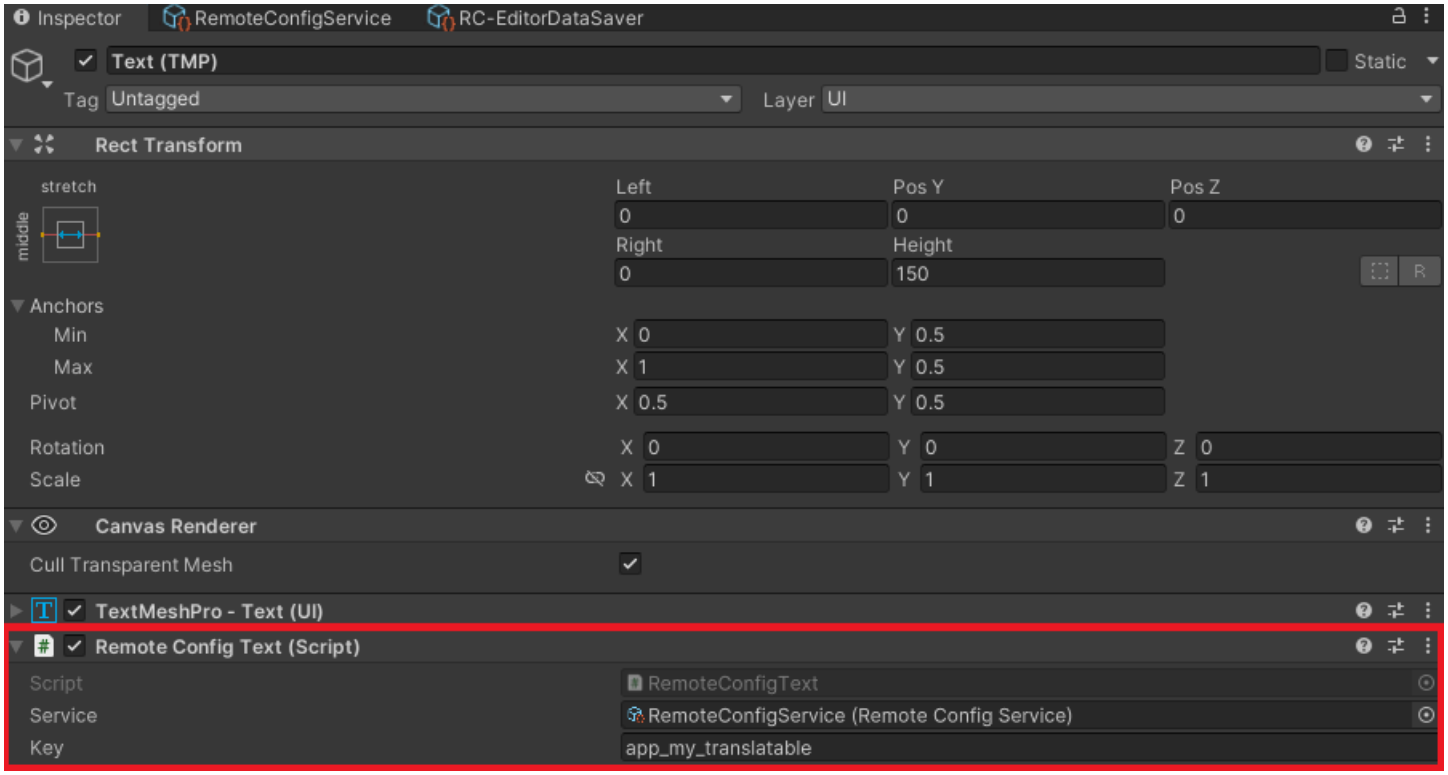
```
private void DownloadSucceeded()
{
    Debug.Log( message: "Remote config successfully downloaded from cloud and loaded into the service !");
    if (service.GetString("app_my_translatable", out string value))
    {
        text.text = value;
    }
}

 Frequently called   1 usage
private void DownloadFailed()
{
    Debug.Log( message: "Failed to download remote config. No data loaded.");
}
```

They don’t do much : the success callback simply assigns a string value to a TextMeshProUGUI element, and the fail callback just logs a message. It also has an Update() method, in which the current language is swapped between english and french whenever I press the space bar :

```
private void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        service.SelectLanguage(service.CurrentLanguage == SystemLanguage.English ? SystemLanguage.French : SystemLanguage.English);
        if (service.GetString("app_my_translatable", out string value))
        {
            text.text = value;
        }
    }
}
```

Here I need to reassign the text every time I change the language, which can be hard to handle if you have dozens of texts. To help you, there is a script you can attach to any TextMeshProUGUI in which you want to display a value from your remote config : the RemoteConfigText script. It has a serialized field to hold a key from which it will try to get a value to display in the TextMeshProUGUI component attached to it :



This script will perform a GetString() during the Start() method and every time the SelectLanguage() method of the service is called.

This concludes the documentation. If you have any questions, feel free to email me at cclbstudio@gmail.com. Enjoy using this tool and I hope it will help you create awesome games !