Additional Provided Class Information

Created with reference to version 2.2.0

SingletonInstance<T>

SingletonInstance is my own implementation of the Unity singleton model. This is applied to most primary gameobjects in this package.

Listing the actions this class takes:

- Stores a static instance of the inherited class
- Moves the gameobject into DontDestroyOnLoad (if PersistBetweenScenes)
- Renames the gameobject to reflect the singleton
- Implements virtual Awake, Start and OnApplicationQuit to be overridden
- Sets execution order to -1 by default (0 is Unity default)

Because of these changes, it is highly recommended not to place additional scripts onto objects hosting singletons using this script. The same applies for overlapping singleton scripts/multiple singletons placed on the same gameobject.

To make a script implement SingletonInstance, the script must inherit SingletonInstance with T being the class that's inheriting it. For example [public class TwitchAPIClient : SingletonInstance<TwitchAPIClient>]

If you wish to acquire the current instance from the scene you can call the static method on the script GetInstance(out T) which will return the instance on the out, if it's not available the function will return false. Alternatively if you need an instance in the scene but there isn't one available you can attempt to create one from the function CreateOrGetInstance(out T).

SingletonDispatcher<T>

SingletonDispatcher is a Singleton class which implements SingletonInstance, its purpose is to implement LateUpdate virtually to be inherited later and is used to execute actions on the main thread. This is done because many actions unity performs can only be done on the main thread and with async capabilities, a dispatcher is required to ensure Unity runs required methods.

InternalSettingsStore

InternalSettingsStore is my own implementation of PlayerPrefs storage in Unity.
Provided is an enum called SavedSettings which you can add to to store additional settings. To save a setting use the function EditSetting, your desired SavedSetting and the value you would like to store as a string, if you would like to store JSON then storing as a string is fine. All values are saved as strings and are extracted into your selected value type in TryGetSetting<T> T being a base type such as string, int ext.

EditorPaged

EditorPaged is an Editor only class provided to simplify adding pages to Unity Inspectors. It is something I created after getting annoyed with the Enum/Dropdown/Switch structure of the Editors.
To use it, instead of inheriting your editor class with Editor, instead inherit it from EditorPaged. EditorPaged does not allow you to inherit the usual OnInspectorGUI and has sealed it so it cannot be changed. Instead you will override the CreateInspectorGUI, inside this function you will return the base.CreateInspectorGUI() as normal at the end.
To add change the label next to the dropdown, inside this function you edit the PopupName value.
To change what page the inspector shows outside of the dropdown, change the SelectedKey value.
And lastly to add a Page, it is stored as a dictionary<int, Page>, the int is the position in the dropdown with the page data. When creating a new page, it will ask you for a Name which appears in the dropdown and a Action. Inside this Action is where you draw your inspector page as you would usually do inside a OnInspectorGUI.

TwitchBadgeManager

TwitchBadgeManager is an autonomous SingletonInstance that is built to download and manage all the images provided to it by the IRC in relation to a chatters badges in the connected IRC channel. It does this by having an assigned set of coroutines to download the images, once downloaded they are catalogued to be built into a TextMeshPro image atlas.

Once available an image can be requested by a TwitchMessage to display the rich text to show the image. Images are identified by their set ID and version value. Global badges unfortunately overlap with channel badges when getting and setting them so both are stored, if a channels badge for a value is not found, it will default to the global badge.

You can access the list of badges using property DownloadedBadges.

If you need to wait for the badges to finish processing you can used the Busy value for if you need to know if it's working.

Currently only works with the IRC, I will re-approach this later to see if I can make this compatible with the EventSub subscriptions and to allow manual input and requests.

TwitchEmoteManager

TwitchEmoteManager is an autonomous SingletonInstance that is built to download and manage all the images provided to it by the IRC in relation to a chat message emotes in the connected IRC channel. It does this by having an assigned set of coroutines to download the images, once downloaded they are catalogued to be built into a TextMeshPro image atlas.

The manager handles both static and animated emotes (using UniGif to assist with gif processing), emotes don't have the issue of overlap like badges do as they all have unique names so when getting the TextMeshPro value only the emote name is required.

You can access the list of emotes using the property DownloadedEmotes.

If you need to wait for the emotes to finish processing you can used the Busy value for if you need to know if it's working.

Currently only works with the IRC, I will re-approach this later to see if I can make this compatible with the EventSub subscriptions and to allow manual input and requests.

TwitchMessageAtlasProducer

TwitchMessageAtlasProducer is an autonomous SingletonInstance that is responsible for taking the downloaded badges and emotes from TwitchBadgeManager and TwitchEmoteManager and compile them into textmesh atlas's.

It works by having requests sent in with the needed emotes and badges to be included and compile them into a new atlas. Requests return a guid to label the work tasks and completed atlas's. Completed atlas's also activate a event returning the id and reference to the work done. It works on a request basis to prevent frame stalling as the production of a atlas can take time.

Examples of how to use this singleton to acquire an Atlas can be found in the classes TwitchMessageGameObject and TwitchMessageSingleDisplayer.

This class has replaced the original TwitchMessageAtlasManager which produced and updated a single atlas texture. This can still be used from the Archive folder however this is not recommended as it has been marked deprecated and will eventually be removed.

FlaggedEnum<T>

FlaggedEnum is my take a large capacity flagged enum class, with standard enums using the flagged attribute you are limited to a total of 32 unique entries (64 if long enum, 128 in NET9). This posed a problem as I required 66~ for my scopes rework. Now with its own property drawer it is cleaned up and simplified the need for listing multiple enums in the inspector. The previous method was a literal list of enums, each entry needing to open the dropdown to select it. Now it's all handled by FlaggedEnum, it includes all typical functions and operators like a typical flagged enum to make it as similar as possible.

ExtendedUnityEvent

This event is exactly the same as a normal UnityEvent, including how it draws in the inspector. The extended part is how it counts both code added listeners and inspector added listeners to count the total listeners, something a standard UnityEvent doesn't do. When a

UnityEvent exists to be serialized, it will not be null and without this value it will be invoked when needed unnecessarily when it's empty. Although this operation in itself isn't very expensive, it adds up when these events need to be invoked on the main thread. With no check for if it has listeners it means in code, it would add an action to the dispatch queue for the main thread to invoke nothing. Extending it like this means we know if there are no listeners and the action to add it to the dispatcher can be ignored. This is significant for events such as PRIVMSG for the IRC, if you use the IRC but dont add a listener to OnPRIVMSG, each message would be triggered in the dispatcher. With the count, this no longer applies.

LightJson

LightJson is a code project by Marcos López C. (MarcosLopezC) to allow for NetStandard processing of JSON data.
This was a life saver as the desired target platform for this work was for NetStandard for maximum project compatibility.

Changes have been made to streamline the process across the reading process. Such methods include ToModelArray<T>() which translates a JsonArray into an array of the specified (T) model.

UniGif

UniGif is a code project by WestHillApps (Hironari Nishioka) to allow for gif processing in Unity and being compatible with NetStandard.

The gif processing saved loads of time from having to do it myself, I had originally done it using System.Drawing however as the project evolved, this is not a NetStandard library so UniGif saved the day for gifs in the project.

Changes were made to allow processing outside of a coroutine, primarily the original code is still present, but now you no longer need a coroutine to use it.