Persistent Asset

Quick usage guide

This tool saves data by allowing scriptable object values to persist between play sessions, even at runtime. Follow these steps to use it:

- **Create a scriptable object that inherits from PersistentScriptableObject.**

The serialized fields within this class will retain their values across sessions.

```
using PersistentAsset;

[CreateAssetMenu(fileName = "Example", menuName = "Persistent Asset/Example")]
public class Example : PersistentScriptableObject
{
    public bool persistentField;

    [field: SerializeField]
    string PersistentAutoProperty { get; set; }
}
```

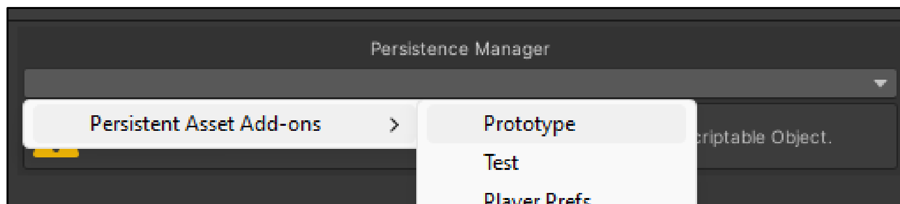
> 💡 You can also create a persistent scriptable object inheriting from any class, see below.

- **Create an instance (or instances) of the asset.**

You can use the CreateAssetMenu attribute or any other method to create one or multiple instances of your type. Each instance will maintain its own persistent data.

- **In the editor, select a persistence manager.**

The persistence manager is responsible for saving and loading, you can configure its settings in the inspector.



> 💡 The **Prototype** persistence manager will work without any configuration or actions from you!
> You can use it to begin with and replace it with another one if needed.

- **That's it! You now have a functioning persistent scriptable object.**

Outside of play mode, you can define default values. Players using your game will start with these.

During play mode, you can modify the persisted data. Note that changing these values will not affect players' data; you are only modifying your own saved data.

Your scripts can access the saved data through the scriptable object instance. Simply use a reference to your asset or the Resources class.

- **Explore the rest of the documentation to discover additional functionalities.**

Table of Contents

How does it work? ***4***

Core Features ***5***

NonPersistent attribute 5

Avoiding inheritance with composition attribute 5

Facultative interfaces 6

Persistence Manager script usage 7

ActivePersistenceManagers 7

Built-ins persistence managers ***8***

List 8

Custom Persistence Managers ***8***

Custom Serialization 8

Custom Save and Load Logic 9

Utilities ***9***

Using the editor scripts ***10***

Persistent scriptable object editor 10

Persistence manager editor 10

Troubleshooting ***10***

Demo ***11***

Content 11

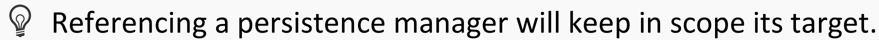
Persistent scriptable objects 11

Final words ***12***

How does it work?

When you select a persistence manager, an instance of it will be created as a sub-asset.

Once your persistent scriptable object is in scope (referenced by a scene or a script), the persistent data will be loaded automatically (unless you disable auto-load). Similarly, when it goes out of scope, it will be saved (unless you disable auto-save on asset unload).



Additionally, an automatically generated singleton **MonoBehaviour** named **AutoSaveSingleton** takes charge of saving any persistent scriptable object. It triggers the saving process during specific application events (such as on pause or on losing focus), as well as at regular intervals.

In the editor, the process becomes slightly more intricate: scriptable object instances are also loaded by opening them and will stay loaded even when no longer used.
To replicate the runtime behavior, we load the persistent data on play mode enter events and save it on play mode exit events.

To reset fields to their default values defined outside of play mode, we save them in files. This system will protect your data in case of a Unity crash.

Core Features

> 💡 Core classes are in the PersistentAsset namespace.

NonPersistent attribute

Fields within a persistent scriptable object are inherently persistent by default.

However, if you want to make a specific field non-persistent, you can achieve this by decorating it with the **NonPersistent** attribute.

```
[NonPersistent] public string nonPersistentField;

[field:SerializeField, NonPersistent]
string NonPersistentAutoProperty { get; set; }
```

By marking a field with the **NonPersistent** attribute, it will behave like any other scriptable object field: Its values will not be retained between play sessions.

This attribute can also be used on a **class** or **struct**, any field of those types will be considered non persistent.

Avoiding inheritance with composition attribute

The simplest way to create a persistent scriptable object is to inherit from **PersistentScriptableObject**, but this is not always possible.

To create a persistent scriptable object inheriting any **ScriptableObject** classes, all you need to do is:

- Add a serialized **PersistenceManager** field to your class.
- Decorate your class with the **CompositePersistentScriptableObject** attribute.
  The constructor parameter is the persistence manager field name.

```
using PersistentAsset;

[CompositePersistentScriptableObject(nameof(persistenceManagerSerializedField))]
public class Example : ScriptableObject
{
    [SerializeField] PersistenceManager persistenceManagerSerializedField;

    // Persistent fields...
}
```

> 💡 If you haven't defined a custom inspector for your class, the persistence manager inspector will not be displayed as a sub-editor. However, you can still access it using the **Open** button.

Facultative interfaces

You can access optional features by implementing specific interfaces in your persistent scriptable object class.

ISaveEnabler

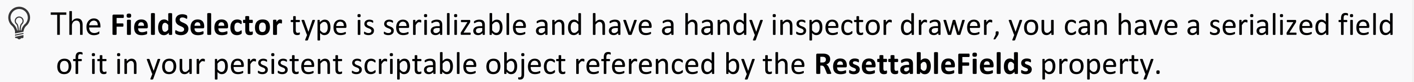
This interface allows you to control whether saves should be performed or ignored based on the **IsSaveEnabled** method.

IResettable

By implementing this interface, you can reset specific fields within your scriptable object to their default values.

The fields to be reset should be included in the **ResettableFields** property.

To reset a value from scripts, you can utilize the **FieldResetter** property of the linked persistence manager. Additionally, you can also perform resets from the inspector.



> The **FieldSelector** type is serializable and have a handy inspector drawer, you can have a serialized field of it in your persistent scriptable object referenced by the **ResettableFields** property.

IDefaultValueInitializer

Implementing this interface enables you to set default values for your asset from a script.

The **DynamicInitialize** method will be called when the asset is loaded, prior to any persistent data being loaded. This allows you to initialize default values programmatically.

IMultiSlot

This interface makes it easy to save and load data from different slots.

This is a better idea than saving a collection because it is simpler, more performant, and gives you more control on how to format your data.

Simply implement the interface in your persistent scriptable object, the **ForceSlot** property defines whether the asset can be saved and loaded without a specified slot.

You can get the slot using the **Slot** property of the linked persistence manager, and set it using the **ChangeSlot** method.

You can also get and set the **GlobalSlot** static property of **PersistenceManager**, this slot will be applied to every persistence manager, including those not loaded yet.

Persistence Manager script usage

Manual save and load

Instead of relying on the automatic save and load mechanisms, you have the flexibility to manually trigger the save and load operations using the Save and Load methods.

These methods allow you to save your data synchronously or asynchronously based on your needs.
When using asynchronous saving, you should utilize the **beforeSave** and **afterSave** delegates to handle any operations before and after the save process, otherwise you may run into some issues:

In this example the saved value will be "After saving.", which may not be what you intend:

```
somePersistentField = "Before saving.";
PersistenceManager.Save(ExecutionMode.Async);
somePersistentField = "After saving.";
```

Logging

Call the **Log** method to add messages that will be displayed in the inspector of the target persistence manager.

Any call to this method will be stripped out when making a non-development build.

Events

There are four events available that allow you to execute methods before and/or after a save and/or a load operation.

These events are triggered only when a save or load operation is not ignored.

The **Result** object passed as a parameter in the after methods provides information about the execution status.

Information properties

Persistence managers expose some properties letting you know about the current state of the target object persistent data.

You can check the load state using **IsLoading** (asynchronous only) and **LastLoadTime**. Same for the save state.

Additionally, you can see whether this persistence manager is in a state allowing auto save by accessing **AutoSaveEnabled**.
They are disabled at first to avoid overwriting not yet loaded data.

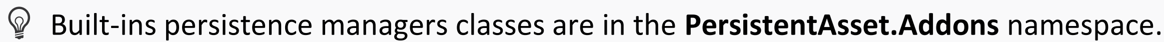
ActivePersistenceManagers

The **ActivePersistenceManagers** static class is a container for every persistence manager.

You can get a list of them or subscribe to add and remove events to be notified when a persistence manager goes in or out of scope.

Built-ins persistence managers

Persistent Asset comes with some persistence manager that should meet most if not all your needs.


Built-ins persistence managers classes are in the **PersistentAsset.Addons** namespace.

List

Prototype

The simplest of all the persistence managers, designed to work with no configuration whatsoever on any platforms.

It is particularly suitable for prototypes, hyper casual games and game jam projects.

However, if you require better performance or desire more control over the timing and method of saving and loading your data, you may want to consider selecting a different persistence manager.

Test

Do not use this persistence manager in your actual game! Its only use is to test everything you can about persistence managers and see how your application reacts to it.

It will not actually save anything.

PlayerPrefs

PlayerPrefs based persistence manager, great for saving small data in a performant manner.

You can change automatic save and load behaviors, including when to call PlayerPrefs.Save.

File

Save persistent data in files on the player computer.

This is the persistence manager with the most features; you can encrypt, compress, and define the format of the generated json. It supports creating back up files to protect player data in case of a crash, and asynchronous save and load.

Custom Persistence Managers

Custom Serialization

By default, persistence managers will follow Unity default serialization rules, but you can change that easily: all you have to do is create a new class that inherit from **Serializer** and implement the abstract members.

You can then select this serialization logic from the inspector.

Custom Save and Load Logic

If you want to save and load your persistent data another way, you can create your own persistence manager.

All you need to do is inherit from the abstract class **PersistenceManager**, you will have to implement every abstract member. Check out the public API reference or see the code summaries to know what to do.

To make this process easier, *Persistent Asset* includes a bunch of utility classes:

Utilities

Utility classes are in the **PersistentAsset.Utilities** namespace.

Text related

- **CheatProtectionUtility** lets you encrypt or append a secured hash to a text.
  It does this using a SecureString, this makes it very secured against memory dump. You should initialize the **secureKey** field yourself, if possible from an external source to avoid malicious user from getting it by decompiling your scripts.
- **GZipUtility** lets you compress and decompress a byte array or a text using Gzip.
- **PersistentJsonUtility** is the equivalent of JsonUtility but serializing and deserializing only persistent fields.

FieldInfo related

- **PersistentFieldInfoUtility** is very useful when overriding the **GetPersistentFields** method.
  It features a cache.
- **IFieldInfoFilter** is an interface, implement it to specify serialization rules to use in **PersistentFieldInfoUtility**.
  You should share a unique instance of each filter type to optimize the cache.

Other

- **SaveFileUtility** includes a bunch of classic file IO related operations (read, write, delete, etc).
  It uses a **SaveLocation** enum as parameter for ease of use.
- **TaskUtility** is very useful when making asynchronous code. Use it to create new tasks on the main thread or a background one.
  It will handle catching exception and log them to the console.

Using the editor scripts

Editor classes are in the **PersistentAssetEditor** namespace.

Persistent scriptable object editor

The **PersistentScriptableObjectEditor** is the default editor for any **PersistentScriptableObject** inheriting class. You can inherit from it to make your own editor class.

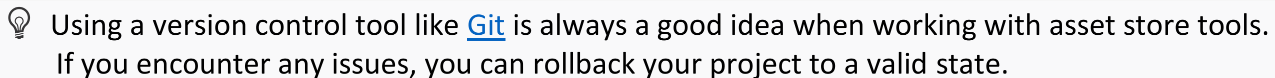
The **PersistentGUILayout** static class can be used like **GUILayout** but dedicated to drawing fields of a persistent scriptable object. Using its **PropertyField** method instead of the default unity one will display the persistent icon or the reset button if applicable.

Persistence manager editor

The **PersistenceManagerEditor** is the default editor for any **PersistenceManager** inheriting class. You can inherit from it to make your own editor class. Just overload the methods you want, calling the base one if needed.

The **PersistenceManagerLogsDrawer** class can be used to draw every log from a given persistence manager.

Troubleshooting


Using a version control tool like Git is always a good idea when working with asset store tools.
If you encounter any issues, you can rollback your project to a valid state.

A great advantage of *Persistent Asset* is that checking for any problem with your saved data is very easy. Just check the inspector for your asset.

In the persistence manager editor, you can see the logs and information properties.

If the problem only happens in build, you can use the **InGamePersistenceManagersLogsViewer** component (found in the *Demo* folder) on a **GameObject** to see any persistence manager logs.
Note that this is active only in development builds.

Demo

Content

The demo shows how you can use *Persistent Asset* in a classic game. It includes a way to see persistence manager logs in a development build, so that you can see exactly what is happening at runtime.

Launch in play mode any scene from the **Scenes** folder to see it in action.

Persistent scriptable objects

Found in *'Persistent Asset/Demo/Scripts/Persistent Asset Related/Persistent Scriptable Objects'*.

Options

Contains player defined options.

Demonstrates the use of **IResettable** and **IDefaultValueInitializer**.

Profiles

Contains the list of profiles that can be loaded using the multi-slot system.

Demonstrates the use of the **CompositePersistentScriptableObject** attribute.

CurrentProfile

Contains the player advancement for the currently used profile (or slot).

Demonstrates the use of **IMultiSlot**.

Achievements

Contains all achievements.

Demonstrates the use of **ISaveEnabler** to make a persistent scriptable object save only when data is modified. Also show how you can load a persistent scriptable object from the **Resources** class to access it with static members.

Final words

Thank you for purchasing this tool!

If you have any questions, discover a bug, or have a feature idea, please don't hesitate to contact me at justetools@gmail.com.

If you enjoy *Persistent Asset*, please consider leaving a review. Your feedback helps improve the tool!