Save Manager

3.x Documentation

Carter Games

Documentation

Documentation Contents

Documentation.....1
- Dependencies.....2
- Initial Set-up.....2
- Asset Settings.....3
- Navigation Menu Options.....5
- Save Objects.....6
- Save Values.....7
- Save Locations.....7
  - Editor Save.....8
- Save Structure.....9
  - Json Converters.....11
  - Meta-data.....11
- Save Editor.....12
  - Save Object & Values GUI.....12
  - Save Categories.....13
  - Save Slots.....13
  - Reset Save Data.....14
- Save Captures.....14
- Save Backups.....15
- Save Encryption.....15
- Legacy (2.x) Save Porting.....16
- Pre Data Load Intercept.....16
- Json Parsing.....16
- Support.....17
  - Help.....17
  - Bug reporting.....17

The save manager is a flexible & modular save system for the Unity game engine with a built-in save editor to edit the save without needing to open the save file itself.

Dependencies

The save manager requires the ***Newtonsoft Json*** package to function. This can be added to your project for free by adding the following package in the package manager from inside a Unity project:

`com.unity.nuget.newtonsoft-json`

If you are in an older version of unity such as the 2020.3.x this package is not shown in the package manager. To add it just enter the above package string via the add from git url option in unity's package manager to add it to the project.

Initial Set-up

There isn't any set-up steps bar importing the package. Any required steps are automatically handled for you. All you need to do is define the save data so the asset can save it for you.

Asset Settings

All of the asset settings can be found in the project settings menu under:
Carter Games > Assets > Save Manager

The settings can also be accessed under the navigation menu item:
Tools > Carter Games > Save Manager > Edit Settings

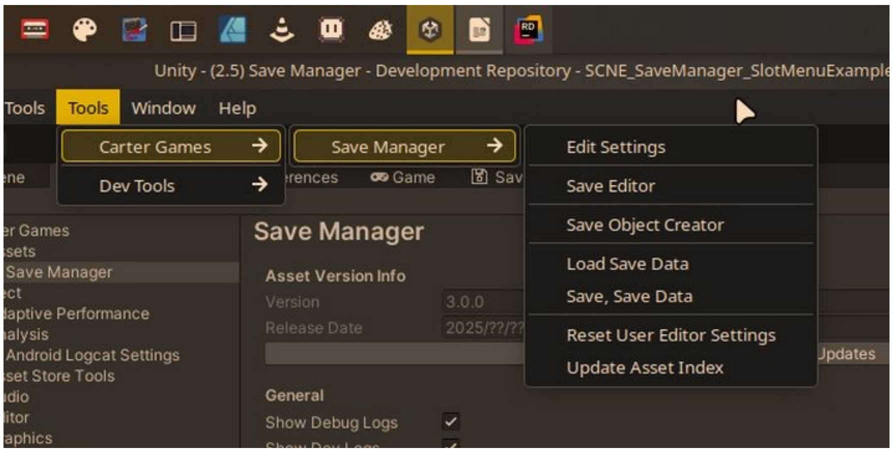
A run-down of all the available settings below:

Setting	Type	Description
Asset Version Info Check for updates	Button	Checks an API, hosted by Carter Games, to see if there is an update to the asset. The result is returned to you in an editor dialogue.
General Show debug logs	Boolean (Toggle)	Toggles if the asset shows any intended debug log messages in the console.
General Show dev logs	Boolean (Toggle)	Toggles if the asset shows more detailed developer logs that can help diagnose issues with the asset.
Save Settings Auto save on exit?	Boolean (Toggle)	Toggles if the asset will try to save the game on exiting the game.
Save Settings Use json converters	Boolean (Toggle)	Toggles if the asset uses its own built-in converters for specific edge cases like Vectors etc.
Save Settings Save location	ISaveDataLocation (Search Assigned)	Defines the active save location the asset will use for your game save.
Save Settings Editor save file	Read-only path + Buttons	Gives options to show the editor save folder & file to view. You cannot change the path of the editor save.
Save Slots Use save slots	Boolean (Toggle)	Defines if the save slots system is enabled or not.
Save Slots Limit available slots	Boolean (Toggle)	Defines if the number of save slots the set-up can make is limited or not.
Save Slots Max user save slots	Int (Slider)	Only available if ***Limit available slots*** is enabled. Defines the number of slots the set-up can produce.

Save Meta Data Game info	Boolean (Toggle)	Defines if the game info meta data the asset provides if used or not.
Save Meta Data System info	Boolean (Toggle)	Defines if the system info meta data the asset provides if used or not.
Backup Settings Total save backups	Int (Slider)	Defines the number of save backups the system will generate whenever it successfully loads a save data different to the last backup.
Backup Settings Backup save location	IBackupSaveLocation (Search Assigned)	Defines the save location for the save backups to be stored at.
Encryption Settings Encrypt save	Boolean (Toggle)	Defines if the save content is encrypted or not. NOTE: Only data under the **$content** tag is encrypted.
Encryption Settings Encryption handler	ISaveEncryptionHandler (Search Assigned)	Defines the encryption handler used to encrypt the game save content. WARNING: If making a custom handler, make sure you test your encryption option before assigning it here to avoid a loss of save data.
Legacy Save Settings Port legacy save?	Boolean (Toggle)	Defines if the asset will try to port a legacy 2.x save.
Legacy Save Settings Legacy save handler	ILegacySaveHandler (Search Assigned)	Defines the handler used for the legacy save porting setup.

Navigation Menu Options

The nav menu gives you some quick asset to useful features as well as the assets editor windows etc.



Version 3.0.0

Release Date 2025/??/??

Show Debug Logs ✓

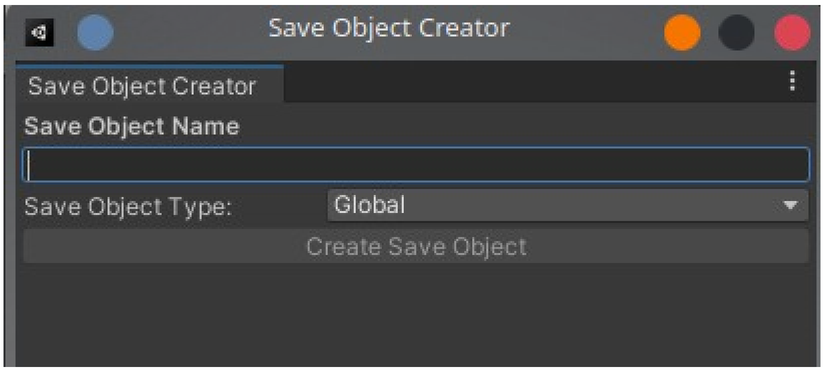
Edit Settings	Opens the settings if it is not already focused on in the editor.
Save Editor	Opens the save editor window to let you access and edit the save data from the editor.
Save Object Creator	Opens a GUI popup to aid in the creation of Save Object classes.
Load Save Data	Loads the current save location manually.
Save, Save Data	Saves the current save state to the save location.
Reset User Editor Settings	Resets any per-user settings related to the asset to their default values.
Update Asset Index	Updates the assets asset lookup in-case of any issues. Should be ignored, but is there for debugging.

Save Objects

A save object is basically a scriptable object that can store save values on it. When defined the save values on each object can be access in the editor and at runtime with ease. To make a save object you just need to make a class that implements the **SaveObject** class, or **SlotSaveObject** class if you want the data to be specifically used in save slots.

You can do this manually by making a class that inherits from the SaveObject clases or you can use the built-in SaveObject maker GUI. This can be found under:

Tools > Carter Games > Save Manager > Save Object Creator



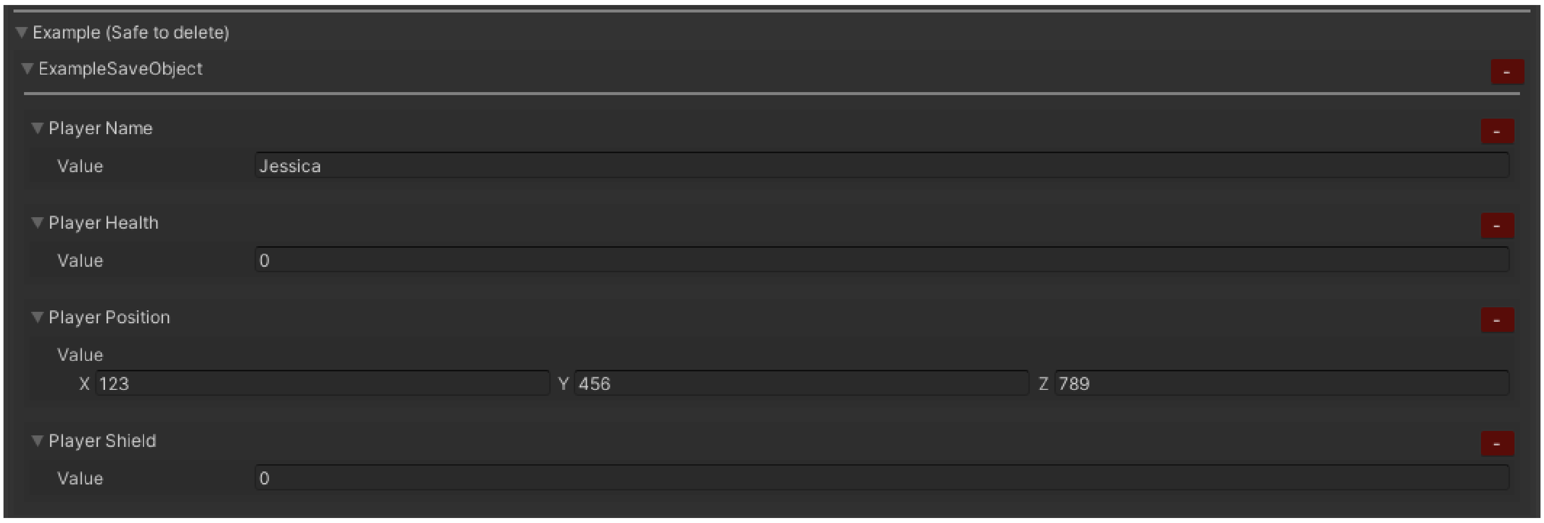
Save Object Name

Save Object Type: Global

The save object creator window has a really simple set-up. You first enter the name of the class you want to make into the ***Save Object Name*** field on the GUI. Then if you have the Save Slots feature enabled you'll be able to select between a ***global*** or ***slot*** save object. If not then it'll be global by default behind the scenes and the option will be hidden from you.

Then all you need to do is press the ***Create Save Object*** button and choose where in the project's assets folder the class should go. Once you confirm the location for the class it will be generated automatically for you.

All correctly set-up save values can be viewed in the ***Save Editor*** window. Either under the ***Global Data*** or ***Save Slots*** tab dependant on which ***Save Object*** class it is under. An example from the example save data:


Example (Safe to delete)

ExampleSaveObject

Player Name
Value Jessica

Player Health
Value 0

Player Position
Value
X 123 Y 456 Z 789

Player Shield
Value 0

Save Values

A save value defines an entry in the game save. You define save values by using the generic save value class ***SaveValue<T>*** as a field on in a ***SaveObject*** class. A valid save value **MUST:**

- Be placed inside of a class that inherits from ***SaveObject/SlotSaveObject***, if not it will not function correctly.
- Be of a serializable type.
- Have a uniquely defined save key.

You have the option to also define a default value for the save value if you wish, but this is totally optional.

An example of a defined save value below:

```
[SerializeField] private SaveValue<int> lastTimestamp = new
SaveValue<int>("lastTime");
```

Save Locations

The asset provides a couple different save locations to use. These are:

Local File	Stores the save data in a file in the application persistent data path.
Player Prefs	Stores the save data in Player prefs, but split into chunks so it can all be saved.

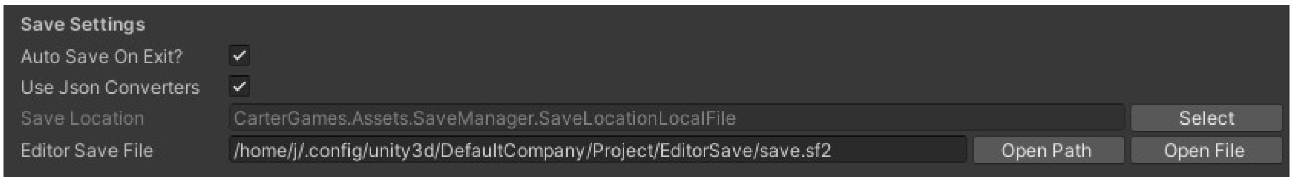
You can define custom storage locations should you wish by implementing the following interfaces into your own classes:

IDataLocation	To define a location where data can be saved to or loaded from.
ISaveDataLocation	To define a ***DataLocation*** that can be used to save/load the game save data specifically.

You can change the location that is used in the asset settings at any time. However, if you have a released product, it is advised you keep the same location throughout to avoid any issues with changing between the different locations. A change in location will be handled on the next load if the previous is different to the currently set.

Editor Save

The editor has a separate save file to your built runtime. This is so you can edit the save in the editor freely without conflict with a build version of your game on the same machine. The editor save will be stored in your projects persistent data path under the **/EditorSave** folder. This can be accessed from the asset settings open folder option next to the editor save path label:



Save Settings

Auto Save On Exit?	✓		
Use Json Converters	✓		
Save Location	CarterGames.Assets.SaveManager.SaveLocationLocalFile		Select
Editor Save File	/home/j/.config/unity3d/DefaultCompany/Project/EditorSave/save.sf2	Open Path	Open File

The editor save will update on certain conditions to avoid performance hits when making edits. These are:

- Saving the project
- Closing Unity
- Entering play mode (pressing play in editor)
- Script recompilation

The editor save will also only save when there are changes to be saved.

Save Structure

The 3.x save structure divides the content of the save into a few different sections. The main game save is stored under the **$content** tag. It is then split the following:

$global	Any game save data that is not used for the save slots system.
$slots	Sores info about the slots system as well as the data for all the slots themselves.

There is also an additional section under the **$metadata** tag which stores read-only info for context about the game & the users system which is completely optional.

Each save value defined is split into a object with 3-4 tags per entry. These are:

$key	Defines the key the save system has defined for the value.
$value	Holds the Json value.
$type	The type the save value is.
$default	Defines the custom default value for the save value. Only appears if defined.

An example:

```json
{
  "$key": "examplePlayerHealth",
  "$value": 6,
  "$type": "System.Int32",
  "$default": 10
}
```

An example of a populated save data from the simple sample scene below:

```
{
  "$content": {
    "$global": [
      {
        "$key": "examplePlayerName",
        "$value": "Bob",
        "$type": "System.String"
      },
      {
        "$key": "examplePlayerHealth",
        "$value": 50,
        "$type": "System.Int32"
      },
      {
        "$key": "examplePlayerPosition",
        "$value": {
          "x": 1.0,
          "y": 2.0,
          "z": 3.0
        },
        "$type": "UnityEngine.Vector3"
      },
      {
        "$key": "examplePlayerShield",
        "$value": 5,
        "$type": "System.Int32"
      }
    ]
  },
  "$meta_data": {
    "$game_info": {
      "$version": "0.1",
      "$save_date": "2026-01-09T18:52:31"
    },
    "$system_info": {
      "$os": "Linux 6.17 Fedora Linux 43 64bit",
      "$cpu": "AMD Ryzen 7 5800X 8-Core Processor",
      "$ram": "32006MB",
      "$gpu": "AMD Radeon RX 6800 (radeonsi, navi21, LLVM 21.1.5, DRM 3.64,
6.17.12-300.fc43.x86_64) (13436MB)"
    }
  }
}
```

Json Converters

The save manager set-up automatically saves any serializable fields which are either:

- public exposed fields
- [SerializeField] private fields

These values need to also be serializable in Unity for them to be saved.

This is a custom set-up that only select fields as the standard set-up for Newtonsoft Json, which the asset uses for its Json set-up, is to serialize both fields and properties. You can freely edit the Json of any of your custom types by making a converter class that inherits from the **SmJsonConverterBase<T>** class.

Meta-data

Meta-data is extra data that is added to a separate section of the game save. The purpose is to show important info such as game version & basic system info to aid with debugging etc. This section of the save is never encrypted, so it is always readable. If no meta-data can be written to the save then the section will be omitted from the save entirely.

The asset has two default implementation that are enabled by default. You can disable them in the asset settings with a simple toggle for each one.The GameInfo meta-data class displays the save date & version number from the player settings. While the SystemInfo meta-data displays the OS/CPU/RAM/GPU info of the users system.

You can easily make your own meta-data set-ups to add to the save by making a new class that implements the **ISaveMetaData** interface.

Save Editor

The save editor is the intended way for you to edit the save of your game. You can open the save editor window from the navigation menu's **Save Editor** option.

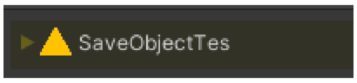
Tools > Carter Games > Save Manager > Save Editor

The save editor is split into 4 tabs:

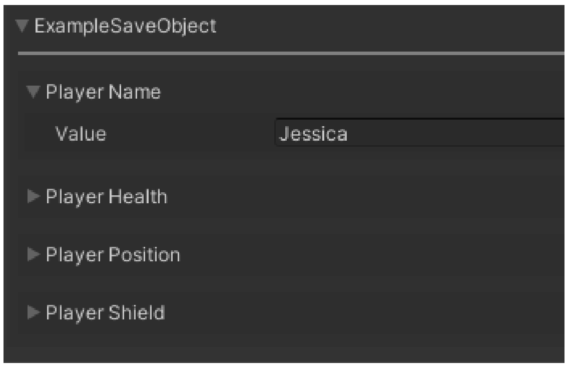
Global Data	Displays all the global save objects and their save values.
Save Slots	Displays all the save slots currently defines in the save and their save objects / save values.
Save Captures	A tool to help you store particular save files as a backup you can reload into your current save at any time.
Save Backups	Lets you view the current save backups and make save captures from any backup should you wish.

Save Object & Values GUI

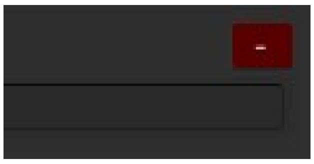
In the save editor GUI each save object is its own drop-down group labelled the save as the save objects class name. If there are any issues with a save value under a save object you will see a warning GUI over it like so:



When you expand a save object in the editor you'll see drop-downs for all the save values defined on that save object. Expanding any of these will reveal the value currently stored on that save value for you to edit:

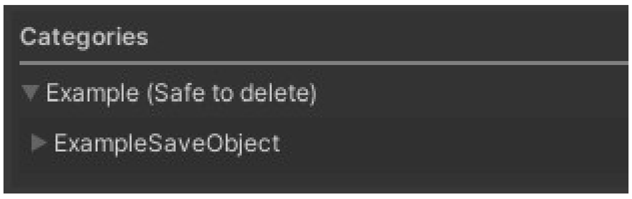


Any edits made to save values here will apply when the editor save next updates. You can press the red minus button next to and save object or save value to reset it to its default value. You will be asked through an editor dialogue to confirm this action.



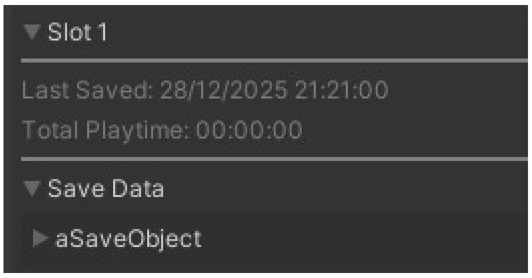
Save Categories

Any save objects that are defined with the **[SaveCategory]** attribute on them will appear under the categories section instead of the uncategorised section above it. If multiple save objects are in the same category, they will appear together under the same drop-down.



Save Slots

Save slots are displayed with each slot being its own drop-down in the save slot stab. Slots will have the last time they were saved as well as their total active playtime stored as read-only data in the save editor. These are just slot data the asset automatically stores for each slot and you should need to touch them. Underneath you'll see the save data drop-down which matches when expanded works just like the global tab with save object / save values for that slot appearing underneath for editing.



Last Saved: 28/12/2025 21:21:00
Total Playtime: 00:00:00

You can reset the meta-data for a slot with the relevant button on the slot.

Pressing the red minus button on the slot itself will delete the slot completely. You will be prompted to confirm this action. You can also add new slots in the editor with the add slot button. Note that at runtime a user will not have any slots. This only works for editor purposes to aid with testing etc.

Reset Save Data

You can reset all save data from the global or slots tabs in the save editor. There is a big reset save button at the bottom of the GUI. You will be prompted to confirm this action before it is performed.

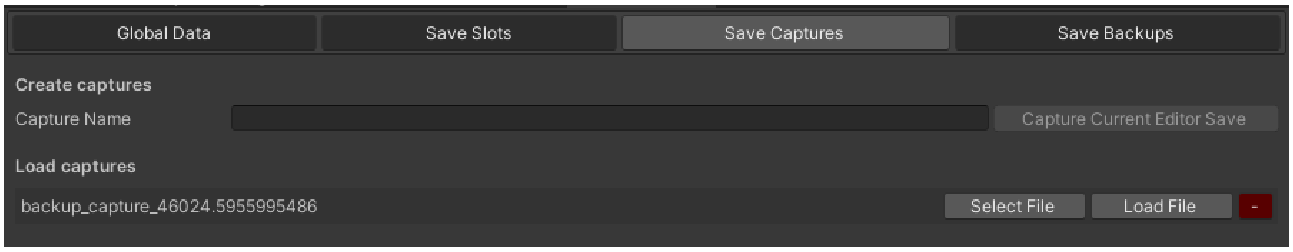
Save Captures

Save captures are save files that are stored in a text file in the project for you to load from at any time. They are good for when you need a backup of a game save state the can be shared and tested with in the editor.

All save captures are stored at the following hard-coded directory:

Assets > Plugins > Carter Games > Save Manager > Captures

You can make captures from the **save editor**



Here you can make new captures and manage them. You can make a capture of the current editor save with the top GUI. Simply enter a name for the capture and press the create capture button.

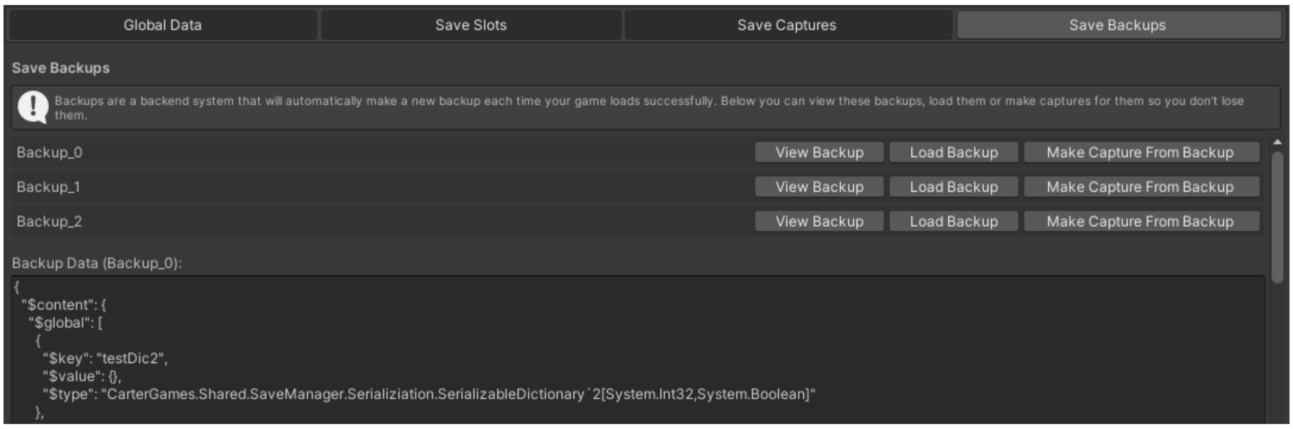
The remaining GUI will show a list of all the save captures in the project. Each capture will display the following options:

Select File	Selects the capture file in the project tab.
Load File	Will attempt to load the capture in to the editor save data.
-	Deletes the capture from the project. You will be prompted to confirm this action.

Save Backups

The asset will automatically store one or several backups of your save data when it loads without errors when the data is different to the last stored in a backup. In the event a save fails to load it will instead try to load a backup. If all backups fail to load you'll get an GameLoadFailed (1210) error code.

You can view save backups from the editor with the save editor window in the save backups tab:



Save Backups

Backups are a backend system that will automatically make a new backup each time your game loads successfully. Below you can view these backups, load them or make captures for them so you don't lose them.

Backup_0	View Backup	Load Backup	Make Capture From Backup
Backup_1	View Backup	Load Backup	Make Capture From Backup
Backup_2	View Backup	Load Backup	Make Capture From Backup

Backup Data (Backup_0):

```
{
 "$content": {
  "$global": [
   {
    "$key": "testDic2",
    "$value": {},
    "$type": "CarterGames.Shared.SaveManager.Serializiation.SerializableDictionary`2[System.Int32,System.Boolean]"
   },
```

Here you can view the backups and perform a few actions:

View Backup	Displays the backup save data in the GUI to view.
Load Backup	Tries to load the backup as the current save data in the editor.
Make Capture From Backup	Tries to make a save capture from the backup data so it is not lost when new backups are made.

Save Encryption

You can encrypt the save content with the built-in set-up. This will encrypt the content data only. So meta-data will still be readable as this is intended for read-only info. By default this set-up is disabled.

The asset provides a super basic AES encryption handler to encrypt the save if you wish. This isn't super secure and any determined attacker will easily get through it. The average user would not be able to read the save right away though.

You can also implement your own encryption set-up by making a class that implements the **ISaveEncryptionHandler** interface. This is recommended if you want a more secure save as I cannot package the best solutions without adding more dependencies or more complex set-ups.

Legacy (2.x) Save Porting

The asset comes with a basic set-up that should port and 2.x save data into 3.x global save data where the keys & types match between versions. It won't work for slot save data due to the legacy set-up not supporting such a feature.

The set-up will try to port a legacy save if one is found and then move it once processed to a legacy location as a backup. This set-up is enabled by default, but can be disabled in the asset settings.

If you want to add your own implementation of a porting of your 2.x save data, you can make a class implementing the **ILegacySaveHandler** interface.

Pre Data Load Intercept

You can add logic before the game loads new data from the save by making a class that implements the **IPreSaveDeserialize** interface. This lets you edit the Json from the save before it is converted into Json object to load it into the game. This is mainly useful for porting older save set-ups to new structures etc. Using this set-up is risky as it can break the loading process easily. Use at your own risk.

Json Parsing

The asset is using the standard ***Newtonsoft Json*** parsing for Json with a few custom edits. These being:

- Only public or [SerializeField] private fields will be captured by the save manager for the game save.
- Unless turned off by the ***Use Json Converters*** *asset* setting, the asset will use its own json converters to make a readable json of the following types:
    - Vector2
    - Vector2Int
    - Vector3
    - Vector3Int
    - Vector4
    - Color
    - Color32
    - Quaternion
- You can add your own for any type using the save managers base class for json converters to make it easier to write. See ***SmJsonConverterBase*** for more info onthis.

Support

Help

I try to respond to queries as within 24 hours, but I am only 1 person. So bear with me if I don't mange this. I may be busy. Should you need any help with the asset, you can reach me by the following channels:

General contact form: https://carter.games/contact/

Direct email: hello@carter.games

Bug reporting

Found an issue? Report it via the following channels. Bugs will be corrected as soon as possible if deemed major. Minor issues may take a little time to resolve. I do this for free after all:

Bug reporting form: https://carter.games/report/

GitHub Issues: https://github.com/CarterGames/SaveManager/issues

Direct email: hello@carter.games

Scripting

Below is a small summary of the main API you'll be accessing from the asset. Note that I've only included the intended API for you to use and not every public property and/or method.

All classes for the asset at runtime are under the following:

Assembly:

- **CarterGames.SaveManager.Runtime**
- **CarterGames.Shared.SaveManager** *(Optional, may be needed for some API)*

Base Namespace:

- **CarterGames.Assets.SaveManager**

The asset is set-up in the mind that you'll be leaving the editor set-up be as most won't ever need to mess with it.

Main Scripting API

Scripting........18
- Save Manager........20
  - IsInitialized [Property]........20
  - IsSaving [Property]........20
  - IsLoading [Property]........20
  - IsBusy [Property]........20
  - InitializedEvt [Evt]........20
  - GameSaveStartedEvt [Evt]........20
  - GameSaveCompletedEvt [Evt]........21
  - GameLoadStartedEvt [Evt]........21
  - GameLoadedEvt [Evt]........21
  - GameLoadFailedEvt [Evt]........21
  - GameLoadFailedCompletelyEvt [Evt]........21
  - SaveGame [Method]........21
  - LoadGame [Method]........22
  - GetGlobalSaveObject [Method]........22
  - TryGetGlobalSaveObject [Method]........22
  - GetActiveSlotSaveObject [Method]........22
  - TryGetActiveSlotSaveObject [Method]........22
  - GetSlotSaveObject [Method]........22
  - TryGetSlotSaveObject [Method]........23
  - TryGetSaveValue [Method]........23
  - TryGetGlobalSaveValue [Method]........23
  - TryGetActiveSlotSaveValue [Method]........23

Save Value.....24
Value [Property].....24
ValueType [Property].....24
ValueObject [Property].....24
HasDefaultValue [Property].....24
DefaultValue [Property].....24
DefaultValueObject [Property].....24
ResetValue() [Method].....25
Save Object / Slot Save Object.....26
Lookup [Property].....26
HasValue() [Method].....26
GetValue() [Method].....26
SetValue() [Method].....26
ResetObjectSaveValues() [Method].....26
SaveCategory [Attribute].....26
Save Slot Manager.....27
SlotsEnabled [Property].....27
HasLoadedSlot [Property].....27
TotalSlotsInUse [Property].....27
ActiveSlotId [Property].....27
HasAnySlots [Property].....27
ActiveSlot [Property].....27
AllSlots [Property].....27
TotalSlotsRestricted [Property].....28
RestrictedSlotsTotal [Property].....28
SlotCreatedEvt [Evt].....28
SlotDeletedEvt [Evt].....28
SlotUnloadedEvt [Evt].....28
SlotLoadedEvt [Evt].....28
SlotLoadFailedEvt [Evt].....29
SlotLoadFailedEvt [Evt].....29
TryCreateSlotAtId() [Method].....29
TryCreateSlot() [Method].....29
LoadSlot() [Method].....29
UnloadCurrentSlot() [Method].....29
DeleteSlot() [Method].....29
Save Slot.....30
SlotId [Property].....30
LastSaveDate [Property].....30
Playtime [Property].....30

Save Manager

The save manager class is the main API class which you’ll use. It is a partial class for readability so the different sections of the class are in different files.

IsInitialized [Property]

```
public static bool IsInitialized { get; }
```

Gets if the save manager class has fully initialized. You should check this before accessing the other API as if the asset is not initialized you may get unexpected errors.

IsSaving [Property]

```
public static bool IsSaving { get; }
```

Gets if the save manager is currently saving data.

IsLoading [Property]

```
public static bool IsLoading { get; }
```

Gets if the save manager is currently loading data.

IsBusy [Property]

```
public static bool IsBusy { get; }
```

Gets if the save manager is currently running a save or load operation.

InitializedEvt [Evt]

```
public static readonly Evt InitializedEvt
```

Is raised when the save manager has initialized.
Add a listener to receive the evt when it is raised.

GameSaveStartedEvt [Evt]

```
public static readonly Evt GameSaveStartedEvt
```

Is raised when the save manager has started to save data.
Add a listener to receive the evt when it is raised.

GameSaveCompletedEvt [Evt]

```
public static readonly Evt GameSaveCompletedEvt
```

Is raised when the save manager has completed saving data.
Add a listener to receive the evt when it is raised.

GameLoadStartedEvt [Evt]

```
public static readonly Evt GameLoadStartedEvt
```

Is raised when the save manager has started to load data.
Add a listener to receive the evt when it is raised.

GameLoadedEvt [Evt]

```
public static readonly Evt GameLoadedEvt
```

Is raised when the save manager has started loaded data successfully.
Add a listener to receive the evt when it is raised.

GameLoadFailedEvt [Evt]

```
public static readonly Evt<LoadFailInfo> GameLoadFailedEvt
```

Is raised when the save manager has failed to load a data set for any reason.
Add a listener to receive the evt when it is raised.

GameLoadFailedCompletelyEvt [Evt]

```
public static readonly Evt GameLoadFailedCompletelyEvt
```

Is raised when the save manager has failed to load data from the current save or any existing backups. Add a listener to receive the evt when it is raised.

SaveGame [Method]

```
public static void SaveGame()
```

Saves the game in its current state when called.

LoadGame [Method]

```
public static void LoadGame()
```

Loads the game from the stored save data when called.

GetGlobalSaveObject [Method]

```
public static T GetGlobalSaveObject<T>()
```

Gets the global save object of the defined type. For a safer call, please use the **TryGetGlobalSaveObject** method instead.

TryGetGlobalSaveObject [Method]

```
public static bool TryGetGlobalSaveObject<T>(out T saveObject)
```

Tries to gets the global save object of the defined type. It returns the result of the call so you can catch any issues instead of dealing with null values.

GetActiveSlotSaveObject [Method]

```
public static T GetActiveSlotSaveObject<T>()
```

Gets a save object from the currently active slot of the defined save object type. For a safer call, please use the **TryGetActiveSlotSaveObject** method instead.

TryGetActiveSlotSaveObject [Method]

```
public static bool TryGetActiveSlotSaveObject<T>(out T slotSaveObject)
```

Tries to get a save object from the currently active slot of the defined save object type. It returns the result of the call so you can catch any issues instead of dealing with null values.

GetSlotSaveObject [Method]

```
public static T GetSlotSaveObject<T>(int slotId)
```

Gets a save object from the defined slot id of the defined save object type. For a safer call, please use the **TryGetSlotSaveObject** method instead.

TryGetSlotSaveObject [Method]

```
public static bool TryGetSlotSaveObject<T>(int slotId, out T saveObject)
```

Tries to get a save object from the defined slot id of the defined save object type. It returns the result of the call so you can catch any issues instead of dealing with null values.

TryGetSaveValue [Method]

```
public static bool TryGetSaveValue<T>(string saveKey, out SaveValue<T>
value, SaveCtx ctx = SaveCtx.Unassigned)
```

Tries to get a save value from anywhere in the save. Use SaveCtx to define the placement of the value in the save data set-up for a slightly faster call.

TryGetGlobalSaveValue [Method]

```
public static bool TryGetGlobalSaveValue<T>(string saveKey, out SaveValue<T>
value)
```

Tries to get a save value from just the global save set-up in the save.

TryGetActiveSlotSaveValue [Method]

```
public static bool TryGetActiveSlotSaveValue<T>(string saveKey, out
SaveValue<T> value)
```

Tries to get a save value from just the active save slot in the save.

Save Value

```
public SaveValue(string key)
public SaveValue(string key, T defaultValue)
```

Defines a value that is stored in the game save. All save values MUST have a save key assigned to them for the system to save them. Any without a key will be flagged in the editor and not be saved to the save data until corrected.

Value [Property]

```
public T Value { get; set; }
```

The value stored in the save value. Use to access or edit the actual value stored in the save.

ValueType [Property]

```
public Type ValueType { get; }
```

The type the save value is, mainly used for the asset itself. But it is public if you wish to use it.

ValueObject [Property]

```
protected object ValueObject { get; set; }
```

The object-typed value for the saved value. Please use T Value for accessing the save value instead of this.

HasDefaultValue [Property]

```
public bool HasDefaultValue { get; }
```

Gets if a default value has been set to this save value. Will return false if the default value is the type default.

DefaultValue [Property]

```
public T DefaultValue { get; set; }
```

Defined the default value of the save value. You can set this post constructor if you wish to. Default values are stored in the save along side the current value.

DefaultValueObject [Property]

```
protected object DefaultValueObject { get; set; }
```

Like the value object variant it is the object-typed value for the saved default value. Please use T DefaultValue for accessing the save default value instead of this.

ResetValue() [Method]

```
public void ResetValue(bool useDefault = true)
```

Reset the value to its user defined default if there is one set, or to its type default value if there is no defined default value.

Save Object / Slot Save Object

Defines a scriptable object that can contain save values that the save manager will detect and handle.

Lookup [Property]

```
public Dictionary<string, SaveValueBase> Lookup { get; }
```

A lookup that is generated when first accessed. It stored a key lookup of all the save values contained on the save object for access. Mainly used by the assets own API, but you can also use it if you wish.

HasValue() [Method]

```
public bool HasValue(string key)
```

Gets if the save object has a save value defined on it with the entered key.

GetValue() [Method]

```
public SaveValue<T> GetValue<T>(string key)
```

Gets a save value on the save object that matches the entered key.

SetValue() [Method]

```
public void SetValue(string key, object value)
```

Sets a save value on the save object that matches the entered key to the entered value.

ResetObjectSaveValues() [Method]

```
public virtual void ResetObjectSaveValues()
```

Another API intended for the assets own use. You may use this method to reset all save values on the save object if need be.

SaveCategory [Attribute]

```
[SaveCategory("ExampleSaveCategory")]
```

A save object specific attribute that allows you to define a save category for a save object to go under in the save editor GUI.

Save Slot Manager

The main manager class for the save slots set-up. Use to manage the save slots for users.

SlotsEnabled [Property]

```
public static bool SlotsEnabled { get; }
```

Gets if the save slots set-up is enabled or not.

HasLoadedSlot [Property]

```
public static bool HasLoadedSlot { get; }
```

Gets if a save slot has been loaded into the set-up.

TotalSlotsInUse [Property]

```
public static int TotalSlotsInUse { get; }
```

Gets the total number of save slots the system has registered for use.

ActiveSlotId [Property]

```
public static int ActiveSlotId { get; }
```

Gets the current active slot id.

HasAnySlots [Property]

```
public static bool HasAnySlots { get; }
```

Gets if any slots have been defined or not.

ActiveSlot [Property]

```
public static SaveSlot ActiveSlot { get; }
```

Gets the active save slot for use.

AllSlots [Property]

```
public static IReadOnlyDictionary<int, SaveSlot> AllSlots { get; }
```

Gets all the save slots in a lookup if needed.

TotalSlotsRestricted [Property]

```
public static bool TotalSlotsRestricted { get; }
```

Gets if the number of save slots is limited by the asset settings.

RestrictedSlotsTotal [Property]

```
public static int RestrictedSlotsTotal { get; }
```

Gets the number of save slots set-up is limited by in the asset settings.

SlotCreatedEvt [Evt]

```
public static readonly Evt<SaveSlot> SlotCreatedEvt
```

Is raised when the slot manager has created a new save slot. The slot is passed as a param.
Add a listener to receive the evt when it is raised.

SlotDeletedEvt [Evt]

```
public static readonly Evt<int> SlotDeletedEvt
```

Is raised when the slot manager is deleted. The id of that slot is passed as a param.
Add a listener to receive the evt when it is raised.

SlotUnloadedEvt [Evt]

```
public static readonly Evt<int> SlotUnloadedEvt
```

Is raised when the slot manager has unloaded a slot. The id of that slot is passed as a param.
Add a listener to receive the evt when it is raised.

SlotLoadedEvt [Evt]

```
public static readonly Evt<int> SlotLoadedEvt
```

Is raised when the slot manager has loaded a slot. The id of that slot is passed as a param.
Add a listener to receive the evt when it is raised.

SlotLoadFailedEvt [Evt]

```
public static readonly Evt<int> SlotLoadFailedEvt
```

Is raised when the slot manager has failed to load a slot. The id of that slot is passed as a param. Add a listener to receive the evt when it is raised.

SlotLoadFailedEvt [Evt]

```
public static readonly Evt<int> SlotLoadFailedEvt
```

Is raised when the slot manager has failed to load a slot. The id of that slot is passed as a param. Add a listener to receive the evt when it is raised.

TryCreateSlotAtId() [Method]

```
public static bool TryCreateSlotAtId(int slotId, out SaveSlot newSlot)
```

Tries to create a new save slot at the entered id.

TryCreateSlot() [Method]

```
public static bool TryCreateSlot(out SaveSlot newSlot)
```

Tries to create a new save slot.

LoadSlot() [Method]

```
public static void LoadSlot(int slotId)
```

Loads the slot of the entered id.

UnloadCurrentSlot() [Method]

```
public static void UnloadCurrentSlot()
```

Unloads the currently loaded slot when called.

DeleteSlot() [Method]

```
public static void DeleteSlot(int slotId)
```

Deletes the slot of the entered id from the save system.

Save Slot

The class that holds the data for a save slot in the slots save set-up.

SlotId [Property]

```
public int SlotId { get; }
```

Gets the id assigned to this slot.

LastSaveDate [Property]

```
public DateTime LastSaveDate { get; }
```

Gets the last time the slot was saved at.

Playtime [Property]

```
public TimeSpan Playtime { get; }
```

Gets the total playtime of the slot. Is calculated from load to save.

Listener Classes

You can listen to some of the assets set-up by implementing specific interfaces without needing any extra logic, these are:

Listeners

Listener Classes..31
    IGameSaveListener..31
    IGameLoadListener..31

IGameSaveListener

Implement to list to when the game saving is called to start and completed.

IGameLoadListener

Implement to list to when the game loading is called to start and completed or fails.

Extending the asset

You can extend the save manager by making your own implementations for core features of the asset to use. These can then be selected in the asset settings so the asset uses them over the default provided options. It is recommended you test any custom implementations well before releasing them for public use in-case of issues. When developing any custom locations or encryption set-ups it is advised to save a backup of your game data to a alternative location so you can restore it should issues arise.

Extension API

Extending the asset........32
- IDataLocation........33
  - HasData [Method]........33
  - SaveToLocation [Method]........33
  - LoadFromLocation [Method]........33
- ISaveDataLocation........34
  - DataLocation [Property]........34
  - HasSaveData [Property]........34
  - SaveDataToLocation [Method]........34
  - LoadDataFromLocation [Method]........34
- ISaveBackupLocation........35
  - Location [Property]........35
  - BackupData [Method]........35
  - GetBackups [Method]........35
- ISaveEncryptionHandler........35
  - Encrypt [Method]........35
  - Decrypt [Method]........35
- SmJsonConverterBase........36
  - ReadFromJson [Method]........36
  - WriteToJson [Method]........36
- ISaveMetaData........37
  - Key [Property]........37
  - CanWriteMetaData [Property]........37
  - GetMetaData [Method]........37
- ILegacySaveHandler........37
  - ProcessLegacySaveData [Method]........37

IDataLocation

Implement this interface to define a location at which any data can be stored at. Implement along-side **ISaveDataLocation** to define a custom save data location.

HasData [Method]

```
public bool HasData(string path)
```

This method should return if the data location has any data currently stored or not.

SaveToLocation [Method]

```
public void SaveToLocation(string path, string data)
```

This method should save the entered data to the location.

LoadFromLocation [Method]

```
public string LoadFromLocation(string path)
```

This method should load the data from the location.

ISaveDataLocation

Implement this interface to define a custom game save data location to store your game save data at. This requires a **IDataLocation** implementation to be set-up for the desired location as well.

DataLocation [Property]

```
public IDataLocation DataLocation { get; }
```

Should get the **IDataLocation** implementation to use for this save method.

HasSaveData [Property]

```
public bool HasSaveData { get; }
```

Should get if there is save data stored in the location currently.

SaveDataToLocation [Method]

```
public void SaveDataToLocation(string json)
```

This method should save the entered json to the location when called.

LoadDataFromLocation [Method]

```
public string LoadDataFromLocation()
```

This method should load the json from the location when called and return the received data.

ISaveBackupLocation

Implement this interface to define a location at which any save backups may be stored. This requires a **IDataLocation** implementation to be set-up for the desired location as well.

Location [Property]

```
public IDataLocation Location { get; }
```

Should get the **IDataLocation** implementation to for the backups.

BackupData [Method]

```
public void BackupData(JToken data)
```

This method should save the entered data to the location.

GetBackups [Method]

```
public IEnumerable<JObject> GetBackups()
```

This method should load all the backups from the location for use.

ISaveEncryptionHandler

Implement this interface to make your own data encryption handler to use with the asset. It is best to test you encryption set-up works before assigning it to the manager to avoid issues. I’d also recommend making a backup of any save data beforehand just in-case.

Encrypt [Method]

```
public string Encrypt(string contentData)
```

This method should encrypt the data entered into it and return the result of the encryption back to the system for use.

Decrypt [Method]

```
public string Encrypt(string encryptedData)
```

This method should decrypt the data entered into it and return the result of the decryption back to the system for use.

SmJsonConverterBase

Implement this abstract class to add your own json converters so the save manager can use them when saving your data. By default all public fields or private fields with the **[SerializeField]** attribute will be saved if possible. You should implement a custom converter if a custom type is not saving correctly or you need to save a Unity type that is not saving correctly in the current set-up.

ReadFromJson [Method]

```
protected abstract void ReadFromJson(ref T value, string name, JsonReader
reader, JsonSerializer serializer)
```

This method should read the json value for the required field if applicable. This should be done through a switch statement or similar.

An example from the Vector2 converter set-up:

```
protected override void ReadFromJson(ref Vector2 value, string name,
JsonReader reader, JsonSerializer serializer)
{
    switch (name)
    {
        case nameof(value.x):
            value.x = (float)reader.ReadAsDouble().GetValueOrDefault(0d);
            break;
        case nameof(value.y):
            value.y = (float)reader.ReadAsDouble().GetValueOrDefault(0d);
            break;
    }
}
```

WriteToJson [Method]

```
protected abstract IEnumerable<KeyValuePair<string, object>>
WriteToJson(Vector2 value, JsonSerializer serializer)
```

This method should define what is written into the json and assign a key/value pair for each value that should be in the json.

An example from the Vector2 converter set-up:

```
protected override IEnumerable<KeyValuePair<string, object>>
WriteToJson(Vector2 value, JsonSerializer serializer)
{
 return new KeyValuePair<string, object>[]
 {
 new KeyValuePair<string, object>("x", value.x),
 new KeyValuePair<string, object>("y", value.y),
 };
}
```

ISaveMetaData

Implement to define extra data that is added to the **$metadata** tag in the save data structure. This is read-only and is mainly intended to aid in context info for debugging etc.

Key [Property]

```
public string Key { get; }
```

Defines the string key the meta-data is saved under.

CanWriteMetaData [Property]

```
public bool CanWriteMetaData { get; }
```

This property should define if the meta-data can be written to the save. Use to toggle the data off if you don’t need it etc.

GetMetaData [Method]

```
public JObject GetMetaData()
```

This should return all the values in a single object to write to the save when called. An example from the game info meta-data set-up:

```
public JObject GetMetaData()
{
    return new JObject
    {
        ["$version"] = $"{Application.version}",
        ["$save_date"] = DateTime.UtcNow.ToString("yyyy-MM-ddTHH:mm:ss"),
    };
}
```

ILegacySaveHandler

Implement to define a custom solution to port older 2.x save manager saves to the 3.x set-up. By default the asset will try to port all 2.x save data to 3.x global data where the keys match. You can implement this interface to change that to something custom should you wish.

ProcessLegacySaveData [Method]

```
public JToken ProcessLegacySaveData(JToken loadedJson,
IReadOnlyDictionary<string, JArray> legacyData)
```

This method should return the processed data, adding the legacy data into the loaded json to produce the converted result.