ESave Documentation

v1.1.0

June 10, 2024

Stylish Esper

Document Version: 1.1.0

Table of Contents

Online Documentation

ESave's documentation may be updated occasionally. To find the latest version of the documentation, use this link.

Installation

Asset Store

You can find the latest version of ESave in the asset store with this link.

Installation Steps:

1. Get ESave from the asset store.
2. Open your Unity project, go into `Window > Package Manager`, switch to `My Assets` from the packages dropdown at the top-left, and then type `ESave` in the search bar.
3. Download and install the package.
4. Once it's complete, a package installer window will popup. Click `Install Newtonsoft JSON`.

Github

You can find the latest version of ESave on GitHub with this link.

Installation Steps:

1. Download one of the `.unitypackage` files from the releases page.
2. Open your Unity project and double click the downloaded Unity package.
3. Click `import`.
4. Once it's complete, a package installer window will popup. Click `Install Newtonsoft JSON`.

Example Setup

You can find the example scene that works with any render pipeline in `Assets/StylishEsper/ESave/Examples/Any RP Example`.
If you'd like to try the URP example, you can import the assets from the `URP_Example.unitypackage` in `Assets/StylishEsper/ESave/Examples`.

Components

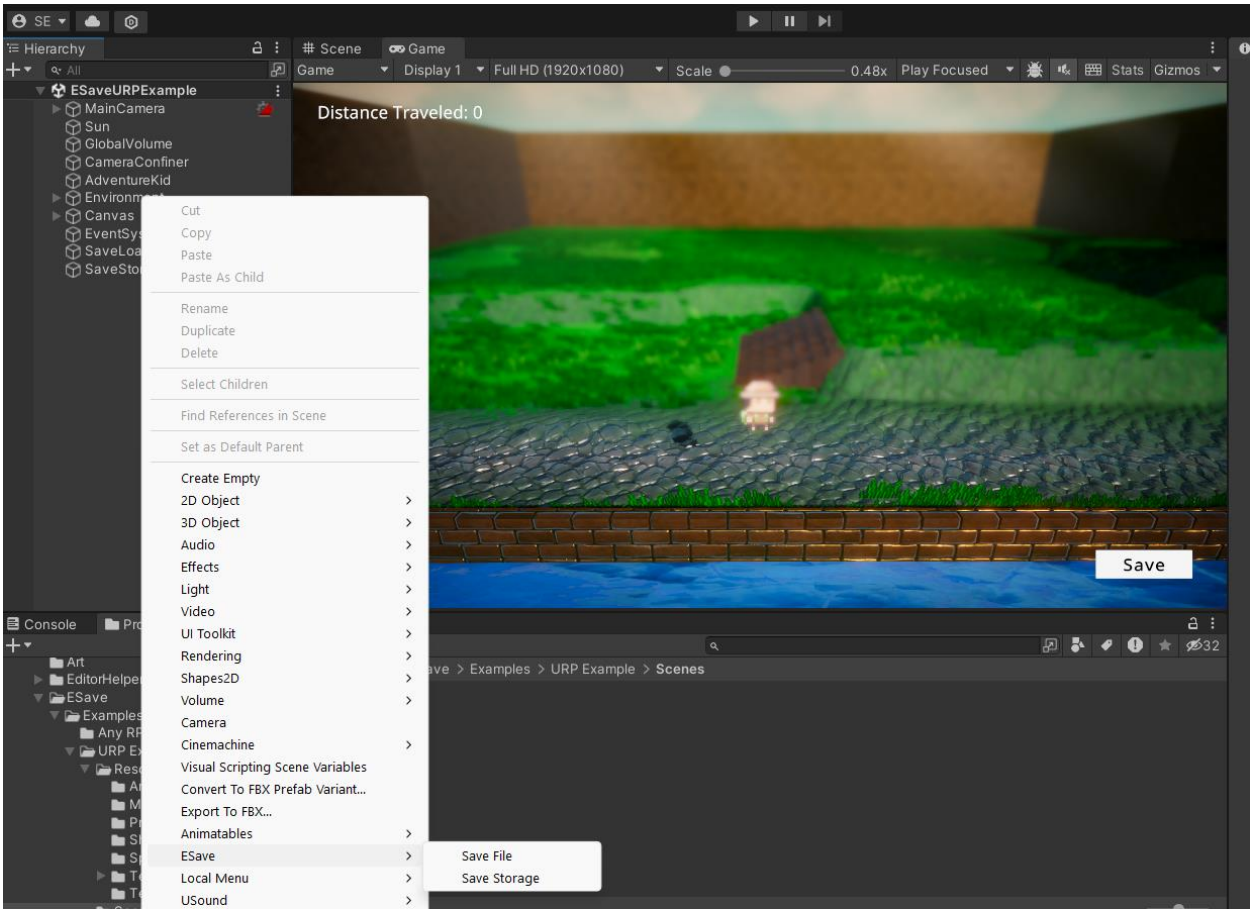
File Structure

Save data is typically stored in separate files, with one for each save state and one for the player's settings. ESave is structured around this idea, which is why it's an important concept to remember.

Separating your saved data is recommended. Having all of the player's data in one file is possible and very common, but it's not a requirement. For example, you could have one save file that

stores the player's inventory data, one for storing the player's progress, and another for the environmental impacts they made.

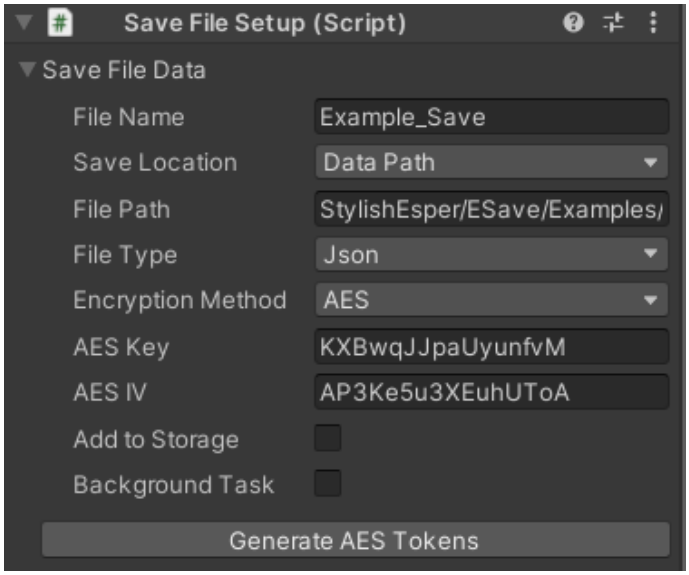
Create Menu Items



You can easily create ESave's save file setup and save storage components by right-clicking on the hierarchy and navigating to ESave.

Save File Setup

Save file setup is the main component you will be working with. This component will work exactly as it's named; it will create a save file in the user's system that you can save and load data from.



File Name
The name of the file that will be saved in the player's system.

Save Location
There are 2 common locations that you can store player files.

1. **Persistent Data Path:** a directory path where data expected to be kept between runs can be stored.
2. **Data Path:** the path to the game data folder on the target device.

It's generally recommended to use persistent data path. Data path may not work on all platforms.

File Path
This is the extra path after the save location path. For example, in the above image, the file `URP_Example_Save` will be stored in the path `StylishEsper/ESave/Example` in the game's data folder.

File Type
The file type determines the saved data format and the file extension. Currently, only the JSON format is supported.

Encryption Method
Only the AES encryption algorithm is supported at this time. The default option is no encryption.

AES Key and IV
The key and IV are used for the AES algorithm. You can click the 'Generate AES Tokens' button to generate a random token for both.

Add to Storage
If add to storage is checked, the save file will be added to the save storage.

Background Task

If background task is checked, both saving and loading operations will be executed in a background thread instead of the main thread. This is great to use if you have a large save file.

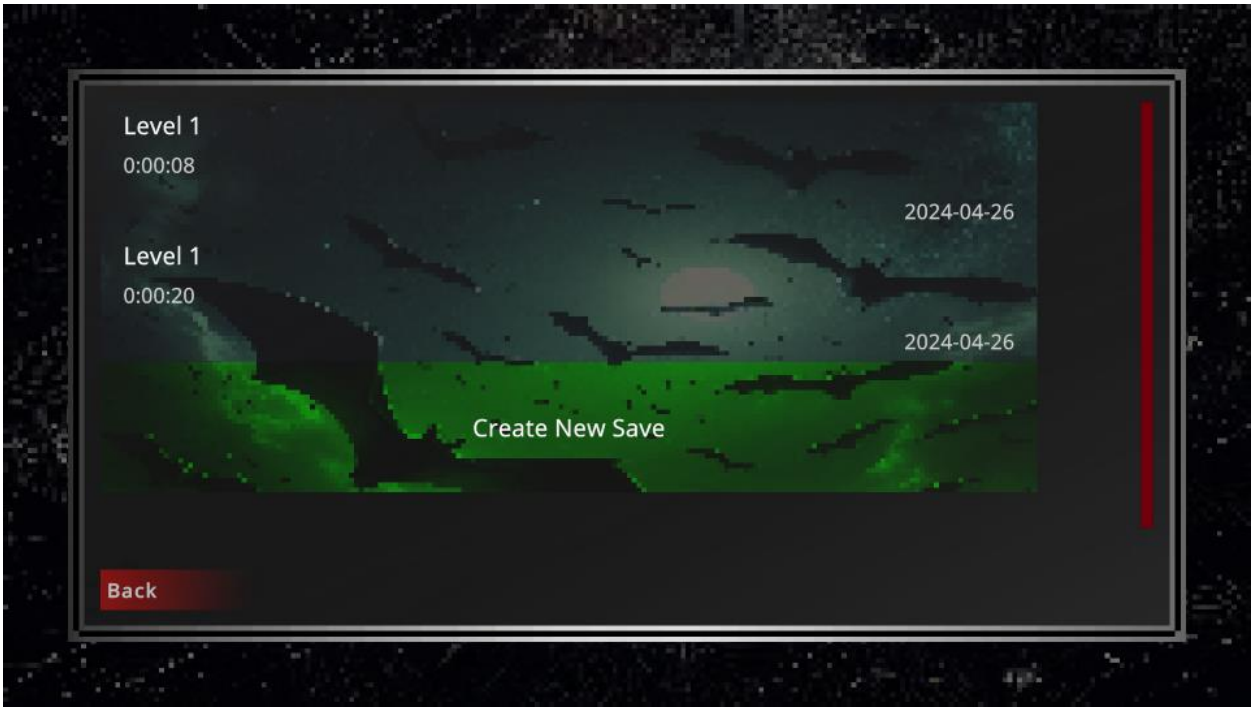
If you need something to happen after saving and loading is done, and if 'background task' is checked, make sure to wait for the operation to complete first.

Save Storage

Save storage is a singleton component that has no inspector properties. Save storage automatically stores the paths of every save file that has 'Add to Storage' checked. It requires a save file setup component to store the paths.

Ensure the 'Add to Storage' box is unchecked for Save Storage's save file setup component.

With this component, you can find any save (that has the 'Add to Storage' value checked) by its file name through code.



Save storage isn't a mandatory component, but having it can significantly simplify tasks like accessing all of a player's saves, particularly useful for games featuring a save list screen.

Saving

Remember to use the `Esper.ESave` namespace when scripting.

1. Getting the Save File

The first step to saving data is getting the save file. This can be done by getting the save file setup component first, which will give you access to the save file with `SaveFileSetup.GetSaveFile`.

```
saveFileSetup = GetComponent<SaveFileSetup>();
saveFile = saveFileSetup.GetSaveFile();
```

It's recommended to use this code in Awake or Start.

2. Saving Data

Each piece of data in a save file has an ID (string). This makes it possible to retrieve specific data by ID. Saving data can be done with a single line with `SaveFile.AddOrUpdateData`.

```
saveFile.AddOrUpdateData("DataID", data);
```

The first parameter is the ID of the data, and the second parameter is the data itself. Any serializable data type can be saved. However, most Unity types are not savable. ESave can save common Unity data types, including (but not limited to) Vector2, Vector3, Quaternion, Color, and Transform.

Loading

1. Getting the Save File

Just like saving, you need a reference to a save file to load data.

```
saveFileSetup = GetComponent<SaveFileSetup>();
saveFile = saveFileSetup.GetSaveFile();
```

2. Loading Data

Each ESave method that loads data only accepts one parameter, which is the ID of the data.

GetData

The main method to load data is `SaveFile.GetData`. This method accepts any type parameter.

```
// Where T is the data type
T data = saveFile.GetData<T>("DataID");
```

You can also retrieve a list of data of the same type:

```
// Where T is the data type
List<T> dataList = saveFile.GetData<T>("DataID", "DataID2",
"DataID3");
```

Special Methods

ESave features special methods to retrieve data for some Unity types.

Vector2

```
Vector2 v2 = saveFile.GetVector2("DataID");
```

Vector3

```
Vector3 v3 = saveFile.GetVector3("DataID");
```

Quaternion

```
Quaternion q = saveFile.GetQuaternion("DataID");
```

Color

```
Color c = saveFile.GetColor("DataID");
```

Transform

```
// Returns a SavableTransform
SavableTransform st = saveFile.GetVector2("DataID");

// Use Transform.CopyTransformValues to set values from a
// SavableTransform to a Transform
transform.CopyTransformValues(st);
```

Saving & Loading Example

This script saves the player's transform values when the game is exited and loads the transform values on start.

When saving a transform with ESave, only the position, rotation, and scale properties are saved.

```
using UnityEngine;

public class SaveLoadExample : MonoBehaviour
{
    // Const data ID
    private const string playerTransformDataKey = "PlayerTransform";

    [SerializeField]
    private Transform playerTransform;

    private SaveFileSetup saveFileSetup;
    private SaveFile saveFile;

    private void Start()
    {
        // Get save file component attached to this object
        saveFileSetup = GetComponent<SaveFileSetup>();
        saveFile = saveFileSetup.GetSaveFile();
```

```
        // Load game
        LoadGame();
    }

    /// <summary>
    /// Loads the game.
    /// </summary>
    public void LoadGame()
    {
        // Check if the data exists in the file
        if (saveFile.HasData(playerPositionDataKey))
        {
            // Get Vector3 from a special method because Vector3 is
not savable data
            var savableTransform =
saveFile.GetTransform(playerPositionDataKey);
            playerTransform.CopyTransformValues(savableTransform);
        }

        Debug.Log("Loaded game.");
    }

    /// <summary>
    /// Saves the game.
    /// </summary>
    public void SaveGame()
    {
        saveFile.AddOrUpdateData(playerPositionDataKey,
playerTransform);
        saveFile.Save();

        Debug.Log("Saved game.");
    }

    private void OnApplicationQuit()
    {
        SaveGame();
    }
}
```

Runtime Save Creation

Important Information

Before we begin, it's important to note that `new SaveFile()` is not always creating a new save file in the users system. It will only create a new save file if there isn't a save file in the specified path with the specified file name.

For example, if we use the same SaveFileSetupData for 2 SaveFile instances, only 1 file will be created in the user's system. The 2 SaveFile instances will simply be editing the same file.

```
SaveFileSetupData saveFileSetupData = new SaveFileSetupData()
{
   fileName = "Save File",
   saveLocation = SaveLocation.PersistentDataPath,
   filePath = "Example/Path",
   encryptionMethod = EncryptionMethod.None,
   addToStorage = true
};
// Both of these will be editing the same save file
SaveFile saveFile1 = new SaveFile(saveFileSetupData);
SaveFile saveFile2 = new SaveFile(saveFileSetupData);
```

This is here for informational purposes only, and **it is not** recommended to do.

Creating a Save File

We know that you can create a save file using the Save File Setup component. Alternatively, you can create a save file through code.

Unencrypted Save

A save file needs save file setup data to be created. The code below creates save file setup data.

```
SaveFileSetupData saveFileSetupData = new SaveFileSetupData()
{
   fileName = "Save File",
   saveLocation = SaveLocation.PersistentDataPath,
   filePath = "Example/Path", // The path AFTER the persistent data
path (can be left empty)
   fileType = FileType.Json,
   encryptionMethod = EncryptionMethod.None,
   addToStorage = true
};
SaveFile saveFile = new SaveFile(saveFileSetupData);
```

However, this code will not create an encrypted save file.

Encrypted Save

The only supported encryption method at the time was AES. To create an AES-encrypted save file, we would need an AES key and IV. These are used for the AES algorithm. Both the AES key and IV are just a string of random alphanumeric characters. It is recommended that they be at least 16 characters in length.

1. Store the AES Key and IV

The key and IV of a save file **must not** change, as the same key and IV are required during both encryption and decryption. So, we should create constants that we will use every time.

```
private const string aesKey = "randomkey1234567";
private const string aesIV = "randomiv12345678";
```

2. Create Setup Data

This will be similar to the previous one, except we will change the encryption method to AES and provide the key and IV.

```
SaveFileSetupData saveFileSetupData = new SaveFileSetupData()
{
   fileName = "Save File",
   saveLocation = SaveLocation.PersistentDataPath,
   filePath = "Example/Path",
   fileType = FileType.Json,
   encryptionMethod = EncryptionMethod.AES,
   aesKey = aesKey,
   aesIV = aesIV,
   addToStorage = true
};
```

3. Create the Save File

After using the code below, you have successfully created an encrypted save file.

```
SaveFile saveFile = new SaveFile(saveFileSetupData);
```

Background Save File

To create a save file that will save and load data in the background, set the `backgroundTask` value to true.

```
SaveFileSetupData saveFileSetupData = new SaveFileSetupData()
{
   fileName = "Save File",
   saveLocation = SaveLocation.PersistentDataPath,
   filePath = "Example/Path",
   fileType = FileType.Json,
   encryptionMethod = EncryptionMethod.None,
   addToStorage = true,
   backgroundTask = true
};
SaveFile saveFile = new SaveFile(saveFileSetupData);
```

You can set this value at any time during runtime, and the save file will use a background or main thread accordingly.

```
saveFile.backgroundTask = false;
```

Infinite Saves

As of v1.0.1, there is an infinite saves example scene in
`Assets/StylishEsper/ESave/Examples/Any RP Examples.`

For this tutorial, the UI setup in ESaveInfiniteSavesExample scene will be used as a reference.

Prerequisites

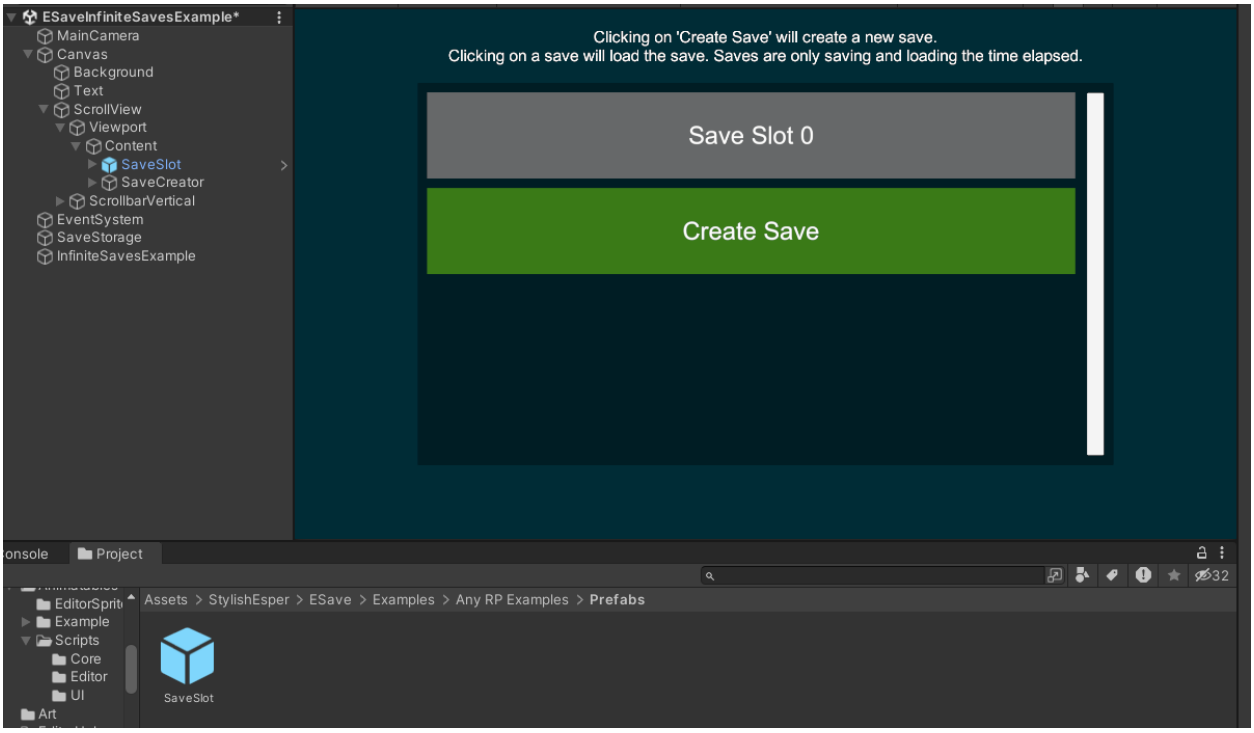
You will need some sort of UI setup that can support infinite saves. Generally, a scroll view is required for this case. The scroll view will be populated with save slot buttons and one save slot creator button.

You will need a way to switch the mode of the save window. For this example, we will be using a toggle.

Create UI Objects

We will start by creating a UI button that will both load and overwrite data. This will be a prefab. You can make a prefab by dragging the object into the project window.

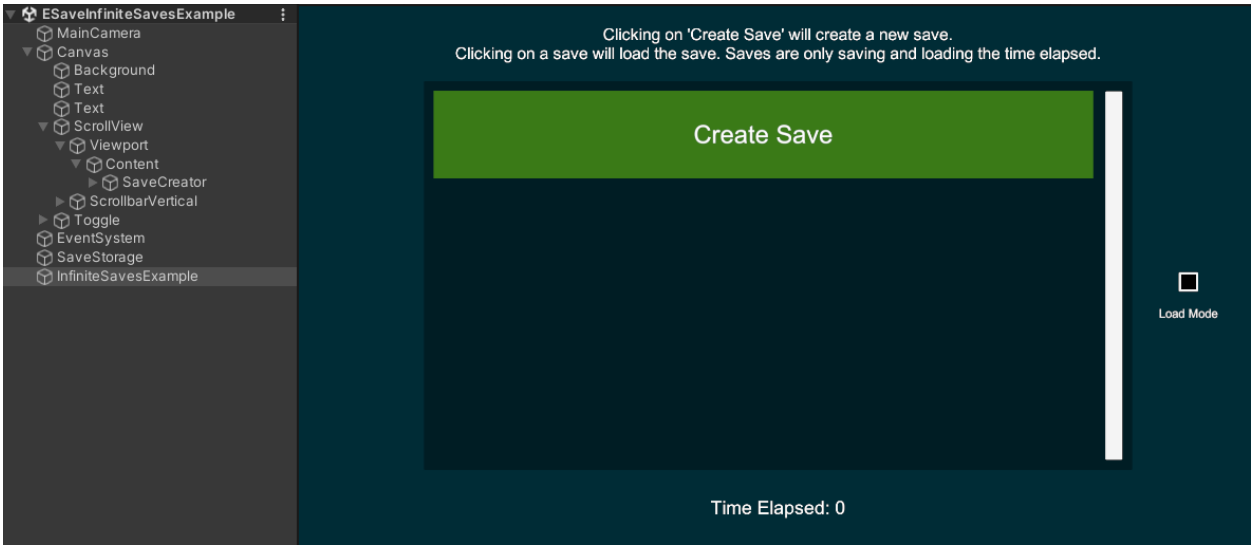
After that, create a button that will create save slots. This does not need to be a prefab. Both buttons will be children of the scroll view's content.



Delete the save slot button from the hierarchy, as it will be instantiated at runtime.

For this example, we will be using a toggle that will change the mode of the save window, which will just sit at the right of the scroll view.

The data we will be saving will only be the time elapsed. So, there is a text object below the scroll view that will display the current time elapsed.



Infinite Save Scripting

Step 1: Create a Script

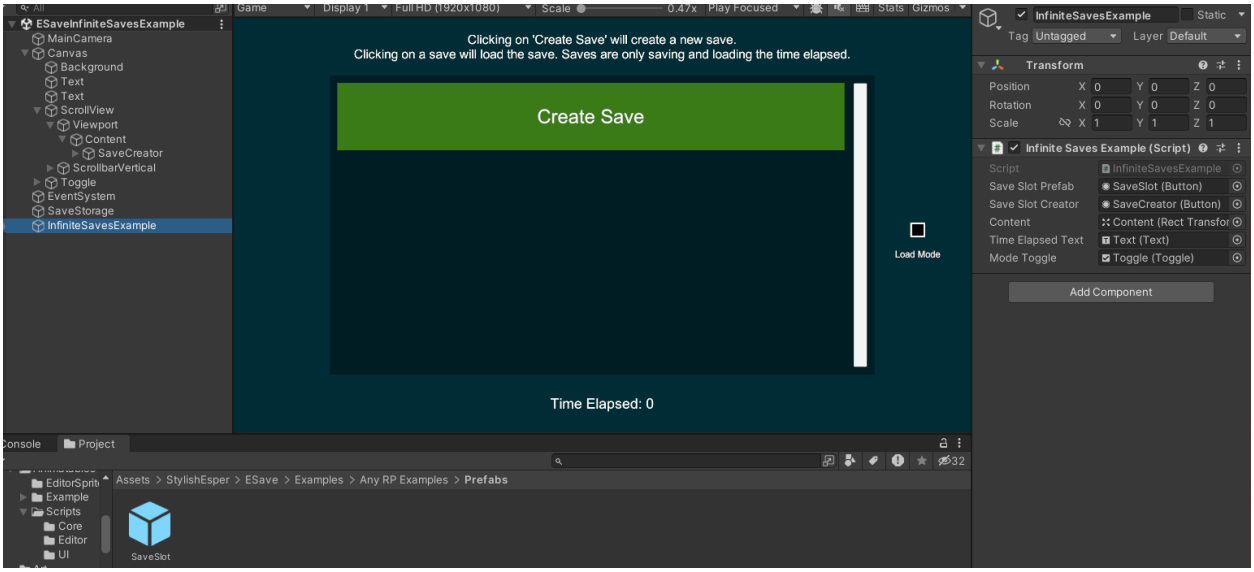
Create a script or multiple scripts that will handle all save menu functionality.

For this example, we will create a single script called `InfiniteSavesExample.cs`.

Start the script by adding the member variables that can be set from the inspector.

```
public class InfiniteSavesExample : MonoBehaviour
{
    [SerializeField]
    private Button saveSlotPrefab;

    [SerializeField]
    private Button saveSlotCreator;

    [SerializeField]
    private Transform content;

    [SerializeField]
    private Text timeElapsedText;

    [SerializeField]
    private Toggle modeToggle;
}
```

Remember to add a GameObject to the scene and add this script to it. At this point, we should have everything to populate the inspector properties.



Properties

Save Slot Prefab: the save slot button prefab.

Save Slot Creator: the button that will create a new save.

Content: the scroll view's content object.

Time Elapsed Text: the text object that will display the time elapsed.

Mode Toggle: the toggle that will toggle the mode between load and save mode.

Step 2: Other Script Members

We will create other members that won't be visible in the inspector.

```
public class InfiniteSavesExample : MonoBehaviour
{
        private const string timeElapsedKey = "TimeElapsed";

        [SerializeField]
        private Button saveSlotPrefab;

        [SerializeField]
        private Button saveSlotCreator;

        [SerializeField]
        private Transform content;

        [SerializeField]
        private Text timeElapsedText;

        [SerializeField]
        private Toggle modeToggle;

        private List<Button> slots = new();
```

```
        private float timeElapsed;

        private bool loadMode { get => modeToggle.isOn; }
}
```

timeElapsedKey: the key (or ID) of the time elapsed for saving purposes.

slots: a list of buttons that will store all of the instantiated save slots.

timeElapsed: the time elapsed as a float.

loadMode: the isOn value of the toggle.

If load mode is true, we will make the save slots load a save. If it's false (save mode), we will make the save slots overwrite a save.

Step 3: Instantiate Existing Saves

When we enter play mode, we need to first load any existing saves that the player may have made in a previous session.

We can do this using the Start method.

```
private void Start()
{
    // Instantiate slots for existing saves
    foreach (var save in SaveStorage.instance.saves.Values)
    {
        CreateNewSaveSlot(save);
    }
}
```

Step 4: Increment Time

In the Update method, we will increment the time elapsed. This will be the only data that will be saved and loaded for this example.

```
private void Update()
{
    // Increment time per frame
    timeElapsed += Time.deltaTime;
    timeElapsedText.text = $"Time Elapsed: {timeElapsed}";
}
```

Step 5: Saving and Loading Data

We will create 3 methods for saving and loading. The first one will load the data from a save file.

```
/// <summary>
```

```
/// Loads a save.
/// </summary>
/// <param name="saveFile">The save file.</param>
public void LoadSave(SaveFile saveFile)
{
    timeElapsed = saveFile.GetData<float>(timeElapsedKey);
}
```

The second one will save (or overwrite) the data in the save file.

```
/// <summary>
/// Overwrites a save.
/// </summary>
/// <param name="saveFile">The save file.</param>
public void OverwriteSave(SaveFile saveFile)
{
    // Save the time elapsed
    saveFile.AddOrUpdateData(timeElapsedKey, timeElapsed);
    saveFile.Save();
}
```

The third one will be used by the save slots and will load or overwrite the data depending on the mode.

```
/// <summary>
/// Loads or overwrites the save based on the active mode.
/// </summary>
/// <param name="saveFile">The save file.</param>
public void LoadOrOverwriteSave(SaveFile saveFile)
{
    if (loadMode)
    {
        LoadSave(saveFile);
    }
    else
    {
        OverwriteSave(saveFile);
    }
}
```

Step 6: Create New Save

We have methods that require a save file, but no save file is being created yet. A save file should be created when the 'Create New Save' button is pressed, along with a save slot.

So, let's create some methods for this. The first method will create a new save file.

```
/// <summary>
/// Creates a new save.
/// </summary>
```

```
public void CreateNewSave()
{
    // Create the save file data
    SaveFileSetupData saveFileSetupData = new()
    {
        fileName =
$"InfiniteExampleSave{SaveStorage.instance.saveCount}",
        saveLocation = SaveLocation.DataPath,
        filePath = "StylishEsper/ESave/Examples/Any RP Examples",
        fileType = FileType.Json,
        encryptionMethod = EncryptionMethod.None,
        addToStorage = true
    };

    SaveFile saveFile = new SaveFile(saveFileSetupData);

    // Save the time elapsed
    // Techincally, nothing is being overwitten at this stage since it
is an empty save file
    OverwriteSave(saveFile);

    // Create ths save slot for this data
    CreateNewSaveSlot(saveFile);
}
```

At the end, CreateNewSaveSlot is called. This has not been created yet. The CreateNewSaveSlot method will instantiate a new save slot, edit the save slot's text, and give the save slot an on-click event that will call LoadOrOverwriteSave.

```
/// <summary>
/// Creates a save slot for a save file.
/// </summary>
/// <param name="saveFile">The save file.</param>
public void CreateNewSaveSlot(SaveFile saveFile)
{
    // Instantiate the save slot
    var slot = Instantiate(saveSlotPrefab, content);
    var slotText = slot.transform.GetChild(0).GetComponent<Text>();
    slotText.text = $"Save Slot {slots.Count}";

    // Move save creator to the bottom
    saveSlotCreator.transform.SetAsLastSibling();

    // Add on-click event for loading
    slot.onClick.AddListener(() => LoadOrOverwriteSave(saveFile));

    slots.Add(slot);
}
```

We still need the save slot creator button to have an on-click event that creates a new save. This can be done by updating our Start method.

```
private void Start()
{
    // Instantiate slots for existing saves
    foreach (var save in SaveStorage.instance.saves.Values)
    {
        CreateNewSaveSlot(save);
    }

    // Save slot creator on-click event
    saveSlotCreator.onClick.AddListener(CreateNewSave);
}
```

Done! You can now test it in play mode.

Understanding Background Save & Load

You don't need to do anything extra to save and load data in the background. It will happen as long as `backgroundTask` is set to true. However, you may have to change the way you code a little when working with save files.

Working With Operations

Normally, if you want something to happen after saving or loading is complete, you would just respect the order of operations and execute your code after the save or load code.

If you've enabled 'Background Task' from the Save File Setup component or set `backgroundTask` to true through code (which is essentially the same thing), then this rule does not apply.

Instead, you must wait for the saving or loading operation to complete. This can be done easily with the `SaveFileOperation` class.

Getting the Operation

`SaveFile.Load` and `SaveFile.Save` return a SaveFileOperation object. This can be used to determine when saving or loading is completed. Whether it's saving or loading, there is no difference in how you work with the operation object.

It's important for saving to complete before loading and loading to complete before saving. Which is why there can only be a single operation running at a time for a save file.

Get Operation From Load

```
SaveFileOperation operation = saveFile.Load();
```

Get Operation From Save

```
SaveFileOperation operation = saveFile.Save();
```

Get Operation After Calling Save or Load

Save files store the most recent save file operation in memory. If you have recently called `SaveFile.Load` or `SaveFile.Save`, you can use `SaveFile.operation` to get the operation. This will be null if load or save was not called.

```
SaveFileOperation operation = saveFile.operation;
```

Waiting for Completion

There are 2 ways that you can check if the save file operation is complete.

1. Checking the State

`SaveFileOperation` has an enum called `OperationState`. This will be updated as the operation state changes.

There are 5 states:

1. **None**: the operation has not started.
2. **Ongoing**: the operation is currently executing (it's either saving or loading),
3. **Completed**: the operation was successfully completed.
4. **Canceled**: the operation was canceled. This can only be done manually with `SaveFileOperation.Cancel`.
5. **Failed**: an error was encountered during the operation and has been aborted. The error should be displayed in the console.

You can use this to check which state the operation is currently in.

```
if (operation.state == SaveFileOperation.OperationState.Completed)
{
    // Do something...
}
```

If you'd like to do something right after saving or loading is complete, this method **is not recommended** because you will have to use this check every frame.

2. On Operation Ended Event

`SaveFileOperation.onOperationEnded` will be invoked when the operation has ended (when the operation state is either completed, canceled, or failed). Here's an example of how you can use this to execute some code after saving has been completed:

Only use the `onOperationEnded` event for save files that will save & load in the background.

```
private void Start()
{
    // Some code...

    // Get the load operation from a save file
```

```
    var operation = saveFile.Load();

    // Add on-ended event
    operation.onOperationEnded.AddListener(LoadGame);
}

private void LoadGame()
{
    // Your loading code here...
}
```

If you'd like to ensure the operation was completed successfully, you can use an event like this instead:

```
// Add on-ended event
operation.onOperationEnded.AddListener(() =>
{
    if (operation.state == SaveFileOperation.OperationState.Completed)
    {
        LoadGame();
    }
    else
    {
        // Do something else...
    }
});
```

Getting Help

In need of assistance?

A Little Heads Up...

I'm a solo developer trying to simplify game development for others (and myself) by creating useful Unity tools. I'm not a part of a large or even small team; it's just me. If I receive your message, I will respond as soon as I can (just give me a day or two).

Help Me Help You

Errors

If you're facing an error, please list them in your message and explain the steps you took that led to that error.

Bug Reports

If you're reporting a bug, please include answers to these questions in your report:

1. What were you trying to do?
2. What did you expect would happen?
3. What actually happened?

Feature Request

I'm always looking to improve my products. If you'd like a specific feature added, don't hesitate to let me know. In your message, please explain how the feature will help your use case.

Options

You can contact me using any of these methods.

1. Discord Server
2. Website Contact Form
3. Emailing developer@stylishesper.com
4. Messaging me on X (Twitter)