Save Load System Documentation

The Save System is a comprehensive and versatile tool designed specifically for Unity developers who need a reliable solution for saving game data. At its core, the system uses JSON for data serialization, which makes the saved data inherently platform-agnostic, ensuring that it can be easily transferred and used across various environments without compatibility issues.

One of the standout features of this Save System is its ability to handle complex object references. In many games, objects are interconnected, with dependencies and relationships that need to be preserved when saving and loading data. This system excels in managing these relationships, even when they involve nested or circular references, which are often challenging to handle. By accurately restoring these references during the loading process, the system ensures that the game's state is fully reconstructed, including the dynamic reloading of any prefabs that were part of the saved state.

Additionally, the Save System is designed to support as many data types as possible, making it highly adaptable to different game structures and requirements. This flexibility allows developers to save a wide variety of data, from simple variables to complex object graphs, ensuring that all necessary information is preserved.

Moreover, the Save System offers various approaches to saving data, allowing developers to choose the method that best suits their specific needs and the complexity of their game. Whether it's through declarative Attribute Saving, using the ISavable interface, or employing Type Converters, the system provides flexible options to ensure that data is efficiently and accurately preserved across different game states. This means that while developers can quickly implement basic saving and loading functionalities with minimal effort, they also have the option to customize the process.

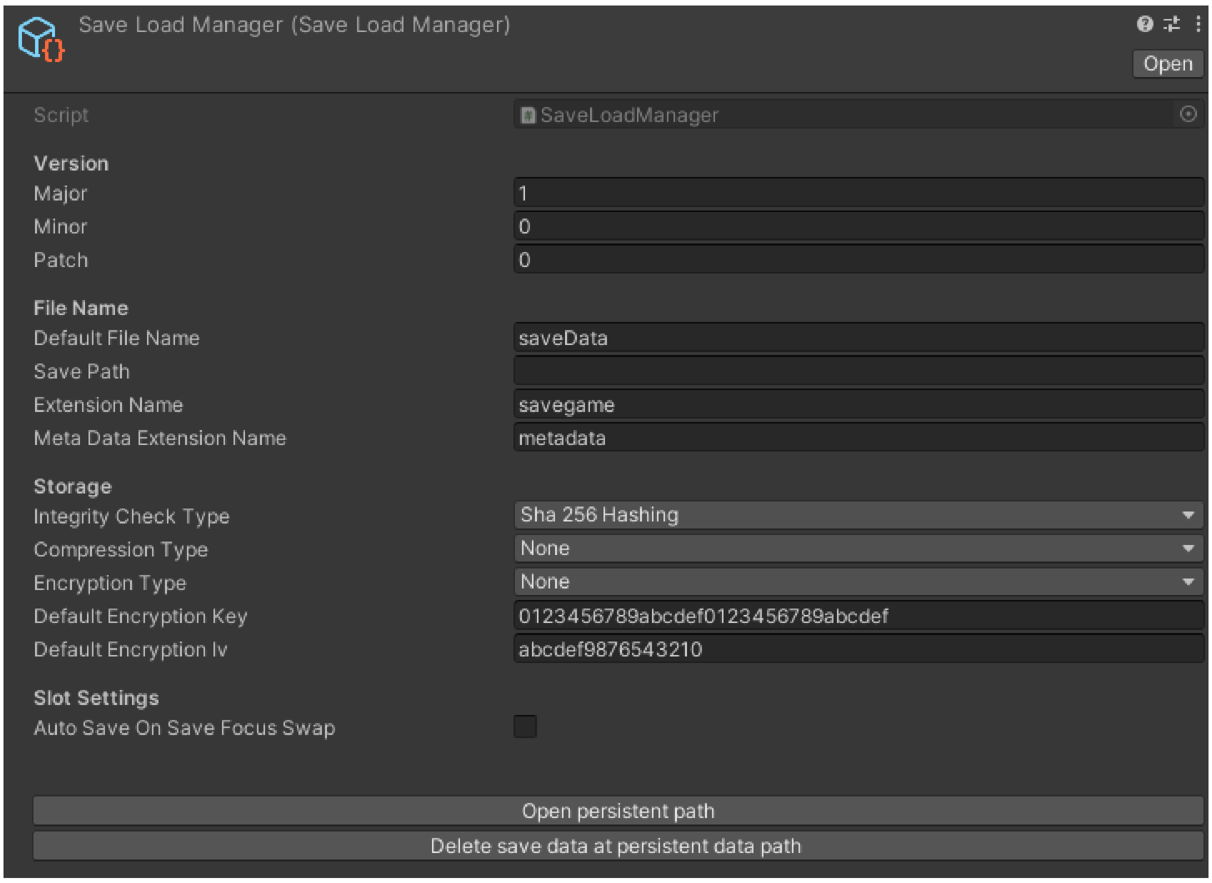
Main Components

Save Load Manager

The **SaveLoadManager** serves as the core of the save system, implemented as a Scriptable Object. Its primary role is to manage and control global settings for saving and loading data and to handle the saving process for all active scenes or a selection of scenes, ensuring data persistence across sessions.

It operates save actions based on a **current save file** that is actively being written to. This save file focus can be set explicitly through methods provided by the system or automatically when using save and load functions. If no specific filename is provided, the system defaults to a predefined save file name.

Beyond standard saving and loading, the system also offers advanced options like **Snapshotting** scenes. This will capture the current state of a scene without fully writing the save to the disk. Additionally, **deleting data** from certain scenes is possible. For more complex scenarios, there is also a method to **reload a scene and then load** the save file, ensuring that the scene is reset to its original state before applying the saved data. This can be particularly useful in cases where the scene's initial setup needs to be restored before loading specific changes from a save.
Additionally, the system provides various events that developers can hook their methods into. These events allow developers to execute custom logic at specific points during the save and load process, such as before or after a scene is saved or loaded, which for example is useful to update UI after loading.


Save Load Manager (Save Load Manager)

Open

Script	SaveLoadManager
Version	
Major	1
Minor	0
Patch	0
File Name	
Default File Name	saveData
Save Path	
Extension Name	savegame
Meta Data Extension Name	metadata
Storage	
Integrity Check Type	Sha 256 Hashing
Compression Type	None
Encryption Type	None
Default Encryption Key	0123456789abcdef0123456789abcdef
Default Encryption Iv	abcdef9876543210
Slot Settings	
Auto Save On Save Focus Swap	☐

Open persistent path

Delete save data at persistent data path

These features provide developers with greater control over how and when game data is managed, allowing for more flexible and robust save management across various gameplay scenarios. All actions related to writing and reading save data to and from disk are fully **asynchronous**, ensuring that these operations do not block the main thread. This async approach helps maintain smooth gameplay performance by offloading potentially time-consuming file operations.

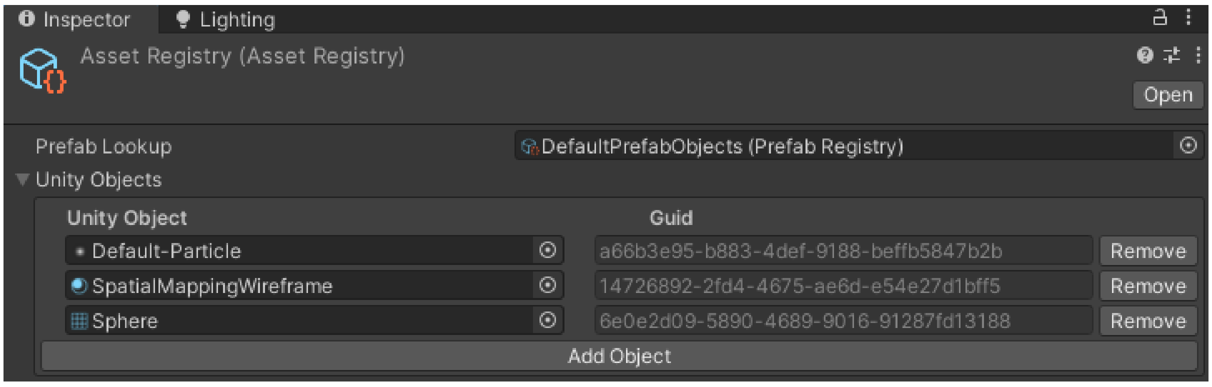
The Save System creates two files for each save operation: a **metadata file** and the actual **save data file**. The metadata file contains essential information such as the creation time of the save, the version of the save file, and a checksum to verify the integrity of the save data.

Additionally, developers can add custom data to the metaData file, allowing for more detailed and specific tracking of save states.

Asset and Prefab Registry

The **Asset Registry** is a crucial part of the save and load system that tracks and manages all assets that need to be persisted. It stores references to assets like textures, materials, and other non-prefab resources, ensuring they are correctly reloaded when a save state is restored. By maintaining a centralized record of these assets, the system can efficiently manage dependencies and reduce the risk of missing resources during loading.

If developers want to support saving data on ScriptableObjects, they must be added to the Asset Registry. This ensures that the ScriptableObjects are properly tracked, and their data is accurately saved and reloaded.



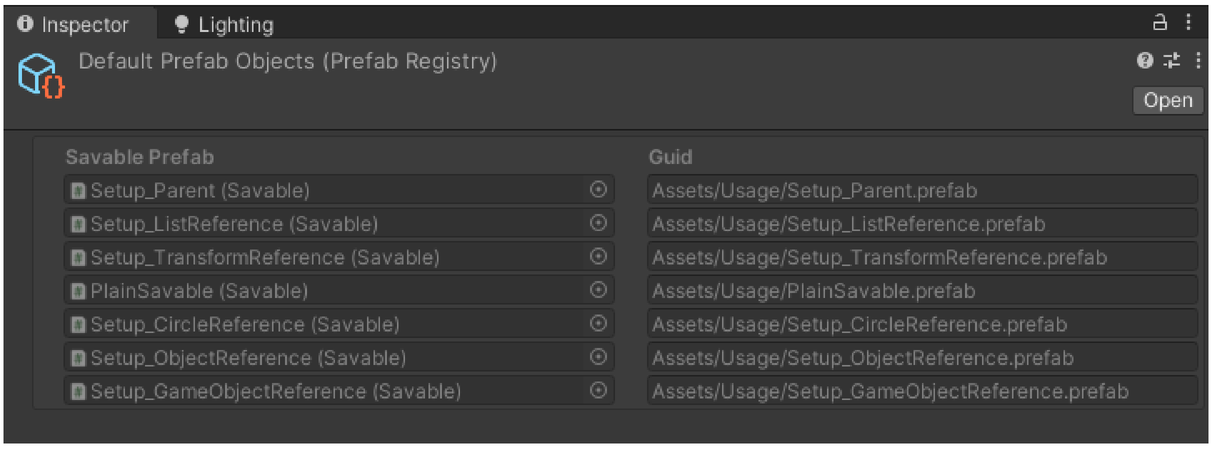
Asset Registry (Asset Registry)

Prefab Lookup: DefaultPrefabObjects (Prefab Registry)

Unity Objects

Unity Object	Guid	
Default-Particle	a66b3e95-b883-4def-9188-beffb5847b2b	Remove
SpatialMappingWireframe	14726892-2fd4-4675-ae6d-e54e27d1bff5	Remove
Sphere	6e0e2d09-5890-4689-9016-91287fd13188	Remove

The **Prefab Registry** is a Scriptable Object that automatically keeps track of all prefabs marked with the Savable Component. It is automatically created if it does not already exist. This registry is used for Dynamic Prefab Loading, where the system can instantiate necessary prefabs and remove unnecessary ones based on the saved state.



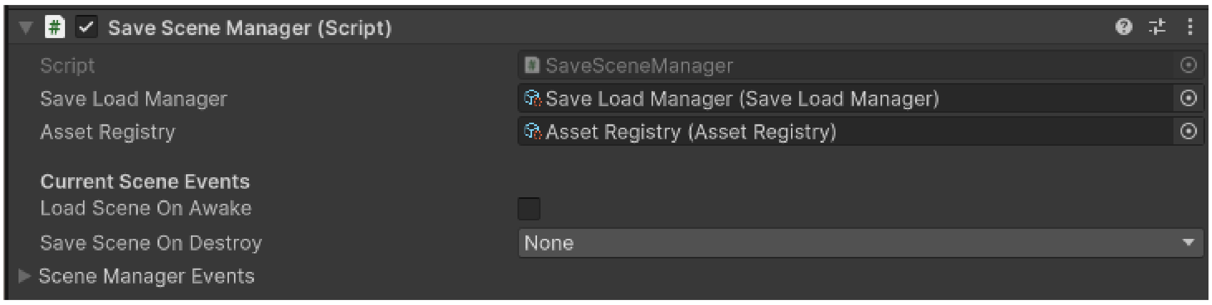
Default Prefab Objects (Prefab Registry)

Savable Prefab	Guid
Setup_Parent (Savable)	Assets/Usage/Setup_Parent.prefab
Setup_ListReference (Savable)	Assets/Usage/Setup_ListReference.prefab
Setup_TransformReference (Savable)	Assets/Usage/Setup_TransformReference.prefab
PlainSavable (Savable)	Assets/Usage/PlainSavable.prefab
Setup_CircleReference (Savable)	Assets/Usage/Setup_CircleReference.prefab
Setup_ObjectReference (Savable)	Assets/Usage/Setup_ObjectReference.prefab
Setup_GameObjectReference (Savable)	Assets/Usage/Setup_GameObjectReference.prefab

Both the Asset Registry and Prefab Registry identify objects based on their type. This requires developers to ensure that the type of a property or field matches the type of the asset in the corresponding registry. For instance, if a field is intended to store a prefab asset, it must be of type `Savable`. This is because the Prefab Registry can only identify prefabs that are of the `Savable` type.

Save Scene Manager Component

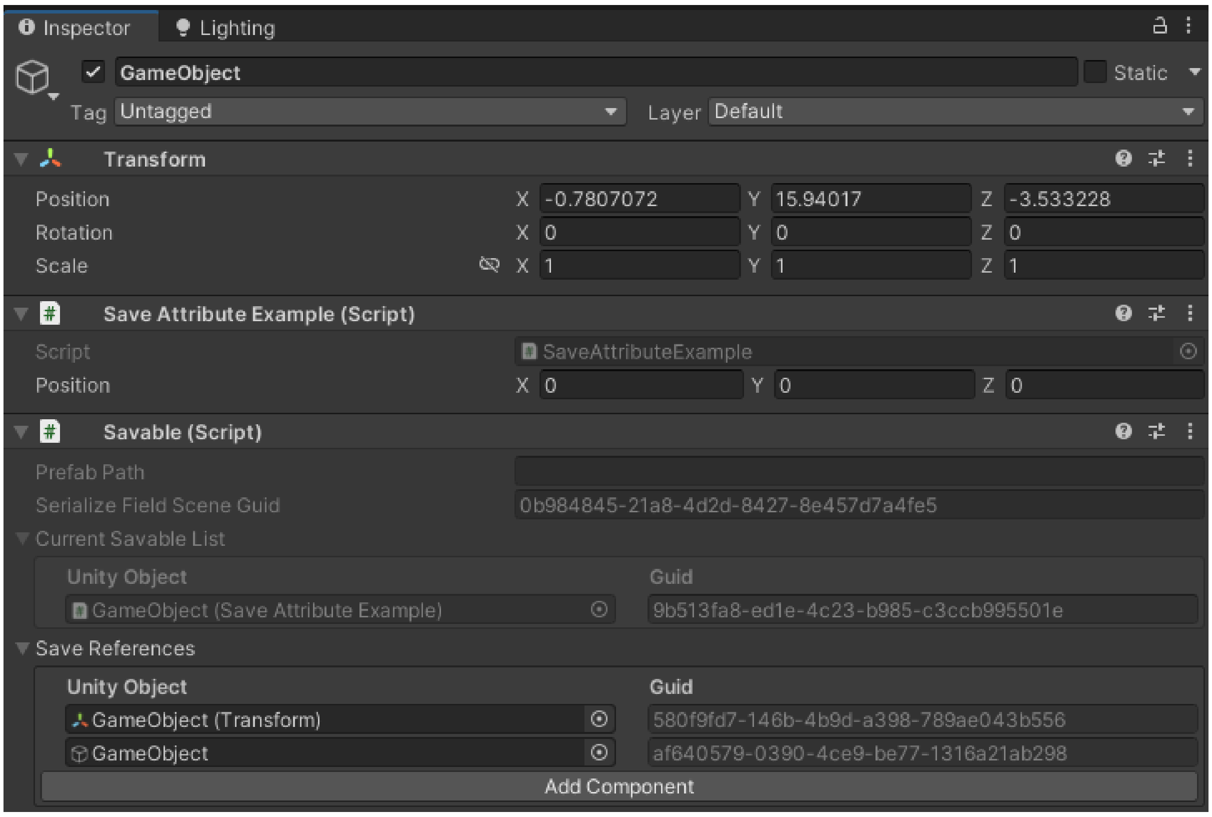
The **SaveSceneManager** is a component specifically responsible for managing save processes within individual scenes. Each scene should contain exactly one SaveLoadSceneManager to independently manage its state and ensure that scene-specific data is saved and restored correctly.


Save Scene Manager (Script)

Script	SaveSceneManager
Save Load Manager	Save Load Manager (Save Load Manager)
Asset Registry	Asset Registry (Asset Registry)
Current Scene Events	
Load Scene On Awake	☐
Save Scene On Destroy	None
Scene Manager Events	

Savable Component

The **Savable Component** is crucial for assigning unique IDs to objects within a scene. These IDs are used to identify and restore all references during loading. It automatically detects scripts that implement the Savable attributes or the ISavable interface and assigns them an ID. To check which objects on a GameObject are tracked for save and loading, developers can check the `Current Savable List` on a Savable Component.



GameObject
Tag Untagged
Layer Default

Transform

	X	Y	Z
Position	-0.7807072	15.94017	-3.533228
Rotation	0	0	0
Scale	1	1	1

Save Attribute Example (Script)

Script: SaveAttributeExample

	X	Y	Z
Position	0	0	0

Savable (Script)

Prefab Path:
Serialize Field Scene Guid: 0b984845-21a8-4d2d-8427-8e457d7a4fe5

Current Savable List

Unity Object	Guid
GameObject (Save Attribute Example)	9b513fa8-ed1e-4c23-b985-c3ccb995501e

Save References

Unity Object	Guid
GameObject (Transform)	580f9fd7-146b-4b9d-a398-789ae043b556
GameObject	af640579-0390-4ce9-be77-1316a21ab298

Additionally, the Savable Component provides a `Save References` field, allowing developers to assign IDs to arbitrary objects for reference purposes. This feature is particularly intended for components on the GameObject that contains the Savable Component. Let's say the child of a GameObject has a Savable component with a SaveParent component, which allows it to save and restore its parent after loading. To ensure that the child can correctly find and reattach to its old parent, you'll need to add the parent's transform to the parent's `Save Reference` list. This step ensures the Save System recognizes the parent's transform, allowing the child to accurately locate and re-establish its connection to the correct parent when the game is reloaded.

Important Concepts

Single Scene and Multi Scene Setup

The Save System in Unity is designed to efficiently handle both single scene and multi-scene setups, offering flexibility for different game architectures. However, there is one restriction regarding multi-scene setups: references are scene-bound. This means that an object reference saved in one scene cannot be directly linked to an object in another scene. If you save an object in Scene A, it will be treated as a separate instance when loaded in Scene B, even if the objects are identical. This design helps maintain clarity and consistency within the game's data, ensuring that each scene can be managed independently without cross-scene conflicts.

However, there is a way to handle references across scenes by using ScriptableObjects. To do this effectively:

1. **Choose One Scene as the Save and Load Origin:** Decide on a single scene to serve as the save and load origin for the ScriptableObject.
2. **Only Save in the Origin Scene:** Ensure that the ScriptableObject is only saved in this designated scene. This prevents duplication and ensures consistency across your game.
3. **Use the ScriptableObject as a Bridge:** Leverage the ScriptableObject to share data across other scenes, acting as a bridge to maintain consistent information.

Additionally, in order to support saving data on ScriptableObjects, they must be added to the Asset Registry. This ensures that the ScriptableObjects are properly tracked, and their data is accurately saved and reloaded.

The Save System saves the data of a ScriptableObject separately for each scene. If a ScriptableObject is saved in two scenes, it will be loaded twice. This could be useful if the scenes are loaded together and contain different objects that need to be applied to the ScriptableObject. However, this approach is generally not recommended. Instead, consider splitting the ScriptableObject into two separate objects to avoid conflicts and ensure a cleaner data structure.

Value Saving and Reference Saving

The Save System offers two primary methods for saving and loading objects: Value Saving and Reference Saving.

Reference Saving: When the Save System saves an object as a reference, it typically creates a separate data container to store that reference. This approach is particularly useful for handling complex objects and relationships, as it allows the system to maintain object identities and dependencies, such as polymorphic types where different derived classes might be saved and restored. While this approach provides flexibility, it is generally slower and consumes more memory due to the overhead of managing references. Therefore, it is recommended to use referenceable saving sparingly and only when necessary for complex objects or relationships that cannot be represented as simple values.

Value Saving: On the other hand, Value Saving simplifies this process. Instead of creating a new, separate container for each reference, the object is directly converted into JSON format and stored within the existing data structure. This approach makes the saving process more efficient by keeping all related data within a single container, reducing the overhead of managing multiple containers. However, Value Saving requires that the correct type be applied directly, as it doesn't support the dynamic nature of polymorphic types like Reference Saving does.

When using the Attribute Saving feature, objects will be saved using Value Saving if they meet certain criteria: they aren’t a MonoBehaviour or ScriptableObject, don’t implement the ISavable Interface, and are not part of the Type Converter system.
On the other hand, when using the ISavable interface, developers have the flexibility to choose between Reference Saving and Value Saving. This allows developers to decide the most appropriate saving method for their objects, depending on the specific needs of their

game. For instance, they can opt for Reference Saving to maintain object references and polymorphic types or use Value Saving to directly store object's data within the current reference data container.

How to mark things for saving

Attribute Saving

There are two main Attributes developers can choose from to mark things to be saved.

- `[Savable]`: This attribute can be applied to individual fields or properties to mark them for saving. It allows developers to specify precisely which data should be persisted without saving the entire class.
- `[SavableObject]`: Applying this attribute to a class automatically marks all fields and properties within that class for saving. By doing so, developers can ensure that all relevant data is persisted. This attribute is particularly suited for Data Transfer Objects (DTOs), which are designed to encapsulate and transport data between different layers of an application without any business logic. Additionally, developers need to use the `declaredOnly` parameter to specify if only the fields of the declared class should be considered, excluding fields from base classes. This is especially useful for MonoBehaviour classes, which cannot be serialized directly by Unity. Using `declaredOnly` ensures that the serialization focuses solely on the class's unique fields, avoiding potential issues with inherited fields that are not directly compatible with the save system.

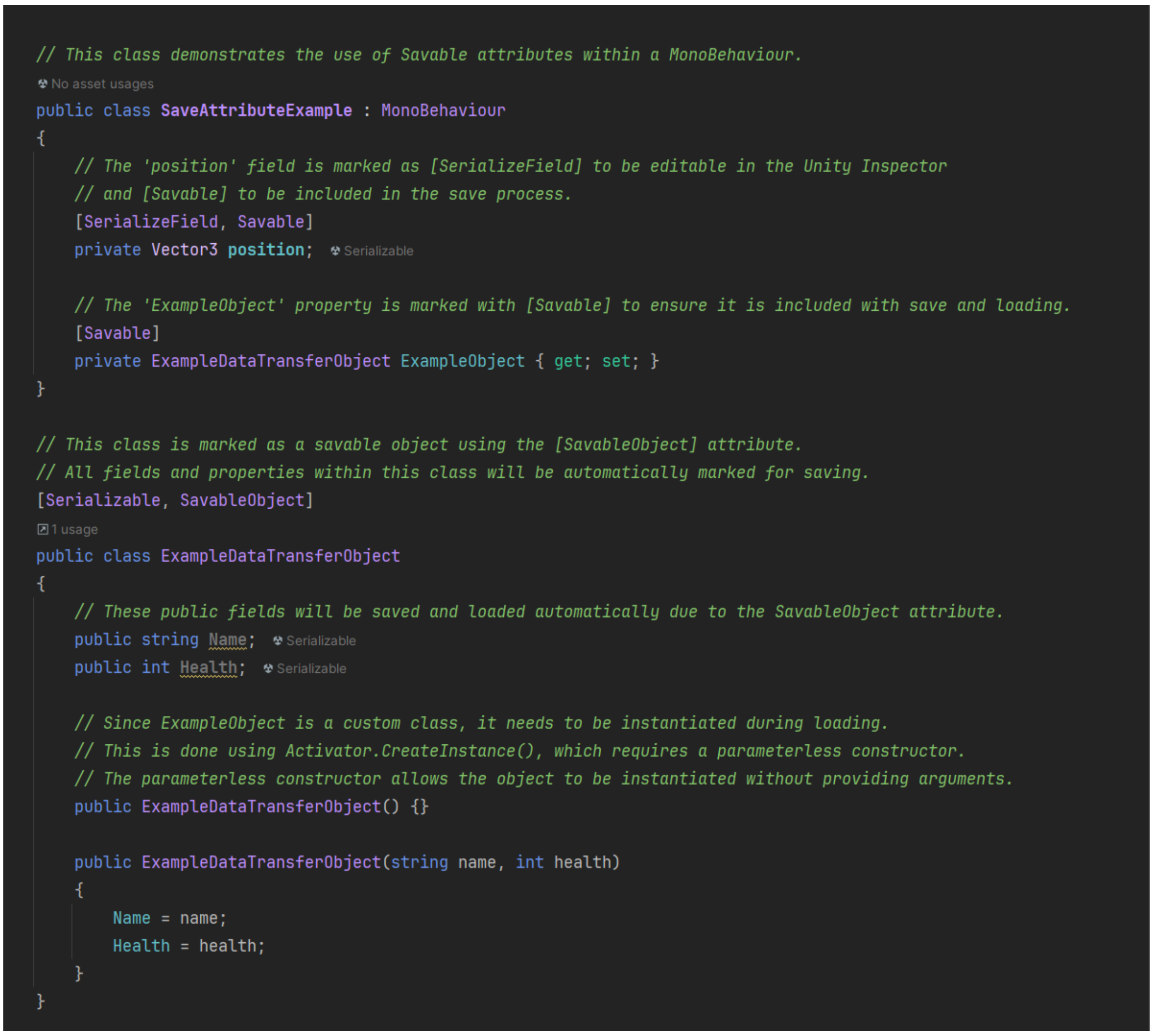
The attributes `[Savable]` and `[SavableObject]` apply to all access modifiers, including public, non-public, static, and instances for fields and properties, offering high flexibility in defining what data to save. For non-MonoBehaviour classes, the system can automatically create new instances of objects and initialize them with saved data, ensuring seamless restoration of the state. Internally it is accomplished using `Activator.CreateInstance`. This requires a parameterless constructor for classes. In contrast, structs do not need a parameterless constructor, as they are automatically instantiated with their default values.

Additionally, these attributes can be used alongside with a ISavable interface. By using both attributes and interface inside a class there is one thing developers must be aware of: it is possible to save and load the same value twice. This should be avoided.

Attribute Save Path

The save system is designed to handle complex data structures by resolving nested objects marked as savable. When nested fields or properties are saved, the system generates a unique identifier (ID) for each reference. Since an object may be used multiple times, the first occurrence path is the origin ID for that object. Every other occurrence will use that path. The ID for nested objects is constructed using the following format:

```
[SavableSceneID]/[ComponentID]/[field/property name]/[field/property name]...
```



```csharp
// This class demonstrates the use of Savable attributes within a MonoBehaviour.
No asset usages
public class SaveAttributeExample : MonoBehaviour
{
    // The 'position' field is marked as [SerializeField] to be editable in the Unity Inspector
    // and [Savable] to be included in the save process.
    [SerializeField, Savable]
    private Vector3 position;  Serializable

    // The 'ExampleObject' property is marked with [Savable] to ensure it is included with save and loading.
    [Savable]
    private ExampleDataTransferObject ExampleObject { get; set; }
}

// This class is marked as a savable object using the [SavableObject] attribute.
// All fields and properties within this class will be automatically marked for saving.
[Serializable, SavableObject]
1 usage
public class ExampleDataTransferObject
{
    // These public fields will be saved and loaded automatically due to the SavableObject attribute.
    public string Name;  Serializable
    public int Health;  Serializable

    // Since ExampleObject is a custom class, it needs to be instantiated during loading.
    // This is done using Activator.CreateInstance(), which requires a parameterless constructor.
    // The parameterless constructor allows the object to be instantiated without providing arguments.
    public ExampleDataTransferObject() {}

    public ExampleDataTransferObject(string name, int health)
    {
        Name = name;
        Health = health;
    }
}
```

ISavable interface

Unity components can’t be directly extended with additional code to support the savable attributes. To address this limitation, the ISavable interface offers a robust solution for making these components savable. Furthermore, the ISavable interface provides the ability to further customize and control the save and load process compared to the Attribute Saving. This is particularly useful when special logic or additional steps are required to correctly save or restore an object's state. You can even add this interface to Non-MonoBehaviour classes, if the instance of the class is marked as a savable.

To make a Unity component savable, implement the ISavable interface in its own MonoBehaviour class and define the `OnSave` and `OnLoad` methods. Inside those methods you can define your custom save and loading for your wanted Unity component. This is done by utilizing the provided `SaveDataHandler` and `LoadDataHandler`. There are two main options for saving and loading data:

To perform Value Saving, use the `SaveAsValue` and `LoadValue<T>` methods. This approach reduces memory usage and improves performance, making it the preferred choice for most scenarios. Reference Saving can be performed by using the `TrySaveAsReferencable` and `TryLoadReferencable` methods. However, Reference Saving requires an additional step compared to Value Saving:

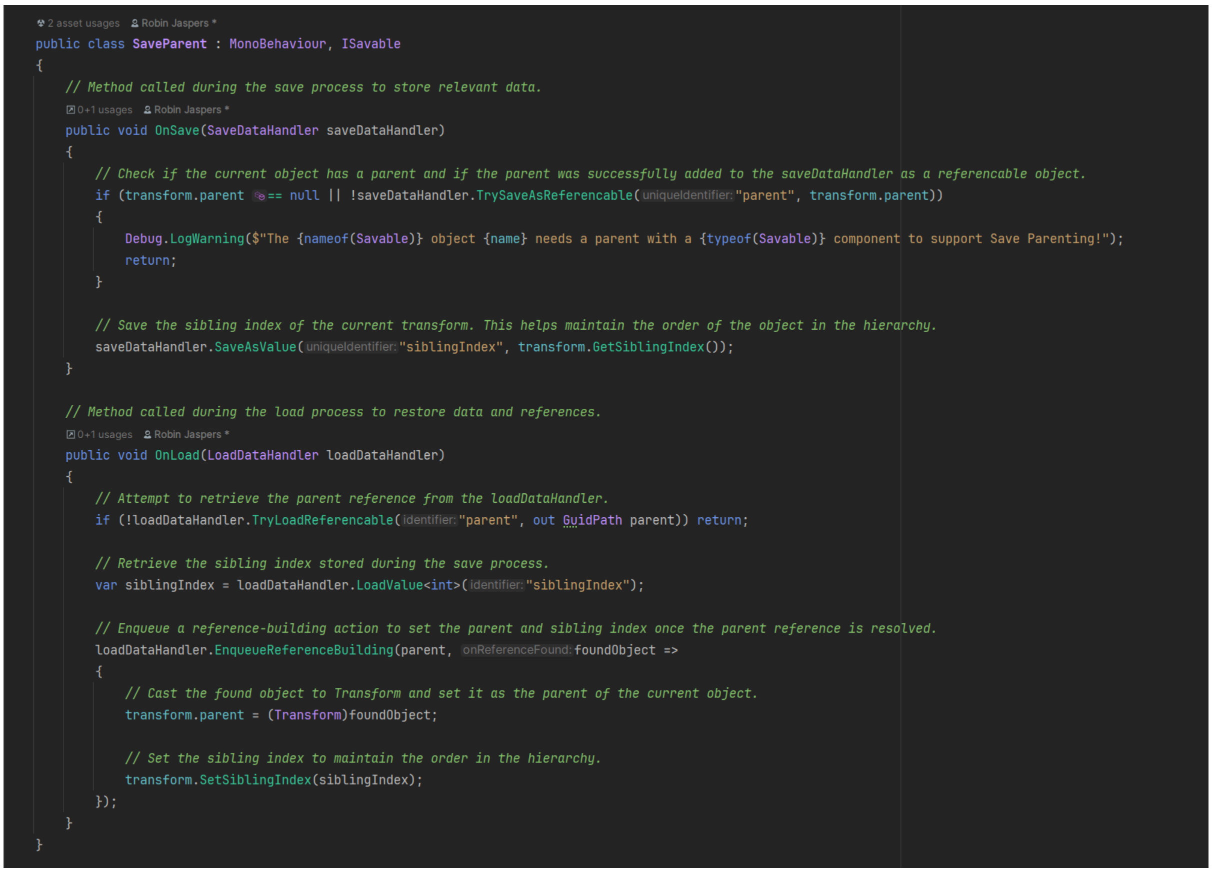
Internally, all objects are created first, and references are resolved afterward. This means that loading references will initially return only the path to the reference. To convert these paths into the desired object once all objects are created, the `LoadDataHandler` includes an additional method called `EnqueueReferenceBuilding`. This method allows you to queue the conversion of one or multiple paths into the actual object. The found object is returned as an action within the `EnqueueReferenceBuilding` method. Inside that action, the developer is responsible for converting the returned object into the required type and resolving the loading logic.

The ISavable interface can be combined with Attribute Saving to mark specific fields or properties as savable. By doing so, it is possible to save and load values twice, which should be avoided.

How to use it

1. **Inherit from ISavable:** Start by inheriting from the ISavable interface in the class you want to make savable. This will allow you to implement the required methods for saving and loading data.
2. **Ensure Proper Tracking:** Make sure that the object implementing ISavable is tracked by a Savable Component. This can be done by adding the ISavable class to the same GameObject that has the Savable Component attached. Alternatively, the object can be identified for saving through another script. For instance, if a field in a MonoBehaviour is marked with the [Savable] attribute, and the object assigned to that field implements ISavable, the Save System will automatically track and save it.

Below is an example demonstrating how the Save and Loading System can save and load the parent of an object, provided that the parent has a Savable Component. This setup ensures that even complex relationships, like parent-child hierarchies, are accurately preserved across save and load operations.



```csharp
2 asset usages  Robin Jaspers *
public class SaveParent : MonoBehaviour, ISavable
{
    // Method called during the save process to store relevant data.
    0+1 usages  Robin Jaspers *
    public void OnSave(SaveDataHandler saveDataHandler)
    {
        // Check if the current object has a parent and if the parent was successfully added to the saveDataHandler as a referencable object.
        if (transform.parent == null || !saveDataHandler.TrySaveAsReferencable(uniqueIdentifier: "parent", transform.parent))
        {
            Debug.LogWarning($"The {nameof(Savable)} object {name} needs a parent with a {typeof(Savable)} component to support Save Parenting!");
            return;
        }

        // Save the sibling index of the current transform. This helps maintain the order of the object in the hierarchy.
        saveDataHandler.SaveAsValue(uniqueIdentifier: "siblingIndex", transform.GetSiblingIndex());
    }

    // Method called during the load process to restore data and references.
    0+1 usages  Robin Jaspers *
    public void OnLoad(LoadDataHandler loadDataHandler)
    {
        // Attempt to retrieve the parent reference from the loadDataHandler.
        if (!loadDataHandler.TryLoadReferencable(identifier: "parent", out GuidPath parent)) return;

        // Retrieve the sibling index stored during the save process.
        var siblingIndex = loadDataHandler.LoadValue<int>(identifier: "siblingIndex");

        // Enqueue a reference-building action to set the parent and sibling index once the parent reference is resolved.
        loadDataHandler.EnqueueReferenceBuilding(parent, onReferenceFound: foundObject =>
        {
            // Cast the found object to Transform and set it as the parent of the current object.
            transform.parent = (Transform)foundObject;

            // Set the sibling index to maintain the order in the hierarchy.
            transform.SetSiblingIndex(siblingIndex);
        });
    }
}
```

Additional ISavable Components

The ISavable interface can also be extended with additional components to manage other aspects of a GameObject's state. In addition to the `SaveParent` component, which handles saving and restoring parent-child relationships, there are other pre-built components available, such as `SavePosition` for saving and restoring an object's position, and `SaveVisibility` for managing its active or inactive state. These components provide ready-made solutions for common saving needs, further simplifying the process of making a Unity component savable.

Interface Save Path

By using the Reference Saving, the system generates a unique identifier (ID) for each reference. Since an object may be used multiple times, the first occurrence path is the origin ID for that object. Every other occurrence will use that path. The ID for nested objects is constructed using the following format:

```
[SavableSceneID]/[ComponentID]/[unique Identifier]/[unique
Identifier]...
```

The unique identifier string originates from the parameter defined inside `SaveDataHandler` and `LoadDataHandler` methods.

Advanced Saving

Type Converter

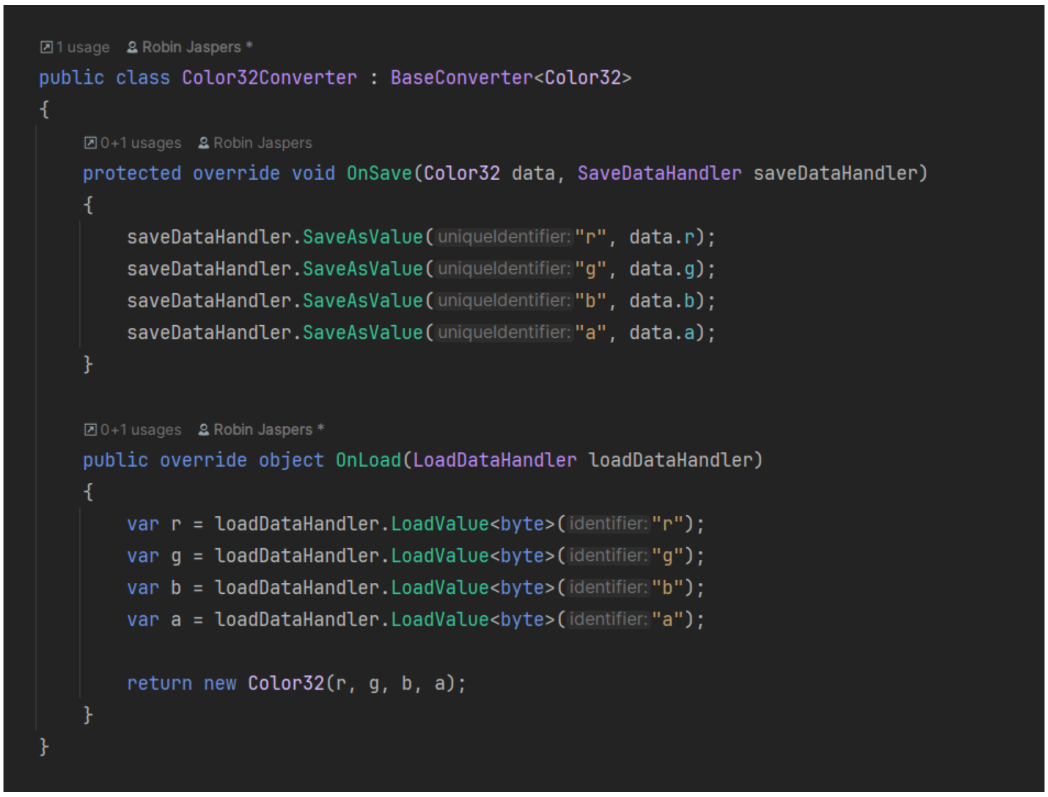
The Type Converter system in the Save System is similar to the ISavable Interface, and it is recommended to familiarize with the ISavable interface system first. The primary purpose of the Type Converter is to convert a specified type directly into a serializable format while maintaining references when loading, making it easier to save and load complex data structures. This system is mainly used for classes that do not provide direct access to change them, such as Unity types.
Both the Type Converter and ISavable interface implement the `OnSave` and `OnLoad` methods, and both provide a `SaveDataHandler` and `LoadDataHandler` for managing the saving and loading processes. The key difference lies in how they handle data conversion: in the `OnSave` method of a Type Converter, you receive the type that needs to be converted into a savable format. In the `OnLoad` method, you are responsible for reconstructing the object from the saved data and returning it.
The Type Converter is already implemented for several Unity types: `Color32`, `Color`, `Vector2`, `Vector3`, `Vector4`, and `Quaternions`. In addition to handling Unity types, the Type Converter is also used for creating references within collections such as `Array`, `List`, `Dictionary`, `Stack`, and `Queue`. This capability ensures that complex data structures, including those that contain multiple references, can be serialized and deserialized accurately, preserving the integrity of the data.

How to Use it

To create a custom Type Converter, developers can either inherit from the `BaseConverter<T>` class or directly implement the `IConvertible` interface, then implement their `OnSave` and `OnLoad` methods for serialization and deserialization:

- **Inheriting from BaseConverter:** For most use cases, inheriting from the `BaseConverter<T>` class is sufficient. The generic T represents the type for which Type Conversion should be enabled. This class provides the foundational functionality needed to create custom type converters, making it easier to handle the conversion process.

- **Implementing IConvertible:** In some specific cases, the `BaseConverter` may not work. In these instances, you will need to implement the `IConvertible` interface directly. When doing so, developers must also define the `CanConvert` method. This method determines whether the Type Converter should handle a given type. If `CanConvert` returns `true`, the Type Converter will use this class for conversion.

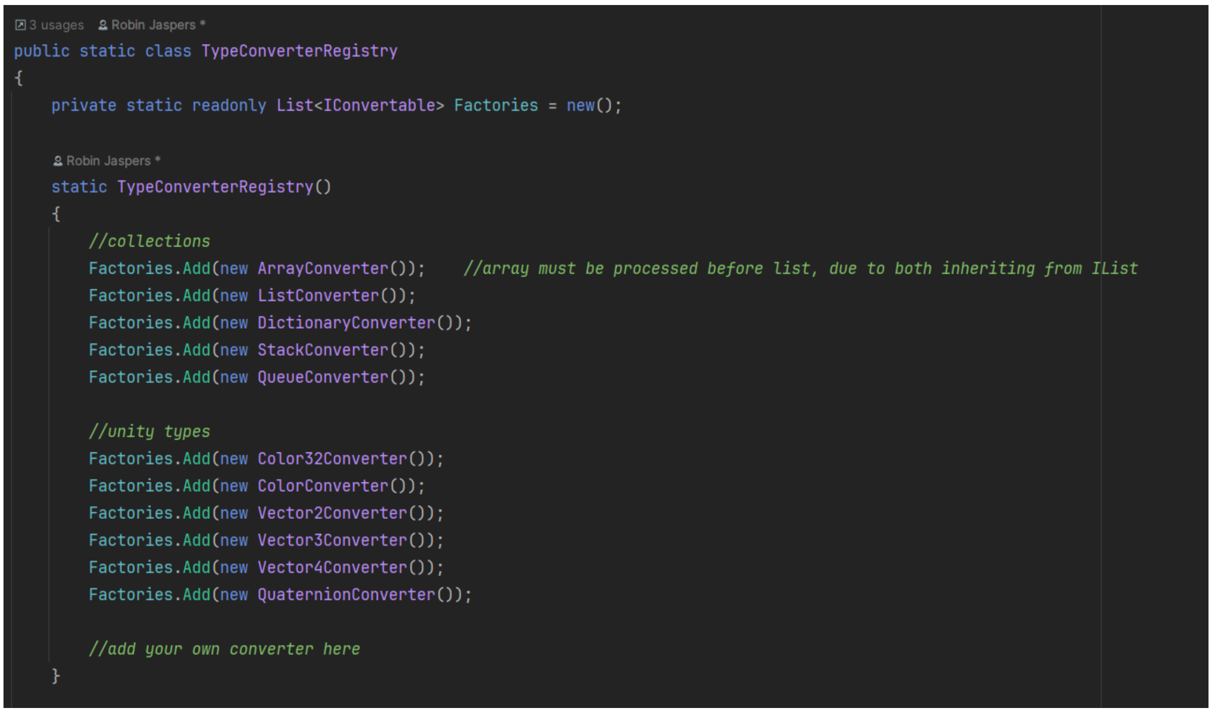


```
1 usage   Robin Jaspers *
public class Color32Converter : BaseConverter<Color32>
{
    0+1 usages   Robin Jaspers
    protected override void OnSave(Color32 data, SaveDataHandler saveDataHandler)
    {
        saveDataHandler.SaveAsValue( uniqueIdentifier: "r", data.r);
        saveDataHandler.SaveAsValue( uniqueIdentifier: "g", data.g);
        saveDataHandler.SaveAsValue( uniqueIdentifier: "b", data.b);
        saveDataHandler.SaveAsValue( uniqueIdentifier: "a", data.a);
    }

    0+1 usages   Robin Jaspers *
    public override object OnLoad(LoadDataHandler loadDataHandler)
    {
        var r = loadDataHandler.LoadValue<byte>( identifier: "r");
        var g = loadDataHandler.LoadValue<byte>( identifier: "g");
        var b = loadDataHandler.LoadValue<byte>( identifier: "b");
        var a = loadDataHandler.LoadValue<byte>( identifier: "a");

        return new Color32(r, g, b, a);
    }
}
```

After creating your custom converter, it must be registered with the Save System. This is done by adding the converter type to the static constructor of the `TypeConverterRegistry` class. Be aware that this registration process is expected to change in a future version of the Save System.



```
3 usages   Robin Jaspers *
public static class TypeConverterRegistry
{
    private static readonly List<IConvertable> Factories = new();

    Robin Jaspers *
    static TypeConverterRegistry()
    {
        //collections
        Factories.Add(new ArrayConverter());    //array must be processed before list, due to both inheriting from IList
        Factories.Add(new ListConverter());
        Factories.Add(new DictionaryConverter());
        Factories.Add(new StackConverter());
        Factories.Add(new QueueConverter());

        //unity types
        Factories.Add(new Color32Converter());
        Factories.Add(new ColorConverter());
        Factories.Add(new Vector2Converter());
        Factories.Add(new Vector3Converter());
        Factories.Add(new Vector4Converter());
        Factories.Add(new QuaternionConverter());

        //add your own converter here
    }
```

Dynamic Prefab Loading

A key feature of this Save and Loading System is its ability to save references and restore them during loading. In a Unity application, prefabs can be instantiated and destroyed during runtime according to requirements. However, if a prefab is missing during loading, its associated references may not be restored correctly. To prevent this, the system saves active prefabs so that the same prefabs exist upon the next load, keeping references intact. This ensures that a scene can be reconstructed seamlessly and without data loss.

Dynamic Prefab Loading compares the current prefabs with the required prefabs. It instantiates necessary prefabs and destroys redundant ones, allowing different save states to be loaded without reloading a scene.

To ensure the Save and Loading System can identify savable prefabs, desired prefabs need the Savable Component. All savable prefabs are automatically added to the Prefab Registry. If a prefab with a Savable Component should not support dynamic prefab loading, it can be disabled using the "Dynamic Prefab Spawning Disabled" inspector boolean.

Prefabs with Overrides

When possible, it is recommended to avoid destroying prefabs with overrides. Developers can identify prefabs with overrides by looking for modifications in the prefab instance compared to its original definition. In Unity, these overrides are typically marked with a blue line in the Inspector, indicating that certain properties have been modified from the prefab's default state.
Prefabs with overrides require special handling, as the dynamic prefab loading system cannot save overrides. If a prefab with overrides is missing during loading because it was destroyed during runtime, it will be lost when the original prefab is re-instantiated in its default state. If destroying prefabs with overrides is necessary, the following solutions are recommended:

- **Set Prefabs to Inactive**: Instead of destroying the prefabs, deactivate them by setting them inactive. Developers can use the `SaveVisibility` component to save and load their active/inactive state. This ensures that the prefabs, along with their overrides, remain intact and are not permanently lost.
- **Reload the Scene**: Before loading a saved state, reload the entire scene to reset it to its original condition. This process ensures that all prefabs, including those with overrides, are instantiated in their original form with all initial modifications and configurations intact. This is particularly useful if the scene's state depends heavily on runtime changes that are not captured by the save system.

Future Plans

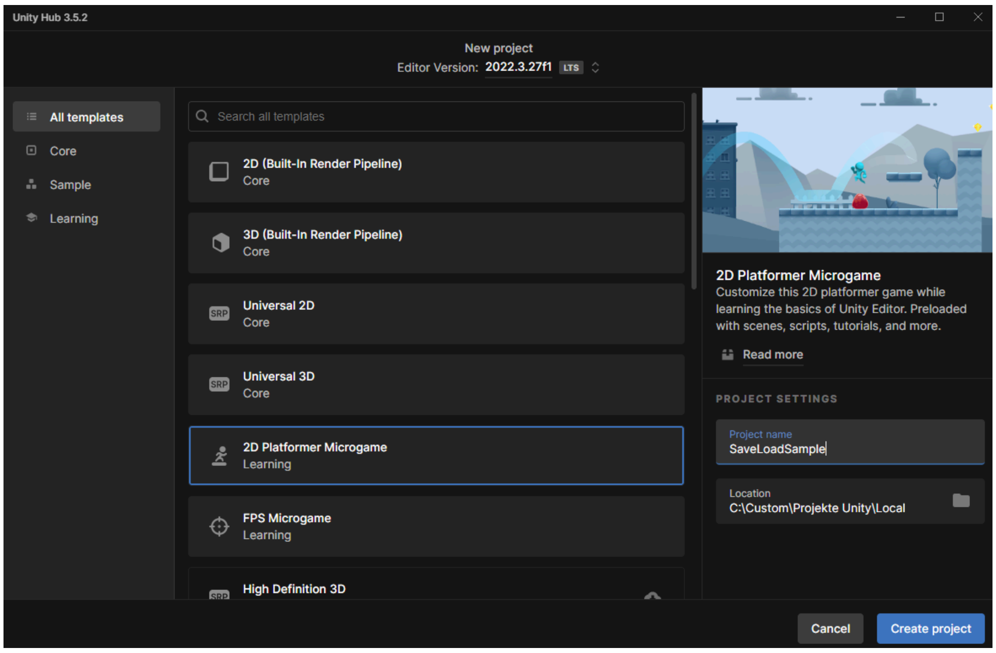
- Save References inspector field on Savable Component: make them resolve itself, so that no user input is needed
- implement a way to easier find save paths inside json and in inspector
- versioning system
- improve security regarding saving of Type.AssemplyQualifiedName (which is currently needed for polymorphic Reference Saving) and make it easy usable for versioning
- improve error handling
- improve adding elements to the Type Converter
- among others

Getting Started with the SaveLoadSystem in Unity

This tutorial will guide you through setting up the SaveLoadSystem in a new Unity project based on the "2D Platformer Microgame" template. Follow these steps to integrate the system and quickly set up saving and loading functionality for key game objects.

Step 1: Download and Import the SaveLoadSystem

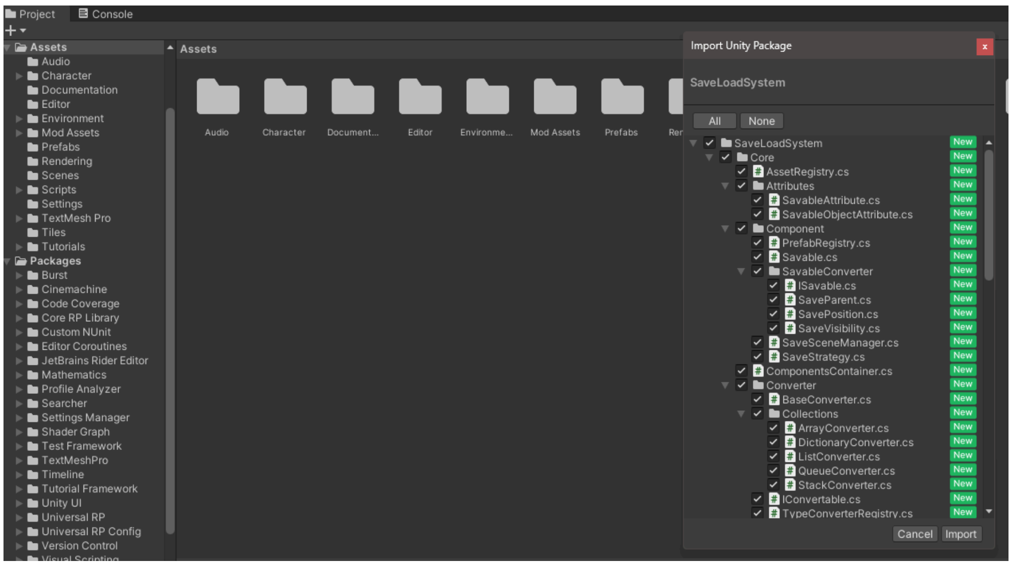
1. Download the "SaveLoadSystem.unityPackage" file containing the SaveLoadSystem from here.
2. Open Unity and create a new project using the "2D Platformer Microgame" template.



2D Platformer Microgame
Customize this 2D platformer game while learning the basics of Unity Editor. Preloaded with scenes, scripts, tutorials, and more.

PROJECT SETTINGS
Project name: SaveLoadSample
Location: C:\Custom\Projekte Unity\Local

3. Drag the `.unitypackage` file into your Unity project window and import all the contents.

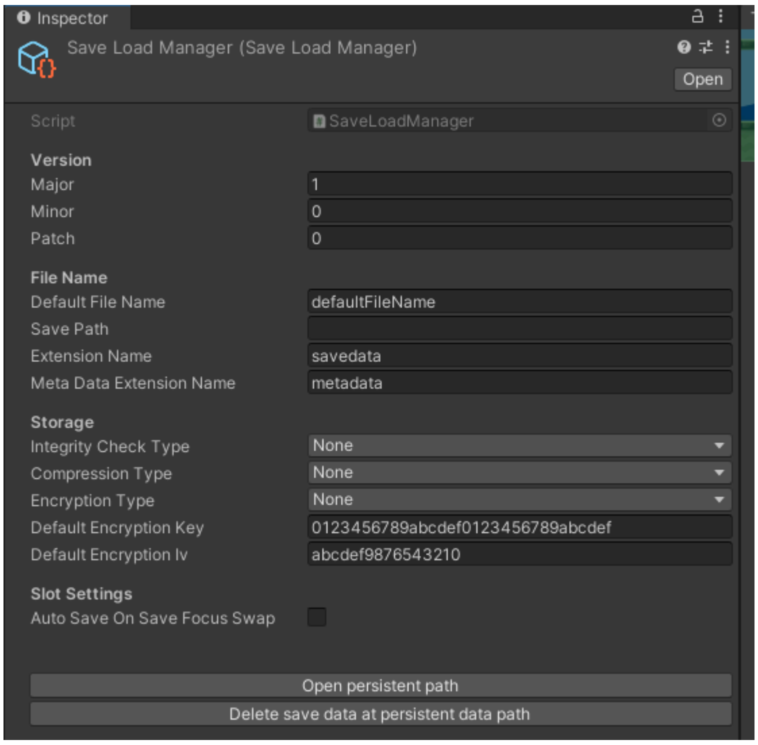


Import Unity Package: SaveLoadSystem

- ✓ SaveLoadSystem (New)
  - ✓ Core (New)
    - ✓ AssetRegistry.cs (New)
    - ✓ Attributes (New)
      - ✓ SavableAttribute.cs (New)
      - ✓ SavableObjectAttribute.cs (New)
    - ✓ Component (New)
      - ✓ PrefabRegistry.cs (New)
      - ✓ Savable.cs (New)
      - ✓ SavableConverter (New)
        - ✓ ISavable.cs (New)
        - ✓ SaveParent.cs (New)
        - ✓ SavePosition.cs (New)
        - ✓ SaveVisibility.cs (New)
      - ✓ SaveSceneManager.cs (New)
      - ✓ SaveStrategy.cs (New)
    - ✓ ComponentsContainer.cs (New)
    - ✓ Converter (New)
      - ✓ BaseConverter.cs (New)
      - ✓ Collections (New)
        - ✓ ArrayConverter.cs (New)
        - ✓ DictionaryConverter.cs (New)
        - ✓ ListConverter.cs (New)
        - ✓ QueueConverter.cs (New)
        - ✓ StackConverter.cs (New)
      - ✓ IConvertable.cs (New)
      - ✓ TypeConverterRegistry.cs (New)

Step 2: Initial Setup

1. **Create Scriptable Objects:**
   - Upon importing the package, a Prefab Registry ScriptableObject will be automatically created
   - Manually create an Asset Registry ScriptableObject.
   - Create a Save Load Manager ScriptableObject.
2. **Configure the Save Load Manager:**
   - Open the Save Load Manager in the Inspector.
   - Set a default file name, extension name, metadata extension name, and version number for the save file.


Inspector
Save Load Manager (Save Load Manager)
Open
Open persistent path
Delete save data at persistent data path

Script: SaveLoadManager

Version
Major: 1
Minor: 0
Patch: 0

File Name
Default File Name: defaultFileName
Save Path:
Extension Name: savedata
Meta Data Extension Name: metadata

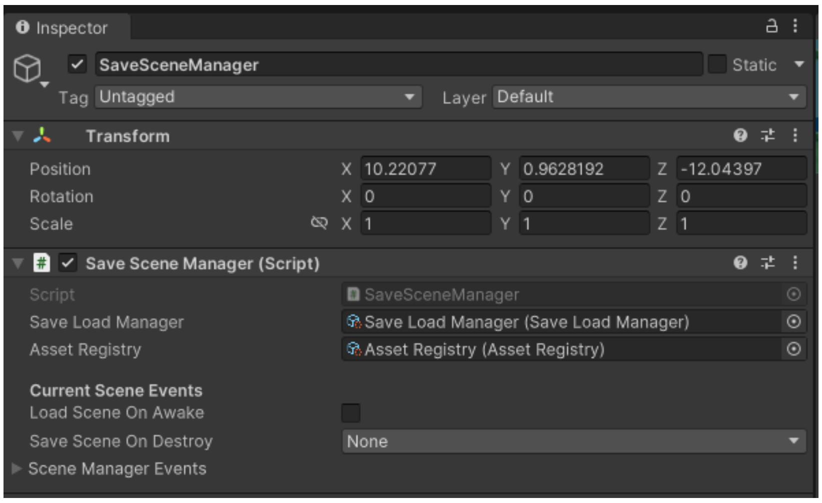
Storage
Integrity Check Type: None
Compression Type: None
Encryption Type: None
Default Encryption Key: 0123456789abcdef0123456789abcdef
Default Encryption Iv: abcdef9876543210

Slot Settings
Auto Save On Save Focus Swap: ☐

3. **Link Registries:**
   - In the Asset Registry Inspector, drag the Prefab Registry into the "Prefab Lookup" field.

Step 3: Set Up the Save Scene Manager

1. **Create Save Scene Manager:**
   - In the Sample Scene of the 2D Platformer Microgame, create a new empty GameObject.
   - Name it SaveSceneManager.
   - Add the “Save Scene Manager” component to this GameObject.
   - In the Inspector, link the associated ScriptableObjects (Asset Registry, Save Load Manager) to the relevant fields.



Position X 10.22077 Y 0.9628192 Z -12.04397
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Save Scene Manager (Script)
Script SaveSceneManager
Save Load Manager: Save Load Manager (Save Load Manager)
Asset Registry: Asset Registry (Asset Registry)

Current Scene Events
Load Scene On Awake
Save Scene On Destroy: None
Scene Manager Events

Step 4: Prepare GameObjects for Saving

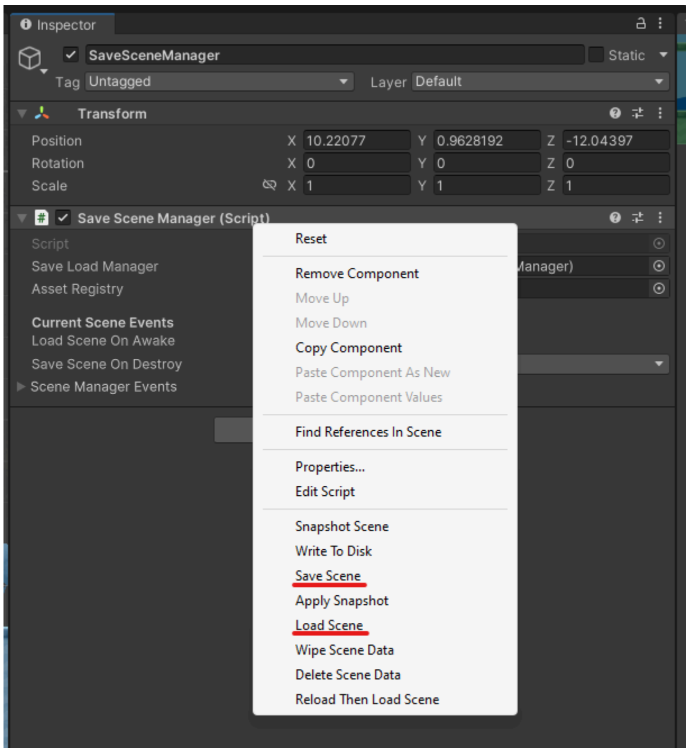
You will need to configure three types of objects for saving: the player prefab, enemy prefab, and all the tokens.

1. **Player Prefab:**
   - Locate the **Player** prefab in the Project window.
   - Open the prefab to make changes directly.
   - Add the Savable component and the **SavePosition** component to the Player prefab.
2. **Enemy Prefab:**
   - Locate the **Enemy** prefab in the Project window.
   - Open the prefab to make changes directly.
   - Add the Savable component and the **SavePosition** component to the Enemy prefab.
3. **Tokens:**
   - In the **Sample Scene** Hierarchy under Tokens, select all token objects (Select the top token, then hold Shift and select the bottom token to highlight all tokens)
   - Add the Savable component and the **SaveVisibility** component to the tokens.

Step 5: Testing the Save and Load Functionality

1. Play and Save the Game:
   - Enter Play mode in Unity and interact with the game to modify the positions and states of the player, enemies, and tokens.
   - Right-click the SaveSceneManager component in the Inspector and select **Save Scene** to save the current state.

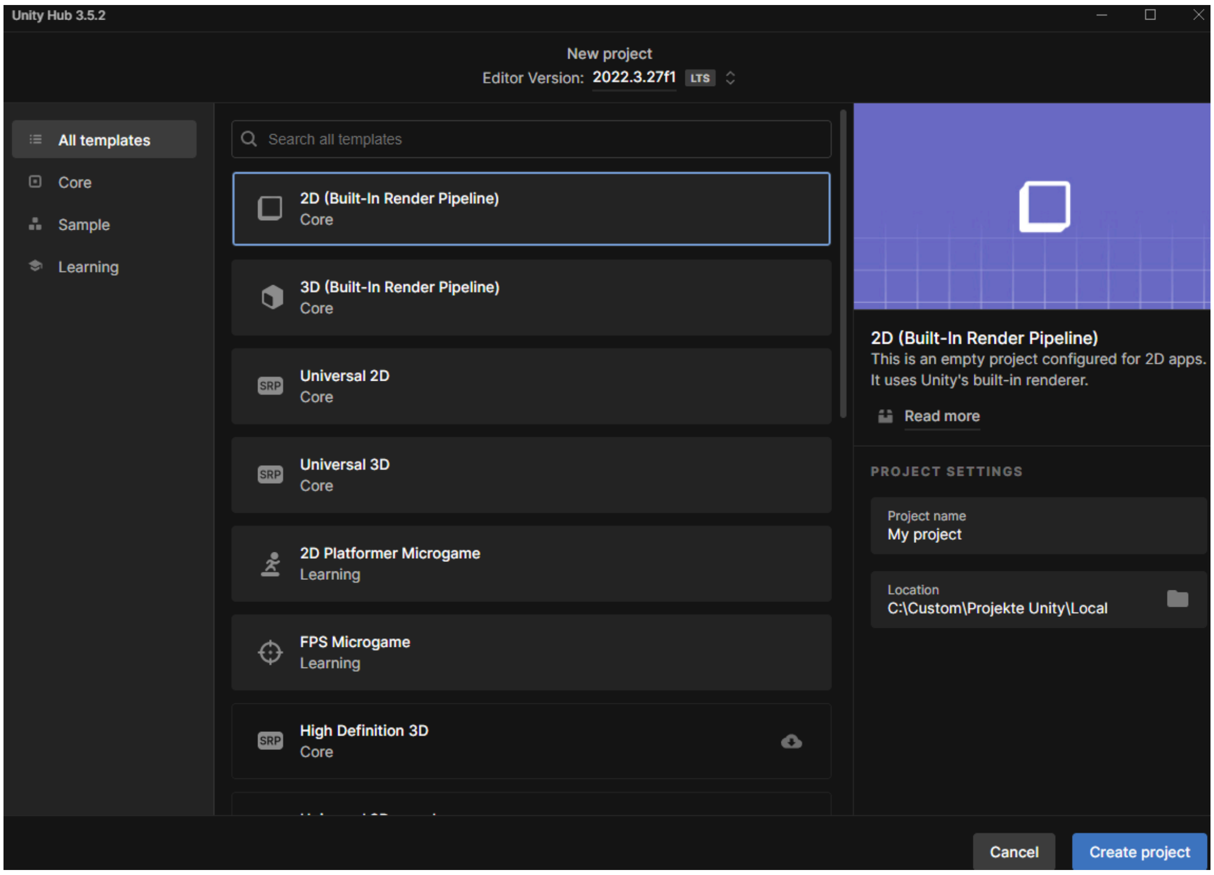
2. **Load the Scene:**
   - Restart Play mode.
   - In the Inspector, click **Load Scene** on the SaveSceneManager component to load the saved state.
   - The player, enemies, and tokens should return to the state they were in when you saved the scene.


Inspector
SaveSceneManager   Static
Tag Untagged   Layer Default
Transform
Position X 10.22077 Y 0.9628192 Z -12.04397
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Save Scene Manager (Script)
Script
Save Load Manager   lanager)
Asset Registry
Current Scene Events
Load Scene On Awake
Save Scene On Destroy
Scene Manager Events

Getting Started with Coding

This tutorial focuses on coding with the SaveLoadSystem. You'll learn how to code, configure dynamic prefab loading and manage references effectively. By the end, you'll have a fully functional save and load system for an Inventory. Note that this Inventory is an example and not production code!

1. **Create a New Project:** Start by creating a new project using the "2D Built-In Render Pipeline." Then, import the "Exercise Two" package from the SaveLoadSystemPackages directory.



Editor Version: 2022.3.27f1 LTS

2D (Built-In Render Pipeline)
This is an empty project configured for 2D apps. It uses Unity's built-in renderer.

Project name
My project

Location
C:\Custom\Projekte Unity\Local

2. **Add "Savable" Component to "InventoryElement" Prefab:** The "InventoryElement" prefab requires a "Savable" component to enable Dynamic Prefab Loading.
3. **Add "SaveParent" to "InventoryElement" Prefab:** Since the "InventoryElement" prefab should maintain the same parent after Dynamic Prefab Loading, you need to add the SaveParent script to it. For the "SaveParent" script to identify its parent reference, the parent must have an ID. Therefore, add a "Savable" component to the parent at "Canvas/Scroll View/Viewport/Content"(this is a hierarchy path) GameObject. After saving and loading, all previously existing InventoryElement prefabs should reappear, though the UI will be empty.
4. **Use the Attribute Saving System:** To save the content of individual class instances, we will use the Attribute Saving System. This system allows fields and properties to be marked so their content, including references, can be saved. Upon loading, all references should be correctly reassigned.

   a. **Apply Attribute to Fields in the “Item” Class:** For the standard C# class “Item”, all fields/properties must be marked as savable. A shortcut for this is the [SavableObject] attribute, which automatically marks all fields and properties. Classes that do not inherit from “MonoBehaviour” or “ScriptableObject”, and do not use Attribute Saving, the ISavable interface, or the Type Converter, will not be saved as references (they will be converted to JSON and directly saved to the current save reference object). In this case, using attributes is necessary to avoid this. Note that even references to collections can be saved. This is internally handled by the Type Converter.
   b. **Apply Attribute [Savable] to Fields/Properties in “Inventory” and “InventoryElement” Scripts:** In the Inventory and InventoryElement scripts, there is a field/property that must be marked with the [Savable] attribute.
   c. **Apply Attribute [Savable] to Fields/Properties in “InventoryView” Script:** The InventoryView script contains elements that must be marked with the [Savable] attribute.
5. **Add “Savable” Component to GameObjects:** To ensure the system recognizes “MonoBehaviour” scripts with attributes, you need to add the “Savable” component to the GameObjects they are attached to. Locate these GameObjects and add the “Savable” component.
6. **Add Necessary Assets for Save and Load System:** To allow assets to be referenced by the save and loading system, you need to add the required assets to the Asset Registry. This includes Sprites as well.
7. **Test Saving and Loading:** When you attempt to save and load now, not only will the prefabs be correctly assigned, but all objects will also be correctly referenced. However, the UI must still be updated to reflect the new values. To do this, find the appropriate event in the “SaveFocus” field of the “SaveLoadManager.” Then, use this event to call the Setup(Item item) method in the “InventoryElement” script, passing the “ContainedItem” property as the parameter.
8. **Done:** Everything should now Save and Load correctly! You should be able to use the button “Delete Random Inventory Item” without issues after loading now! To further check if everything is correctly loaded, you can use the debugging mode inside the inspector.