Save Design User Guide - Table of Contents

1. Introduction

1.1 Thank You for Your Purchase & Script Reference Guide

Thank you very much for purchasing **Save Design**. We hope this asset makes the implementation and maintenance of save features in your project more comfortable.

This document explains the **basic usage of Save Design** and the setup flow. For more technical information such as detailed API specifications, attribute usage, and interface definitions, please also refer to the separate **Script Reference**.

> *The Script Reference is included in `Documentation/save-design-script-reference.pdf`. Reading it together with this usage guide will help you use Save Design more smoothly.*

1.2 What is Save Design (Overview of Features and Mechanism)

Save Design is a save data management framework designed for Unity. Its most distinctive feature is that you can **access save data without using string keys at all**.

Developers explicitly define target classes for saving using attributes like `[SharedData]` or `[SlotData]`. Save Design then **automatically generates static APIs**, enabling type-safe save/load/initialize operations with IDE auto-completion.

🔧 Components of Save Design

Component	Description
Data Definition by Attributes	Use `[SharedData]`, `[SlotData]`, `[TempData]`, etc. to classify save targets
Automatic Static API Generation	Functions such as `SD.Save.Slot(...)` or `SD.Load.Shared()` are automatically generated
Strong Support for Completion & Renaming	All operations are based on type information, preventing key typos and missed updates
Easy to Introduce	Just add attributes to start, with minimal initialization and setup
Optional Encryption Support	AES + HMAC encryption can be enabled when needed

---

🎯 Goals of Save Design

- Maintain **organized save design** even as data structures become complex
- Implement and maintain save processing in a **type-safe and simple way**
- Flexibly accommodate realistic save requirements such as **multiple slots and temporary states**

For developers with these needs, Save Design is a reliable tool that can **eliminate the hassle around saving**.

---

2. Minimal Setup Steps

2.1 Requirements (Unity Version / Installation)

✅ Supported Unity Versions

- **Unity 2022.3 LTS or later** is recommended
- Must be a version that supports APIs equivalent to .NET Standard 2.1

📦 Importing the Package

Save Design is distributed as a Unity package. After import, the following structure will be deployed under `Assets/Plugins/Save Design/` :

```
Assets/Plugins/Save Design/
 ├── Editor/        // Editor extensions (setup, encryption settings, code generation)
 ├── Runtime/       // Core functionality such as attributes and interfaces
 ├── Samples/       // Usage examples and sample scenes
 └── Documentation/  // Script reference and this usage guide
```

🔧 Easy Setup via Editor Tool

Save Design provides a **dedicated editor tool** to simplify initial setup. From script scaffolding to config asset creation, everything can be done in just a few clicks.

🔹 Script Auto-Generation

1. In Unity's menu, select **Tools** > **Save Design** > **Setup** to open the **Save Design Setup Window**.

2. Enter your `Namespace` , then click the **Generate scripts** button.

3. The following scripts will be automatically generated in the specified path:

    - An SD root class annotated with `SaveDesignRoot`
    - A `SaveDesignConfig` class implementing `ISaveDesignConfig` (for configuration)

This prepares the minimum structure required to start using Save Design.

🔹 Creating the Config Asset

Next, select **Tools** > **Save Design** > **Create Save Design Config** from the Unity menu.

This creates a `SaveDesignConfig` asset under the `Assets/Resources` folder. In this asset, you can configure **file-related settings** such as:

- Save folder name
- File names for SharedData / SlotData
- File extension
- How to handle exceptions during initialization and read/write

Once configuration is complete, simply define your data classes with `[SharedData]` or `[SlotData]` attributes, and you can start using Save Design immediately.

This setup step only needs to be done once per project. After that, you can safely and simply implement save processing through the auto-generated APIs with IDE autocompletion.

⚠️ Remove Unnecessary Samples

Save Design comes with simple **sample classes** and **sample scenes** for quick verification and reference right after installation.

However, these are usually unnecessary for your actual game, and **if left as is, they may be included in the final build**.

Delete the `Assets/Plugins/Save Design/Samples` folder to ensure your production environment does not contain unwanted data.

🧩 Optional Libraries

MessagePack for C# (High-speed Serializer)

- If you want to use it, install MessagePack for C#
- Supported by adding `[SaveDesignRoot(SerializerType.MessagePack)]` to your root class

Newtonsoft.Json (Feature-rich JSON Serializer)

- If you want to use it, install Newtonsoft.Json
- Supported by adding `[SaveDesignRoot(SerializerType.NewtonsoftJson)]` to your root class
- For saving common Unity types like `Vector3` , it is also recommended to install Newtonsoft.Json-for-Unity.Converters

MemoryPack (High-speed Serializer)

- If you want to use it, install MemoryPack
- Supported by adding `[SaveDesignRoot(SerializerType.MemoryPack)]` to your root class

> *These libraries are **not bundled** with Save Design. Please install them separately as needed and use them in compliance with their licenses.*

2.2 Creating Data Definition Classes

In Save Design, you define the data to be saved as classes, and classify them by attaching attributes such as `[SharedData]` or `[SlotData]` .

This “data definition” allows Save Design to automatically generate save APIs.

🔹 Types of Data and Their Use Cases

Attribute Name	Usage Example
`[SharedData]`	Player settings or progress shared across all slots
`[SlotData]`	Slot-specific save content (character info, inventory, etc.)
`[TempData]`	Temporary data not saved to file (session flags, etc.)
`[SlotMetaData]`	Metadata for slot display (save date, playtime, etc.)

> *For details on attribute usage and structures, see Section 2. Attributes in the Script Reference.*

🧩 Example: Player Data Per Slot

```
[SlotData]            // Slot-specific data is annotated with SlotData
[System.Serializable] // Required for serialization with JsonUtility
public class Player
{
    public int level;
    public float hp;
    public Vector3 position;
}
```

By simply defining a class like above, you can save and load it via the auto-generated APIs:

```
SD.Load.Slot(slotIndex);          // Load

var level = SD.Slot.Player.level; // Access values

SD.Slot.Player.level++;           // Modify values

SD.Save.Slot(slotIndex);          // Save
```

> 💡 ***Note: Which fields are saved depends on the serializer you use***
>
> *In Save Design, actual serialization and deserialization are handled by external serializers such as `JsonUtility` or `MessagePack for C#`. Therefore, "which fields are saved" follows the rules of the chosen serializer.*
>
> *For example, with `UnityEngine.JsonUtility`:*
>
> - *`public` fields are saved*
> - *`private` fields are saved if marked with `[SerializeField]`*
> - *Properties and fields marked with `[NonSerialized]` are not saved*
>
> *With `MessagePack for C#`, `[MessagePackObject]` and `[Key(n)]` are required.*
>
> ***Always check the serializer's specifications to ensure your intended data is correctly saved and restored.***

3. Practical Usage Examples

3.1 Reading and Writing Shared Data

Data annotated with `[SharedData]` is treated as **shared data** accessible across all save slots. This is suitable for saving information that does not depend on individual slots, such as user settings or overall game progress.

✅ Recommended Timing and Method for Loading

Shared data is generally loaded **once at game startup** and treated as resident data thereafter. We recommend implementing this loading process using a static initialization function in your root class:

```
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
static void InitSaveDesignConfig()
{
    // Load configuration settings
    config = Resources.Load<SaveDesignConfig>(”SaveDesignConfig”);

    // Load shared data, or initialize if none exists
    if (!Load.Shared()) Initialize.Shared();
}
```

This ensures that **shared data is prepared before any scene loads**, so subsequent code can safely access `SD.Shared.xxx` .

💾 Saving Timing and Design Policy

There are two main approaches to saving shared data:

1. **Save immediately after a change occurs (recommended)**
2. **Save collectively at game exit**

The first approach is more reliable because the latest changes will always be preserved, even if the application closes unexpectedly.

✅ Approach 1: Save Immediately After Changes (Recommended)

```
SD.Shared.settings.volume = newVolume;
SD.Save.Shared();
```

This guarantees that the most recent change is preserved, making it ideal for settings and progress data.

⚠️ Approach 2: Save on Application Exit

```
[RuntimeInitializeOnLoadMethod]
private static void RegisterSaveOnExit()
{
    Application.quitting += () => SD.Save.Shared();
}
```

This is simple to implement, but only works if Unity triggers a “normal exit.” It may not run in cases such as forced termination from the Task Manager or OS process management (e.g., on mobile).

3.2 Slot Saves and Auto-Saves

Save Design supports **slot-based saving**, where you can specify a numeric index (e.g., Slot 1, Slot 2, Slot 3) to manage multiple save files. This makes it easy to implement common game systems such as “three save slots” or “multiple character profiles.”

In addition to numeric slots, Save Design also allows you to save using **string identifiers**. This provides more flexibility for cases where saves don’t fit neatly into a numbered slot system. Typical examples include:

- `”autosave”` → For automatic save points
- `”checkpoint-3”` → For stage or checkpoint-specific saves

✨ Explicit Control, No Hidden Auto-Saving

A key design principle of Save Design is that **it never saves automatically behind the scenes**. All saves occur only when the developer explicitly calls the API.

This has several advantages:

- You avoid unexpected overwrites caused by hidden auto-saves
- You always know exactly **when** and **what** was saved
- It keeps the flow of your game logic transparent and easy to reason about

As a result, you can implement both **slot-based saves** and **identifier-based saves** in a consistent, predictable manner while retaining full control over your game’s save behavior.

3.3 Using and Cautions for SlotMetaData

`SlotMetaData` is used for **metadata about save slots**, such as display in slot selection screens. Examples: player name, chapter, playtime, last saved timestamp.

If your game does not require slot-based saves, you don’t need to define `SlotMetaData`.

✅ Usage Examples

- Display player name, chapter, and playtime in slot selection UI
- Record last save date/time
- Flag whether a slot is empty

`SlotMetaData` is automatically saved alongside slot data with `SD.Save.Slot(...)`. It can be read back with `SD.Load.SlotMeta(...)`.

⚠️ Important Differences from Other Data Types

Unlike `SharedData` or `SlotData`, `SlotMetaData` **cannot be used in a workflow where you load an instance, modify its values, and then save it again**.

For example, the following pattern does **not** work as expected:

```
// ❌ Incorrect usage — do not modify and save directly
SD.Load.SlotMeta(slotIndex, out var meta);
meta.value = 100;
SD.Save.Slot(slotIndex); // Will not persist changes correctly
```

This design is based on the principle that *metadata should always be automatically generated from the values of other save data*, rather than being edited and saved independently.

✅ Recommended Implementation

The best practice is to generate a fresh instance of `SlotMetaData` every time you save a slot, and populate its fields appropriately. One reliable approach is to implement `IBeforeSaveCallback` in your slot data class. This allows you to set up `SlotMetaData` fields just before saving, based on the current state of your actual save data:

```
public class MySlotData : IBeforeSaveCallback
{
    public Player player;
    public float playTime;

    public void OnBeforeSave()
    {
        // Populate SlotMetaData right before saving
        SD.SlotMeta = new SlotMetaData
        {
            playerName = player.name,
            playTime   = this.playTime,
            savedAt    = DateTime.Now
        };
    }
}
```

This guarantees that each save operation creates a **fresh and consistent `SlotMetaData` instance** derived from the actual slot data, ensuring reliability and avoiding subtle bugs.

3.4 Managing Temporary States with TempData

`TempData` is for **temporary states not saved to files**. It resets automatically when quitting the game or switching slots.

Use cases include:

- Temporary item acquisition or dialogue progress
- Battle results or score history
- One-time flags that reset after reload

You can also control **when it resets** via `TempDataResetTiming` (e.g., on game start or on slot load). Default reset timing is `OnSharedDataLoad`.

By correctly specifying `TempDataResetTiming`, you can prevent common bugs such as *forgetting to reset and unintentionally carrying over the previous state*. This ensures temporary state management is implemented in a way that is both **safer and cleaner**.

3.5 How to Use Initialization

Initialization functions are intended for cases such as **when no save data exists** or when starting a **new game**. They explicitly create a new instance of the save data in memory.

- `SD.Initialize.Shared()` → Creates a fresh shared data instance
- `SD.Initialize.Slot()` → Creates a fresh slot data instance

These two functions provide the basic means to reset save data at runtime.

⚠️ Important Note

When you call an initialization function, the **current in-memory instance** of the data class is discarded and replaced with a new one. However, **this does not delete the save data file itself**. This means you can reinitialize the runtime state without destroying the existing files on disk.

✅ Custom Logic After Initialization

If you need to run additional setup immediately after initialization, you can implement the `IAfterInitializeCallback` interface in your data classes.

For example, this can be used to:

- Set default values for player stats
- Configure initial states for tutorials
- Prepare any runtime-only data required right after a new game starts

This makes initialization flexible, ensuring that your save system always starts in a consistent and predictable state.

📖 Further Reference

> *`IAfterInitializeCallback` is only processed for classes marked with either `[SharedData]` or `[SlotData]`.*
>
> *For detailed specifications, please refer to "1.1 IAfterInitializeCallback" in the Script Reference.*

3.6 How to Use Callbacks During Read/Write

Save Design provides two interfaces for hooks around save/load:

- `IAfterLoadCallback` → Called immediately after data is loaded
- `IBeforeSaveCallback` → Called right before data is saved

These let you separate pre/post-processing per data class.

✅ Example: `IAfterLoadCallback`

Useful for:

- Reconstructing runtime-only fields

- Rebuilding caches or dictionaries
- Updating UI or audio after load

✅ Example: `IBeforeSaveCallback`

Useful for:

- Converting runtime state into savable form
- Removing unused data
- Recording timestamps or playtime before save

💡 Why Callbacks Are Useful

- Keep save/load calls **simple and clean**
- Encapsulate complex pre/post logic inside each data class

⚠️ Dependency Considerations

If one data class depends on another (e.g., `Inventory` referencing `Player`), you need to be careful. By default, **the order of callback execution is not guaranteed**, and this can lead to issues such as:

- A class attempting to reference another class that hasn't finished loading yet
- Inconsistent initialization order causing missing or invalid values

To avoid such problems, you should **explicitly declare dependencies** using attributes, ensuring that related data classes are initialized and callbacks are invoked in the correct order.

```
[SharedData(typeof(Profile)), Serializable]
public class GameSettings : IAfterLoadCallback
{
    ...
}
```

> ✅ *Dependent classes must also be of the same data type (e.g., SharedData → SharedData)*
>
> ✅ *For `[TempData]` , `TempDataResetTiming` must also match*

🔄 Rollback Support

In addition to standard callbacks, Save Design provides **dedicated rollback interfaces** to revert external side effects when exceptions occur:

- `IAfterInitializeRollback.OnAfterInitializeRollback`
- `IAfterLoadRollback.OnAfterLoadRollback`
- `IBeforeSaveRollback.OnBeforeSaveRollback`

These rollback methods are always invoked in the **reverse order (LIFO) of the original callbacks**, so that resources are released safely and in the opposite order they were acquired.

- **Initialization and Load:** After rolling back side effects, data classes are reverted to their previous instances, restoring them to a known-good state.

- **Save:** Since save operations do not create new instances, the data itself remains unchanged, and only side effects are rolled back.

⚠️ Important Notes on Rollback

- Rollback is performed on a **best-effort basis**; not all side effects can be guaranteed to revert completely.
- Rollback implementations should be **idempotent and exception-safe**—throwing exceptions inside rollback can make recovery unstable.
- If no side effects are present, you don’t need to implement rollback interfaces; normal behavior remains unchanged.

---

4. Common Use Cases

4.1 How to Handle Changes in Data Structures

As game development progresses, it is common to **add or remove fields** in save data classes. Before release, such changes don’t cause issues, but after release you must take care to maintain compatibility.

- With `JsonUtility` or `MessagePack`, **missing fields in loaded save data are filled with default values**.
- Likewise, extra fields from older versions are ignored when loading.

For more complex changes (renaming fields, changing types, etc.), we recommend strategies like:

- Adding a `Version` field and branching logic after loading
- Using `IAfterLoadCallback` to convert old structures into the new format
- Discarding old data and regenerating it using `Initialize`

This ensures save data remains stable even when your data model evolves.

The following is an example of changing the type of some fields.

```
[SlotData, Serializable]
public class ExampleData : IAfterLoadCallback
{
    public int version;    // Save data version
    public int oldField;   // Must remain unchanged to prevent data loading failures if the old field is renamed or deleted
    public float newField; // New field with changed type

    void IAfterLoadCallback.OnAfterLoad()
    {
        // Migration processing when loading old data
        if (version == 0)
        {
            newField = (float)oldField;
            version = 1;
        }

        // Further version change handling
        if (version == 1)
        {
            ...
        }
    }
}
```

4.2 Handling Exceptions with ExceptionPolicy

When an exception occurs during **initialization**, **load**, or **save**, Save Design handles it according to the configured `ExceptionPolicy`.

Available Policies

- `Throw`
  Rethrows the exception as-is. Choose this when you want the caller to `try/catch` and control error handling explicitly (e.g., during development or for critical operations).

- `LogAndSuppress`
  Logs the exception via `UnityEngine.Debug.LogException` and then suppresses it. This most closely mirrors the previous `OnGameDataError` behavior.

- `Suppress`
  Silently swallows the exception with no log output. Use in cases where you do not want to surface errors to the user.

Default Behavior

By default, initialization and read/write functions follow the `ExceptionPolicy` provided by **`ISaveDesignConfig`**. Setting it once applies a consistent policy across your project.

Handling Special Cases

Each function also provides an **overload that accepts `ExceptionPolicy`** as a parameter, allowing you to override the default per call when needed. For example:

```
// Force throwing just for this call
SD.Load.Shared(ExceptionPolicy.Throw);

// Log & suppress only for this save
SD.Save.Slot(slotIndex, ExceptionPolicy.LogAndSuppress);
```

Behavior Common to All Policies

- Regardless of which policy you choose, Save Design performs **best-effort rollback** after an exception.
- **Initialization / Load:** Roll back side effects in reverse order (LIFO), then revert data classes to their previous instances (restore to a known-good state).
- **Save:** Since saving does not create new data instances, only **side effects are rolled back**; the data itself remains as-is.
- Rollback cannot be guaranteed to fully restore every external side effect. Implement rollback to be **idempotent** and avoid throwing further exceptions.

Migration Guide

The previous mechanism **`partial void OnGameError(Exception ex)`** has been removed. To emulate similar behavior, set `ExceptionPolicy.LogAndSuppress`. Choose `Suppress` or `Throw` where appropriate, and **test your exception paths** to verify the resulting UX and logs.

4.3 Initializing Only Part of the Data

`Initialize` will initialize all data of the specified type.

If you only want to initialize some data, you can use the following method:

```
[SharedData(”Settings”), Serializable]
public class Audio : IAfterInitializeCallback
{
    public event System.Action<float> OnVolumeChanged;
    [SerializedField] float volume;

    public void SetVolume(float volume)
    {
        this.volume = volume;
        OnVolumeChanged?.Invoke(volume);
    }

    public void Initialize()
    {
        SetVolume(1f);
    }

    void IAfterInitializeCallback.OnAfterInitialize()
    {
        Initialize();
    }
}

SD.Shared.Settings.Audio.Initialize();
```

By encapsulating reset logic within the data class itself, you also keep consistency across your project and make it easier to maintain.

---

5. Security and Encryption (Optional)

5.1 How to Enable Encryption (Editor Tool)

Save Design provides a dedicated **editor extension** that makes it very easy to add encryption functionality to your save system. You don't need to write encryption code yourself—just follow the setup steps below in the Unity Editor.

🔹 Steps to Enable Encryption

1. From the Unity menu bar, open **Tools** > **Save Design** > **Encrypt Settings**.
2. In the displayed window, enter an AES key (32 characters) and an HMAC key (32 characters). ※ Secure random strings are recommended for both.
3. Click the [**Generate Encryptor.cs**] button to automatically generate an encryption script named `Encryptor.cs` .

This process only takes a few clicks, and once enabled, all subsequent save/load operations will transparently use encryption.

🔒 What Is Actually Integrated

The encryption functionality added through this editor tool uses:

- **AES (Advanced Encryption Standard)** → to protect confidentiality of the save data
- **HMAC (Hash-based Message Authentication Code)** → to verify data integrity and detect tampering

Together, AES + HMAC ensures that save files are both **confidential** and **protected from unauthorized modification**. This makes it far more difficult for players or third parties to directly edit or tamper with save files.

5.2 How to Implement Custom Encryption

For finer control or when using specialized encryption methods, Save Design also allows you to replace the encryption logic with your own implementation.

Simply implement the `Encrypt` and `Decrypt` functions in a class with the `Encryptor` attribute to incorporate custom processing.

```
using SaveDesign.Runtime;

[Encryptor]
public static class CustomEncryptor
{
    public static void Encrypt(ref byte[] data)
    {
        ...
    }

    public static void Decrypt(ref byte[] data)
    {
        ...
    }
}
```

These methods are hook points called **before saving and after loading**, respectively.

By applying any encryption or decryption processing to `data` here, you can accommodate custom security policies.

📄 Licenses and Third-Party Acknowledgements

Save Design optionally integrates with third-party libraries. These are distributed under the MIT license.

■ Optional Third-Party Libraries

Library	License	Purpose
MessagePack for C#	MIT	High-speed binary serializer
Newtonsoft.Json	MIT	High-performance JSON serializer (optional)
MemoryPack	MIT	High-speed binary serializer

These libraries are **not included** in Save Design by default. They are **optional**, and users must manually install and manage them.

Whether or not to include them in your build is up to you as the developer.

If you choose to integrate them, please ensure that you comply with their licenses and provide the required license text to end users where necessary.

■ License for Save Design

This asset “Save Design” is distributed under the standard **Unity Asset Store EULA**, which allows commercial use of all code and assets provided.

For details, please refer to the Unity Asset Store End User License Agreement.