Save Design User's Guide - Table of Contents

1. Introduction

1.1 Thank You & Script Reference

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

This document explains the **basic usage** and setup process of Save Design. For more technical information such as detailed specifications of each API, attribute usage, and interface definitions, please refer to the separately provided **"Script Reference"**.

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

1.2 What is Save Design? (Features and Architecture Overview)

Save Design is a save data management framework designed for Unity. Its most notable feature is the ability to **access save data without using any key strings**.

Developers explicitly define save-target classes using attributes like `[SharedData]` and `[SlotData]`. Save Design then **auto-generates static APIs** based on those classes, allowing safe and intuitive save/load/initialize operations with IDE auto-completion.

🔧 Core Components of Save Design

Component	Description
Data definition via attributes	Classify save targets using `[SharedData]`, `[SlotData]`, `[TempData]`, etc.
Automatic generation of static APIs	Functions like `SD.Save.Slot(...)` and `SD.Load.Shared()` are generated automatically
Strong auto-completion and rename safety	All operations are type-safe, preventing typos and update omissions
Easy to integrate	Simply add attributes – minimal setup and initialization required
Optional support for encryption and async processing	AES + HMAC encryption and `async/await`-based APIs can be enabled if needed

---

🎯 Goals of Save Design

- Maintain a **well-organized save structure** even with complex data
- Make save logic **type-safe and easy to manage**
- Flexibly handle **practical save requirements**, such as multiple slots and temporary session data

For developers with these goals, Save Design is a **reliable tool that eliminates the hassle of save systems**.

---

2. Minimum Setup Steps

2.1 Prerequisites (Unity Version / Installation)

✅ Supported Unity Versions

- **Unity 2022.3 LTS or later** is recommended
- Your version must support APIs equivalent to .NET Standard 2.1

📦 Importing the Package

Save Design is distributed as a Unity package. After importing it, the following structure will appear under `Assets/Plugins/Save Design/`:

```
Assets/Plugins/Save Design/
├── Runtime/        // Source code required for Save Design to work
├── Editor/         // Editor extensions (setup, encryption settings, code generation)
├── Samples/        // Usage examples and sample scenes
└── Documentation/  // Script reference and usage guide (this file)
```

🔧 Easy Setup from the Editor

Save Design comes with a **dedicated editor tool** to simplify initial setup. You can complete script scaffolding and configuration asset creation in just a few clicks.

🔹 Auto-Generate Scripts

1. Open the Save Design Setup window from the menu: **Tools** > **Save Design** > **Setup**
2. Enter the `Namespace` and `Output Path`, then click the **Generate scripts** button.
3. The following scripts will be auto-generated at the specified path:
    - An SD root class with the `SaveDesignRoot` attribute
    - A `SaveDesignConfig` class implementing `ISaveDesignConfig`

🔹 Create the Config Asset

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

This will generate a `SaveDesignConfig` asset under the `Assets/Resources` folder. With this asset, you can configure various **file-related settings**:

- Save folder name
- File names for shared and slot data
- File extension

Once configuration is done, you can immediately start using Save Design just by defining data classes with `[SharedData]`, `[SlotData]`, etc.

You only need to perform this setup once per project. Afterward, you can implement safe and clean save logic using auto-generated APIs with IDE auto-completion.

⚠️ Remove Unused Samples

Save Design includes simple **sample classes** and **sample scenes** for testing and learning after import. However, they are not needed in actual game projects, and if left as-is, **they may be included in your production build**.

To avoid this, keep only the components you need and delete the rest.

🧩 Optional Libraries

UniTask (Async Support)

- If you want async support, install UniTask
- Define the `SAVE_DESIGN_SUPPORT_UNITASK` scripting symbol to auto-generate UniTask-based async APIs

MessagePack for C# (High-Speed Serializer)

- If needed, install MessagePack for C#
- Add `[SaveDesignRoot(SerializerType.MessagePack)]` to the root class

> These libraries are **not included** with Save Design. Please install and use them as needed, following their respective licenses.

2.2 Creating Data Definition Classes

In Save Design, you define the data you want to save as classes, and categorize them by attaching attributes like `[SharedData]` or `[SlotData]`.

This "data definition" step allows Save Design to automatically generate appropriate save APIs.

🔹 Types of Data and Their Usage

Attribute Name	Use Case
`[SharedData]`	For global settings or progress shared across slots
`[SlotData]`	For per-slot saves (character info, items, etc.)
`[TempData]`	For session-only values (e.g., temporary flags)
`[SlotMetaData]`	For metadata shown in save slot UI (timestamp, playtime)

> For attribute usage details and structure, see section “2. Attributes” in the Script Reference.

🧩 Example: Slot-Based Player Data

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

Once this class is defined, you can read and write the data using the auto-generated API like this:

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

var level = SD.Slot.Player.level; // Read value

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

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

> 💡 **Note: Which fields are saved depends on the serializer used.**
>
> Save Design delegates actual data serialization/deserialization to external libraries such as JsonUtility or MessagePack for C#. Therefore, which fields are included in save/load depends on the behavior of the selected serializer.
>
> For example, when using `UnityEngine.JsonUtility`:
>
> - `public` fields are saved
> - `private` fields with `[SerializeField]` are saved
> - Properties and fields marked with `[NonSerialized]` are ignored
>
> When using `MessagePack for C#`, you must use `[MessagePackObject]` and `[Key(n)]`.

> Please review the specifications of the serializer you plan to use to ensure all intended data is correctly saved and restored.

3. Practical Usage Examples

3.1 Reading & Writing Shared Data

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

✅ Recommended Timing & Method for Reading

Shared data is generally loaded **once at game startup** and treated as persistent in memory afterward. We recommend implementing this loading process in a static initializer function inside the root class, like so:

```
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
static void InitSaveDesignConfig()
{
    // Load the file-related configuration first
    config = Resources.Load<SaveDesignConfig>("SaveDesignConfig");

    // Load shared data, or initialize if no data exists
    if (!Load.Shared()) Initialize.Shared();
}
```

This ensures that **shared data is ready before any scenes load**, so that any following logic can safely access `SD.Shared.xxx`.

💾 Save Timing and Design Guidelines

There are two main options for when to save shared data:

1. Save **immediately after changes** (recommended)
2. Save **once upon application exit**

While both are valid, **the first option is preferred for clarity and reliability.**

✅ Option 1: Save Immediately After Value Change (Recommended)

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

This method is safest, ensuring that **recent changes are preserved** even if the application is terminated unexpectedly. It's especially suited for data like settings or progress that must persist.

⚠️ **Option 2: Save on Application Exit (Backup Strategy)**

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

This is simpler and good for lightweight projects or infrequently changed settings. However, note that `Application.quitting` is only triggered if Unity shuts down normally — **it won't run on task manager termination or mobile OS force-closes.**

3.2 Slot Saves and Auto Save

Save Design allows you to flexibly manage **slot-based save data**, such as “Slot 1–3” setups using numeric indices. In addition to numbers, it also supports saving using **string-based identifiers**.

For example, a player might select a save slot from the title screen using a numeric index, while automatic checkpoints or auto-saves during gameplay can use identifiers like `"autosave"` or `"checkpoint-3"`.

However, Save Design **does not save data automatically**. This is intentional—to prevent unexpected bugs or hard-to-debug issues caused by saving behind the scenes.

Instead, Save Design follows a clear rule: **"Save only when explicitly instructed to."** It never performs automatic I/O or saves data without your awareness, which ensures that developers retain full control and the game state remains consistent.

This makes it easy to manage when and what data is saved, ultimately improving reliability and debuggability.

3.3 Using SlotMetaData + Cautions

`SlotMetaData` is designed to store **slot metadata**, which is useful for displaying in slot selection UIs. It’s ideal for storing information like the player’s name, playtime, or save timestamp.

If your game doesn’t use a slot system, defining `SlotMetaData` is optional.

✅ Common Uses of SlotMetaData

- Displaying "player name", "current chapter", or "playtime" on the slot select screen

- Storing the last saved timestamp
- Flag to indicate whether a slot is empty

The data is saved automatically alongside the slot when calling `SD.Save.Slot(...)`, and loaded via `SD.Load.SlotMeta(...)`.

⚠️ Important: SlotMetaData Is Always Overwritten with a New Instance

Unlike `[SharedData]` or `[SlotData]`, SlotMetaData does **not support modifying an existing instance** and saving it. Each save must generate a **new instance**, and **all fields must be explicitly assigned** before saving.

Example of what **does not work**:

```
if (SD.Load.SlotMeta(slotIndex, out var meta))
{
    meta.value += 100; // ❌ This modification won't be saved
}
```

This design is intentional and encourages developers to **derive metadata values from game data** just before saving.

Use `IBeforeSaveCallback` to populate fields just in time:

```
[SlotMetaData, System.Serializable]
public class SlotMetaData : IBeforeSaveCallback
{
    public string playerName;
    public int level;
    public int money;

    void IBeforeSaveCallback.OnBeforeSave()
    {
        var player = SD.Slot.Player;
        playerName = player.name;
        level = player.level;
        money = player.money;
    }
}
```

3.4 Managing Temporary State with TempData

`TempData` is used to handle **temporary values that are not saved to file**. It is automatically reset when the game exits or a slot is switched, making it ideal for storing session-only flags or volatile data.

Common use cases include:

- Tutorial flags valid only for the current session
- Temporary inventory or dialogue progress
- Scores or battle results during a single match
- Flags for one-time events triggered after loading

To define TempData, just mark a class with `[TempData]`. You can also specify a `TempDataResetTiming` to control **when it gets reset**, such as:

- `OnGameStart` (when the game starts)
- `OnSlotDataLoad` (when switching slots)

This feature helps prevent bugs from uninitialized or leftover temporary states.

The default reset timing is `OnSharedDataLoad`.

3.5 Leveraging Initialize Processing

When starting a new game or creating a new save slot, you can explicitly initialize data using `SD.Initialize.Shared()` or `SD.Initialize.Slot()`.

Each of these methods creates a new instance of the data.

Use these during:

- "New Game" flow
- Creating a new slot from the title screen
- Developer tests that require a clean state

⚠️ Do not call initialize if data already exists unless intentional, as it will overwrite existing saves.

To perform custom logic after data creation, implement `IAfterInitializeCallback`. Its `OnAfterInitialize()` method is called **once immediately after the data is initialized** (not during load).

Good use cases include:

- Setting default values for runtime-only fields
- Deriving initial state from other data
- Triggering tutorial setup or setting a starting stage

This separates new-game logic cleanly from general load logic.

> Note: `IAfterInitializeCallback` only applies to `[SharedData]` and `[SlotData]` classes. See section "1.2 IAfterInitializeCallback" in the script reference for details.

3.6 Using Callbacks During Read/Write

Save Design provides two interfaces for hooking into the save/load process:

- `IAfterLoadCallback`
- `IBeforeSaveCallback`

These let you define **pre/post-processing logic within each data class**, making your code modular and easier to manage.

✅ `IAfterLoadCallback` Usage

Called automatically after a data class is deserialized from disk.

Useful for:

- Reinitializing runtime-only fields
- Rebuilding caches or dictionaries
- Triggering visual/UI updates after load

✅ `IBeforeSaveCallback` Usage

Called just before the data is serialized to disk.

Useful for:

- Transforming data into a serializable format
- Removing circular references
- Updating timestamps or playtime info

These callbacks help maintain data accuracy and modularity.

💡 Why Callbacks Help

Instead of placing logic directly in the calling code, these callbacks **encapsulate logic inside the data class**.

This keeps your save/load function calls clean (e.g., just `SD.Save.Slot(...)`) while offloading complex behavior to where it belongs — in the data model.

🔔 Caution When Depending on Other Data

In Save Design, each data type (`SharedData`, `SlotData`, `TempData`, etc.) is defined as a separate class. By default, **the order in which data is initialized, loaded, or saved is not guaranteed**.

This becomes critical when a callback method (such as `IAfterLoadCallback`, `IBeforeSaveCallback`, or `IAfterInitializeCallback`) **accesses another data class** during execution.

❌ What Can Go Wrong Without Ordered Execution

For example, let's say you want to initialize a value in your `GameSettings` (SharedData) by referencing a value from `Profile` (SharedData):

```
[SharedData]
public class GameSettings : IAfterLoadCallback
{
    public int volume;

    public void OnAfterLoad()
    {
        volume = SD.Shared.Profile.defaultVolume; // ❌ May throw NullReferenceException
    }
}
```

If `Shared.Profile` hasn't been loaded yet, this code may result in a `NullReferenceException` or initialize with an incorrect value.

✅ Save Design's Solution: Declaring Dependencies

To resolve this, Save Design lets you **declare dependencies explicitly** by passing the type of the dependent data to the attribute:

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

With this declaration, Save Design guarantees that `GameSettings` will be initialized or loaded **after** `Profile`, ensuring it is safe to reference.

> ✅ The dependent type must be of the **same data kind** (e.g., `SharedData` can only depend on other `SharedData`)
> ✅ In the case of `TempData`, the `ResetTiming` must also match

This mechanism ensures **safe access to other data** inside callbacks and helps you build reliable save systems even in projects with complex dependencies.

3.7 Async Processing (UniTask / await Support)

Save Design can auto-generate **async variants** of its load/save APIs. This is ideal when save/load operations take time (e.g., large files or mobile platforms).

These methods are compatible with `async/await` and execute without blocking the main thread.

✅ Using Async Namespaces

Async APIs live under `.Async`, e.g., `SD.Load.Async.Shared()`.

Example:

```
if (await SD.Load.Async.Shared())
{
    // Data loaded successfully
}
```

This keeps the UI responsive and avoids frame hitches during I/O operations.

⚠️ Warning: Unity APIs Not Allowed During Async

Async operations are run **on a background thread**. Therefore, you must not use Unity-specific APIs (e.g., `Instantiate`, `Find`) inside:

- Constructors
- Field initializers
- `OnAfterLoad()`
- `OnBeforeSave()`

Split Unity-related code into separate logic on the main thread if needed.

---

4. Common Use-Cases

4.1 How to Handle Data Structure Changes

As your game development progresses, it's common to **add or remove fields** from save-target data classes. Before release, such changes pose no problem. However, for a game already released to users, extra caution is necessary.

When using `JsonUtility` or `MessagePack`, **any fields missing from the save file will be filled with default values** during loading. Conversely, if older save files contain now-unused fields, they will simply be ignored.

If you need to make more complex changes — such as renaming fields or changing data types — consider the following strategies:

- Add a `Version` field to your data, and branch your logic based on version during load
- Use `IAfterLoadCallback` to transform older structures into the new format
- In some cases, discard the old data entirely and regenerate a new structure via `Initialize`

Save Design allows you to implement these custom strategies freely. It's important to ensure your save system **can evolve safely** as your project grows.

🔁 Example: Changing Field Types with Version Migration

```
[SlotData, Serializable]
public class ExampleData : IAfterLoadCallback
{
    public int version;    // Version number of the save data
    public int oldField;   // Keep this to read old data; don't remove it yet
    public float newField; // New version using float

    void IAfterLoadCallback.OnAfterLoad()
    {
        // Migration logic for version 0
        if (version == 0)
        {
            newField = (float)oldField;
            version = 1;
        }

        // Further version upgrades if necessary
        if (version == 1)
        {
            ...
        }
    }
}
```

4.2 How to Check Errors During Read/Write

By default, Save Design is built on the principle of **not throwing exceptions**. If an operation fails (e.g., due to corrupted data or I/O error), the method simply returns `false`.

This approach ensures the game won't crash, even if something goes wrong during save/load, which is especially useful in production environments.

If you'd like to track and handle these errors, implement the generated partial method `OnGameDataError(System.Exception e)` in your root class.

```
[SaveDesignRoot]
internal partial class SD
{
    static partial void OnGameDataError(Exception e)
    {
        Debug.LogException(e);
    }
}
```

4.3 Initializing Only Part of the Data

Normally, calling `Initialize` will **fully reset** all data of the specified type. But in some situations, you may want to **initialize just a subset of the data** manually.

Although this is a bit unconventional, the following approach works:

```
[SharedData("Settings"), Serializable]
public class Audio : IAfterInitializeCallback
{
    public float volume;

    public void OnAfterInitialize()
    {
        volume = 1f;
    }
}

// Manually invoke the initialization callback
SD.Shared.Settings.Audio.OnAfterInitialize();
```

This lets you manually reset values without wiping the entire data class.

---

5. Security and Encryption (Optional)

5.1 Enabling Encryption (Editor Tool)

Save Design provides a built-in editor tool that lets you configure encryption directly within the Unity Editor.

Follow these steps to enable encryption:

1. From the Unity top menu bar, go to **Tools** > **Save Design** > **Encrypt Settings**
2. In the opened window, enter your AES key (32 characters) and HMAC key (32 characters). ※ We strongly recommend using secure, random strings for both.
3. Click the **[Generate Encryptor.cs]** button. This will automatically generate a script named `Encryptor.cs`.

This script hooks into Save Design's runtime processing and enables the following:

- **All saved data is encrypted** after this point
- **HMAC-based tamper detection** is automatically performed during load

---

5.2 Implementing Custom Encryption Logic

If you need more advanced control — such as integrating a different encryption scheme — Save Design also supports **custom encryption logic**.

You can incorporate your own processing by simply implementing the `Encrypt` and `Decrypt` functions in a class with the `Encryptor` attribute.

```
using SaveDesign.Runtime;

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

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

These hooks are called:

- **Immediately before saving**, and

- **Immediately after loading**

This gives you full control over the `data` byte array, so you can adapt Save Design to your desired security policy.

Example use cases:

- Integrate special encryption used in proprietary or enterprise titles
- Use randomized IVs or salt values to strengthen encryption per save
- Apply HMAC validation across multiple files

Save Design provides both **a simple encryption solution** and **the flexibility to support advanced scenarios**.

We recommend using encryption if the integrity and confidentiality of saved data is important in your game.

⚠️ Warning: Apply Encryption Before Public Release

Encrypted data cannot be read without the encryption logic.

Likewise, encrypted save files will fail to load unless decryption logic is present.

This means: **If you release your game without encryption, and add encryption later, older save files will be unreadable.** Or vice versa.

To avoid this, always apply encryption **before** your game's public release, and **do not change the encryption logic** after shipping.

📄 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
UniTask	MIT	Enables async operations (Async API)
MessagePack for C#	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.