SaveSmart Easy

A lightweight, schema-first save manager for Unity that stores game data as JSON (with optional basic encryption). **Schema-first save system** — define your schema in the editor, generate typed C# accessors, then read and write with simple APIs. **No string keys in gameplay code.** No boilerplate, no manual serialization.

Best for: Indie games, VR games, mobile games; single-player or local save; settings, progress, unlocks.

Not ideal for: Massive live-service, per-frame telemetry, server-authoritative MMO.

Overview

- **Best for:** Settings, level progress, and player state — anything you need to persist between sessions.
- **Caching & performance:** All reads and writes use an **in-memory JSON cache** per slot. After the first load, subsequent `Get` / `SaveData.XXX` reads return instantly from memory — no disk I/O. Each write updates both the cache and disk, so follow-up reads remain instant. **Fast enough for typical gameplay usage, with no runtime allocation concerns after initial load.**
- **Avoid:** Writing every frame or inside tight loops; each write hits disk.
- **Paths:** Current slot is stored in **PlayerPrefs** (key: `SaveSmartEasy_CurrentSlot`, default 1). Per-slot data: `SaveData1.json` … `SaveDataN.json` under `Application.persistentDataPath`. Slot index is **1-based** and must be **≥ 1**.
- **Schema:** Fields are defined in **SaveSmartEasyConfig**. The config asset must be in a folder named **Resources** (e.g. `Assets/Resources/SaveSmartEasyConfig.asset`) so Unity can load it. On first import, it may be auto-created there. Edit via **Tools → SaveSmart Easy → Edit SaveData Schema**, then **Save and Generate C#** to regenerate `SaveDataAccessor.generated.cs`.

- **Undo:** Full Undo supported in the Schema Editor (add/remove/reorder/edit type, name, comment).
- **Save File Editor:** It includes a built-in Save File Editor (Play Mode Supported) that lets you inspect and modify save data directly as raw JSON.
- **Autocomplete in code:** After defining variables in the schema, you get IntelliSense (autocomplete) when writing code, no need to worry about typos.
- **Read:** Use `SaveSmartEasy.SaveSmart.SaveData.XXX` (or `SaveSmart.SaveData.XXX` with `using SaveSmartEasy;`) or the key-based getters below.
- **Write:** Assign to `SaveSmart.SaveData.XXX = value` or use the key-based setters. **Each assignment writes to disk immediately (For best practice, avoid writing every frame or inside tight loops.)**.
- **Encryption:** Optional. Set `UseEncryption = true` and `EncryptionKey` for basic protection (e.g. deter casual save editing). Not intended for high-security use; see **Optional encryption** below.

Why Resources: The config must sit in a Resources folder so Unity can load it at runtime:

- Used only to load the Config — nothing else.
- No Addressables required.
- Does not cause runtime memory leaks.
- Loaded once and cached.
- The config is a **ScriptableObject** — stored as a `.asset` file and persisted with your project; schema data will not be lost when you close the editor.

Schema Change Notes: Rename and Delete both remove API access to the old key. The old value may still sit in `SaveDataN.json` until the file is cleared or overwritten.

- **Rename** = new key in schema; existing data stays under the old key (invisible to the API).
- **Delete** = field removed from schema; same as above — you can't read or write it, value may remain in the file.

Quick usage

Typed properties (after schema + generate):

```
using UnityEngine;
using System.Collections.Generic; // Required when using List<T> (e.g. List<Vector3>)
using SaveSmartEasy;

public class SaveSmartEasyDemo : MonoBehaviour
{
    void Start()
    {
        // Slots (if you only use one save slot, you can skip this section)
        SaveSmart.SetCurrentSlot(1);   // switch active slot (1-based); optional – default is 1; slot index must be ≥ 1
        SaveSmart.ClearSlot(2);       // delete SaveData2.json
        int slot = SaveSmart.GetCurrentSlot();

        // SaveDataAccessor (typed properties from schema)
        // The code below is demo usage. You must edit the schema (Tools → SaveSmart Easy → Edit SaveData Schema)
        // to add your own variable names, then click "Save and Generate C#" to get typed properties like SaveSmart.SaveData.YourVariableName.
        // Int
        SaveSmart.SaveData.CurrentStage = 5;
        int stage = SaveSmart.SaveData.CurrentStage;

        // String
        SaveSmart.SaveData.Username = "player1";
        string name = SaveSmart.SaveData.Username;

        // Float, Bool, Vector3, Color
        SaveSmart.SaveData.SomeFloatField = 3.14f;
        SaveSmart.SaveData.SomeBoolField = true;
```

```
        SaveSmart.SaveData.LastCheckpoint = new Vector3(1f, 2f, 3f);
        SaveSmart.SaveData.ThemeColor = Color.red;

        // DateTime
        System.DateTime dateTime = SaveSmart.SaveData.LastLoginTime;
        dateTime = System.DateTime.UtcNow;
        SaveSmart.SaveData.LastLoginTime = dateTime;
        // OR
        SaveSmart.SaveData.SkillCooldownEndTime =
System.DateTime.UtcNow.AddDays(1);

        // List<T> (define field as IntList, FloatList, StringList,
Vector3List, etc. in schema)
        // New one-liner - Add/Remove/Clear sync to disk automatically:
        SaveSmart.SaveData.UnlockedStage.Add("Stage1");
        SaveSmart.SaveData.ScoreList.Add(100);
        // Old pattern - List copy, modify, assign back - also
supported:
        List<int> scoreList = SaveSmart.SaveData.ScoreList;
        scoreList.Add(2);
        SaveSmart.SaveData.ScoreList = scoreList;
        // Replace entire list: SaveSmart.SaveData.UnlockedStage = new
List<string> { "Stage1", "Stage2" };

        // Arrays
        SaveSmart.SaveData.LevelScores = new int[] { 1, 2, 3 };
        int[] scores = SaveSmart.SaveData.LevelScores;
        SaveSmart.SaveData.PlayerNames = new string[] { "Alice", "Bob",
"Carol" };
        string[] names = SaveSmart.SaveData.PlayerNames;

        // Vector3 array (e.g. checkpoints, waypoints)
        SaveSmart.SaveData.Checkpoints = new Vector3[] { new Vector3(1f,
0f, 0f), new Vector3(2f, 0f, 0f) };
        Vector3[] points = SaveSmart.SaveData.Checkpoints;
```

```
        // From List<Vector3> to Vector3[]
        List<Vector3> waypoints = new List<Vector3> { Vector3.zero,
Vector3.one };
        SaveSmart.SaveData.Checkpoints = waypoints.ToArray();

        // Key-based get/set (key = variable name in Config)
        // You can also use this approach, but be careful of typos in
the key.
        SaveSmart.SetInt("CurrentStage", 5);
        // Dynamic key: build key string at runtime (e.g. "Stage1",
"Stage2"). Each key must exist in Config with correct type.
        int stageIndex = 1;
        int stageValue = SaveSmart.GetInt("Stage" + stageIndex, 1);
        SaveSmart.SetInt("Stage" + stageIndex, 3);
        stage = SaveSmart.GetInt("CurrentStage", 1);
        SaveSmart.SetString("Username", "player1");
        string username = SaveSmart.GetString("Username", "");
        SaveSmart.SetFloat("SomeFloatField", 3.14f);
        float value = SaveSmart.GetFloat("SomeFloatField", 0f);
        SaveSmart.SetBool("SomeBoolField", true);
        bool flag = SaveSmart.GetBool("SomeBoolField", false);
        SaveSmart.SetVector3("LastCheckpoint", new Vector3(1f, 2f, 3f));
        Vector3 pos = SaveSmart.GetVector3("LastCheckpoint",
Vector3.zero);
        SaveSmart.SetVector2("LastTouchPos", new Vector2(10f, 20f));
        Vector2 pos2 = SaveSmart.GetVector2("LastTouchPos",
Vector2.zero);
        SaveSmart.SetLong("TotalCoins", 999999L);
        long coins = SaveSmart.GetLong("TotalCoins", 0L);
        SaveSmart.SetDouble("HighScoreTime", 99.5);
        double time = SaveSmart.GetDouble("HighScoreTime", 0.0);
        SaveSmart.SetColor("ThemeColor", Color.red);
        Color themeColor = SaveSmart.GetColor("ThemeColor",
Color.white);
        SaveSmart.SetIntArray("LevelScores", new int[] { 1, 2, 3 });
        scores = SaveSmart.GetIntArray("LevelScores", null);
```

```
        SaveSmart.SetFloatArray("LevelTimes", new float[] { 10.5f, 20f
});
        float[] times = SaveSmart.GetFloatArray("LevelTimes", null);
        SaveSmart.SetStringArray("UnlockedLevelIds", new string[] {
"level_1", "level_2" });
        string[] ids = SaveSmart.GetStringArray("UnlockedLevelIds",
null);
        SaveSmart.SetVector3Array("Checkpoints", new Vector3[] { new
Vector3(1f, 0f, 0f), new Vector3(2f, 0f, 0f) });
        points = SaveSmart.GetVector3Array("Checkpoints", null);
        waypoints = new List<Vector3> { Vector3.zero, Vector3.one };
        SaveSmart.SetVector3Array("Checkpoints", waypoints.ToArray());

        // Key-based List (returns List<T>; same storage as Array, use
when you prefer List)
        List<int> unlockLevels = SaveSmart.GetIntList("UnlockLevels",
new List<int>());
        unlockLevels.Add(9);
        SaveSmart.SetIntList("UnlockLevels", unlockLevels);
        // Or use typed accessor for one-line Add:
SaveSmart.SaveData.UnlockLevels.Add(9);
        List<Color> colors = SaveSmart.GetColorList("ThemeColors",
null);

    }
}
```

Schema and code generation

1. Open **Tools** → **SaveSmart Easy** → **Edit SaveData Schema**.
2. Add/remove/reorder fields. Each field has: **Type**, **Variable name**, and optional **Comment**. Use **valid C# identifiers** for variable names (no spaces, no leading digits) so the generated properties compile.
3. Click **Save and Generate C#**. This updates the config asset and regenerates

`SaveDataAccessor.generated.cs` in `SaveSmartEasy/Generated/` (the file is overwritten each time).

4. Use `SaveSmart.SaveData.YourVariableName` in code (with `using SaveSmartEasy;`). The key-based APIs (`GetInt`, `SetString`, etc.) use the same variable names — they must match a variable in the config and the correct type, or you get a warning and the call is ignored.

Dynamic key usage: You can build the key string at runtime for key-based API, e.g. `SaveSmart.GetInt("Stage" + stageIndex, 1)` or `SaveSmart.SetInt("Stage" + stageIndex, value)`. Each resulting key (e.g. `"Stage1"`, `"Stage2"`) must be defined in the schema with the correct type; otherwise you get a warning and the get returns the default (set is ignored). For many dynamic keys, consider using an array field (e.g. `int[] LevelScores`) and access by index instead.

Supported types: Int, Long, Float, Double, Bool, String, Vector2, Vector3, Color, Color32, Quaternion, Rect, DateTime, IntArray, FloatArray, StringArray, BoolArray, Vector3Array, Vector2Array, ColorArray, Color32Array, QuaternionArray, RectArray, DateTimeArray, DoubleArray, LongArray. **List variants:** IntList, FloatList, StringList, BoolList, Vector3List, Vector2List, ColorList, Color32List, QuaternionList, RectList, DateTimeList, DoubleList, LongList — these produce `IList<T>` properties backed by `SaveSmartList<T>`. **List note:** You can use one-line Add: `SaveSmart.SaveData.UnlockedStage.Add("Stage1");` — mutations sync to disk automatically. To replace the entire list: `SaveSmart.SaveData.UnlockedStage = new List<string> { "a", "b" };`. Key-based API: use `GetIntList` / `SetIntList`, etc. (see **Key-based get/set (lists)** below).

In-Editor Save File Editor (Play Mode Supported)

Tools → SaveSmart Easy → Edit SaveData File opens an editor window to view and edit the raw JSON for any save slot. This tool works even while the game is running (Play Mode), making it ideal for rapid testing and debugging without restarting the game.

- **Save Slot Selection** — Quickly switch between existing save slots (SaveData1.json, SaveData2.json, etc.).
- **Reload From Disk** — Re-read the file from disk to reflect changes written by the

game at runtime.

- **Direct JSON Editing** — Edit save values directly in a text editor. JSON is automatically formatted and ordered based on your schema for readability.
- **Safe Save & Hot Reload** — JSON is validated before saving. Changes are written to disk and the in-memory cache is updated immediately. The running game sees the updated values instantly.
- **Reveal in File Explorer** — Open the save folder directly in the system file manager for inspection or backup.

Typical use cases:

- Test different save states (e.g. set `CurrentStage = 5`) while the game runs.
- Inspect or fix corrupted or partial JSON.
- Debug cloud sync or encryption by inspecting readable JSON after decryption.

Storing custom objects as JSON

To save your own classes or structs (e.g. settings, inventory), use a **String** field and serialize/deserialize with `JsonUtility` (or another JSON library).

1. **Schema:** In the Save Data Schema editor, add a field with type **String** and a name (e.g. `PlayerSettingsJson`, `InventoryData`).
2. **Save and Generate C#** so the typed property exists.
3. **Write:** Serialize your object to a JSON string, then set it via the typed property (or key-based `SetString`).

```
using SaveSmartEasy;
using UnityEngine;

[System.Serializable]
public class MyPlayerSettings
{
    public float MusicVolume = 1f;
```

```
    public float SfxVolume = 0.8f;
    public bool Fullscreen = true;
}

// Save (typed accessor first)
var settings = new MyPlayerSettings { MusicVolume = 0.5f, SfxVolume = 1f
};
string json = JsonUtility.ToJson(settings);
SaveSmart.SaveData.PlayerSettingsJson = json;
// Or key-based: SaveSmart.SetString("PlayerSettingsJson", json);
```

4. **Read:** Get the string via the typed property, then deserialize with `JsonUtility.FromJson` (or use key-based `GetString`).

```
// Load (typed accessor first)
string json = SaveSmart.SaveData.PlayerSettingsJson;
if (string.IsNullOrEmpty(json)) json = "{}";
MyPlayerSettings settings = JsonUtility.FromJson<MyPlayerSettings>
(json);
Debug.Log("Music volume: " + settings.MusicVolume);
Debug.Log("Sfx volume: " + settings.SfxVolume);
Debug.Log("Fullscreen: " + settings.Fullscreen);
// Or key-based: string json = SaveSmart.GetString("PlayerSettingsJson",
"{}");
```

The entire JSON string is stored as one String value (quotes and special characters are escaped correctly). No need to add a separate "Json" type to the schema — String + your own serialize/deserialize is enough.

API

All APIs are in namespace `SaveSmartEasy`, static class `SaveSmart` (and typed accessor `SaveSmart.SaveData`).

Slot & path

API	Description
`int GetCurrentSlot()`	Current save slot (1-based).
`void SetCurrentSlot(int slotIndex)`	Set current slot (1-based, must be ≥ 1; slotIndex can be any number, e.g. 1, 5, 100). Persisted via PlayerPrefs.
`void ClearSlot(int slotIndex)`	Delete `SaveData{slot}.json` for the given slot.
`void InvalidateSlotCache()` / `InvalidateSlotCache(int slotIndex)`	Clear in-memory cache (no arg = current slot). Next read loads from disk. Use after manually editing the save file.
`string GetSavePath()` / `GetSavePath(int slotIndex)`	Save file path (no arg = current slot; e.g. `.../SaveData1.json`).

Typed accessor (from schema)

API	Description
`SaveDataAccessor SaveData`	Typed read/write for current slot. Use `SaveSmart.SaveData.YourVariableName` (properties generated from schema). Assign to write and auto-save; read returns current value.

Key-based get/set (scalars)

`key` must match a variable name in SaveSmartEasyConfig; type must match or the call is ignored with a warning.

API	Description
`void SetInt(string key, int value)` / `int GetInt(string key, int defaultValue = 0)`	Int.
`void SetLong(string key, long value)` / `long GetLong(string key, long defaultValue = 0L)`	Long.
`void SetFloat(string key, float value)` / `float GetFloat(string key, float defaultValue = 0f)`	Float.
`void SetDouble(string key, double value)` / `double GetDouble(string key, double defaultValue = 0.0)`	Double.
`void SetBool(string key, bool value)` / `bool GetBool(string key, bool defaultValue = false)`	Bool.
`void SetString(string key, string value)` / `string GetString(string key, string defaultValue = "")`	String.
`void SetVector2(string key, Vector2 value)` / `Vector2 GetVector2(string key, Vector2 defaultValue)`	Vector2.
`void SetVector3(string key, Vector3 value)` / `Vector3 GetVector3(string key, Vector3 defaultValue)`	Vector3.
`void SetColor(string key, Color value)` / `Color GetColor(string key, Color defaultValue)`	Color.

Key-based get/set (arrays)

API	Description
`void SetIntArray(string key, int[] value)` / `int[] GetIntArray(string key, int[] defaultValue = null)`	Int array.
`void SetLongArray(string key, long[] value)` / `long[] GetLongArray(string key, long[] defaultValue = null)`	Long array.
`void SetFloatArray(string key, float[] value)` / `float[] GetFloatArray(string key, float[] defaultValue = null)`	Float array.
`void SetDoubleArray(string key, double[] value)` / `double[] GetDoubleArray(string key, double[] defaultValue = null)`	Double array.
`void SetBoolArray(string key, bool[] value)` / `bool[] GetBoolArray(string key, bool[] defaultValue = null)`	Bool array.
`void SetStringArray(string key, string[] value)` / `string[] GetStringArray(string key, string[] defaultValue = null)`	String array.
`void SetVector2Array(string key, Vector2[] value)` / `Vector2[] GetVector2Array(string key, Vector2[] defaultValue = null)`	Vector2 array.
`void SetVector3Array(string key, Vector3[] value)` / `Vector3[] GetVector3Array(string key, Vector3[] defaultValue = null)`	Vector3 array.
`void SetColorArray(string key, Color[] value)` / `Color[] GetColorArray(string key, Color[] defaultValue = null)`	Color array.
`void SetColor32Array(string key, Color32[] value)` / `Color32[] GetColor32Array(string key, Color32[] defaultValue = null)`	Color32 array.
`void SetQuaternionArray(...)` / `Quaternion[] GetQuaternionArray(...)`	Quaternion array.
`void SetRectArray(...)` / `Rect[] GetRectArray(...)`	Rect array.
`void SetDateTimeArray(...)` / `DateTime[] GetDateTimeArray(...)`	DateTime array.

Key-based get/set (lists)

Same storage as arrays; use when you prefer `List<T>`. `key` must match a variable of the corresponding List type in Config.

API	Description
`void SetIntList(string key, List<int> value)` / `List<int> GetIntList(string key, List<int> defaultValue = null)`	Int list.
`void SetFloatList(...)` / `List<float> GetFloatList(...)`	Float list.
`void SetStringList(...)` / `List<string> GetStringList(...)`	String list.
`void SetBoolList(...)` / `List<bool> GetBoolList(...)`	Bool list.
`void SetLongList(...)` / `List<long> GetLongList(...)`	Long list.
`void SetDoubleList(...)` / `List<double> GetDoubleList(...)`	Double list.
`void SetVector3List(...)` / `List<Vector3> GetVector3List(...)`	Vector3 list.
`void SetVector2List(...)` / `List<Vector2> GetVector2List(...)`	Vector2 list.
`void SetColorList(...)` / `List<Color> GetColorList(...)`	Color list.
`void SetColor32List(...)` / `List<Color32> GetColor32List(...)`	Color32 list.
`void SetQuaternionList(...)` / `List<Quaternion> GetQuaternionList(...)`	Quaternion list.
`void SetRectList(...)` / `List<Rect> GetRectList(...)`	Rect list.
`void SetDateTimeList(...)` / `List<DateTime> GetDateTimeList(...)`	DateTime list.

Readable JSON (decrypts when slot was encrypted)

API	Description
`string ReadJson()`	Full JSON string for current slot (decrypted if needed). Returns `"{}"` if missing or error.
`string ReadJson(int slotIndex)`	Full JSON string for the given slot (decrypted if needed). Returns `"{}"` if missing or error.

Raw JSON (no decrypt; for sync/debug)

API	Description
`string ReadRawJson()`	File contents for current slot, no decrypt. Returns `null` if missing or error.
`string ReadRawJson(int slotIndex)`	File contents for the given slot, no decrypt. Returns `null` if missing or error.
`void WriteRawJson(string json)`	Write string to current slot (no encrypt).
`void WriteRawJson(string json, int slotIndex)`	Write string to the given slot (no encrypt).

Encryption (static fields) — basic protection only

API	Description
`bool UseEncryption`	When `true`, save files are encrypted on write and decrypted on read (fallback to plain JSON on failure).
`string EncryptionKey`	Key string for AES (any length, derived to 128-bit). Must match for read/write.

Slots

- The “current” slot is stored in **PlayerPrefs** (key: `SaveSmartEasy_CurrentSlot`, default 1). All normal read/write goes to `save{CurrentSlot}.json`.

- `SetCurrentSlot(slot)` — switch slot (1-based); saves via PlayerPrefs. **Note: slot must be ≥ 1**; slot numbers do not need to be sequential (e.g. 1, 2, 5).
- `GetCurrentSlot()` — returns the current save slot (1-based). Useful for UI display or logic that depends on which slot is active.
- `ClearSlot(slot)` — deletes `save{slot}.json`. Next load for that slot is empty.
- `GetSavePath()` / `GetSavePath(slotIndex)` — useful for debugging or custom tools. To check if save data exists for a slot, use `System.IO.File.Exists(SaveSmart.GetSavePath(slotIndex))`.
- **Manual edit on disk:** If you edit the save file at runtime, call `SaveSmart.InvalidateSlotCache()` (current slot) or `SaveSmart.InvalidateSlotCache(slotIndex)` so the next read loads the updated content from disk.

Optional encryption (basic protection)

Encryption is a **bonus** for basic protection — e.g. to deter casual cheating by editing save files. It is **not intended for high-security use**; treat it as obfuscation, not security-grade.

- Set `SaveSmartEasy.SaveSmart.UseEncryption = true` to encrypt on write and decrypt on read (AES, key derived from `EncryptionKey`).
- Set `SaveSmartEasy.SaveSmart.EncryptionKey` to any string (e.g. in code). Same key must be used for read and write.
- If decryption fails (wrong key or corrupted data), the loader falls back to treating the file as plain JSON.
- Current slot (PlayerPrefs) is not encrypted; only the per-slot save files are.

Raw JSON and readable JSON (advanced)

To get readable JSON for a slot (including when it was saved with encryption):

- `ReadJson()` / `ReadJson(slotIndex)` — loads the slot and decrypts when needed. Returns the full JSON string in readable form. Returns `"{}"` if the file is missing or on

error. Use this when you need the entire save as a JSON string for a given slot.

For cloud sync or custom pipelines (raw file, no decrypt):

- `ReadRawJson()` / `ReadRawJson(slotIndex)` — returns the file contents as-is (no decryption). Returns `null` if missing or on error. **If the slot was saved with encryption enabled, the returned string is encrypted (garbled) raw bytes, not readable JSON.**
- `WriteRawJson(json)` / `WriteRawJson(json, slotIndex)` — writes the string directly (no encryption). The content does not have to be valid JSON; it can be plain JSON or already encrypted/garbled data (e.g. from `ReadRawJson` of an encrypted slot).

Example: cloud backup with your own server

1. Read the slot with `ReadRawJson(slotIndex)` (raw bytes, no decrypt).
2. Upload that string to your server (e.g. via HTTP POST) and store it per user.
3. When the user logs in on another device or re-installs, download the saved string from your server.
4. Write it back with `WriteRawJson(downloadedString, slotIndex)` so the slot is restored exactly (including encryption if it was originally encrypted).

ReadJson vs ReadRawJson vs internal load

	ReadJson(slotIndex)	ReadRawJson	Internal loader (used by Get/Set)
Decrypt	Yes. Returns readable JSON (decrypts when slot was encrypted).	No. File bytes only; if encrypted, you get garbled text.	Yes. If content does not start with `{`, decrypts then uses result.
On missing / error	Returns `"{}"`.	Returns `null`.	Returns `"{}"`.
Use case	Get full readable JSON for a slot (e.g. slot 2).	Debugging, external sync, or when you handle encryption yourself.	Used automatically by `GetInt`, `SetString`, `SaveData.XXX`, etc.

So: use **`ReadJson()`** or **`ReadJson(slotIndex)`** when you want the entire save as readable JSON for a slot (including decryption). Use `ReadRawJson` when you want the raw file with no decryption (e.g. to send to a server or inspect on disk).

Summary

What you need	Where
Unity version	2020.3 or later (LTS recommended)
Edit schema	**Tools** → **SaveSmart Easy** → **Edit SaveData Schema** → Save and Generate C#
Edit save JSON (runtime testing)	**Tools** → **SaveSmart Easy** → **Edit SaveData File** — view/edit JSON while game runs
Full Undo	Schema Editor supports Undo (Ctrl+Z / Cmd+Z) for all field changes
Config asset	Must be in a **Resources** folder (e.g. `Assets/Resources/SaveSmartEasyConfig.asset`). Auto-created on first import if missing.
Generated accessor	`SaveSmartEasy/Generated/SaveDataAccessor.generated.cs`
Read/write in code	`SaveSmartEasy.SaveSmart.SaveData.XXX` or `GetXXX` / `SetXXX` by key
Current slot	PlayerPrefs key `SaveSmartEasy_CurrentSlot`
Files on disk	`SaveData1.json`, … under `Application.persistentDataPath`

For more detail, see the XML remarks on `SaveSmart` and `SaveDataAccessor` in the source, or the in-code examples in `SaveSmartEasy.cs`.