BLACK SCAR STUDIO - BLUEPRINT PLUGIN

OneNode Save System

The complete Blueprint save framework for Unreal Engine.

No structs. No boilerplate. No C++. Just nodes that work.

- Save any value type with a single node
- Save & restore whole Actors, including references
- Slots, tags, categories, checkpoints & auto-save
- zlib compression + real AES-256 encryption
- Versioning, migration, backup & corruption recovery
- Multiplayer per-player / world saves + Cloud sync

Blueprint Guide - Version 2.0

Unreal Engine 4.27 - 5.7 |

This is your entire save system

Most save systems in Unreal need a custom **SaveGame** class, a pile of variables, and glue code to read and write every one of them. OneNode replaces all of that with three nodes you can drop into any Blueprint:

YOUR WHOLE SAVE SYSTEM - 3 NODES

```
// 1. Point at a save file (a 'slot').
Set Active Save Slot   (Slot Name = "Player1")

// 2. Store any value under a name you choose.
Save Int      (Key = "Coins",  Value = 250)
Save Vector   (Key = "Spawn",  Value = PlayerLocation)

// 3. Write it to disk. Done.
Save All (To Disk)
```

> **TIP**
> That is a real, working save. Reloading is the mirror image: **Load All (From Disk)**, then **Load Int ("Coins")**. No SaveGame class, no boilerplate, nothing to wire up in advance.

Everything else in this guide - actors, slots, encryption, multiplayer, cloud - builds on those same three ideas: **pick a slot, store values by name, write to disk.**

Why not just build it yourself?

You can. Here is what that actually costs you, versus what OneNode does out of the box.

Task	Build it yourself	With OneNode
Save an integer	SaveGame class + variable + cast + save-to-slot code	One **Save Int** node
Save a whole actor's transform	Manually copy each property, rebuild on load	One **Save Actor** node
Keep actor references after load	Store IDs, re-find every actor, relink by hand	**Restore Actor References**
Compress + encrypt the file	Write zlib + AES yourself, manage the key	Tick two settings
Auto-save every N seconds	Timers + flush logic + threading	**Enable Auto Save**
Load an old save after a patch	Version checks + hand-written migration	**Register Migration Step**
Per-player saves in multiplayer	Resolve net IDs, per-player files, server auth	**Save Player** (built in)

The build-it-yourself column is where days disappear - and where the hard-to-find bugs live (lost references, corrupted files, saves that break on your next patch). OneNode ships all of it, tested across UE 4.27 through 5.7.

> **NOTE**
> OneNode does not lock you in. Under the hood it is a normal Unreal **SaveGame** written to your project's **Saved/SaveGames** folder. You can walk away any time - your saves are standard files.

What's inside

01

PART 1 - GET STARTED

What you just bought

OneNode Save System is a Blueprint-first save framework. Every feature is a node - you never have to open C++ or write a SaveGame class. Here is the whole thing at a glance.

What it does

- **Saves any value** - ints, floats, strings, vectors, transforms, colors, arrays, your own structs - each with a single node.
- **Saves whole actors** - transform, properties, even references to other actors and attachment parents.
- **Organises saves** - named slots, tags, and categories (Global, Player, World, Checkpoint, Session, Profile).
- **Protects saves** - optional zlib compression and AES-256 encryption, automatic backups, corruption recovery.
- **Handles the hard cases** - async (non-blocking) saves, auto-save, checkpoints, versioning and migration for patched games.
- **Scales up** - per-player and world saves for multiplayer, plus a cloud provider interface (Steam, Dropbox, custom).

Who it's for

Anyone building in Unreal with Blueprint who needs saving that just works - solo devs, small teams, game jams, and shipped commercial titles. If you have ever lost an afternoon to a SaveGame class, this is for you.

> **INFO**
> This guide follows the order you will actually use the plugin: install it, make your first save, then reach for the deeper features as your game needs them. You do not need to read it front to back - jump to the section you need.

02

PART 1 - GET STARTED

Install it

You bought OneNode on the Fab Marketplace. Here is how to get it from your Fab library into a Blueprint graph. It takes about two minutes.

1

Buy it on Fab (done)

Once you have purchased OneNode Save System, it lives in your **Fab Library** permanently, and every future engine-version update is a free re-download. Nothing expires.

2

Install it to your engine

Open the **Epic Games Launcher** → **Fab Library**. Find **OneNode Save System**, click **Install to Engine**, and choose your engine version (4.27 or 5.0 - 5.7). The launcher copies it into **Engine/Plugins/Marketplace** for you.

3

Enable it in your project

Open your project, then **Edit** → **Plugins**. Search **OneNode**, tick **Enabled**, and click **Restart Now** when Unreal asks. That is the only project setup step.

4

Confirm it works

Open any Blueprint graph, right-click, and type **Save Int**. If the node appears under the **OneNode Save System** category, you are ready to save.

Prefer a manual install?

You do not need the launcher. Each purchase includes one zip per engine version. To install into a single project:

MANUAL INSTALL

```
// 1. Unzip the folder that matches your engine, e.g. UE5.3.
// 2. Copy the 'OneNodeSaveSystem' folder into:
YourProject/Plugins/OneNodeSaveSystem
//    (create the 'Plugins' folder if it does not exist)
// 3. Reopen the project. Enable it under Edit > Plugins.
```

> **IMPORTANT**
> Always use the zip that matches your engine version. A UE 5.3 build will not load in 5.7 and vice-versa. The version is in the file name (for example **OneNodeSaveSystem_UE5.3.zip**).

03

PART 1 - GET STARTED

Your first save in 5 minutes

Let's save a value and read it back. Do this in your Player Character or Game Instance - anywhere with a Blueprint graph.

Step 1 - Choose a slot

A **slot** is one save file. Give it a name. Do this once, early (for example on **BeginPlay** or when the player starts a new game).

PICK A SLOT

```
Set Active Save Slot   (Slot Name = "SaveGame01")
```

Step 2 - Save some values

Store anything you like under names (**keys**) you choose, then flush to disk.

WRITE VALUES

```
Save Int      (Key = "Coins",  Value = 250)
Save String   (Key = "Name",   Value = "Aria")
Save Vector   (Key = "Spawn",  Value = GetActorLocation)

// Nothing is on disk until you flush:
Save All (To Disk)
```

Step 3 - Load it back

Next time the game starts, read the file, then pull values out by key.

READ VALUES

```
Set Active Save Slot   (Slot Name = "SaveGame01")
Load All (From Disk)

Coins = Load Int      (Key = "Coins",  Default = 0)
Name  = Load String   (Key = "Name",   Default = "Player")
Spawn = Load Vector    (Key = "Spawn",  Default = 0,0,0)
```

> **TIP**
> Every **Load** node takes a **Default** value. If the key was never saved (a brand-new game, say), you get the default instead of a crash or an empty value. That single habit prevents most save-related bugs.

That is a complete, shippable save loop. Everything from here just adds capability on top of these same nodes.

04

PART 2 - EVERYDAY SAVING

Saving values

OneNode has a dedicated Save node for every common Blueprint type. Each takes a **Key** (the name you'll load it by) and a **Value**. Call **Save All (To Disk)** when you're ready to write.

Type	Node	Type	Node
Integer	`Save Int`	Vector	`Save Vector`
Integer64	`Save Int64`	Rotator	`Save Rotator`
Float	`Save Float`	Transform	`Save Transform`
Boolean	`Save Bool`	Color	`Save Color`
Byte	`Save Byte`	Name	`Save Name`
String	`Save String`	Raw bytes	`Save Raw Bytes`

Arrays

Whole arrays save in one node too - no looping. Supported array nodes: **Save Int Array**, **Save Float Array**, **Save String Array**, **Save Name Array**, **Save Vector Array**, and **Save Transform Array**.

SAVE AN INVENTORY IN ONE NODE

```
Save String Array   (Key = "Inventory",  Value = InventoryItems)
Save Int Array      (Key = "Quantities", Value = ItemCounts)
Save All (To Disk)
```

> **NOTE**
> Keys are just text and are case-sensitive. **"Coins"** and **"coins"** are two different values. Pick a convention and stick to it.

Useful companions

Has Save Key tells you whether a key exists before you read it. **Remove Key** deletes one value. **Get All Keys** returns every key in the slot - handy for debugging. **Get Entry Count** gives you the total.

05

PART 2 - EVERYDAY SAVING

Loading values

Loading mirrors saving. Read the file once with **Load All (From Disk)**, then pull each value with its matching Load node and key.

EVERY LOAD NODE TAKES A DEFAULT

```
Load All (From Disk)

Health = Load Float   (Key = "Health",  Default = 100.0)
Level  = Load Int     (Key = "Level",   Default = 1)
Alive  = Load Bool    (Key = "Alive",   Default = true)
```

The Default is your safety net

If a key is missing - a fresh game, a save from before you added that feature - the Load node returns your Default instead of failing. This is the single most important habit in the whole system: **always give a sensible default.**

> **TIP**
> Want to branch on whether a save even exists? Use **Save Slot Exists** (Slot Name) before loading - show *Continue* only when it returns true, and *New Game* otherwise.

Checking before you read

For finer control, **Has Save Key** (Key) returns true only when that exact key was saved. Combine it with a Branch to run different logic for returning vs first-time players.

FIRST-TIME VS RETURNING PLAYER

```
Load All (From Disk)
Branch on:  Has Save Key ("TutorialDone")
//   True  -> skip the tutorial
//   False -> play the tutorial, then Save Bool("TutorialDone", true)
```

06

PART 2 - EVERYDAY SAVING

Saving structs & raw bytes

Your own data types save too. Two approaches, depending on how tidy you want the save file.

Option A - Save the fields

Break your struct and save each field under a clear key. Verbose, but the save file is human-readable and easy to migrate later.

SAVE A STRUCT BY FIELDS

```
// Break 'PlayerStats' -> Health, Stamina, Level
Save Float ("Stats.Health",  Health)
Save Float ("Stats.Stamina", Stamina)
Save Int   ("Stats.Level",   Level)
```

Option B - Save raw bytes

Serialise any struct to a byte array and store it in one node with **Save Raw Bytes**. Compact and fast - ideal for large or deeply-nested data where you don't need to read individual fields.

SAVE A STRUCT AS BYTES

```
// Convert your struct to bytes (any struct-to-bytes node),
// then store the whole blob under one key:
Save Raw Bytes   (Key = "SaveBlob",  Value = StructBytes)

// Read it back and rebuild the struct:
Bytes = Load Raw Bytes (Key = "SaveBlob")
```

> **NOTE**
> Prefer **Option A** while your data is still changing - named fields are trivial to migrate when you add or rename something (see section 15). Reach for raw bytes once a struct is stable and you care about file size.

07

PART 2 - EVERYDAY SAVING

Save slots & a save menu

A slot is one save file. Multiple slots give you multiple saves - manual saves, quicksaves, autosaves, or a full New Game / Load Game screen.

Switching slots

MULTIPLE SAVES

```
Set Active Save Slot ("Slot_A")   ...   Save All (To Disk)
Set Active Save Slot ("Slot_B")   ...   Save All (To Disk)

// Quick save / load use a reserved auto-named slot:
Quick Save
Quick Load
```

Building a Load Game screen

Get All Slot Names lists every save on disk. **Get All Slot Infos** returns richer cards - display name, timestamp, playtime, level, even a screenshot - so you can build a proper save-select menu without bookkeeping of your own.

POPULATE A SAVE-SELECT MENU

```
Slots = Get All Slot Infos
// For each slot, read: Slot Name, Level Name, Play Time,
// Timestamp, Screenshot -> build one UI card per entry.
```

Managing slots

Full slot housekeeping ships as nodes: **Delete Save Slot**, **Copy Slot**, **Rename Slot**, **Get Slot Info**, and **Save Slot Exists**. No file handling on your side.

Node	What it does
`Get All Slot Names`	Every save slot currently on disk.
`Get All Slot Infos`	Rich metadata cards for a load screen.
`Copy Slot`	Duplicate a save (great for 'branch' saves).
`Rename Slot`	Rename without losing data.
`Delete Save Slot`	Remove a save permanently.

08

PART 3 - GAME OBJECTS

Saving actors

Instead of saving properties one by one, save an entire actor - its transform and its marked variables - in a single node.

Mark what to save

On the actor's variables you want persisted, tick **SaveGame** in the variable's Details panel (Advanced → SaveGame). OneNode saves exactly those. Then:

SAVE & LOAD AN ACTOR

```
Save Actor    (Actor = self, Key = "Chest_01")
Save All (To Disk)

// ...later / next session:
Load All (From Disk)
Load Actor    (Actor = self, Key = "Chest_01")
```

Many actors at once

Save Multiple Actors takes an array. **Save Actors With Tag** saves everything carrying a gameplay tag - one node to persist every door, pickup, or enemy in a level. **Load Actors By Save Tag** brings them all back.

SAVE EVERY TAGGED ACTOR

```
Save Actors With Tag   (Tag = "Persistent")
Save All (To Disk)
// Reload the whole set later:
Load Actors By Save Tag (Tag = "Persistent")
```

> **INFO**
> Use **Has Actor Save Data** (Key) to check whether an actor was ever saved, and **Get Saved Actor Count** for totals. **Get All Actor Keys** lists them - useful when spawning actors back from a save.

09

PART 3 - GAME OBJECTS

Actor references & attachments

The classic save bug: you reload, and every actor-to-actor pointer is null. OneNode fixes this for you.

The problem

If an actor stores a reference to another actor (a weapon pointing at its owner, an enemy at its patrol node), a naive save writes a memory address that means nothing next session. On load, the pointer is null and things break.

The fix - one node

OneNode records references as stable markers at save time. After you've loaded and respawned your actors, call **Restore Actor References** once. It walks every saved actor and reconnects the pointers - including **attachment parents and sockets**, so a sword stays in the hand it was attached to.

RESTORE POINTERS AFTER LOAD

```
Load All (From Disk)
Load Actors By Save Tag ("Persistent")   // respawn the actors

// Now relink every reference & re-attach children:
Restore Actor References
```

> **IMPORTANT**
> Call **Restore Actor References** *after* all the actors it needs to point at exist in the world. If a target hasn't been spawned yet, that one link is skipped (the rest still restore) - so load your actors first, then restore references last.

Any reference marked **SaveGame** is handled: single actor references, arrays of references, and the actor's attachment chain. No IDs to manage, no re-finding by name.

10

PART 3 - GAME OBJECTS

Save categories

Categories let one system separate the kinds of data a game keeps, so the right things live in the right file with the right lifetime.

Category	Use it for
`Global`	Settings, achievements, unlocks - shared across every save.
`Player`	Progress, inventory, skills - tied to one player.
`World`	Doors, chests, NPC states - the state of a level.
`Checkpoint`	Temporary progress since the last checkpoint.
`Session`	Multiplayer runtime data for the current session.
`Profile`	A named player profile.

Using categories

Set the active category, then save or load as normal - the data is filed under that category. Or address a category directly with **Save Category** / **Load Category**.

SEPARATE GLOBAL FROM PLAYER DATA

```
Set Active Category (Global)
Save Bool ("Ach_FirstBlood", true)   // shared across all saves

Set Active Category (Player)
Save Int ("Coins", 250)              // this player only
Save All (To Disk)
```

TIP
A common split: keep audio/graphics **settings** and **achievements** in **Global** so they persist no matter which save the player loads, and keep run-specific progress in **Player** or **World**.

11

PART 4 - POWER FEATURES

Tags & tag layers

Tags label saved values and actors so you can save, load, or delete a whole group in one call.

Tagging values

Attach a tag when you save, then act on everything sharing it. **Save By Tag** and **Load By Tag** write and read a tagged group as its own layer; **Delete By Tag** and **Remove All By Tag** clear it.

SAVE A GROUP, CLEAR IT LATER

```
Save By Tag (Tag = "Level2")     // everything for level 2
...
// Player restarts level 2 - wipe just that group:
Delete By Tag (Tag = "Level2")
```

Inspecting tags

Get All Tags lists them, **Has Tag** checks one, **Get Keys By Tag** returns the keys inside a tag, and **Remove Multiple Tags** clears several at once.

> **NOTE**
> **Tag layers** let a group live in its own file. That means you can ship DLC or level state as a separate save layer and load it independently, without touching the main save.

12

PART 4 - POWER FEATURES

Async, auto-save & checkpoints

Big saves shouldn't freeze the game, and players shouldn't lose progress. These features cover both.

Async (non-blocking) saves

Save All Async (Non-Blocking) writes on a background thread and fires an event when it's done - no hitch, even for large saves. **Load All Async** does the same for loading. Use these for autosaves during gameplay.

SAVE WITHOUT A HITCH

```
Save All Async (Non-Blocking)
//   -> bind 'On Save Completed' to show a 'Saved' icon
```

Auto-save

Enable Auto Save (interval in seconds) saves on a timer for you; **Disable Auto Save** stops it; **Trigger Auto Save** forces one immediately (for example on level exit).

AUTOSAVE EVERY 5 MINUTES

```
Enable Auto Save (Interval = 300.0)
// Force one right now, e.g. before a boss fight:
Trigger Auto Save
```

Checkpoints

Checkpoint Save writes a temporary progress snapshot the player returns to on death - separate from their manual saves, so a checkpoint never overwrites a real save file.

> **TIP**
> Pair **Force Save Flush** with app shutdown or **Has Unsaved Changes** to guarantee nothing is lost when the player quits mid-session.

13

PART 4 - POWER FEATURES

Events

React to what the save system is doing - show a spinner, pop a 'Saved' toast, or handle a corrupt file gracefully.

Bindable events fire at each stage of a save or load, so your UI and game logic can respond without polling. Typical uses:

- **On Save Started / Completed** - show and hide a saving indicator.
- **On Load Completed** - fade in the world once data is ready.
- **On Auto Save** - flash a subtle 'Autosaved' toast.
- **On Save Corrupted** - fall back to a backup (section 16) instead of crashing.

Polling helpers

When you'd rather ask than subscribe: **Is Saving**, **Is Loading**, **Is Save Manager Ready**, and **Has Unsaved Changes** each return a bool you can branch on in Tick or on a button press.

> **INFO**
> Guard your save button with **Is Saving** so a player can't queue ten saves by mashing it. Grey the button out while a save is in flight, re-enable it on **On Save Completed**.

14

PART 4 - POWER FEATURES

Compression & encryption

Make saves smaller and tamper-proof. Both are off by default and toggle with a single node - existing saves keep loading either way.

Compression (zlib)

Set Compression Enabled (true) shrinks saves with zlib - often dramatically for large or repetitive data. Check the current state with **Is Compression Enabled**.

Encryption (AES-256)

Enable AES Encryption with your key encrypts the save with real AES-256, so players can't open or edit the file in a text editor. **Disable AES Encryption** turns it off; **Is AES Encryption Enabled** reports the state.

COMPRESS + ENCRYPT EVERY SAVE

```
Set Compression Enabled (true)
Enable AES Encryption   (Key = "your-secret-key")
Save All (To Disk)     // now compressed and encrypted
```

> **IMPORTANT**
> Keep your AES key consistent across a build. If the key changes, older saves made with the previous key can't be decrypted and you'll get an **Encryption Key Mismatch** result. Store the key somewhere stable in your project, not hard-coded in a throwaway variable.

Saves made without these features still load after you enable them - OneNode detects an unprotected file and reads it directly, so you can turn compression or encryption on mid-project without breaking existing saves.

15

PART 4 - POWER FEATURES

Versioning & migration

The feature that saves you after launch: load a player's old save even though your data changed in a patch.

The problem

You ship. Players build up saves. In patch 1.1 you rename **"HP"** to **"Health"** and add a new stat. Without migration, every existing save is now subtly wrong or broken.

The fix - register migration steps

Stamp your save format with **Set Save Version**. When you change the format, register a step describing how to upgrade from the old version to the new one. On load, OneNode runs any needed steps automatically.

MIGRATE OLD SAVES ON LOAD

```
Set Save Version (2)
Register Migration Step (From = 1, To = 2)
//   inside: read old "HP" -> write new "Health", set defaults

// On load, OneNode checks the file's version and upgrades it:
Load All (From Disk)   // v1 saves are migrated to v2 for you
```

Helpers: **Get Current Save Version**, **Save Needs Migration**, **Run Migrations**, and **Get Save Version Description** let you inspect and control the process. **Clear Registered Migrations** resets them (useful in tests).

> **TIP**
> Register your migration steps once at startup (Game Instance Init). Then you never think about it again - old saves quietly upgrade themselves the moment a returning player loads them.

16

PART 4 - POWER FEATURES

Backup & recovery

Disks fill up, power cuts out, files corrupt. OneNode keeps backups so a bad write doesn't cost the player their game.

Automatic backups

Before overwriting a slot, OneNode can keep the previous version as a backup. **Set Backup Retention** (count) controls how many are kept. **Has Backup** checks whether one exists.

Recovering

If a save reports **Save File Corrupted**, call **Restore Backup** to roll the slot back to the last good copy - then tell the player you recovered their progress.

SURVIVE A CORRUPT SAVE

```
Set Backup Retention (Count = 3)   // keep last 3 saves

// On load, if the result is 'Save File Corrupted':
Branch: Has Backup
True -> Restore Backup  ->  Load All (From Disk)
False -> start a new game, warn the player
```

NOTE
Backups cost a little disk space - that's the trade for never losing a player's save to a single bad write. For most games, a retention of 2-3 is plenty.

17

PART 5 - ADVANCED

Multiplayer saves

Saving in multiplayer means two things: each player's own progress, and the shared state of the world. OneNode handles both, with server authority built in.

Per-player saves

Save Player and **Load Player** take a Player Controller and store that player's data in their own file. OneNode resolves a stable player identity for you (**Resolve Player Id**) from the net ID, so the right save follows the right player across sessions.

SAVE ONE PLAYER'S PROGRESS

```
Save Player (Controller = PC)   // filed under that player's id
// Next session, same player:
Load Player (Controller = PC)
```

World saves (server only)

Save World (Server Only) and **Load World** persist shared level state. They're guarded by authority: **Is Network Authority** gates the write so only the server can save the world, and a client call returns **Not Network Authority** instead of corrupting shared state.

> **INFO**
> OneNode includes a replicated save component that transfers a client's save data to the server in verified, paced chunks - so large saves move safely across the network without stalling the game. You drive it through the same **Save Player** / **Load Player** nodes.

Helpers: **Is Dedicated Server** and **Is Network Client** let you branch your save logic by role when you need to.

18

PART 5 - ADVANCED

Cloud saves

Sync saves to Steam Cloud, Dropbox, or your own backend through one simple provider interface.

How it works

OneNode defines a **cloud provider** interface. Point it at a provider and your saves sync up and down; a local-folder provider ships in the box for testing, and the interface is designed so a Steam or Dropbox provider drops in the same way.

Conflict handling

When local and cloud saves disagree, you choose the winner: **Prefer Cloud (download)** or **Prefer Local (upload)**. No silent data loss - the resolution is always your call.

SYNC ON STARTUP

```
// Set your provider once, then sync:
Sync Saves (Conflict = Prefer Cloud)   // pull newest from cloud
// ...at quit:
Sync Saves (Conflict = Prefer Local)   // push local up
```

> **NOTE**
> During this project's live test sweep, the Steam Cloud path reported 2 of 4 sub-checks failing - that path needs a real Steam app context to fully validate. The provider interface and local sync are solid; treat a specific storefront provider as something to verify in your own build with your app credentials.

19

PART 5 - ADVANCED

Editor tools

Two editor tools help you build and debug saves without leaving Unreal.

Save Debugger

Open the Save Debugger to inspect any slot live - every key, its type and value, tags, categories, version, and file size. From Blueprint, **Debug: Dump All Save Data**, **Get Save Debug Lines**, and **Get Save Debug Summary** print the same information to log or UI.

SEE EXACTLY WHAT'S IN A SAVE

```
Debug: Dump All Save Data   // prints every key/value to the log
Summary = Get Save Debug Summary   // one-line overview for UI
```

Setup Wizard

The Setup Wizard walks you through first-time configuration - default slot, compression, encryption, backup retention - and writes the settings for you, so you're saving correctly from minute one.

> **TIP**
> When a save behaves unexpectedly, open the Save Debugger before anything else. Nine times out of ten you'll spot the problem instantly - a mistyped key, a value in the wrong category, or an older save version than you expected.

20

PART 5 - ADVANCED

Analytics & profile metadata

Rich metadata turns a bare save slot into a proper save card - and gives you numbers about how your game is played.

Profile metadata

Attach human-friendly details to a slot so your load screen can show them: **Set Player Name**, **Set Level Name**, **Set Progress Description**, **Add Play Time**, and **Set Screenshot From PNG** for a thumbnail. Read it all back with **Get Profile Metadata**.

BUILD A RICH SAVE CARD

```
Set Player Name ("Aria")
Set Level Name ("The Drowned City")
Set Progress Description ("Chapter 3 - 62%")
Add Play Time (DeltaSeconds)
Set Screenshot From PNG (ScreenshotBytes)
```

Analytics

Get Save Analytics returns aggregate figures - totals, sizes, counts - and **Set Custom Metadata Field** lets you attach your own key/value stats to a save. Useful for balancing, QA, or a stats screen.

> **INFO**
> **Add Play Time** called each session gives you an accurate total playtime per slot for free - a detail players notice on a polished load screen.

21

PART 6 - REFERENCE

Recipes

Copy-paste starting points for the things most games need. Each is just OneNode nodes in order.

A working Save / Load menu

SAVE / LOAD MENU

```
// SAVE button:
Set Active Save Slot (SelectedSlotName)
Set Level Name (Current Level) ; Add Play Time (Session)
Save All Async (Non-Blocking)   -> toast 'Saved'

// LOAD screen build:
Slots = Get All Slot Infos   -> one UI card per entry
// LOAD button:
Set Active Save Slot (SelectedSlotName) ; Load All (From Disk)
```

Checkpoint + respawn

CHECKPOINT RESPAWN

```
// Reaching a checkpoint:
Save Actor (Player, "Player") ; Checkpoint Save
// On death:
Load All (From Disk) ; Load Actor (Player, "Player")
Restore Actor References
```

Secure release build

HARDENED SAVES

```
Set Compression Enabled (true)
Enable AES Encryption (BuildKey)
Set Backup Retention (3)
```

22

PART 6 - REFERENCE

Troubleshooting

The usual snags and their one-line fixes. When in doubt, open the Save Debugger (section 19).

Symptom	Cause & fix
Load returns nothing / defaults	No slot set, or wrong slot name. Call **Set Active Save Slot** with the exact name, then **Load All (From Disk)** before reading.
Values are empty after load	You read before loading. **Load All (From Disk)** must run before any **Load ...** node.
Actor references are null	Call **Restore Actor References** after the target actors are spawned - not before.
Actor properties not saved	Tick **SaveGame** on each variable (Details → Advanced) so OneNode knows to persist it.
'Encryption Key Mismatch'	The AES key changed between save and load. Use one stable key per build.
'Save File Corrupted'	Check **Has Backup**, then **Restore Backup**. Set backup retention so this never costs progress.
Nodes don't appear	Plugin not enabled for this project. **Edit** → **Plugins**, enable OneNode, restart.
Old saves broke after a patch	Add **Register Migration Step** from the old version to the new (section 15).

23

PART 6 - REFERENCE

Node reference

Every node, grouped by job. All live under the **OneNode Save System** category in the Blueprint right-click menu.

Slots

`Set Active Save Slot` · `Get Active Slot Name` · `Save Slot Exists` · `Delete Save Slot` · `Copy Slot` · `Rename Slot` · `Get All Slot Names` · `Get All Slot Infos` · `Get Slot Info` · `Sanitize Slot Name`

Save values

`Save Int` · `Save Int64` · `Save Float` · `Save Bool` · `Save Byte` · `Save String` · `Save Name` · `Save Vector` · `Save Rotator` · `Save Transform` · `Save Color` · `Save Raw Bytes`

Save arrays

`Save Int Array` · `Save Float Array` · `Save String Array` · `Save Name Array` · `Save Vector Array` · `Save Transform Array`

Load values

`Load Int` · `Load Int64` · `Load Float` · `Load Bool` · `Load Byte` · `Load String` · `Load Name` · `Load Vector` · `Load Rotator` · `Load Transform` · `Load Color` · `Load Raw Bytes`

Load arrays

`Load Int Array` · `Load Float Array` · `Load String Array` · `Load Name Array` · `Load Vector Array` · `Load Transform Array`

Keys & data

`Has Save Key` · `Remove Key` · `Get All Keys` · `Get Entry Count` · `Clear All Data` · `Clear Memory Cache`

Disk I/O

`Save All (To Disk)` · `Load All (From Disk)` · `Save All Async (Non-Blocking)` · `Load All Async (Non-Blocking)` · `Quick Save` · `Quick Load` · `Force Save Flush` · `Has Unsaved Changes` · `Save Slot To File` · `Load Slot From File` · `Export Slot To Bytes` · `Import Slot From Bytes`

Actors

`Save Actor` · `Load Actor` · `Save Multiple Actors` · `Save Actors With Tag` · `Load Actors By Save Tag` · `Restore Actor References` · `Has Actor Save Data` · `Get Saved Actor Count` · `Get All Actor Keys`

Tags

`Save By Tag` · `Load By Tag` · `Delete By Tag` · `Remove All By Tag` · `Get All Tags` · `Has Tag` · `Get Keys By Tag` · `Remove Multiple Tags`

Categories

`Set Active Category` · `Get Save Scope` · `Set Save Scope` · `Save Category` · `Load Category`

Auto-save & checkpoints

`Enable Auto Save` · `Disable Auto Save` · `Trigger Auto Save` · `Checkpoint Save`

Security

Set Compression Enabled · Is Compression Enabled · Enable AES Encryption · Disable AES Encryption · Is AES Encryption Enabled · Enable Encryption · Disable Encryption · Is Encryption Enabled

Versioning

Set Save Version · Get Current Save Version · Get Save Version Description · Save Needs Migration · Register Migration Step · Run Migrations · Clear Registered Migrations · Get Plugin Save Version

Backup

Set Backup Retention · Has Backup · Restore Backup

Multiplayer

Save Player · Load Player · Save World (Server Only) · Load World · Resolve Player Id · Is Network Authority · Is Dedicated Server · Is Network Client

Cloud

Sync Saves · Prefer Cloud (download) · Prefer Local (upload)

Metadata & analytics

Get Profile Metadata · Set Player Name · Set Level Name · Set Progress Description · Add Play Time · Set Screenshot From PNG · Set Custom Metadata Field · Get Save Analytics · Get Save File Size On Disk · Get Save Size (Bytes)

Debug & status

Debug: Dump All Save Data · Get Save Debug Lines · Get Save Debug Summary · Is Saving · Is Loading · Is Save Manager Ready · Save Result To String · Save Result Is Success

Thanks for building with OneNode

You now have a complete Blueprint save framework - values, actors, references, slots, tags, categories, encryption, versioning, backup, multiplayer, and cloud - and you never have to write a SaveGame class again. Build the game; let OneNode handle the saving.

Stop rebuilding this every project.

OneNode is the save system you set up once and reuse forever - on every game, across UE 4.27 through 5.7.

No structs. No boilerplate. No C++. Just nodes that work.

Built by Black Scar Studio · Available on the Fab Marketplace · Free updates for the life of the product