Hierarchical.Tag.Forge Manual

Introduction 1
Installation and Requirements 1
Supported Unity versions 2
Dependencies 2
Importing the package 2
Getting Started 2
Core 2
Authoring Tags 3
Naming conventions 3
Designing roots 3
Baking 3
Generated code references 4
Default Database 4
Runtime settings asset 4
GameplayTag 4
What it is 4

Practical guidance 5
Matching semantics 5
GameplayTagContainer 5
What it is 5
GameplayTagQuery 6
What it is 6
GameplayTagRequirements 6
What it is 6
What it contains (the four checks) 6
RequireAll 6
RequireAny 7
BlockAny 7
Query (optional advanced logic) 7
Match semantics (TagMatchType) 7
How evaluation works (short-circuit order) 7
How to use it 7
1) Add it to a gateable thing 8
2) Evaluate it against a tag owner 8
Performance behavior (important for users) 8
Fast path: IGameplayTagAssetQuery 8
Generic path: IGameplayTagAsset only 8
Practical guidance 8
Editor Windows/Drawers 9
Tag Manager Window 9
Opening the Tag Manager 9
Toolbar 9
Search field 9
Expand All / Collapse All toggle button 10
Add Tag (“+”) button 10
Tag Tree 10
Hierarchical structure 11
Alternating row backgrounds (zebra striping) 11

Tag labels with inline comments 11
Tag nodes (“⋮” menu) 11
Structural folder nodes (“+” button) 12
Tag Actions Menu (⋮) 12
Add Subtag… 12
Update Comment… 12
Rename… 13
Delete (with children)… 13
Create / Edit / Rename Popups 13
Create Tag popup 13
Edit Comment popup 14
Rename Tag popup 14
GameplayTag 14
How it appears in the Inspector 14
How to use it in the editor 14
Assign a tag 14
Clear a tag 14
Copy tag information (for debugging or documentation) 15
Locate the active database 15
Common display states (what they mean) 15
GameplayTagContainer 15
How it appears in the Inspector 15
How to use it in the editor 15
Add or remove tags (primary workflow) 15
Clear the container 16
Browse tags in the expanded list 16
Remove a single tag quickly 16
Per-tag quick actions (row menu) 16
Visual status cues in the list 16
Practical guidance 16
GameplayTagQuery 17
How it appears in the Inspector 17

The two representations you manage (what the UI is really controlling) 17
Definition (authoring) 17
Compiled data (runtime) 17
Status line (what it means) 17
How to use it in the editor 18
Create a new query definition 18
Edit the query logic 18
Compile the definition to runtime data 18
Clear compiled data 18
Practical guidance 19
Tag Picker Dropdown 19
What it is 19
Requirements and prerequisites 19
Default database must be set 19
Database should be baked for full functionality 19
Layout and controls 19
1) Search field 20
2) Expand All / Collapse All 20
3) Tag tree (hierarchical list) 20
4) Multi-select action bar (multi mode only) 20
Single-select vs Multi-select behavior 21
Single-select mode (GameplayTag) 21
Multi-select mode (GameplayTagContainer) 21
Visual indicators and row interaction 21
Checkbox/toggle indicator 21
Double-click behavior 21
What the picker uses as “identity” 22
Stable identity is GUID-based 22
Special rows use negative internal IDs 22
Expected user workflow 22
Picking a single tag 22
Editing a container 22

Troubleshooting and common issues 22
"No default database set" 22
"Database is not baked" 22
"Expand/Collapse doesn't work during search" 22
GameplayTagQuery Editor Window 22
What the Query Editor is 23
Opening the Query Editor 23
Header Toolbar 23
Title 23
Auto-Compile toggle 23
Compile button 23
Tag Source Banner 24
Tag Source display 24
Ping button 24
Creating and Editing a Definition 24
Definition creation 24
Description field 24
Authoring the Query Tree 25
Node layout and navigation 25
Root node 25
Node Type 25
Group Nodes: Children Management 25
Add buttons 26
Child ordering and removal 26
Leaf Nodes: Tag Lists 26
Match field 26
Tags list (Edit / Clear / Remove) 26
Tag display 27
Definition Utilities 27
Validate 27
Prune Orphans 27
Reset and Close 28

Reset Definition To Default 28
Close 28
Assets 28
GameplayTagOwner 28
What it is 28
When to use it 28
Inspector authoring 29
Owned tags 29
Change notifications 29
1) Runtime C# event (primary runtime hook) 29
2) Inspector events (UnityEvent, no code required) 30
On Tags Changed 30
On Tags Changed Detailed 30
Query API (read-only checks) 30
HasMatchingGameplayTag 30
HasAnyMatchingGameplayTags 30
HasAllMatchingGameplayTags 30
Mutation API (safe modifications) 30
AddGameplayTag(tag) 31
RemoveGameplayTag(tag) 31
AddGameplayTags(container) 31
RemoveGameplayTags(container) 31
ClearGameplayTags() 31
Direct container access (advanced) 31
NotifyTagsChanged() 32
Diff computation strategy (why bulk operations are efficient) 32
Recommended usage patterns 32
Pattern 1: Tag-driven UI refresh 32
Pattern 2: Systems that need precise deltas 32
Pattern 3: High-frequency checks (AI / combat rules) 32
Common Pitfalls: 32
GameplayTagAssetUtility (Query/Mutation Helpers for Arbitrary Unity Objects) 33

What it is 33
Resolution model (how it finds tags) 33
Query families: Strict vs Safe (why both exist) 33
Strict queries 33
Safe queries 34
Mutation helpers 34
Observable helpers 34
GameplayTagQueryAsset (Reusable Query Definition Asset) 35
What it is 35
What it contains 35
Practical usage pattern 35
Best practice note 35
GameplayTagRequirementsAsset (Reusable Requirements Rule Asset) 35
What it is 35
What it contains 36
Design intent: description is non-operative 36
Practical usage pattern 36
Project Settings 36
Where to find the settings 36
Definitions 37
Auto Create Default Database 37
Auto Generate Code On Bake 37
Auto Bake And Generate On Tag Edit 37
Warn On Invalid Tags 37
Clear Invalid Tags 38
Invalid Tag Characters 38
Auto Compile Query On Definition Change 38
Fail Build On Stale Queries 39
Auto Fix Stale Queries On Build 39
Presets 39
Recommended default (most teams) 39
Strict production / locked pipelines 39

Menu Commands 40
Bake GameplayTagDatabase 40
Menu Command 40
What it does 40
How to use 40
Notes 40
Generate Tag Code 40
Menu Command 40
What it does 40
How to use 41
Notes 41
API Usage and Examples 41
Performance and Best Practices 41
Public Runtime API Overview 42
Authoring Tags in Code (Attribute Declarations) 43
What the attribute does 43
Recommended declaration pattern 43
Runtime Querying and Common Patterns 44
GameplayTag 44
TagMatchType 45
GameplayTagContainer 45
GameplayTagAssetUtility 46
GameplayTagQuery 47
GameplayTagRequirements 48
Example Patterns 48
Demo Scene 49
Demo_AssetUtilityTester.cs 50
What it does 50
How to use it 50
What to copy into your project 50
Demo_GameplayTagVisual.cs 50
What it does 51

How to use it 51
What to copy into your project 51
Demo_GameplayTagSingleOwner.cs 51
What it does 51
How to use it 51
What to copy into your project 51
Demo_GameplayTagSingleCaller.cs 52
What it does 52
How to use it 52
What to copy into your project 52
Demo_GameplayTagContainerOwner.cs 52
What it does 52
How to use it 53
What to copy into your project 53
Demo_GameplayTagContainerCaller.cs 53
What it does 53
How to use it 53
What to copy into your project 53
Demo_GameplayTagQuery.cs 54
What it does 54
How to use it 54
What to copy into your project 54
Demo_GameplayTagQueryCaller.cs 54
What it does 54
How to use it 54
What to copy into your project 55
Demo_RequirementsGate.cs 55
What it does 55
How to use it 55
What to copy into your project 55
Testing and Diagnostics 56
Build Validation 56

GameplayTagQuery Build Validation 56
What the validator checks 56
Important limitation (definition-backed checks) 56
Auto-fix behavior 56
Running validation manually 57
Project preferences (validation policy) 57
How to read the report 57
GameplayTag Stress Harness 57
Overview 58
Adding the harness to a scene 58
Inspector UI and organization 58
Core settings 58
Run Control 58
Load 58
Owner setup and tag assignment 58
Container inputs (HasAny / HasAll benchmarks) 59
Query inputs (GameplayTagQuery.Matches) 59
Diagnostics 59
Profiling the Stress Harness 59
Finding the correct profiler markers 59
How to interpret results 60
Recommended profiling workflow 60
FAQ and Troubleshooting 60
“No GameplayTagDatabase is set as default” 60
“Database is not baked” appears 60
Tags don’t match as expected 60
Builds can’t find default database 61
Generated constants are missing or stale 61
Conclusion 61

Don't like reading? Go to the following link for video tutorials:
https://www.youtube.com/@CARRYFORGE

Introduction

HierarchicalTagForge is a production-grade, Unreal-style hierarchical gameplay tag framework for Unity. It provides a single, consistent vocabulary you can use across your entire project; abilities, AI states, items, units, weapons, buffs/debuffs, events, UI flows, quest flags, and more; while keeping runtime lookups fast, deterministic, and scalable as content grows.

Instead of relying on fragile string comparisons, scattered enums, one-off manager scripts, or concrete type references, you maintain a centralized Gameplay Tag Database and query it efficiently at runtime through optimized access patterns designed for real game workloads.

A core focus of **HierarchicalTagForge** is performance: runtime APIs and query evaluation are structured for zero-GC allocation during steady-state gameplay querying. This supports frequent checks and complex evaluations without avoidable garbage collections, ideal for AI, combat, state machines, and systemic gameplay.

Installation and Requirements

Supported Unity versions

HierarchicalTagForge is designed for Unity 2022 LTS and newer, including Unity 6+.

Dependencies

No third-party runtime dependencies.

Editor tooling uses UnityEditor IMGUI controls.

Importing the package

Import the package into your project.

Allow Unity to recompile scripts.

Open Tag Manager once to trigger editor-time initialization.

Getting Started

When initially installed, there are a few key assets that are either **A: already created** or **B: generated automatically** depending on if this package was brought over from another project or if it was installed via the package manager. If for some reason your database/tags are not correctly referenced, see: **(Faqs and Troubleshooting)** for possible fixes. To get started (after the package has been imported), in the header of the Unity Engine Editor window go to: **Tools->CarryForge->HierarchicalTagForge-**

>TagManager. This will automatically create a default database if the default database already exists, then it will ensure that the database is found and registered.

NOTE:
The database should always exist in the following file path:
CarryForge->HierarchicalTagForge->Runtime->Resources

Now that that the database is registered, you can either go to the demo scene and begin reviewing the demo objects and how they work with one another or, you can begin creating your own tags inside of the **Tag Manager Window**. If this is the first time you have used **HierarchicalTagForge**, it is highly recommended that you read the rest of this documentation as it will give you important code/performance insights to get the most out of this asset.

NOTE: When the Tag Manager is open, you will see a Demo tag hierarchy. This is referenced by the demo scene and is not explicitly required if you no longer plan to use the demo scene and/or you already generated your own tags to test. You can quickly delete the whole hierarchy by selecting the options button and selecting **Delete With Children**.

Core

This section covers the foundational concepts and primary runtime types that everything else in **HierarchicalTagForge** builds on. This section defines the tag model (hierarchy, identity, and matching semantics), explains the central data structures and workflows (tags, containers, queries, requirements, registry/database), and establishes the mental model you need before moving into authoring, tooling, and advanced patterns.

Authoring Tags

Naming conventions

Use dot-separated segments to express hierarchy (Root.Category.Subcategory.Leaf).

Avoid embedding implementation details in tag names (prefer meaning and intent).

Prefer singular nouns and consistent tense (e.g., State.Stunned, Effect.Buff.DamageUp).

Keep segments short and searchable; avoid redundant words (Ability.Magic.Fire.Fireball, not Ability.Magic.AbilityFireball).

Designing roots

Your roots are the '**namespaces**' of your vocabulary. A few consistent roots improve discoverability and reduce collisions.

Common roots include:

Ability

State

Effect

Item

Unit

Weapon

AI

UI

Quest

Event

Faction

Baking

Baking converts the editable database into a runtime-friendly form: string paths and hierarchy are transformed into deterministic integer IDs, parent chains, and lookup tables suitable for high-frequency checks.

Editor tooling uses baked data to drive pickers and previews reliably.
Runtime evaluation depends on baked data for fast hierarchical matching.
Generated code (optional) is typically based on the baked data and/or database contents.

Generated code references

If enabled, the package can generate a flat static class containing constant tag references. This provides clean, typo-proof access in code without manually declaring tags or typing strings.

Default Database

HierarchicalTagForge uses a **GameplayTagDatabase** as the source of truth for your project's tag vocabulary. Many editor tools (Tag Manager, tag pickers, query authoring UI) resolve a **default database** from these project settings. Currently **Hierarchical Tag Forge** only allows for one tag database at a time.

Runtime settings asset

Player builds cannot use UnityEditor APIs. To make the default database discoverable at runtime, HierarchicalTagForge uses a runtime settings ScriptableObject stored under a Resources path. The runtime loads this settings asset to locate the default database and optional auto-initialize behavior.

Recommended: keep the runtime settings asset in a dedicated Resources folder owned by the package.

Confirm the settings references your intended default GameplayTagDatabase.

If you ship builds and the default database is missing, this asset is the first item to verify.

GameplayTag

What it is

GameplayTag is the core **single-tag** value used throughout HierarchicalTagForge. It represents one tag from your hierarchy (for example: Ability.Magic.Fire.Fireball) and is stored internally as a stable identifier (GUID-backed), so you do not rely on fragile string comparisons at runtime. The database stores the authoring representation (including the path), while runtime logic uses a stable identifier (GUID) and baked, integer-based lookup data for fast evaluation.

Use a GameplayTag field when an object needs **exactly one** semantic label—such as:

- An ability type tag
- A weapon category tag
- A unit “class” tag
- A state tag
- A UI flow tag
- A single gating/interaction tag

(If you need multiple tags at once, use a tag container type instead.)

Practical guidance

- Prefer GameplayTag for “one thing is this one category/state/type.”
- Prefer a container when you want “this object currently has these many tags.”
- In day-to-day authoring, always assign tags through the Inspector picker rather than typing strings—this keeps your project vocabulary consistent and prevents typos and mismatched paths.

Matching semantics

Different checks may use different semantics. In practice you will use a mix of:

Exact match: only matches the exact tag.

Hierarchical match: a child tag can match a parent query depending on the operation (e.g., Fireball matches Ability.Magic.Fire).

Set logic: Any/All/None operations over a container.

The query/requirements assets expose these semantics in a designer-friendly way; runtime APIs typically expose both exact and hierarchical checks.

GameplayTagContainer

What it is

A Tag Container represents a set of owned tags for an entity (player, AI, item, ability spec, UI state, etc.). Containers are designed for frequent runtime access. Internally, containers generally maintain a stable identity list (GUIDs) and a runtime cache derived from baked data to make hierarchical checks fast and allocation-free.

GameplayTagContainer is the core **multi-tag** value in HierarchicalTagForge. It represents a set of tags owned by an object (for example: { Unit.Humanoid, Faction.Player, Status.Buffed }) and is designed for gameplay scenarios where an entity can have multiple labels at once.

Use a GameplayTagContainer field when an object needs to hold **many tags simultaneously**, such as:

- Current status effects (buffs/debuffs)
- Faction/alignment + role + state tags
- Item properties and modifiers
- AI state/context tags
- Interaction eligibility tags (what this thing “is” and “can do”)

GameplayTagQuery

What it is

A Tag Query is a reusable boolean expression over tags, typically authored in the Query Builder UI. Queries can be nested (groups within groups) to express complex conditions without bespoke code.

GameplayTagQuery is the system’s **boolean query object** for evaluating tag sets. Conceptually, it answers:

“Does this target match my tag rules?”

A GameplayTagQuery is authored as a **definition tree** (human-editable logic), and it is evaluated at runtime using a **compiled** representation (fast, deterministic data).

Typical use cases include:

- Requirements checks (abilities, items, upgrades, interactions)
- AI decision filters (“only if target is Unit.Humanoid AND NOT Status.Stunned”)
- UI gating (“show this button only if player has Tech.Unlock.X”)
- Gameplay state validation and conditional triggers

GameplayTagRequirements

What it is

A Requirements Query is a streamlined gating asset inspired by GAS: it covers the most common requirement patterns using three buckets (Require All, Require Any, Block Any). This is ideal for ability gating, interactions, equippables, AI behaviors, quest conditions, and UI state gating.

GameplayTagRequirements is a reusable, serializable **ruleset** for gating actions based on a target's gameplay tags.

It is designed to answer:

"Is this operation allowed for this tag owner right now?"

- Use **Require/Block containers** for the majority of gating logic; they're simple, readable, and fast.
- Use the **Query** only when the rule cannot be expressed cleanly as Require/Any/Block.
- If you expect a requirement to be evaluated frequently (e.g., per-frame UI gating or AI decisions), consider implementing IGameplayTagAssetQuery on the evaluated targets to enable the fast path.

What it contains (the four checks)

GameplayTagRequirements combines up to **four** independent checks into a single evaluation:

RequireAll

- **Meaning:** *All* tags in RequireAll must be present on the target.
- **Use for:** strict prerequisites (e.g., must be State.Alive and Class.Mage).

RequireAny

- **Meaning:** *At least one* tag in RequireAny must be present (only when the container is non-empty).
- **Use for:** alternative prerequisites (e.g., Weapon.Rifle **or** Weapon.Pistol).

BlockAny

- **Meaning:** If *any* tag in BlockAny is present, the requirements **fail** immediately.
- **Use for:** disqualifiers (e.g., cannot be State.Stunned or State.Silenced).

Query (optional advanced logic)

- **Meaning:** A GameplayTagQuery that can express nested boolean rules (AND/OR/NOT groups, etc.).
- **Ignored when:** the query has no compiled content (treated as "not used").
- **Use for:** complex rules that don't fit neatly into the three-container model.

Match semantics (TagMatchType)

All checks use a single TagMatchType value (MatchType) that controls **how tags are considered a match** during evaluation.

This match mode is applied consistently across:

- RequireAll
- RequireAny
- BlockAny
- Query (when used)

This is the lever that determines whether “exact” vs “hierarchy-aware” matching applies (depending on how your project defines TagMatchType behavior).

How evaluation works (short-circuit order)

IsMetBy(asset) evaluates in a performance-friendly, failure-first order:

1. **BlockAny** (fast fail)
2. **RequireAll**
3. **RequireAny**
4. **Query** (only if present)

This means disqualifying tags stop the check immediately, which is important when these gates run frequently (abilities, AI, interaction scans).

How to use it

1) Add it to a gateable thing

Use GameplayTagRequirements anywhere you need a serialized ruleset:

- Ability activation rules
- Interactable requirements
- Item equip restrictions
- AI task prerequisites
- Tech/upgrade gating

You typically expose it as a serialized field:

- Configure RequireAll, RequireAny, BlockAny in the Inspector (via the container UI).
- Optionally author a GameplayTagQuery if you need more than simple require/block logic.
- Choose the TagMatchType once for the whole requirements object.

2) Evaluate it against a tag owner

Call:

- requirements.IsMetBy(asset)

Where asset is an object implementing IGameplayTagAsset.

Performance behavior (important for users)

GameplayTagRequirements supports two evaluation paths depending on what the target implements:

Fast path: IGameplayTagAssetQuery

If the target also implements IGameplayTagAssetQuery, then:

- Simple checks (RequireAll, RequireAny, BlockAny) are evaluated **directly** without materializing a full container snapshot.
- This reduces overhead for frequently-called checks.

The query evaluation still requires a container snapshot, because queries evaluate over a concrete tag set. In that case the implementation pulls tags into a temporary container and evaluates Query.Matches(...).

Generic path: IGameplayTagAsset only

If the target does **not** support the query-oriented interface:

- The target's owned tags are pulled once into a temporary container.
- All checks run against that snapshot.

Practical guidance

- Use **Require/Block containers** for the majority of gating logic; they're simple, readable, and fast.
- Use the **Query** only when the rule cannot be expressed cleanly as Require/Any/Block.
- If you expect a requirement to be evaluated frequently (e.g., per-frame UI gating or AI decisions), consider implementing IGameplayTagAssetQuery on the evaluated targets to enable the fast path.

Editor Windows/Drawers

This section documents the Unity Editor-facing tooling that ships with **HierarchicalTagForge**, including custom windows, inspectors, and property drawers. This section explains what each editor UI is for, how to use it to author and manage tags and queries efficiently, and how the editor experience maps to the underlying runtime data so you can confidently move from configuration in the editor to behavior in play mode.

Tag Manager Window

The **Tag Manager** is the primary editor UI for creating, browsing, searching, and maintaining hierarchical gameplay tags within your active **GameplayTagDatabase**. It is built for day-to-day production iteration: fast authoring, clean hierarchy navigation, and lightweight editing workflows that do not require typing fragile strings throughout your project.

The Tag Manager targets the **default database** configured in **HierarchicalTagForgeProjectSettings** and relies on editor auto-setup to ensure required settings/assets are present.

Opening the Tag Manager

Open the Tag Manager from the Unity menu under the HierarchicalTagForge tools entry (Tools menu). The window opens as an EditorWindow titled **"Gameplay Tags"**.

When opened/enabled, the Tag Manager:

Ensures HierarchicalTagForge editor setup is ready.

Resolves the current default GameplayTagDatabase from project settings.

Restores persisted UI state such as search text, scroll position, and foldout expansion.

Toolbar

The toolbar provides the highest-frequency actions: searching, expanding/collapsing the tree, and creating new tags.

Search field

What it does

Filters the visible tag tree based on a case-insensitive substring match.

Matching is evaluated against:

The tag's full path (e.g., Ability.Magic.Fire.Fireball)

The tag comment

When searching, the tree keeps ancestor nodes visible if they contain matching descendants.

How to use

Click the **Search** field.

Enter a partial path, name, or keyword from a comment.

The tree immediately updates to show only matching tags and the hierarchy needed to reach them.

Expand All / Collapse All toggle button

What it does

Acts as a context-aware toggle:

If not all folders are expanded, it displays **Expand All** and expands all folder nodes in the current tree.

If all folders are expanded, it displays **Collapse All** and collapses all folders.

Immediately persists expansion state so hierarchy navigation remains stable even if later operations trigger an editor refresh.

How to use

Click **Expand All** to fully open the hierarchy.

Click **Collapse All** to fully collapse the hierarchy back to a compact view.

Add Tag (“+”) button

What it does

Opens a tag creation popup for creating a new tag at any path you specify.

Supports creating deep paths and automatically creating missing parent segments (for example, creating Ability.Magic.Fire.Fireball will create missing parents if needed).

Ensures the newly created tag is visible by expanding its full hierarchy chain.

How to use

Click **+**.

Enter the full tag path.

Optionally enter a comment.

Click **Create**.

Tag Tree

The tag tree is the main browser for your hierarchy and displays tags as dot-delimited paths broken into parent/child nodes.

Hierarchical structure

What it does

Displays tags as a hierarchy based on the . separator.

Intermediate segments are represented as structural “folder” nodes.

Leaf nodes represent the final segment of each tag path.

How to use

Use foldouts to expand/collapse branches.

Scroll to navigate large hierarchies.

Use search to jump directly to a tag or category.

Alternating row backgrounds (zebra striping)

What it does

Improves readability in large lists by drawing alternating background rows.

Automatically selects appropriate tones for the current Unity editor skin (Light vs Pro).

How to use

No setup required; this is purely visual to improve scanning speed.

Tag labels with inline comments
What it does

Structural nodes display their name only.

Tag nodes display their name plus their comment (if present) for quick scanning:

TagName — Comment

How to use

Add comments to tags to describe intent, constraints, or usage notes so the Tag Manager becomes a lightweight "tag glossary" during production.

Each visible row includes a right-side action button. Which button appears depends on whether the node is a real tag or only a structural folder.

Tag nodes (":" menu)
What it does

Opens a context menu for operations that apply to that tag:

Add Subtag…

Update Comment…

Rename…

Delete (with children)…

How to use

Click ⋮ on a tag row.

Choose the desired action from the menu.

Structural folder nodes ("+" button)
What it does

Opens the Create Tag popup prefilled with the folder's prefix path so you can quickly create a child tag under that folder.

How to use

Click **+** on a folder row.

The popup opens with the prefix already filled (e.g., Ability.Magic.Fire.).

Enter the remaining leaf (or a full deeper path) and click **Create**.

Tag Actions Menu (⋮)
The context menu provides focused operations for an existing tag.

Add Subtag…
What it does

Opens the Create Tag popup with the selected tag's path used as the prefix.

How to use

Click **⋮** on the parent tag.

Select **Add Subtag…**

Enter the child tag path (prefilled prefix included).

Click **Create**.

Update Comment…
What it does

Opens a comment editor for the selected tag.

Updates the tag's comment field.

How to use

Click **⋮** on the tag.

Select **Update Comment…**

Edit the comment.

Click **Apply**.

Rename…
What it does

Renames the leaf segment of the selected tag (the final portion of the path).

Preserves navigation context by maintaining expansion state around the edited area.

How to use

Click **⋮** on the tag.

Select **Rename…**

Enter the new leaf name.

Click **Rename**.

Delete (with children)…
What it does

Deletes the selected tag and its entire subtree of subtags.

Clears expansion state entries for the removed subtree so the tree view stays consistent.

How to use

Click ⋮ on the tag.

Select **Delete (with children)…**

Confirm the deletion.

Create / Edit / Rename Popups

The Tag Manager uses small utility windows for focused edits.

Create Tag popup
What it does

Creates a new tag using:

Name (full path)

Comment (optional)

Includes a “Source” field that currently reflects the project’s default database (displayed but disabled).

Automatically focuses the Name field on open for fast keyboard-driven entry.

How to use

Enter the full path in **Name**.

Optionally enter a **Comment**.

Click **Create** to apply or **Cancel** to exit.

Edit Comment popup
What it does

Displays the tag path read-only and allows editing the comment.

Applies the new comment to the tag.

How to use

Enter or update the **Comment** text.

Click **Apply**.

Rename Tag popup
What it does

Renames the tag’s leaf segment.

How to use

Enter the new leaf value.

Click **Rename**.

GameplayTag

How it appears in the Inspector

When you expose a GameplayTag field in a MonoBehaviour or ScriptableObject, Unity displays it as a compact, database-backed selector rather than a raw text field.

You will typically see:

- A **dropdown selector** showing the currently selected tag path
- A **menu button** (⋮) for quick actions

This UI is driven by your project's **Default GameplayTagDatabase**, so tag selection is consistent across the entire project.

How to use it in the editor

Assign a tag

1. Click the **dropdown** for the GameplayTag field.
2. Choose a tag from the picker.

Result: the field stores the selection as a stable identifier and displays the tag's resolved path.

Clear a tag

1. Click the ⋮ button.
2. Select **Clear**.

Result: the field becomes unassigned and displays **(None)**.

Copy tag information (for debugging or documentation)

1. Click the ⋮ button.
2. Use the **Copy** options:
    a. **Copy/Path** copies the resolved tag path (e.g., Ability.Magic.Fire.Fireball)
    b. **Copy/Guid** copies the internal identifier (useful for troubleshooting or audits)

Locate the active database

1. Click the ⋮ button.
2. Select **Ping Default Database**.

Result: Unity highlights the Default GameplayTagDatabase in the Project window.

Common display states (what they mean)

- **(None)**: No tag is assigned.
- **A full tag path**: The tag is valid and resolves cleanly from the Default Database.
- **(Missing Tag)**: The tag reference can't be resolved—most commonly because no Default Database is configured, or the referenced tag no longer exists in the database.

GameplayTagContainer

How it appears in the Inspector

When you expose a GameplayTagContainer field, Unity displays it as a compact, database-backed container UI with:

- A **foldout header** showing the container name and tag count: MyTags (N)
- An **Edit** button for multi-select tag picking
- A **Clear** button to remove all tags
- When expanded, a **list of all tags** currently in the container

Tags are resolved against your project's **Default GameplayTagDatabase**, so what you see in the Inspector always reflects your project's canonical tag vocabulary.

How to use it in the editor

Add or remove tags (primary workflow)

1. Click **Edit**.
2. Use the multi-select picker to choose tags.
3. Apply/close the picker.

Result: the container updates to exactly match your selection.

Notes:

- This flow is the intended day-to-day authoring path.
- The container normalizes the stored tag set so it stays consistent (no duplicates, stable ordering).

Clear the container

1. Click **Clear**.

Result: all tags are removed and the container count becomes (0).

Browse tags in the expanded list

1. Expand the container foldout.
2. Review each tag row.

Each row shows:

- The resolved **tag path** (and any badge suffixes if the tag carries metadata)
- A tooltip with additional info when available

Remove a single tag quickly

1. Expand the container foldout.
2. Click **X** on the row you want to remove.

Result: that tag is removed from the container without opening the picker.

Per-tag quick actions (row menu)

1. Expand the container foldout.

2. Click the row's ⋮ menu.

Available actions typically include:

- **Copy/Path** (copies the resolved tag path)
- **Copy/Guid** (copies the internal identifier)
- Container actions (for convenience) such as removing the current row or clearing all
- **Ping Default Database** (when a default database is configured)

Visual status cues in the list

When expanded, each tag entry can communicate its status at a glance:

- **Invalid / Empty Guid**: the row contains an empty internal identifier (unassigned/corrupt entry).
- **Missing Tag**: the GUID does not resolve in the current Default Database (database not set, tag deleted, or moved to a different database).
- **Restricted / Deprecated indicators** (when supported by your database metadata): rows may display badge suffixes and subtle highlighting to warn you the tag is restricted or deprecated.

These cues are intended to help you catch broken references early during authoring.

Practical guidance

- Use GameplayTagContainer for any system that needs to evaluate *sets* of tags: requirements checks, AI decisions, equipment rules, state machines, interaction gating, etc.
- Prefer the **Edit** workflow for bulk changes; use **X** for quick, surgical removals.
- If you see **Missing Tag** rows, it usually means the container is referencing tags that are not present in the currently configured Default Database.

GameplayTagQuery

How it appears in the Inspector

When you expose a GameplayTagQuery field, it displays as a foldout containing:

1. A **status line** summarizing definition/compiled state
2. A row of **workflow buttons** to create/edit/compile/clear compiled data

This UI is intentionally designed to make it obvious whether the query is ready for runtime evaluation and whether the compiled data matches the current authored definition.

The two representations you manage (what the UI is really controlling)

Definition (authoring)

- This is the editable "source of truth" tree (a managed-reference definition object).
- You edit it through the dedicated query editor window.

Compiled data (runtime)

- This is the generated, deterministic runtime representation used for evaluation.
- It includes:
    - Compiled node buffer
    - Compiled tag dictionary (GUID list)
    - A compiled definition hash used to detect staleness

You generally author the definition and keep the compiled data synchronized.

Status line (what it means)

The Inspector status line tells you whether your query is valid and synchronized:

- **Definition: (none) | Compiled: (empty → matches all)**
  No definition exists and there is no compiled data. This effectively behaves as a "no restrictions" query.
- **Definition: (none) | Compiled: … [Compiled-only]**
  Compiled buffers exist but there is no authoring definition attached.
- **Definition: present | Compiled: … [Up-to-date]**
  Definition exists and the compiled representation matches it (ready for runtime use).
- **Definition: present | Compiled: … [STALE]**
  Definition exists but the compiled data no longer matches the definition (you edited the definition since the last compile, or compilation has not run yet).
- **Definition: present [INVALID] …**
  The definition exists but cannot be hashed/validated safely (typically indicates corrupt or malformed data). The field remains safe to draw; you must correct or reset the definition through the editor workflow.

How to use it in the editor

Create a new query definition

1. Expand the GameplayTagQuery field.
2. Click **Create Def**.

Result:

- A default definition tree is created (authoring becomes available).
- The system schedules a compile so the query is not "born stale."

Use this when you are adding a query field for the first time and want to start authoring logic.

Edit the query logic

1. Expand the field.
2. Click **Edit...**

Result:

- Opens the dedicated **GameplayTag Query Editor** window bound to this specific field.

Use this to build the query tree (groups, leaves, match types, etc.). The Inspector field remains your status + control surface.

Compile the definition to runtime data

1. Expand the field.
2. Click **Compile**.

Result:

- The current definition is validated and compiled into the runtime buffers.
- The status should become **[Up-to-date]** if compilation succeeds.

Use this when:

- You have Auto-Compile disabled, or
- You want to force a compile immediately before testing/building.

Clear compiled data

1. Expand the field.
2. Click **Clear Compiled**.

Result:

- Removes the compiled node buffer, compiled dictionary, and compiled hash.

This is primarily used when you want to intentionally strip compiled content and force a clean rebuild later.

Practical guidance

- During normal authoring, keep the status at **[Up-to-date]** before you rely on the query in runtime logic.
- If you see **[STALE]**, compile (or enable Auto-Compile in your project settings/workflow).
- Treat the Inspector UI as the operational workflow surface: create definition, open editor window, compile, and verify status.

Tag Picker Dropdown

What it is

The **Tag Picker Dropdown** is the standard HierarchicalTagForge UI for selecting gameplay tags in the editor. It appears as a **popup window** (a lightweight dropdown) that hosts a searchable **hierarchical TreeView** of tags sourced from your **default GameplayTagDatabase**.

It is used in two primary contexts:

- **Single-select**: picking one GameplayTag
- **Multi-select**: editing a GameplayTagContainer (selecting many tags)

The picker is designed to be fast, compact, and safe to use inside inspectors and editor tools.

Requirements and prerequisites

Default database must be set

The picker relies on the project's **default GameplayTagDatabase**. If no default database is configured, the picker does not open and instead shows a dialog explaining how to fix the setup (typically: open Tag Manager and set a default database).

Database should be baked for full functionality

The picker reads tags from the database's **baked data**. If the database is not baked (or has no baked tags), the picker opens but shows a **message row** such as:

- "(Database is not baked)"
- "(No tags in database)"

In that state, most meaningful selection actions are unavailable because there is no stable baked tag tree to display.

Layout and controls

When opened, the dropdown has four functional areas:

1) Search field

A search bar at the top filters the tree.

Search behavior

- Searches against the **full path**, not just the leaf segment.
    - Searching "Ability.Fire" or "Fire" will match the tag as long as the substring appears anywhere in the full path.
- Matching is **case-insensitive** and uses an ordinal comparison (stable and fast).

Important UX note

When a TreeView is filtered, Unity may effectively present a flattened subset of results. In that mode, **Expand/Collapse may not appear to do anything**, which is expected behavior for Unity's TreeView filtering.

2) Expand All / Collapse All

Two buttons toggle hierarchy expansion.

- **Expand All**: expands all top-level branches recursively.
- **Collapse All**: collapses all branches recursively.

These are disabled when the picker does not have a valid baked tree (unbaked database, no tags, etc.).

3) Tag tree (hierarchical list)
The main body is a TreeView representing tags in their natural hierarchy.

Display

- Each row shows the **leaf name** (last segment of the path).
- When metadata is available, the label includes **badges** (for example restricted/deprecated markers), and the row provides a richer tooltip.

Tooltips
Hovering a row can show details such as:

- Full path
- Tag GUID
- Status metadata (restricted/deprecated)
- Additional tag info surfaced from the database's editor read helpers

Tooltips are intended to help users distinguish similarly named tags across branches.

4) Multi-select action bar (multi mode only)
When editing containers, a bottom row appears:

- **Clear**: empties the current selection state in the picker UI.
- **Apply**: commits the current selection back to the caller (returns a sorted, de-duplicated list).
- **Cancel**: closes without applying changes.

Apply is disabled when the picker has no valid baked tree.

Single-select vs Multi-select behavior

Single-select mode (GameplayTag)
This mode is used when a field represents a single tag.

Selection semantics

- Clicking a tag selects it, immediately invokes the selection callback, and closes the popup.
- Clicking the **currently-selected** tag toggles it off (clears it), then closes the popup.
- Selecting **(None)** clears the field and closes the popup.

(None) row

- Only present in **single-select** mode.
- Represents "no tag assigned."

Multi-select mode (GameplayTagContainer)
This mode is used when selecting multiple tags into a container.

Selection semantics

- Clicking tags toggles them on/off inside the picker, but **does not close** the popup.
- The picker maintains selection state internally until you click **Apply** or **Cancel**.
- **Apply** returns:
    - a list that is **filtered** (no empty GUIDs),
    - **sorted deterministically** (by GUID high/low components),
    - and **de-duplicated**.

This ensures stable serialization output and avoids accidental duplicates.

Visual indicators and row interaction

Checkbox/toggle indicator
Each tag row displays a right-aligned toggle state:

- In **single mode**, the toggle indicates which tag is selected (or (None)).
- In **multi mode**, the toggle indicates membership in the selection set.

Double-click behavior
Double-clicking a tag row is treated as an activation:

- In single mode: selects/clears and closes, just like clicking.
- In multi mode: toggles selection, just like clicking.

What the picker uses as "identity"

Stable identity is GUID-based
While the TreeView row IDs are based on **baked TagId** for efficiency and hierarchy building, the **authoritative identity returned to the user is the tag GUID** (plus the resolved full path when available).

Special rows use negative internal IDs
The picker includes "special" rows like:

- (None)
- informational message rows (unbaked / no tags)

These rows use **negative IDs** to guarantee they never collide with real baked tag IDs (which are positive). This is an internal correctness detail that prevents broken selection, expansion state, or incorrect item lookups in Unity's TreeView.

Expected user workflow

Picking a single tag

1. Click the dropdown field.
2. (Optional) type in the search bar.

3. Click a tag.
4. Popup closes immediately; the GameplayTag field updates.

Editing a container

1. Click **Edit** (or equivalent) on the container UI.
2. Search and toggle multiple tags.
3. Click **Apply** to commit the selection (or **Cancel** to discard).
4. Container updates using a stable sorted/deduped set.

Troubleshooting and common issues

"No default database set"

- The picker will not open.
- Fix by setting a default database in your Tag Manager / Project Settings workflow.

"Database is not baked"

- The picker opens, but shows a message row instead of tags.
- Bake the database to generate baked data, then reopen the picker.

"Expand/Collapse doesn't work during search"

- Normal TreeView behavior when filtered. Clear the search field to restore normal hierarchy expansion interaction.

GameplayTagQuery Editor Window

What the Query Editor is

The **GameplayTag Query Editor Window** is a dedicated editor window for building and editing a **GameplayTagQuery** as a structured definition tree (a readable, hierarchical set of logical rules). It exists to make authoring complex tag logic practical and consistent without manually managing nested data in the inspector.

A query definition is composed of **nodes** arranged as a tree:

Group nodes combine child expressions (e.g., "All of these must be true").

Leaf nodes evaluate selected tags (e.g., "Any of these tags match").

Constant nodes are always true/false (useful for testing or temporary disable/enable cases).

This window edits one bound query field on one specific object at a time (single-target editing), ensuring edits apply deterministically.

Opening the Query Editor

You open this window from a **GameplayTagQuery** field in the inspector (via the query drawer's editor/open control).

When opened, the window binds to:

The **target object** that owns the query field

The **property path** of the query field on that target object

The window title appears as **"GameplayTag Query"** and opens as a utility window.

Header Toolbar

Title
The left side displays **GameplayTag Query** for context.

Auto-Compile toggle
What it does

Enables or disables automatic compilation of your authoring tree after edits.

This is a project-scoped preference (intended to be consistent for the team/project).

How to use

Turn **Auto-Compile** on if you want compiled query data kept up to date automatically while editing.

Turn it off if you prefer to compile only when you explicitly choose.

Compile button
What it does

Manually compiles the current query definition into its compiled/runtime representation.

How to use

Make edits to the query tree.

Click **Compile** to update the compiled form.

Tag Source Banner

Tag Source display
What it does

Shows the active/default **GameplayTagDatabase** used for resolving and displaying tag selections in the editor.

Ping button
What it does

Highlights ("pings") the current database asset and selects it in the project.

How to use

Click **Ping** to jump to the database asset in the Project window.

Creating and Editing a Definition

Definition creation
If a query has no authoring definition yet, the window presents a **Create Definition** action.

What it does

Creates a new default query definition structure and attaches it to the query.

How to use

Click **Create Definition**.

The window initializes a default root node and node list.

Description field
What it does

Provides a free-form description text area stored with the query definition.

Intended for human-readable intent such as: what the query represents, where it's used, or design constraints.

How to use

Enter a short explanation of the query's purpose (recommended for shared projects and long-lived content).

Authoring the Query Tree

Node layout and navigation
The query is displayed as a tree of nodes, with each node shown inside a boxed section. Nodes are drawn top-to-bottom in hierarchical order with indentation to represent depth.

Each node has:

A header showing its **index** and **type**

A foldout to expand/collapse node details

Context actions depending on node role (root vs non-root, group vs leaf)

Newly created nodes are visually marked briefly so you can immediately identify what was added.

Root node
The query definition has a single **Root** node index that represents the starting point for evaluation.

Make Root

What it does

Sets the selected node to be the root of the query (non-root nodes only).

How to use

Expand a node and click **Make Root** to designate it as the query's root.

Node Type
Each node includes a **Type** field that defines what the node represents.

Common categories used in authoring:

Group expression nodes

Combine child expressions into a single result.

The window treats these as "group" nodes that hold a list of child node indices.

Leaf tag-match nodes

Evaluate a list of selected tags according to the node's match behavior.

Constant nodes

Evaluate as always true or always false and do not contain tags or children.

To change a node's function, change its **Type**.

Group Nodes: Children Management

Group nodes display a **Children** header and controls for building out the expression structure.

Add buttons
What they do

Add Leaf: Adds a new leaf node (tag-based evaluation) as a child of the current group.

Add Group: Adds a new group node as a child of the current group.

Add True / Add False: Adds constant nodes as children.

How to use

Expand a group node.

Use **Add Leaf** or **Add Group** to build the logic.

Use **Add True/False** when you need an always-on/off clause (commonly used during iteration).

Child ordering and removal
Each child row provides:

Up / Down: Reorders children within the group.

Remove: Removes the child reference from the group's child list.

How to use

Use **Up/Down** to reorder evaluation structure for readability and organization.

Use **Remove** to detach a child from a group.

After the child list, the editor draws each child node recursively, so you can edit the entire subtree in one view.

Leaf Nodes: Tag Lists

Leaf nodes are where you define tag-based checks.

Match field
What it does

Controls how the leaf interprets selected tags (match semantics).

This is typically used to define whether tags must match exactly or via other match rules supported by your query system.

How to use

Expand a leaf node.

Set **Match** to the desired matching behavior.

Edit the tag list to reflect the tags being evaluated.

Tags list (Edit / Clear / Remove)
Leaf nodes show a **Tags** line with controls:

Edit

What it does

Opens a tag picker and allows selecting multiple tags for this leaf node.

How to use

Click **Edit**.

Select the desired tags from the picker.

Apply the selection to update the leaf node's tag list.

Clear

What it does

Removes all tags from the leaf node.

How to use

Click **Clear** to reset the tag list to empty.

Per-tag remove (“X”)

What it does

Removes a single tag entry from the leaf node without reopening the picker.

How to use

Click **X** next to the tag you want removed.

Tag display
Each selected entry is displayed as a resolved tag path (when resolvable through the active tag database). Tooltips provide additional tag details for quick inspection.

Definition Utilities

Validate
What it does

Runs a validation pass on the definition tree and reports whether it is structurally valid.

How to use

Click **Validate** when you want a quick structural check of the current definition.

Prune Orphans
What it does

Removes nodes that exist in the node list but are not reachable from the current root (unreferenced nodes).

How to use

Click **Prune Orphans** to clean the definition’s internal node list and keep it minimal and readable.

Reset and Close

Reset Definition To Default
What it does

Resets the entire query definition back to its default initial structure.

How to use

Click **Reset Definition To Default** when you want to discard the current tree and start from a clean default.

Close
Closes the query editor window.

Assets

This section documents the MonoBehaviour components, ScriptableObject assets, and supporting data files that **HierarchicalTagForge** uses to store, serialize, and ship tag-related data. It explains what each asset type represents, when to create it, how it is referenced by runtime systems and editor tooling, and how these assets fit into the overall workflow from authoring to baking to runtime lookup and querying.

GameplayTagOwner

What it is

GameplayTagOwner is the standard **runtime tag owner** for HierarchicalTagForge. You attach it to any GameObject that needs to **own a GameplayTagContainer** and expose:

- **Query APIs** (HasTag / HasAny / HasAll)
- **Mutation APIs** (Add / Remove / Bulk Add / Bulk Remove / Clear)
- **Change notifications** (C# event + Inspector UnityEvents)

This is intended to be the core bridging point between “tags as data” and “systems that react to tags” (AI, abilities, gating, UI indicators, triggers, filtering, etc.).

When to use it

Use GameplayTagOwner when:

- You want tags to be **authorable in the Inspector** (serialized container).
- You want other systems to query tags without knowing your internal data layout.
- You want reliable **change notifications** when tags mutate.
- You want a single owner API that mirrors common GAS-style tag ownership patterns.

Typical owners:

- Units, NPCs, enemies
- Items, pickups, interactables
- Ability or status effect carriers
- UI proxies (e.g., a widget representing an entity state)
- World objects you want to filter or gate

Inspector authoring

Owned tags

The component serializes a GameplayTagContainer field:

- Tags (backed by _tags)

Because it is serialized, you can:

- Pre-seed tags on prefabs

- Inspect tag state during play mode
- Use the same Tag Picker UI as other tag fields (single/multi selection)

Important behavioral rule

Even though the container is serialized, **mutations should go through the component API** (Add/Remove/etc.) so that notifications and versioning remain correct.

Change notifications

GameplayTagComponent supports two notification channels:

1) Runtime C# event (primary runtime hook)

public event GameplayTagsChangedHandler TagsChanged;

Subscribers receive:

- The GameplayTagComponent instance that changed
- A GameplayTagChangeSet payload containing:
    - Added (container)
    - Removed (container)

This is the preferred mechanism for code-driven systems (AI state machines, ability systems, UI controllers, etc.).

2) Inspector events (UnityEvent, no code required)

Under **Events (Inspector)** the component exposes:

On Tags Changed

_onTagsChanged : UnityEvent

- Fired after any **successful** mutation.
- No payload.
- Intended for simple "refresh something" responses in the editor:
    - refresh UI
    - recompute a display
    - re-run a query
    - update indicators

On Tags Changed Detailed

_onTagsChangedDetailed : GameplayTagChangeSetUnityEvent

- Fired after any **successful** mutation.
- Payload is GameplayTagChangeSet:
    - Added/Removed containers

Use this when you want Inspector-bound listeners to know *what* changed rather than only "something changed".

Query API (read-only checks)

These methods query the owned tags. All default to TagMatchType.IncludeDescendants unless specified.

HasMatchingGameplayTag

Checks a single tag against the owned set.

HasAnyMatchingGameplayTags

Checks whether any tag in a provided container matches the owned set.

HasAllMatchingGameplayTags

Checks whether all tags in a provided container match the owned set.

Match semantics

TagMatchType controls whether matching is strict/exact or hierarchical (descendant/ancestor inclusion). You should document TagMatchType once centrally and reference it here.

Mutation API (safe modifications)

All mutation methods share an important guarantee:

- **Notifications only fire after successful mutations.**
- If the operation is a **no-op**, nothing is fired and version does not advance.

This avoids "event spam" and makes TagsVersion meaningful.

AddGameplayTag(tag)

- Rejects invalid tags (empty GUID).
- Adds tag if not present.
- On success:
    - Added set contains that tag
    - Removed set is empty
    - Events fire

RemoveGameplayTag(tag)

- Rejects invalid tags (empty GUID).
- Removes tag if present.
- On success:
    - Removed set contains that tag
    - Added set is empty
    - Events fire

AddGameplayTags(container)

Bulk add.

- If nothing is added, returns 0 and does not notify.
- On success, Added set is computed as **(after - before)** using canonical sorted GUID arrays.

RemoveGameplayTags(container)
Bulk remove.

- If nothing is removed, returns 0 and does not notify.
- On success, Removed set is computed as **(before - after)** using canonical sorted GUID arrays.

ClearGameplayTags()

- If already empty: no-op.
- Otherwise:
    - Removed set contains the entire previous set
    - Added set is empty
    - Events fire

Direct container access (advanced)

public ref GameplayTagContainer Tags => ref _tags;

This is intentionally provided for advanced systems that need to operate on the container by ref (e.g., low-level runtime code, wrapper adapters, or performance-sensitive batching).

Critical rule

If you mutate via Tags directly, you bypass the component's ability to compute Added/Removed sets and bypass automatic notifications. In that scenario you must call:

NotifyTagsChanged()

This publishes a change notification with:

- Added = empty
- Removed = empty

This is a deliberate compromise:

- Direct ref mutation can be arbitrary and may not preserve enough info to compute diffs cheaply or correctly.
- Observers still get notified that "something changed."

Diff computation strategy (why bulk operations are efficient)

For bulk adds/removes, the component computes changes using a linear merge-walk over **sorted, unique GUID arrays** (GameplayTagContainer.TagGuids is assumed canonical).

Key properties:

- O(n) difference computation
- No per-tag allocations
- Only allocates an output array when there are actual diffs
- Uses a deterministic GUID ordering (Hi/Lo) so results are stable

This is why the component can provide Added/Removed sets in bulk operations without sacrificing performance.

Recommended usage patterns

Pattern 1: Tag-driven UI refresh

- Bind UI scripts to TagsChanged (code) or _onTagsChanged (Inspector).

Pattern 2: Systems that need precise deltas

- Subscribe to TagsChanged or _onTagsChangedDetailed.
- Use Added and Removed to apply incremental updates (e.g., add/remove status icons).

Pattern 3: High-frequency checks (AI / combat rules)

- Prefer HasMatchingGameplayTag/HasAny/HasAll for direct queries.
- Avoid pulling full containers unless you actually need them.

Common Pitfalls:

- **Do not mutate _tags directly** from external code. Always go through the component mutation API unless you deliberately accept "manual notify + no diff payload".
- **Direct ref mutation requires NotifyTagsChanged()** or observers won't react.
- **Invalid tags (empty GUID) are rejected** in single-tag add/remove calls.

GameplayTagAssetUtility (Query/Mutation Helpers for Arbitrary Unity Objects)

What it is

GameplayTagAssetUtility is your high-level "do tags exist on this object?" utility. It solves a common Unity problem:

"I have a UnityEngine.Object (maybe a component, maybe a gameobject, maybe an asset).
Can I query tags from it? Can I mutate tags on it?"

It consolidates:

- tag owner resolution
- querying (strict + safe variants)
- optional mutation (try-add/remove/clear, bulk variants)
- observable detection/version reads

Resolution model (how it finds tags)

The utility prefers interfaces and falls back to common Unity patterns.

Owned Tags fetch resolution order:

1. IGameplayTagAssetQuery (preferred; can query and fetch tags)
2. IGameplayTagAsset (fetch only)
3. IGameplayTagAsset on a component attached to a Component/GameObject
4. GameplayTagComponent on a Component/GameObject

If no owner is found, the output container remains empty and the call fails gracefully (no exceptions).

Query families: Strict vs Safe (why both exist)

Strict queries

- HasMatchingGameplayTag
- HasAnyMatchingGameplayTags
- HasAllMatchingGameplayTags

Strict variants:

- prefer IGameplayTagAssetQuery (best path)
- otherwise pull a container snapshot and evaluate via GameplayTagContainer
- may rely on **registry-backed hierarchical matching** (depending on your container implementation)

Use strict when:

- you expect GameplayTagRegistry to be initialized normally
- you want full hierarchy semantics when matchType is descendant-based

Safe queries

- HasMatchingGameplayTagSafe
- HasAnyMatchingGameplayTagsSafe
- HasAllMatchingGameplayTagsSafe

Safe variants:

- never require the registry for hierarchy semantics
- if the requested matchType is hierarchical and the registry is missing/uninitialized, they **degrade to Exact matching** (fail-closed)

Use safe when:

- you are early in boot / tool mode / unusual runtime lifecycle
- you cannot guarantee registry initialization
- you want deterministic “no accidental broad matches” behavior

This “fail-closed” approach is a real production safeguard: it prevents cases where “IncludeDescendants” becomes effectively uncontrolled because hierarchy data isn’t loaded.

Mutation helpers

Mutation methods are “Try” style and never throw:

- TryAddGameplayTag

- TryRemoveGameplayTag
- TryAddGameplayTags
- TryRemoveGameplayTags
- TryClearGameplayTags

Mutation preference:

1. mutate through a resolved IGameplayTagAssetQuery if possible
2. else mutate through a resolved GameplayTagComponent
3. else fail (returns false / 0)

Important behavioral note:

- TryClearGameplayTags returns true if it invoked a supported surface; it does not guarantee the set was non-empty.

Observable helpers

- TryAsObservable(obj, out observable)
- TryGetTagsVersion(obj, out version)

These are used by systems that want to subscribe or track versions but are given an unknown Unity reference.

GameplayTagQueryAsset (Reusable Query Definition Asset)

What it is

GameplayTagQueryAsset is a lightweight ScriptableObject wrapper around a GameplayTagQuery.

Use cases:

- Centralize complex boolean queries as content assets
- Reuse the same query across multiple systems (AI, abilities, interactables, UI gating)
- Enable drag/drop assignment in inspectors instead of embedding queries inline everywhere

What it contains

- _query : GameplayTagQuery (serialized)
- public GameplayTagQuery Query => _query;

This wrapper deliberately keeps behavior minimal:

- It does not implement evaluation itself
- It does not duplicate compiled/baked query data

Practical usage pattern

- Designers author a query in the asset.
- Runtime systems read Query and evaluate it against a GameplayTagContainer snapshot (or via a tag owner that can supply tags).

Best practice note

If your GameplayTagQuery supports compilation/baking:

- the compiled data should live inside GameplayTagQuery or its supporting compiler/registry system
- the asset should remain a stable serialized “definition” container

That keeps the asset stable, diff-friendly, and avoids mismatched compiled state.

GameplayTagRequirementsAsset (Reusable Requirements Rule Asset)

What it is

GameplayTagRequirementsAsset is the asset wrapper for GameplayTagRequirements. It exists so designers can author reusable GAS-style gating rules and assign them via drag/drop.

Use cases:

- Common gating rules reused across many interactables/abilities:
  - “Require All: X, Y”
  - “Require Any: A, B”
  - “Block Any: Stunned, Silenced”
  - plus optional query (if used in your requirements struct)
- Shareable “rulesets” as project content

What it contains

- _description : string (editor-facing, non-functional)
- _requirements : GameplayTagRequirements
- public GameplayTagRequirements Requirements => _requirements;

Design intent: description is non-operative

The _description is intentionally not used for evaluation; it is for:

- authoring clarity
- tooltips / inspector readability
- auditability (“why does this gate exist?”)

That is a good content pipeline feature: it reduces tribal knowledge.

Practical usage pattern

- Assign this asset anywhere you want consistent gating logic.
- In editors/drawers (like your Observer Rule drawer), if an asset is assigned, inline requirements are typically disabled to avoid “two sources of truth” ambiguity.

Project Settings

This section explains the project-wide configuration options that control how **HierarchicalTagForge** behaves in both the editor and at runtime. It covers where these settings live, what each option changes (authoring rules, defaults, automation behavior, validation, and generation workflows), and how to choose sensible values for different team pipelines and build environments.

Where to find the settings

HierarchicalTagForge stores its project-wide configuration in a project settings asset, exposed through Unity's Project Settings UI.

You can access it here:

Edit → Project Settings → CarryForge → HierarchicalTagForge → Gameplay Tags

These settings are project-scoped, meaning they apply consistently for the entire project and can be committed to source control for team-wide consistency.

Definitions

Auto Create Default Database

What it does

When enabled, HierarchicalTagForge can automatically create and wire a default database if none is currently configured.

How to use

Leave this on for most projects, especially early in development or when onboarding new team members.

Turn it off if your project requires strict control over asset creation (e.g., locked-down production pipelines).

Auto Generate Code On Bake

What it does

When enabled, baking a database can automatically trigger **code generation** (for generated constants / tag reference code).

How to use

Enable this if your workflow relies on generated code references (recommended for codebases that want typo-proof tag access).

Disable it if you prefer to manually control when generated code updates occur.

Auto Bake And Generate On Tag Edit

What it does

When enabled, edits made in the **Tag Manager** automatically trigger bake (and generation, if enabled) using the system's scheduling/debouncing rules.

How to use

Keep enabled for the most "it just stays in sync" authoring workflow.

Disable if you want to batch large tag edits and bake manually at controlled times.

Warn On Invalid Tags

What it does

When enabled, the editor surfaces warnings when tag paths contain invalid characters or formatting.

How to use

Keep enabled in most projects to catch mistakes early during authoring.

Clear Invalid Tags

What it does

Controls how invalid authored tags are handled during baking:

Off (strict): invalid tags are treated as errors (preferred when enforcing correctness).

On (permissive): invalid tags are removed during bake instead of failing the bake.

How to use

Leave **off** in disciplined production pipelines where you want invalid authoring to be corrected.

Turn **on** only if your project prioritizes keeping builds moving and you accept that invalid entries will be dropped.

Invalid Tag Characters

What it does

Defines the set of characters that are not allowed in tag paths.

Dots (.) are hierarchy separators and are always allowed; this list typically includes whitespace.

How to use

Use this to enforce a consistent tag naming policy across your project.

Most teams leave this set to block whitespace and control characters.

Auto Compile Query On Definition Change

What it does

When enabled, compiled query data is automatically updated when you change a **GameplayTagQueryDefinition** (authoring tree).

How to use

Enable this for a smooth authoring experience where queries remain compiled and ready without manual compile steps.

Disable it if you want manual control (for example, when performing large structural changes and compiling only at the end).

Fail Build On Stale Queries

What it does

When enabled, build validation fails if any **GameplayTagQuery** assets contain stale compiled data relative to their authoring definition.

How to use

Enable this in production/CI to ensure the shipped build always reflects current query definitions.

Disable only if your pipeline does not require strict validation at build time.

Auto Fix Stale Queries On Build

What it does

When enabled, stale compiled query data can be automatically rebuilt during build validation.

How to use

Enable this if you want CI/build steps to self-heal stale query compiled data automatically.

Leave disabled if you want builds to fail and force explicit correction (strict enforcement).

Presets

Recommended default (most teams)

Auto Create Default Database: **On**

Auto Bake And Generate On Tag Edit: **On**

Auto Generate Code On Bake: **On** (if using generated constants)

Warn On Invalid Tags: **On**

Clear Invalid Tags: **Off**

Auto Compile Query On Definition Change: **On**

Fail Build On Stale Queries: **On**

Auto Fix Stale Queries On Build: **Off** (strict) or **On** (CI convenience)

Strict production / locked pipelines

Auto Create Default Database: **Off**

Auto Bake And Generate On Tag Edit: **Off** (manual bake)

Warn On Invalid Tags: **On**

Clear Invalid Tags: **Off**

Fail Build On Stale Queries: **On**

Auto Fix Stale Queries On Build: **Off**

Menu Commands

HierarchicalTagForge provides a small set of selection-driven Unity Editor menu commands under **Tools**. These commands operate on the **currently selected GameplayTagDatabase asset** in the Project window and are intended to support a consistent "author → bake → (optionally) generate code" workflow.

Bake GameplayTagDatabase

Menu Command

Tools → CarryForge → HierarchicalTagForge → BakeSelectedTagDatabase

What it does

"Baking" is the editor build step that validates and compiles authored tag data into the database's derived/baked runtime tables used for fast lookups and hierarchical checks.

How to use

1. In the **Project** window, select the **GameplayTagDatabase** asset you want to bake.
2. Run the Bake command from the Tools menu.
3. Review warnings/errors in the Console (the bake report logs them using the database asset as context).

Notes

- The menu command is intentionally selection-driven so projects can support multiple databases.
- If no database is selected, a dialog explains what to select.
- The bake is invoked with includeCodeDeclaredTags: true, meaning code-declared tags (via HierarchicalTagAttribute) are included when supported.
- After a **successful** bake (no errors), code generation **may** run automatically if your project setting **Auto Generate Code On Bake** is enabled.

Generate Tag Code

Menu Command

Tools → CarryForge → HierarchicalTagForge → Generate Selected Database Tag Code

What it does

Generates a strongly-typed C# source file (.g.cs) containing a static class of tag constants/helpers produced from the **baked** database. The generated file is runtime-safe (no UnityEditor references) and is imported immediately, so Unity compiles it.

How to use

1. In the **Project** window, select the **GameplayTagDatabase** asset you want to generate code from.
2. Run the Generate Tag Code command from the Tools menu.
3. Unity will write the output file into the package's configured output directory and force an import so it compiles right away.

Notes

- If the selected database is missing baked data, **code generation will force a bake** to ensure consistent baked tables and IDs are available to the generator (forceBakeIfMissing: true).
- If no valid database is selected, the command shows an instructional dialog (and the menu item is also guarded by a validation function).
- Generation is deterministic by canonical tag path ordering and uses GUID-first identity principles (GUID is authoritative; runtime IDs are treated as a cache).

API Usage and Examples

NOTE: If you are looking for the quickest way to understand how to use this asset via code and are looking to create a demo/portfolio project or prototype, go directly to the Demo Scene in your Unity Project and review the Demo Scene section of this documentation.

This section provides practical, code-focused guidance for integrating **HierarchicalTagForge** into real gameplay systems. It summarizes the key runtime APIs, outlines performance-minded usage patterns, and includes concise examples for common tasks such as authoring tags in code, storing and mutating tag sets, evaluating queries and requirements, and using utility helpers to work with tag owners across diverse Unity object types.

Performance and Best Practices

HierarchicalTagForge is designed to support high-frequency runtime queries (AI decisions, ability gating, item usage, state machines, UI visibility rules) without forcing allocations or heavy object graphs. The guidance below is the difference between 'it works' and 'it scales'.

Key performance rules:

- Prefer data that can be evaluated as value types (GameplayTag, GameplayTagContainer, GameplayTagQuery, GameplayTagRequirements).
- Avoid repeatedly materializing tag containers for the same owner; cache owned tags in a GameplayTagContainer field when possible.
- When evaluating a GameplayTagQuery frequently, prefer query.Matches(ref tags) to allow both the query and container to reuse their runtime caches.
- Remember that GameplayTagContainer is a struct: mutating a copy does not mutate the original. When your owner stores a container field, mutate via a ref accessor or a ref local.
- Use TagMatchType.Exact when you do not explicitly need hierarchy semantics, as it is the most performant. Exact matching is GUID-based and does not require a registry.
- If you rely on hierarchical matching (IncludeDescendants / IncludeAncestors), ensure the GameplayTagRegistry is initialized early in your app lifecycle.
- **HasMatching** is the most performant of the query semantics, use this when checking for single tag comparisons.

Cache and registry notes:

- GameplayTag uses a stable GUID as its authoritative identity; any runtime integer ID is a derived cache that can change across rebakes.
- GameplayTagContainer stores authoritative GUIDs and may store a derived baked-ID cache for hierarchical evaluation. This cache is automatically rebuilt when needed, but only when the registry is initialized.
- When the registry is missing or not initialized, hierarchical checks intentionally fail closed (return false) to avoid accidental broad matches.

Public Runtime API Overview

The types below form the core public runtime surface for querying and operating on gameplay tags. Most projects can be implemented using these alone.

Core value types:

- GameplayTag: a lightweight handle for a single tag (stable GUID identity, optional runtime cache ID).
- GameplayTagContainer: canonical storage for a set of tags (sorted + unique), with exact and hierarchical matching helpers.
- GameplayTagQuery: a compiled boolean expression evaluated against a GameplayTagContainer snapshot.
- GameplayTagRequirements: a reusable 'gate' (RequireAll / RequireAny / BlockAny + optional query) evaluated against an owner.

Interfaces and utilities:

- IGameplayTagAsset: minimal allocation-free interface to expose owned tags.
- IGameplayTagAssetQuery: optional fast-path interface that supports direct matching and mutation without forcing container copies.
- IGameplayTagAssetObservable: extension that exposes an event when tags change (useful for UI/state synchronization).
- GameplayTagAssetUtility: convenience helpers that query/mutate tags on arbitrary UnityEngine.Object references.

Match semantics:

- TagMatchType: controls how a single tag comparison is evaluated (Exact vs hierarchy-aware modes).
- TagMatchOptions / TagMatchPresets: flag-based convenience options for systems that prefer bitfields.

Authoring Tags in Code (Attribute Declarations)

HierarchicalTagForge supports a workflow where tags can be declared in code using the HierarchicalTagAttribute. Declarations are discovered by editor tooling at edit time (for example, via UnityEditor.TypeCache) and baked into your GameplayTagDatabase. This is useful for teams that want compile-time discoverability of important tags, Unreal-style “static tag catalogs,” and a repeatable pipeline across branches and CI.

What the attribute does

[HierarchicalTag("Some.Path", ...)] declares a tag path for registration at edit time. The attribute can also carry metadata used by editor tooling and the database:

- DevComment: Optional developer comment/tooltip stored as tag metadata.
- Restricted: Marks the tag as restricted (enforcement is defined by your database’s restricted roots/policies).
- RestrictedOwner: Optional ownership hint used by policy enforcement (assembly/package/team name, etc.).

Recommended declaration pattern

1. Create a dedicated tag catalog class (for example, GameTags, AbilityTags, AITags).
2. Declare tags as public const string fields (or properties) containing the full tag path.
3. Annotate each field/property with [HierarchicalTag("Full.Path.Here", ...)].
4. Run your manual bake so the declared tags are registered into the database and become available everywhere (inspector pickers, global lookups, etc.).
5. Use the string constant in code when you need a stable, refactor-friendly way to reference tag paths (and/or use the baked tag handles when your workflow calls for it).

Example: a dedicated tag catalog class.

```
using CarryForge.HierarchicalTagForge;

public static class GameTags
{
    [HierarchicalTag("Damage.Magic.Fire", DevComment = "Fire-based magic damage.")]
    public const string Damage_Magic_Fire = "Damage.Magic.Fire";

    [HierarchicalTag("Effect.Poisoned", DevComment = "A poison-based effect.")]
    public const string Effect_Poisoned = "Effect.Poisoned";

    // Example of restricted tag metadata (policy enforcement happens in editor tooling):
    [HierarchicalTag("Internal.DebugOnly", Restricted = true, RestrictedOwner =
"CarryForge")]
    public const string Internal_DebugOnly = "Internal.DebugOnly";
}
```

Notes and constraints:

- **Manual bake is required.** In this version of **HierarchicalTagForge**, declared tags are not automatically registered at runtime; you must **manually Bake** for code-declared tags to become visible in the inspector and to be globally usable via the database/registry. See **MenuCommands** for more details on how to manually bake in the editor.
- **No exceptions at declaration time.** The attribute constructor intentionally does not throw; validation is performed during the editor bake, which reports errors/warnings.
- **Field/property targets only.** The attribute is valid on fields and properties (not methods/classes), and only one attribute instance is allowed per member.

Runtime Querying and Common Patterns

This section provides practical runtime patterns using the primary public types: GameplayTag, GameplayTagContainer, GameplayTagQuery, GameplayTagRequirements, GameplayTagAssetUtility, and TagMatchType.

GameplayTag

GameplayTag is a lightweight handle representing a single tag. Its authoritative identity is a stable GUID. A runtime cache ID may be present for performance, but user code should treat the GUID as authoritative.

- Use tag.IsSet / tag.IsValid to check whether the tag has a non-empty identity.
- Use tag.IsRegistered to verify the tag exists in the currently loaded registry (useful for defensive checks and debugging).
- Use tag.IsResolved only when diagnosing cache behavior; it reflects whether a cached runtime ID is currently valid for the active registry.

```
// Defensive checks
if (!myTag.IsValid)
   return;

// Optional: verify the tag exists in the active registry
if (!myTag.IsRegistered)
   UnityEngine.Debug.LogWarning("Tag is not present in the current registry bake.");
```

TagMatchType

TagMatchType defines how a single tag comparison should be evaluated. In most gameplay systems, you will use either Exact or IncludeDescendants.

- Exact: only matches if the candidate tag is the same tag.
- IncludeDescendants: matches if the candidate is the same tag OR any descendant of the query tag (common for 'category' queries).
- IncludeAncestors: matches if the candidate is the same tag OR any ancestor of the query tag (useful for symmetric or tooling comparisons).

```
// Examples of match semantics:
//
// Candidate: Damage.Magic.Fire
// Query:     Damage.Magic
// IncludeDescendants => true
//
// Candidate: Damage.Magic
// Query:     Damage.Magic.Fire
// IncludeAncestors   => true
```

GameplayTagContainer

GameplayTagContainer represents a canonical set of tags (sorted + unique). It supports allocation-conscious mutation and both exact and hierarchical matching. It is the standard storage type for 'owned tags'.

Core patterns:

- Store owned tags on your runtime objects as a GameplayTagContainer field (component, ScriptableObject, or plain class).
- Mutate using AddTag / RemoveTag for single tags, or AddGameplayTags / RemoveGameplayTags for bulk operations.
- Use HasTag / HasAny / HasAll for membership tests. Exact matching does not require the registry; hierarchy matching does.

Important: struct copy semantics:

- GameplayTagContainer is a struct. If you read it out of a property/field by value, you receive a copy.
- Mutating that copy does not mutate the owner's stored container.
- Prefer ref accessors for owner types, or use IGameplayTagAssetQuery / GameplayTagAssetUtility mutation helpers.

```
// Example owner that stores tags directly.
public sealed class TagOwnerExample : UnityEngine.MonoBehaviour, IGameplayTagAssetQuery
{
    [UnityEngine.SerializeField] private GameplayTagContainer _tags;

    public void GetOwnedGameplayTags(ref GameplayTagContainer outTags) => outTags = _tags;

    // Implement query/mutation by delegating to the container.
    public bool HasMatchingGameplayTag(GameplayTag tag, TagMatchType matchType =
TagMatchType.IncludeDescendants)
        => _tags.HasTag(tag, matchType);

    public bool HasAnyMatchingGameplayTags(in GameplayTagContainer tags, TagMatchType matchType =
TagMatchType.IncludeDescendants)
        => _tags.HasAny(tags, matchType);

    public bool HasAllMatchingGameplayTags(in GameplayTagContainer tags, TagMatchType matchType =
TagMatchType.IncludeDescendants)
        => _tags.HasAll(tags, matchType);

    public int AddGameplayTags(in GameplayTagContainer tags) => _tags.AddGameplayTags(tags);
    public int RemoveGameplayTags(in GameplayTagContainer tags) => _tags.RemoveGameplayTags(tags);
    public void ClearGameplayTags() => _tags.Clear();
    public bool AddGameplayTag(in GameplayTag tag) { int before = _tags.Count; _tags.AddTag(tag); return
_tags.Count != before; }
    public bool RemoveGameplayTag(in GameplayTag tag) { int before = _tags.Count; _tags.RemoveTag(tag); return
_tags.Count != before; }
}
```

GameplayTagAssetUtility

GameplayTagAssetUtility solves the common 'I have some UnityEngine.Object reference; does it have tags?' problem. It prefers interfaces when available and falls back to common Unity patterns (component lookup, GameplayTagOwner).

When to use it:

- You have an unknown object type (ScriptableObject, MonoBehaviour, GameObject) and you want to query tags without writing type-specific code.
- You want to add/remove tags without knowing whether the object implements an interface or uses the default GameplayTagOwner component.
- You need a safe query path that never relies on registry initialization for hierarchy semantics (Safe helpers).

```
using CarryForge.HierarchicalTagForge;
using UnityEngine;

public static class TagChecks
{
    public static bool IsFireDamageSource(Object obj)
    {
        // Strict: uses hierarchy semantics when registry is initialized.
        return GameplayTagAssetUtility.HasMatchingGameplayTag(obj, GameTags.Damage_Magic_Fire,
TagMatchType.IncludeDescendants);
    }

    public static bool IsFireDamageSourceSafe(Object obj)
    {
        // Safe: if registry is missing/uninitialized, hierarchy semantics degrade to Exact.
        return GameplayTagAssetUtility.HasMatchingGameplayTagSafe(obj, GameTags.Damage_Magic_Fire,
TagMatchType.IncludeDescendants);
    }
}
```

GameplayTagQuery

GameplayTagQuery is a compiled boolean query evaluated against a GameplayTagContainer snapshot. It is intended to be stored on user data and evaluated frequently at runtime.

Evaluation guidance:

- Prefer query.Matches(ref tags) in high-frequency code so the container and query can reuse runtime caches.
- Use query.Matches(owner) only for convenience; it materializes a temporary container copy (still allocation-free, but does not retain cache warmups back to the owner).

- In Editor/Development builds, a query may retain an authoring definition for stale detection and diagnostic recompilation.

```
// High-frequency evaluation (recommended when you own the container storage)
if (myQuery.Matches(ref myTags))
{
   // query passed
}

// Convenience evaluation against an owner interface (does not persist container cache warmups)
if (myQuery.Matches(myOwnerQueryInterface))
{
   // query passed
}
```

GameplayTagRequirements

GameplayTagRequirements is a reusable gate for gameplay logic. It combines three common GAS-style checks (RequireAll, RequireAny, BlockAny) and optionally an advanced GameplayTagQuery for complex boolean logic.

Evaluation order (short-circuited):

1. BlockAny (fast fail)
2. RequireAll
3. RequireAny
4. Query (only if present)

If the evaluated owner implements IGameplayTagAssetQuery, the simple checks are executed through that interface to avoid unnecessary container copies; the query path still evaluates against a GameplayTagContainer snapshot.

```
// Example: ability activation gate
public sealed class AbilityGate : UnityEngine.MonoBehaviour
{
   [UnityEngine.SerializeField] private GameplayTagRequirements _requirements;

   public bool CanActivate(IGameplayTagAsset owner)
   {
      return _requirements.IsMetBy(owner);
   }
}
```

Example Patterns

1) Ability gating with TagMatchType.IncludeDescendants (common 'category' check):

```
// Require any 'State.CrowdControl.*' tag blocks activation
// (for example: State.CrowdControl.Stun, State.CrowdControl.Fear, etc.)

var blocked = GameplayTagContainer.Temp();
blocked.AddTag(GameTags.State_Stunned); // Example; add more tags as needed

if (tags.HasAny(blocked, TagMatchType.IncludeDescendants))
{
   // Block ability use
}
```

2) Temporary tag mutation (status effects / buffs):

```
// Add a status tag when an effect starts
GameplayTagAssetUtility.TryAddGameplayTag(targetObject, GameTags.State_Stunned);

// ... later, remove it when the effect ends
GameplayTagAssetUtility.TryRemoveGameplayTag(targetObject, GameTags.State_Stunned);
```

3) Unknown-owner query (UI visibility or interaction prompts):

```
// UI: show an icon if the target has any 'Interactable.*' tag
var required = GameplayTagContainer.Temp();
// required.AddTag(GameTags.Interactable); // example tag

bool show = GameplayTagAssetUtility.HasAnyMatchingGameplayTagsSafe(
   targetObject,
   required,
   TagMatchType.IncludeDescendants);
```

4) Query + Requirements combined (recommended for designer-authored logic):

```
// Simple structure:
// - Use requirements for common Require/Block lists.
// - Use a query for complex boolean logic (AND/OR/NOT nesting).

if (_requirements.IsMetBy(owner) && _advancedQuery.Matches(owner as IGameplayTagAssetQuery))
```

```
{
    // Gate passed
}
```

Demo Scene

The package includes a demo scene and template code demonstrating expected usage patterns. Use it as a reference implementation and a fast way to validate that your database, runtime settings, and baking pipeline are configured correctly.

> ("These scripts are intentionally small, readable templates. You can copy them into your project and replace the demo response (logging/visual color changes) with real gameplay behavior.")

Demo_AssetUtilityTester.cs

What it does

Demo_AssetUtilityTester demonstrates the **Asset Utility** workflow: interacting with tag ownership through a generic helper layer rather than hard-coding a specific owner type. In the demo scene, it acts as a "test driver" for common operations such as checking tags, listing tags, or applying tag changes (depending on what your utility layer exposes).

This script exists to show users the intended pattern for building:

- debug tools,
- editor-side "quick actions,"
- generic interaction code,
  where the caller does not need to know the concrete implementation details of tag ownership.

How to use it

1. Add Demo_AssetUtilityTester to any GameObject in the Demo scene.
2. In the Inspector, assign the target references the tester operates on (typically a tag owner/component or a GameObject that has one).
3. Assign any tag/tag-container fields that the tester uses for its example operations (e.g., a single tag for checks, a container for bulk operations).
4. Enter Play Mode and run the demo actions as the script intends (often via Inspector-exposed buttons or context menu actions, if present in your demo).

What to copy into your project

Use this as a template whenever you want **generic tag operations** that work across many object types. In production, the usual change is to replace “demo triggers” with:

- an in-game debug menu,
- a developer console command,
- interaction scripts (e.g., “apply tag on pickup,” “remove tag on consume”).

Demo_GameplayTagVisual.cs

What it does

Demo_GameplayTagVisual provides lightweight **visual feedback** for the demo scene. It typically controls a Renderer (or other simple visual target) so the demo can display a clear pass/fail state when a tag check, container check, query, or requirements gate is evaluated.

This script is intentionally simple so other demo scripts can depend on it without duplicating “change color / flash” logic.

How to use it

1. Add Demo_GameplayTagVisual to a GameObject that has a Renderer (or assign a Renderer reference in the Inspector).
2. Assign the target Renderer in the Inspector if the script does not auto-resolve it.
3. Wire this component into other demo scripts by dragging it into their Visual field (or equivalent reference).
4. Enter Play Mode and observe the Renderer update when those systems evaluate.

What to copy into your project

This is a template for **turning tag/query outcomes into immediate feedback**. In production you usually:

- rename it (remove the Demo_ prefix),
- swap color changes for your game’s UX (UI icon, outline shader, material swap, etc.),
- keep the “single place to visualize outcomes” pattern.

Demo_GameplayTagSingleOwner.cs

What it does

Demo_GameplayTagSingleOwner is a minimal “tag owner” example that exposes a **single GameplayTag field** in the Inspector. Its purpose is to demonstrate:

- how a tag looks/behaves in the Inspector (picker/drawer),
- how a simple component can provide tag data to another script.

This is the “hello world” of tag ownership: one component, one tag, no extra behavior.

How to use it

1. Add Demo_GameplayTagSingleOwner to a GameObject.
2. In the Inspector, select a tag for its Tag field using the HierarchicalTagForge picker.
3. Use this object as input to tag-checking demo scripts (for example Demo_GameplayTagSingleCaller).

What to copy into your project

Use this pattern any time you want a **single tag field** on a MonoBehaviour (e.g., "ItemType," "Faction," "InteractionCategory"). In production you often expand it into a richer owner, but the Inspector-facing tag field remains the same.

Demo_GameplayTagSingleCaller.cs

What it does

Demo_GameplayTagSingleCaller demonstrates evaluating one tag against another under an explicit **match policy** (TagMatchType). It typically:

- reads a tag from an "owner" component,
- evaluates that tag under match rules (exact / hierarchical behavior depending on your system),
- optionally routes the result into Demo_GameplayTagVisual for immediate feedback.

This script exists to show users how to do **fast, explicit tag checks** with a known match semantic.

How to use it

1. Add Demo_GameplayTagSingleCaller to any GameObject.
2. Assign the two tag sources it compares (commonly Demo_GameplayTagSingleOwner references).
3. Set MatchType in the Inspector to control how matching behaves.
4. Optionally assign a Demo_GameplayTagVisual reference to display pass/fail.
5. Enter Play Mode and observe the evaluation result.

What to copy into your project

This is a template for:

- interaction checks ("does the player have Ability.Magic.*?"),
- state validation ("is this actor in State.Stunned?"),
- simple gating logic where one tag is compared to a requirement tag.

Replace demo visuals/logging with your gameplay response (enable interaction, trigger VFX, etc.).

Demo_GameplayTagContainerOwner.cs

What it does

Demo_GameplayTagContainerOwner is the container-based equivalent of Demo_GameplayTagSingleOwner. It exposes a **GameplayTagContainer** in the Inspector, allowing an object to own **multiple tags**.

Its purpose is to demonstrate:

- authoring a tag set in the Inspector,
- using a container as the input for queries and gating systems.

How to use it

1. Add Demo_GameplayTagContainerOwner to a GameObject.
2. In the Inspector, populate the container with one or more tags using the container UI/pickers.
3. Use this component as the "tag source" for container checks, queries, and requirements demo scripts.

What to copy into your project

This is the standard template for "this object has many tags," such as:

- status effects,
- equipment states,
- faction/role flags,
- interaction permissions.

In production you may store the container internally (or via a runtime component), but the conceptual pattern is identical.

Demo_GameplayTagContainerCaller.cs

What it does

Demo_GameplayTagContainerCaller demonstrates common **container-to-container** checks such as:

- HasAny: "does the owner contain any of these tags?"
- HasAll: "does the owner contain all of these tags?"

It exists to show the most common production use case: **gating** based on a set of required tags.

How to use it

1. Add Demo_GameplayTagContainerCaller to any GameObject.
2. Assign a target container owner (commonly Demo_GameplayTagContainerOwner) whose tags will be evaluated.
3. Populate the "check" container (the set of tags you require).

4. Select the evaluation mode (Any/All) if the script exposes it.
5. Optionally assign Demo_GameplayTagVisual to show pass/fail.
6. Enter Play Mode and observe results.

What to copy into your project

This is a direct template for:

- "can interact?" rules,
- ability prerequisites,
- UI state gating,
- AI permission checks.

Replace demo behavior with your actual response (enable UI button, allow equip, transition AI state, etc.).

Demo_GameplayTagQuery.cs

What it does

Demo_GameplayTagQuery is a small holder component for a **GameplayTagQuery** field. It exists to separate:

- query authoring/storage (on one component),
  from
- query evaluation (handled elsewhere, e.g., by Demo_GameplayTagQueryCaller).

This keeps the demo clean and mirrors production design: data lives with the thing that owns the rule; evaluation is performed by the runtime system.

How to use it

1. Add Demo_GameplayTagQuery to a GameObject.
2. In the Inspector, author or assign a GameplayTagQuery using the query tooling.
3. Reference this component from a caller script that evaluates queries.

What to copy into your project

Use this pattern for:

- "ability has a query,"
- "interactable has a query,"
- "AI transition has a query,"
- "quest step has a query."

Keeping "query data holder" separate from "query evaluator" scales well.

Demo_GameplayTagQueryCaller.cs

What it does

Demo_GameplayTagQueryCaller evaluates a GameplayTagQuery against a tag source (typically a container owner). It demonstrates:

- consuming an authored query at runtime,
- producing a deterministic match result,
- optionally visualizing that result via Demo_GameplayTagVisual.

How to use it

1. Add Demo_GameplayTagQueryCaller to any GameObject.
2. Assign the tag source (commonly a Demo_GameplayTagContainerOwner).
3. Assign the query holder (commonly a Demo_GameplayTagQuery component).
4. Optionally assign Demo_GameplayTagVisual for pass/fail visualization.
5. Enter Play Mode and observe query results.

What to copy into your project

This is the template for "evaluate complex gating logic in one call." In production, the match result typically drives:

- interaction availability,
- ability activation,
- AI decision-making,
- equipment rules,
- UI visibility/enablement.

Demo_RequirementsGate.cs

What it does

Demo_RequirementsGate demonstrates a "requirements" style gating workflow (GAS-style semantics):

- Require All
- Require Any
- Block Any

It exists to show the most content-friendly approach for gating gameplay behavior: author requirements in data, evaluate at runtime against a tag owner/container, then enable/disable behavior.

How to use it

1. Add Demo_RequirementsGate to the GameObject that represents the gated behavior (or the controller that drives it).
2. Assign the requirements asset/data (your "requirements query" object).

3. Assign the tag source (the owner/container to evaluate).
4. Enter Play Mode; the gate evaluates and updates the demo response (often paired with visuals).

What to copy into your project

This is the production template for:

- interactable locks,
- equipment restrictions,
- quest/trigger gating,
- ability prerequisites,
- UI state constraints.

Replace the demo “response” with your own action (enable component, unlock door, permit equip, etc.).

Avoid LINQ in hot paths; avoid allocating collections during per-frame checks.

Prefer query assets and precomputed containers over ad-hoc logic each frame.

Use the TestHarness and Profiler to verify allocations in your specific project.

Testing and Diagnostics

This section documents HierarchicalTagForge tooling intended for **verification, profiling, and build-time safety checks**. Use these tools to catch stale query data before shipping and to benchmark tag query performance using consistent, profiler-visible workloads.

Build Validation

GameplayTagQuery Build Validation

HierarchicalTagForge includes an editor-only build-time validator that scans your project for serialized **GameplayTagQuery** fields and detects cases where a query’s **compiled runtime data is stale** relative to its authoring definition. This helps prevent situations where a query was edited but not recompiled, leading to mismatches at runtime.

What the validator checks

At a high level, validation runs the following steps:

1. **Scans Prefab assets** under Assets/.
2. **Scans ScriptableObject assets** under Assets/.
3. **Scans enabled Build Settings scenes**, opening each scene additively, iterating components (including inactive objects), then restoring the user’s scene setup.

For each serialized GameplayTagQuery field found, the validator compares:

- The stored **compiled definition hash** (_compiledDefinitionHash)
- Against the **current hash** computed from the query definition (when definition-backed authoring data is available)

If the hashes differ, the query is treated as **stale**.

Important limitation (definition-backed checks)

The stale-check is **definition-driven** and relies on definition-backed authoring data (Unity 2022.3+ managed references). If a query has *no stored definition* (release-like serialized data), the validator may not be able to perform a hash check and will not flag it.

Auto-fix behavior

When **Auto Fix** is enabled and a stale query is detected, the validator attempts to:

- Recompile the current definition
- Replace the entire query struct **atomically** via SerializedProperty.boxedValue
- Save and refresh assets if any fixes were applied

If auto-fix fails (compile error), the report records an issue including the context and guidance to recompile manually.

Running validation manually

HierarchicalTagForge exposes a menu command to run the same validation pipeline on demand and display results immediately:

- **Tools → CarryForge → HierarchicalTagForge → Run Validation Now**

Output is:

- Logged to the Console
- Shown in a modal dialog for visibility

Project preferences (validation policy)

Build validation behavior is controlled by project settings exposed through a small editor-only preferences façade:

- **Fail Build On Stale**
  When enabled, stale queries are treated as build-failing issues (policy enforcement depends on your build integration path).
- **Auto Fix Stale**
  When enabled, validation will attempt to auto-fix stale queries by recompiling them during the validation pass.

How to read the report

The validation report includes:

- **CheckedQueries**: total number of GameplayTagQuery fields inspected
- **FixedQueries**: number of queries successfully auto-fixed (if enabled)
- **Issues**: list of problems encountered (stale queries, exceptions while scanning, compile failures during auto-fix)

If you see an issue describing stale compiled data, the intended fix is:

- Open the query asset/field authoring UI
- Click **Compile** (or enable your auto-compile workflow)
- Save the asset (then rerun validation)

GameplayTag Stress Harness

Overview

The **GameplayTag Stress Harness** is a runtime benchmark component used to measure the cost of different tag query entry points under load. It can spawn or discover tag owners, assign tags from a configurable pool, and execute a large number of queries per frame while emitting Unity Profiler markers under the HTF.Stress.* namespace.

This harness is intended for:

- Benchmarking query costs on your target hardware
- Comparing query paths (Owner vs Container vs AssetUtility vs Query.Matches)
- Confirming steady-state profiling behavior using consistent workloads

Adding the harness to a scene

1. Create an empty GameObject (for example, HTF_StressHarness).
2. Add the component:
   Add Component → CarryForge → HierarchicalTagForge → Tests → GameplayTag Stress Harness
3. Assign **Tag Pool** (Tag Pool) with at least one GameplayTag. The harness disables itself at runtime if the pool is empty.

Inspector UI and organization

The harness ships with a custom inspector that groups settings into foldouts and provides inline tooltips for each field (Run Control, Load, Owner Setup, Tag Pool, Container Inputs, Query Inputs, Diagnostics).

Core settings

Run Control

- **Run**: Master toggle. When false, the harness does nothing in Update().
- **Mode**: Which benchmark path runs each frame.
- **Match Type**: Exact vs hierarchical semantics. If using hierarchical matching, ensure the registry is initialized (the harness can warn if it is not).

Load

- **Queries Per Frame**: The primary load knob. This is the number you scale up/down to stress the system.
- **Spread Across Owners**:

- Enabled: distributes queries across many owners
- Disabled: repeatedly queries owner[0] (useful for isolating cache behavior)

Owner setup and tag assignment

The harness supports two owner sources:

- **Owners Override**: If set, those owners are used and spawn/discovery options are ignored.
- **Auto Spawn Owners**:
  - If enabled, spawns Owner Count owners as children of the harness GameObject.
  - If disabled, discovers existing GameplayTagOwner instances in children.

Owner tag assignment is controlled by **Owner Tag Mode** and related count fields. The harness supports selection strategies such as:

- All from pool
- Fixed or random **unique** counts
- Fixed or random **with replacement** (duplicates allowed; unique result may be lower than attempt count)

Container inputs (HasAny / HasAll benchmarks)

- **Any Query Tags**: used by HasAny benchmarks
- **All Query Tags**: used by HasAll benchmarks

If either container is empty at runtime, the harness auto-builds it from **Tag Pool** using the corresponding auto-build mode/count settings.

Query inputs (GameplayTagQuery.Matches)

To benchmark GameplayTagQuery.Matches, populate **Compiled Queries** with one or more GameplayTagQueryAsset references. If this list is empty, Query_Matches mode still runs (for marker visibility) but does not evaluate real queries.

Diagnostics

- **Warn If Registry Not Initialized**: When enabled, the harness warns if you select a hierarchical match type but the registry is not initialized (results may fail-closed).
- **Log Once On Start**: Logs a single line describing harness configuration at startup.

Profiling the Stress Harness

Finding the correct profiler markers

The harness emits explicit CPU profiler markers. In the Unity Profiler:

1. Open **Window → Analysis → Profiler**
2. Select **CPU Usage**

3. In **Hierarchy** view, search for:
   HTF.Stress

Primary markers include:

- HTF.Stress.Total
- HTF.Stress.Component.HasMatchingGameplayTag
- HTF.Stress.Component.HasAny
- HTF.Stress.Component.HasAll
- HTF.Stress.Container.HasTag
- HTF.Stress.Container.HasAny
- HTF.Stress.Container.HasAll
- HTF.Stress.AssetUtility.HasMatchingGameplayTag
- HTF.Stress.AssetUtility.HasAny
- HTF.Stress.AssetUtility.HasAll
- HTF.Stress.Query.Matches

How to interpret results

- **HTF.Stress.Total**: overall harness cost for the frame
- The nested marker for your selected **Mode** is the one you compare across runs
- Use **Queries Per Frame** as your independent variable for scaling load
- Watch the **GC Alloc** column during steady-state runs; the harness is designed to avoid allocations during the benchmark loops

Recommended profiling workflow

For consistent results:

1. Start with a moderate owner count (e.g., 256–1024) and a stable tag pool.
2. Choose one mode (e.g., Container.HasTag) and baseline timing.
3. Increase **Queries Per Frame** until the marker cost is easily visible but not saturating the frame.
4. Switch modes (Component vs Container vs AssetUtility vs Query) while keeping all other settings identical.
5. If comparing hierarchical semantics, set **Match Type** accordingly and ensure the registry is initialized (or keep Match Type = Exact for strict baselines).

FAQ and Troubleshooting

“No GameplayTagDatabase is set as default”

Open Tag Manager and set a default database in the settings UI.

Confirm you saved the database asset and it exists in the project.

If your project uses multiple databases, ensure the correct one is configured as default.

“Database is not baked” appears

Open Tag Manager and run Validate/Bake.

Confirm the database you edited is the same one referenced by runtime settings.

If using generated code, regenerate and recompile.

Tags don’t match as expected

Confirm whether your check is exact or hierarchical.

Confirm runtime initialization loaded baked data.

Test the same tags/queries in the demo scene to isolate custom code issues.

Builds can’t find default database

Confirm the runtime settings asset exists under a Resources folder.

Confirm the runtime settings references the correct database.

Confirm the database asset is included in the build (Resources inclusion or your content pipeline).

Generated constants are missing or stale

Regenerate from Tag Manager after tag edits.

Ensure the generated file compiles in a runtime assembly if used by gameplay code.

Check .gitignore/source control filters to ensure generated file is not ignored.

Conclusion

Thank you for using HierarchicalTagForge. For additional help or feature requests, please reach out to the developer via email:
anthony.halstead@carryforgestudio.com