Gameplay Tags

An overview of the hierarchical label system.

Gameplay Tags are user-defined strings that function as conceptual, hierarchical labels. You can apply Gameplay Tags to objects in your project and evaluate them to drive your gameplay implementation, similarly to checking for booleans or flags.

You can use them to communicate many different concepts, including the following:

- Attributes of an object, such as `Character.Enemy.Zombie`
- What an object is doing or capable of doing, such as `Movement.Mode.Swimming`
- Game events and triggers, such as `GameplayEvent.RequestReset`

Gameplay Tags can have any number of hierarchical levels, as indicated by `.` character separations. For example, the tag `Event.Movement.Dash` has three levels, with `Event` being the broadest identifier in the hierarchy and `Dash` being the most specific.

Defining Gameplay Tags

You must add Gameplay Tags to the tag dictionary for Unreal Engine to be aware of them. You can add (or remove) tags with one of the following methods:

- Directly adding or removing them in **Project Settings**
- Importing them from **Data Table** assets
- Defining them with C++

All of the above methods are set in **Project Settings** in the **GameplayTags** section under the **Project** heading.

Adding Tags in Project Settings

The simplest way to define a new Gameplay Tag is to add it directly in **Project Settings**.

To add a tag in **Project Settings**, do the following:

1. Enable **Import Tags From Config**. This imports all Gameplay Tags from `.ini` files, including `Config/DefaultGameplayTags.ini` and any in `Config/Tags`.
2. (Optional) Click the **Add new Gameplay Tag source** button to create a new source `.ini` file in `Config/Tags` to store the Gameplay Tag. Creating separate source files for different aspects of a project can be useful for organization and collaboration on large projects.
3. Click the **Manage Gameplay Tags** button next to the **Gameplay Tag List** entry. This opens a **Gameplay Tag Manager** window.
4. In the **Gameplay Tag Manager** window, click the **Add (+)** button in the top-left corner.
5. Enter the desired **Name**, **Comment**, and **Source**. The Comment displays on the tag's tooltip, and the Source is the `.ini` file that stores the tag.
6. Click the **Add New Tag** button.

You can rename, delete, duplicate, or add a new sub-tag to a tag by right-clicking it in the list and selecting the option from the context menu. Tags from sources other than `.ini` files cannot be renamed or deleted in the **Gameplay Tag Manager** window.

> You can edit the tag `.ini` source files with a text editor, but you must restart the editor to load the changes.

Importing Tags from Data Table Assets

You can import Gameplay Tags from Data Table assets with the row type `GameplayTagTableRow`. With this approach, you can:

- Manage tags in the **Data Table Editor**.
- Make changes to Data Tables while the editor is running.
- Add tags by importing a `.csv` or `.json` file as a Data Table.

To import tags from a Data Table, do the following in **Project Settings**:

1. Click the **Add Element (+)** button next to **Gameplay Tag Table List**.
2. Click the dropdown for the new index and select your Data Table.

Defining Tags with C++

You can define Gameplay Tags with C++ by using the following macros defined in `NativeGameplayTags.h`:

- `UE_DECLARE_GAMEPLAY_TAG_EXTERN`: Used in a `.h` file to declare a tag defined in a `.cpp` file.
- `UE_DEFINE_GAMEPLAY_TAG`: Used in a `.cpp` file to define a tag declared in a `.h` file without a tooltip comment.
- `UE_DEFINE_GAMEPLAY_TAG_COMMENT`: Used in a `.cpp` file to define a tag declared in a `.h` file with a tooltip comment.

- `UE_DEFINE_GAMEPLAY_TAG_STATIC`: Used in a `.cpp` file to define a tag only available to the defining file. Unlike the other `DEFINE` macros, this should not be paired with a `DECLARE` macro call.

> You must add the `GameplayTags` module to your project's `Build.cs` file to access Gameplay Tags functionality in C++.

Example Implementation

```
// In .h file
UE_DECLARE_GAMEPLAY_TAG_EXTERN(Movement_Mode_Walking);

// In .cpp file
UE_DEFINE_GAMEPLAY_TAG_COMMENT(Movement_Mode_Walking,
"Movement.Mode.Walking", "Default Character movement tag");
```

Copy full snippet

For a more detailed example implementation, see `LyraGameplayTags.h` and `LyraGameplayTags.cpp` in the Lyra Sample Game project.

Using Defined Gameplay Tags

Once defined, you can apply tags to objects and evaluate the tags to drive gameplay in your project.

Applying Tags to Objects

Apply tags to objects by doing the following:

1. Add a **Gameplay Tag Container** (`FGameplayTagContainer`) type variable to the object. This variable stores multiple Gameplay Tags.
2. Use the Add Gameplay Tag (`AddTag`) function to add a specified tag to the container.

You can also remove tags from a container with the Remove Gameplay Tag (`RemoveTag`) function and append Gameplay Tag Containers together using the Append Gameplay Tag Containers (`AppendTags`) function.

> You can use a Gameplay Tag (`FGameplayTag`) type variable directly, but objects frequently have multiple tags, so Gameplay Tag Containers are often required.

Evaluating Tags with Conditional Functions

You can drive your gameplay implementation based on an object's tags. To evaluate tags stored in an object's Gameplay Tag Container, you can use a variety of conditional functions, such as:

- Has Tag (`HasTag`)
- Has Any Tags (`HasAny`)
- Has All Tags (`HasAll`)

See the `FGameplayTagContainer` C++ API reference and the GameplayTags Blueprint API reference for more information.

> Calling a conditional function with an empty Gameplay Tag Container as the input parameter returns false, except for the `All` functions such as `HasAll`. This is because no tags in the container are missing from the source set.

Gameplay Tag Queries

Gameplay Tag Query (`FGameplayTagQuery`) type variables combine conditional functions to establish complex logic in a more straightforward and condensed way.

Gameplay Tag Queries support the following expressions:

- **Any Tags Match**: Tests if at least one tag from the query is found in the container.
- **All Tags Match**: Tests if all tags in the query are in the container. If the query is empty, this returns true.
- **No Tags Match**: Tests if no tags in the query are in the container. If the query is empty, this returns true.

Additionally, Gameplay Tag Queries support the following root expressions that are evaluated based on sub-expressions:

- **Any Expressions Match**: Tests if any of the sub-expressions returns true.
- **All Expressions Match**: Tests if all of the sub-expressions return true. If there are no sub-expressions, this returns true.
- **No Expressions Match**: Tests if none of the sub-expressions return true. If there are no sub-expressions, this returns true.

Advanced Topics

Setting Tag Editing Restrictions

You can restrict Gameplay Tag editing (at any hierarchy level) by user.

To restrict editing, set the following settings in the **Project Settings** under **Advanced Gameplay Tags > Advanced**:

- **Restricted Config Files**: A list of `.ini` files used to store restricted tags paired with a list of **Owners** who have editing privileges.

- **Restricted Tag List**: Displays a **Gameplay Tag Manager** window that can modify restricted tags.

If a user (other than the listed owners) attempts to edit a restricted tag, a warning message displays, asking the user to confirm that they have permission from the owners to make any edits. If the user does not confirm, the edit is not made.

> Restricted tags cannot be deleted in the editor once they are created. To delete a restricted tag, you must edit the `.ini` file directly.

Streamlining Tag Access in C++

You can improve your Gameplay Tag implementation by using the `IGameplayTagAssetInterface`. The interface provides the following benefits:

- You can get an object's tags without explicitly casting the object.
- You can write custom code for each possible type.

Implementing the interface and overriding the `GetOwnedGameplayTags` function creates a Blueprint-accessible method to populate a Gameplay Tag Container with the tags associated with that object. In most cases, this means copying the tags from the base class into a new container, but your implementation could gather tags from multiple containers or call a Blueprint function to access Blueprint-declared tags or whatever your object requires.

For an example implementation, see the `ALyraTaggedActor` class in the Lyra Sample Game project.