

Developer

/ Documentation

/ Unreal Engine ▾

/ Unreal Engine 5.4 Documentation

/ Making Interactive Experiences

/ Gameplay Ability System

/ Gameplay Attributes and Attribute Sets

Gameplay Attributes and Attribute Sets

Using Gameplay Attributes and Attribute Sets



The **Gameplay Ability System** uses **Gameplay Attributes** (`FGameplayAttribute`) to store, calculate, and modify gameplay-related floating-point values. These values can describe any trait of their owners, such as a character's remaining health points, a vehicle's top speed, or the number of times an item can be used before it breaks. Actors in the Gameplay Ability System store their Gameplay Attributes in an **Attribute Set**, which helps to manage interactions between Gameplay Attributes and other parts of the system, and registers itself with the Actor's Ability System Component. These interactions include clamping to value ranges, performing calculations that apply temporary value changes, and reacting to events that permanently alter their base values.

Gameplay Attributes

Gameplay Attributes keep a current and base value. The "current" value is commonly used in calculations and logic that use and can be affected by any [Gameplay Effects](#) that are currently active, while the "base" value is more likely to remain fixed for a longer period. As an example, a "Jump Height" Gameplay Attribute might have a base value of 100.0, but if the character has an active Gameplay Effect stating that they're tired and can only jump to 70% of their normal height, then the current value would be 70.0. If that character becomes skilled

at jumping through a level-up system, then the base value might increase to 110.0, while the current value would be calculated as 77.0 as long as the Gameplay Effect remains.

To create Gameplay Attributes, you must first establish an Attribute Set. You can then add your Gameplay Attributes (`FGameplayAttributesData`) into the Attribute Set, a class derived from `UAttributeSet`.



In some cases, Gameplay Attributes can exist without an Attribute Set. This generally indicates that a Gameplay Attribute has been stored on an **Ability System Component** that does not have an Attribute Set containing the appropriate type of Gameplay Attribute. This is not recommended, because the Gameplay Attribute will have no defined behaviors that interact with any part of the Gameplay Ability System other than being storage for a floating-point value.

Attribute Sets

Definition and Setup

To begin, set up an Attribute Set with one or more Gameplay Attributes, and register it with your Ability System Component.

1. Extend the base Attribute Set class, `UAttributeSet`, and add your Gameplay Attributes as `FGameplayAttributeData` UProperties. A simple Attribute Set with one Gameplay Attribute might look like this:

```
1 UCLASS()  
2 class MYPROJECT_API UMyAttributeSet : public UAttributeSet  
3 {  
4     GENERATED_BODY()  
5  
6     public:  
7         /** Sample "Health" Attribute, publicly accessible */  
8         UPROPERTY(EditAnywhere, BlueprintReadOnly)  
9         FGameplayAttributeData Health;  
10 };
```

 Copy full snippet

2. Store the Attribute Set on the Actor, and expose it to the engine. Use the `const` keyword to ensure that code cannot modify the Attribute Set directly. Add this to your Actor's class definition:

```
1 /** Sample Attribute Set. */
2 UPROPERTY()
3 const UMyAttributeSet* AttributeSet;
```

 Copy full snippet

3. Register the Attribute Set with the appropriate Ability System Component. This happens automatically when you instantiate the Attribute Set, which you can do in the Actor's constructor, or during `BeginPlay`, as long as the Actor's `GetAbilitySystemComponent` function returns a valid Ability System Component at the moment of instantiation. You can also edit the Actor's Blueprint and add the Attribute Set type to the Ability System Component's Default Starting Data. A third method is to instruct the Ability System Component to instantiate the Attribute Set, which will then register it automatically, as in this example:

```
1 // Get the appropriate Ability System Component. It could be on another
Actor, so use GetAbilitySystemComponent and check that the result is
valid.
2 AbilitySystemComponent* ASC = GetAbilitySystemComponent();
3 // Make sure the AbilitySystemComponent is valid. If failure is
unacceptable, replace this if() conditional with a check() statement.
4 if (IsValid(ASC))
5 {
6 // Get the UMyAttributeSet from our Ability System Component. The
Ability System Component will create and register one if needed.
7 AttributeSet = ASC->GetSet<UMyAttributeSet>();
8
9 // We now have a pointer to the new UMyAttributeSet that we can use
later. If it has an initialization function, this is a good place to
call it.
10 }
```

 Copy full snippet



An Ability System Component can have multiple Attribute Sets, but each Attribute Set must be of a different class from all the others.

Finally, applying a Gameplay Effect that modifies a Gameplay Attribute that the Ability System Component doesn't have will cause the Ability System Component to create a matching Gameplay Attribute for itself. However, this method does not create an Attribute Set or add the Gameplay Attribute to any existing one.

1. As an optional step, add some basic helper functions to interact with the Gameplay Attribute. It is a good idea to make the Gameplay Attribute itself protected or private, while leaving the functions that interact with it public. The Gameplay Ability System provides a set of macros to set up some default functions:

Macro (with Parameters)	Signature of Generated Function	Behavior/Usage
<code>GAMEPLAYATTRIBUTE_PROPERTY_GETTER(UMyAttributeSet, Health)</code>	<code>static FGameplayAttribute GetHealth()</code>	Static function, returns the <code>FGameplayAttribute</code> struct from the engine's reflection system
<code>GAMEPLAYATTRIBUTE_VALUE_GETTER(Health)</code>	<code>float GetHealth() const</code>	Returns the current value of the "Health" Gameplay Attribute
<code>GAMEPLAYATTRIBUTE_VALUE_SETTER(Health)</code>	<code>void SetHealth(float NewVal)</code>	Sets the "Health" Gameplay Attribute's value to <code>NewVal</code>
<code>GAMEPLAYATTRIBUTE_VALUE_INITTER(Health)</code>	<code>void InitHealth(float NewVal)</code>	Initializes the "Health" Gameplay Attribute's value to <code>NewVal</code>

After adding these, your Attribute Set class definition should look like this:

```
1 UCLASS()  
2 class MYPROJECT_API UMyAttributeSet : public UAttributeSet  
3 {  
4     GENERATED_BODY()
```

```

5
6 protected:
7 /** Sample "Health" Attribute */
8 UPROPERTY(EditAnywhere, BlueprintReadOnly)
9 FGameplayAttributeData Health;
10
11 //~ ... Other Gameplay Attributes here ...
12
13 public:
14 //~ Helper functions for "Health" attributes
15 GAMEPLAYATTRIBUTE_PROPERTY_GETTER(UMyAttributeSet, Health);
16 GAMEPLAYATTRIBUTE_VALUE_GETTER(Health);
17 GAMEPLAYATTRIBUTE_VALUE_SETTER(Health);
18 GAMEPLAYATTRIBUTE_VALUE_INITTER(Health);
19
20 //~ ... Helper functions for other Gameplay Attributes here ...
21 };

```

 Copy full snippet

Although these helper functions are not strictly required, they are considered a best practice.

This establishes a basic Attribute Set with a single Gameplay Attribute. You will also need to implement code that controls the behavior of your Gameplay Attributes based on an understanding of how these values are meant to interact with each other, and what they mean in the context of your project or the specific Actor class you are developing. You can build this functionality by controlling access to the Gameplay Attributes themselves, or by directing the way that Gameplay Effects work at the Ability Set level.

Initialization

If you choose not to initialize your Attribute Set and its Gameplay Attributes by calling an initialization function with hard-coded values, you can do so with a [Data Table](#) using the Gameplay Ability System row type called "AttributeMetaData". You can import data from an external file, or manually populate the Data Table in the editor.

</

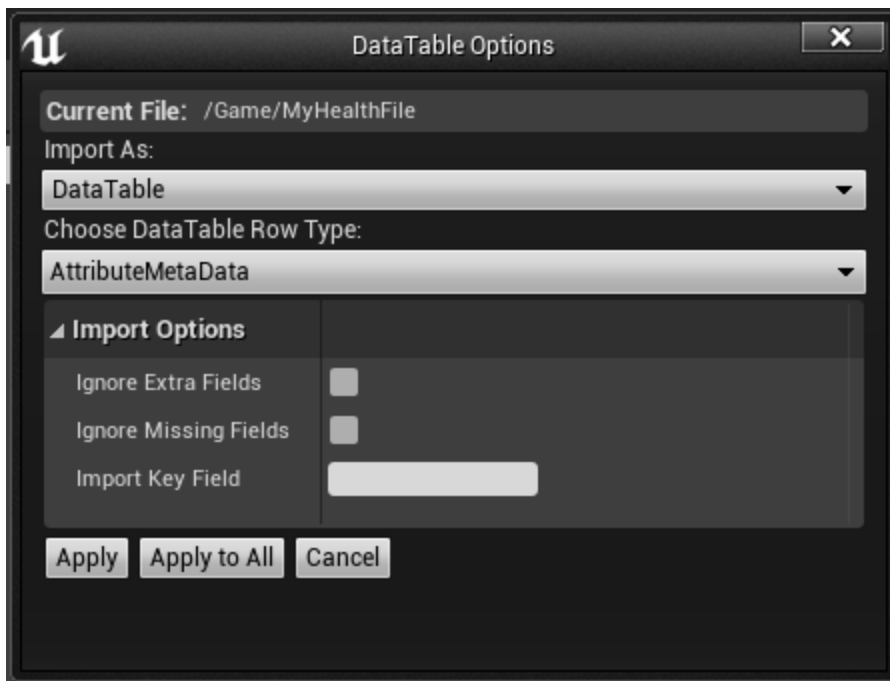
When creating the Data Table Asset, choose "AttributeMetaData" as the row type.

Importing Data Tables

Developers usually import their tables from .csv files like the following:

```
1 ---,BaseValue,MinValue,MaxValue,DerivedAttributeInfo,bCanStack
2 MyAttributeSet.Health,"100.000000","0.000000","150.000000","", "False"
```

 Copy full snippet



When importing a .csv file as a Data Table Asset, select the "AttributeMetaData" row type.

You can append additional rows to support Attribute Sets with multiple Gameplay Attributes. In the file shown above, the "Health" Gameplay Attribute within `UMyAttributeSet` (the reflection system drops the "U" prefix) will initialize with a value of 100. It has no derived information and does not stack.



Although there are columns for MinValue (0.0) and MaxValue (150.0), Gameplay Attributes and Attribute Sets do not feature a built-in clamping behavior; the values in these columns

have no effect.

Manually Populating Data Tables

If you prefer to edit values in Unreal Editor rather than an external spreadsheet or text-editor program, you can do so by creating your table and opening it like any other Blueprint Asset. Use the Add button at the top of the window to add a row for each Gameplay Attribute. Keep in mind that the naming convention is AttributeSetName.AttributeName, and it is case-sensitive.



The "Min Value" and "Max Value" columns are not implemented in the default Gameplay Ability System Plugin; these values do not have any effect.

Controlling Gameplay Attribute Access

Controlling direct access to the Gameplay Attributes is a good way to ensure that their values are always within the limits that you set for them. This is done through the Ability Set, not by extending `FGameplayAttributeData`; `FGameplayAttributeData` only stores and provides access to the Gameplay Attribute's data.

To restrict the value of the "Health" Gameplay Attribute so that it can never go below zero, you can write your own getter and setter functions. Remove the

`GAMEPLAYATTRIBUTE_VALUE_GETTER` and `GAMEPLAYATTRIBUTE_VALUE_SETTER` macros, replacing them with function headers:

```
1 GAMEPLAYATTRIBUTE_PROPERTY_GETTER(UMyAttributeSet, Health);  
2 float GetHealth() const;  
3 void SetHealth(float NewVal);  
4 GAMEPLAYATTRIBUTE_VALUE_INITIALIZER(Health);
```

Copy full snippet

Define these functions in your Attribute Set's source file:

```

1 float UMyAttributeSet::GetHealth() const
2 {
3     // Return Health's current value, but never return a value lower than zero.
4     // This is the value after all modifiers that affect Health have been
       considered.
5     return FMath::Max(Health.GetCurrentValue(), 0.0f);
6 }
7
8 void UMyAttributeSet::SetHealth(float NewVal)
9 {
10    // Do not accept values lower than zero.
11    NewVal = FMath::Max(NewVal, 0.0f);
12
13    // Make sure we have an Ability System Component instance. This should
       always be the case.
14    UAbilitySystemComponent* ASC = GetOwningAbilitySystemComponent();
15    if (ensure(ASC))
16    {
17        // Set the base value (not the current value) through the appropriate
           function.
18        // This makes sure that any modifiers we have applied will still work
           properly.
19        ASC->SetNumericAttributeBase(GetHealthAttribute(), NewVal);
20    }
21 }
22
23 AbilitySystemComponent-
   >GetGameplayAttributeValueChangeDelegate(AttributeSet-
   >GetHealthAttribute()).AddUObject(this,
   &AGASAbilityDemoCharacter::OnHealthChangedInternal);

```

 Copy full snippet

Interactions with Gameplay Effects

A common way to exercise control over the value of a Gameplay Attribute is to handle [Gameplay Effects] as they relate to it.

1. Begin by overriding the `PostGameplayEffectExecute` function in your Attribute Set's class definition. This function should be at the public access level.


```
void PostGameplayEffectExecute(const struct FGameplayEffectModCallbackData& Data)
```

 Copy full snippet

2. Write the function body in the Attribute Set's source file, making sure to call the parent class' implementation.

```
1 void UMyAttributeSet::PostGameplayEffectExecute(const struct
  FGameplayEffectModCallbackData& Data)
2 {
3   // Remember to call the parent's implementation.
4   Super::PostGameplayEffectExecute(Data);
5
6   // Check to see if this call affects our Health by using the Property
  Getter.
7   if (Data.EvaluatedData.Attribute == GetHealthAttribute())
8   {
9     // This Gameplay Effect is changing Health. Apply it, but restrict the
  value first.
10    // In this case, Health's base value must be non-negative.
11    SetHealth(FMath::Max(GetHealth(), 0.0f));
12  }
13 }
```

 Copy full snippet

Replication

For multiplayer projects, you can replicate your Gameplay Attribute through the Attribute Set similar to how you would replicate any other property.

1. Begin by adding the `ReplicatedUsing` [Specifier](#) to your property's definition in the Attribute Set header file. This will set up a callback function that helps with prediction on remote systems.


```
1 protected:
2   /** Sample "Health" Attribute */
```

```
3 UPROPERTY(EditAnywhere, BlueprintReadOnly, ReplicatedUsing =  
    OnRep_Health)  
4 FGameplayAttributeData Health;
```

 Copy full snippet

2. Declare your replication callback function:

```
1 /** Called when a new Health value arrives over the network */  
2 UFUNCTION()  
3 virtual void OnRep_Health(const FGameplayAttributeData& OldHealth);
```

 Copy full snippet

3. In the Attribute Set's source file, define your replication callback function. The body of the function can be expressed as a single macro that the Gameplay Ability System defines.

```
1 void UMyAttributeSet::OnRep_Health(const FGameplayAttributeData&  
    OldHealth)  
2 {  
3 // Use the default Gameplay Attribute System renotify behavior.  
4 GAMEPLAYATTRIBUTE_REPNOTIFY(UMyAttributeSet, Health, OldHealth);  
5 }
```

 Copy full snippet

4. If this is the first replicated property in your Attribute Set, set up an override for the public `GetLifetimeReplicatedProps` function.

```
1 /** Marks the properties we wish to replicate */  
2 virtual void GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&  
    OutLifetimeProps) const override;
```

 Copy full snippet

5. Add the Gameplay Attribute to the Attribute Set's `GetLifetimeReplicatedProps` function in its source file, as follows:

```
1 void
  UMyAttributeSet::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
    OutLifetimeProps) const
2 {
3   // Call the parent function.
4   Super::GetLifetimeReplicatedProps(OutLifetimeProps);
5
6   // Add replication for Health.
7   DOREPLIFETIME_CONDITION_NOTIFY(UMyAttributeSet, Health, COND_None,
    REPNOTIFY_Always);
8 }
```

 Copy full snippet