

2. AbilitySystemComponent

AbilitySystemComponent is the of the ability system. Is **thrived** from the **actor components**.

It contains:

1. All the granted abilities
2. Applied tags
3. Header replications

And many other things.

```
/** The core ActorComponent for interfacing with the GameplayAbilities System */
UCLASS(ClassGroup=AbilitySystem, hideCategories=(Object, LOO_Lighting, Transform, Sockets, TextureStreaming), editInLineNew, meta=(BlueprintSpawnableComponent))
class GAMEPLAYABILITIES_API UAbilitySystemComponent : public UGameplayTasksComponent, public IGameplayTagAssetInterface, public IAbilitySystemReplicationProxyInterface
```

This is the component that we have to add to a character to be able to work with the ability.

AbilitySystemComponent **Definition** is inside of [AbilitySystemComponent.h](#) and its implementation is splited in to two files:

1. AbilitySystemComponent.cpp
2. AbilitySystemComponent_Abilities.cpp

We also have a useful global functionality located inside of the [AbilitySystemGlobal.cpp](#)

```
/** Helping function to avoid having to manually cast */
UAbilitySystemComponent* UAbilitySystemGlobals::GetAbilitySystemComponentFromActor(const AActor* Actor, bool LookForComponent)
{
    if (Actor == nullptr)
    {
        return nullptr;
    }

    const IAbilitySystemInterface* ASI = Cast<IAbilitySystemInterface>(Actor);
    if (ASI)
    {
        return ASI->GetAbilitySystemComponent();
    }

    if (LookForComponent)
    {
        /** This is slow and not desirable */
        ABILITY_LOG(Warning, TEXT("GetAbilitySystemComponentFromActor called on %s that is not IAbilitySystemInterface. This slow!"), *Actor->GetName());
        return Actor->FindComponentByClass<UAbilitySystemComponent>();
    }

    return nullptr;
}
```

It has for example a useful function named [GetAbilitySystemComponentFromActor\(\)](#) that we'll using a lot.

[AbilitySystemBlueprintLibrary.cpp](#) will give you the access to global helper functions from Blueprints.

```
UAbilitySystemComponent* UAbilitySystemBlueprintLibrary::GetAbilitySystemComponent(AActor *Actor)
{
    return UAbilitySystemGlobals::GetAbilitySystemComponentFromActor(Actor);
```

In the code above we can see that this function provides us with a Blueprint wrapper for to get an [GetAbilitySystemComponentFromActor\(\)](#) from [AbilitySystemGlobal](#).

Going back to the [AbilitySystemComponent.h](#) can see from the comments:

```
/** UAbilitySystemComponent
 *
 * A component to easily interface with the 3 aspects of the AbilitySystem:
 *
 * GameplayAbilities:
 * -Provides a way to give/assign abilities that can be used (by a player or AI for example)
 * -Provides management of instanced abilities (something must hold onto them)
 * -Provides replication functionality
 * -Ability state must always be replicated on the UGameplayAbility itself, but UAbilitySystemComponent provides RPC replication
 *   for the actual activation of abilities
 *
 * GameplayEffects:
 * -Provides an FActiveGameplayEffectsContainer for holding active GameplayEffects
 * -Provides methods for applying GameplayEffects to a target or to self
 * -Provides wrappers for querying information in FActiveGameplayEffectsContainers (duration, magnitude, etc)
 * -Provides methods for clearing/remove GameplayEffects
 *
 * GameplayAttributes
 * -Provides methods for allocating and initializing attribute sets
 * -Provides methods for getting AttributeSets
 */

```

That provides us with the interface to work with the [GamplayAbilities](#), [GamplayEffects](#), [GameplayAttributes](#).

GamplayAbilities: before we can Activate an ability we should [give it to our Actor](#) first. We can do it with these functions:

1. UAbilitySystemComponent::[Giveability](#)(const FGameplayAbilitySpec& spec)
2. (In the Photo)

```

/*
 * Grants an Ability.
 * This will be ignored if the actor is not authoritative.
 * Returns handle that can be used in TryActivateAbility, etc.
 *
 * @param AbilitySpec FGameplayAbilitySpec containing information about the ability class, level and input ID to bind it to.
 */
FGameplayAbilitySpecHandle GiveAbility(const FGameplayAbilitySpec& AbilitySpec);

/*
 * Grants an ability and attempts to activate it exactly one time, which will cause it to be removed.
 * Only valid on the server, and the ability's Net Execution Policy cannot be set to Local or Local Predicted
 *
 * @param AbilitySpec FGameplayAbilitySpec containing information about the ability class, level and input ID to bind it to.
 * @param GameplayEventData Optional activation event data. If provided, Activate Ability From Event will be called instead of ActivateAbility, passing the Event Data
 */
FGameplayAbilitySpecHandle GiveAbilityAndActivateOnce(FGameplayAbilitySpec& AbilitySpec, const FGameplayEventData* GameplayEventData = nullptr);

/**
 * Grants a Gameplay Ability and returns its handle.
 * This will be ignored if the actor is not authoritative.
 *
 * @param AbilityClass Type of ability to grant
 * @param Level Level to grant the ability at
 * @param InputID Input ID value to bind ability activation to.
 */
UFUNCTION(BlueprintCallable, BlueprintAuthorityOnly, Category = "Gameplay Abilities", meta = (DisplayName = "Give Ability", ScriptName = "GiveAbility"))
FGameplayAbilitySpecHandle K2_GiveAbility(TSubclassOf<UGameplayAbility> AbilityClass, int32 Level = 0, int32 InputID = -1);

/**
 * Grants a Gameplay Ability, activates it once, and removes it.
 * This will be ignored if the actor is not authoritative.
 *
 * @param AbilityClass Type of ability to grant
 * @param Level Level to grant the ability at
 * @param InputID Input ID value to bind ability activation to.
 */
UFUNCTION(BlueprintCallable, BlueprintAuthorityOnly, Category = "Gameplay Abilities", meta = (DisplayName = "Give Ability And Activate Once", ScriptName = "GiveAbilityAndActivateOnce"))
FGameplayAbilitySpecHandle K2_GiveAbilityAndActivateOnce(TSubclassOf<UGameplayAbility> AbilityClass, int32 Level = 0, int32 InputID = -1);

/** Wipes all 'given' abilities. This will be ignored if the actor is not authoritative. */
UFUNCTION(BlueprintCallable, BlueprintAuthorityOnly, Category="Gameplay Abilities")
void ClearAllAbilities();

```

in the [Giveability](#) :

```

FGameplayAbilitySpecHandle UAbilitySystemComponent::GiveAbility(const FGameplayAbilitySpec& Spec)
{
    if (!IsValid(Spec.Ability))
    {
        ABILITY_LOG(Error, TEXT("GiveAbility called with an invalid Ability Class."));

        return FGameplayAbilitySpecHandle();
    }

    if (!IsOwnerActorAuthoritative())
    {
        ABILITY_LOG(Error, TEXT("GiveAbility called on ability %s on the client, not allowed!"), *Spec.Ability->GetName());

        return FGameplayAbilitySpecHandle();
    }

    // If locked, add to pending list. The Spec.Handle is not regenerated when we receive, so returning this is ok.
    if (AbilityScopeLockCount > 0)
    {
        AbilityPendingAdds.Add(Spec);
        return Spec.Handle;
    }

    ABILITYLIST_SCOPE_LOCK();
    FGameplayAbilitySpec& OwnedSpec = ActivatableAbilities.Items[ActivatableAbilities.Items.Add(Spec)];

    if (OwnedSpec.Ability->GetInstancingPolicy() == EGameplayAbilityInstancingPolicy::InstancedPerActor)
    {
        // Create the instance at creation time
        CreateNewInstanceOfAbility(OwnedSpec, Spec.Ability);
    }

    OnGiveAbility(OwnedSpec);
    MarkAbilitySpecDirty(OwnedSpec, true);

    return OwnedSpec.Handle;
}

```

In above we can see that we **shouldn't give an ability by the client.**

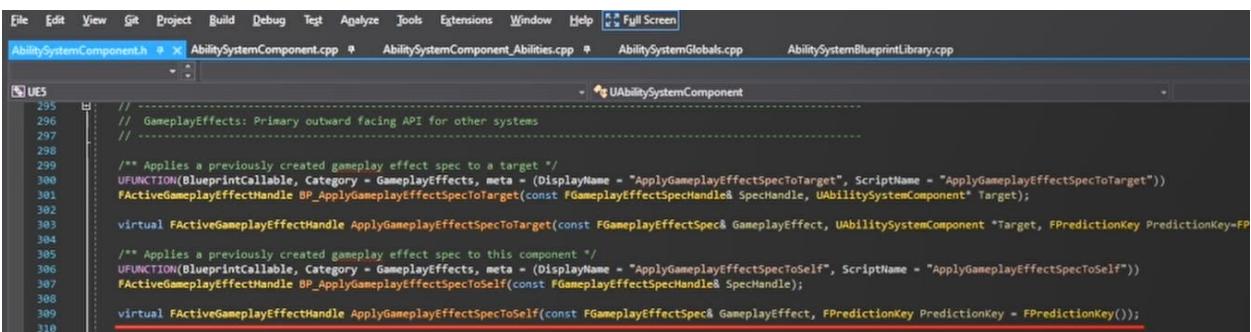
In below code we see a nice method.

```
1600 bool UAbilitySystemComponent::ReplicateSubobjects(class UActorChannel *Channel, class FOutBunch *Bunch, FReplicationFlags *RepFlags)
1601 {
1602     bool WroteSomething = Super::ReplicateSubobjects(Channel, Bunch, RepFlags);
1603
1604     for (const UAttributeSet* Set : GetSpawnedAttributes())
1605     {
1606         if (IsValid(Set))
1607         {
1608             WroteSomething |= Channel->ReplicateSubobject(const_cast<UAttributeSet*>(Set), *Bunch, *RepFlags);
1609         }
1610     }
1611
1612     for (UGameplayAbility* Ability : AllReplicatedInstancedAbilities)
1613     {
1614         if (IsValid(Ability))
1615         {
1616             WroteSomething |= Channel->ReplicateSubobject(Ability, *Bunch, *RepFlags);
1617         }
1618     }
1619
1620     return WroteSomething;
1621 }
1622
1623 }
```

AllReplicatedInstancedAbilities container is marked as not replicated. But we iterating over it and explicitly call for Subobject Replication, this is a very important trick, because [that's how we can Replicated an Array of Dynamically Allocated UObject](#)s. in the above for loop we can see we also using it to replicate AttributeSet too.

Remember this approach because it may be very useful in the future.

GamplayEffects: we can apply GamplayEffects with the `ApplyGameplayEffectSpecToSelf` function.



```
File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen
AbilitySystemComponent.h AbilitySystemComponent.cpp AbilitySystemComponent_Abilities.cpp AbilitySystemGlobals.cpp AbilitySystemBlueprintLibrary.cpp
UES UAbilitySystemComponent
295 // ...
296 // GamplayEffects: Primary outward facing API for other systems
297 // ...
298
299 /** Applies a previously created gameplay effect spec to a target */
300 UFUNCTION(BlueprintCallable, Category = GamplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToTarget", ScriptName = "ApplyGameplayEffectSpecToTarget"))
301 FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpecHandle& SpecHandle, UAbilitySystemComponent* Target);
302
303 virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpec& GameplayEffect, UAbilitySystemComponent *Target, FPredictionKey PredictionKey=FPredictionKey);
304
305 /** Applies a previously created gameplay effect spec to this component */
306 UFUNCTION(BlueprintCallable, Category = GamplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToSelf", ScriptName = "ApplyGameplayEffectSpecToSelf"))
307 FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpecHandle& SpecHandle);
308
309 virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpec& GameplayEffect, FPredictionKey PredictionKey = FPredictionKey());
310
```

If the GamplayEffects have a [non-instance](#) effect, we will add it to the [ActiveGameplayEffects](#) container



```
791 FGameplayEffectSpec* OurCopyOfSpec = nullptr;
792 TSharedPtr<FGameplayEffectSpec> StackSpec;
793 {
794     if (Spec.Def->DurationPolicy != EGameplayEffectDurationType::Instant || bTreatAsInfiniteDuration)
795     {
796         AppliedEffect = ActiveGameplayEffects.ApplyGameplayEffectSpec(Spec, PredictionKey, bFoundExistingStackableGE);
797         if (!AppliedEffect)
798         {
799             return FActiveGameplayEffectHandle();
800         }
801     }
802 }
```

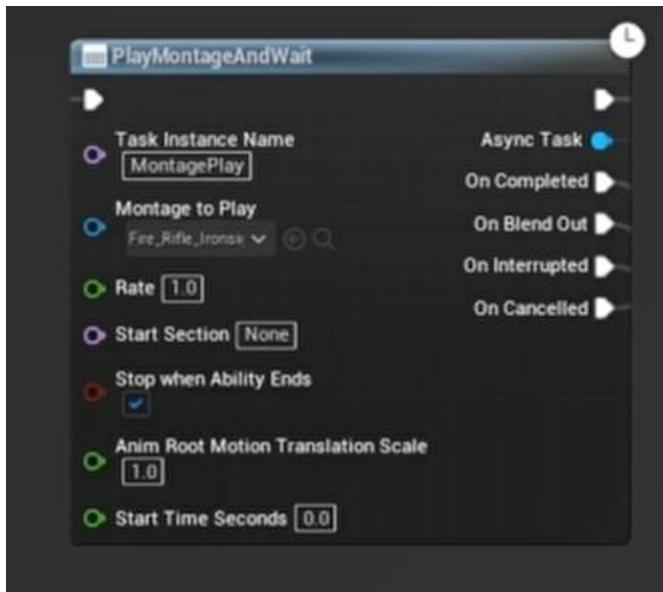
And that container among Cue containers are marked as Replicated.

```
1755     /** Contains all of the gameplay effects that are currently active on this component */
1756     UPROPERTY(Replicated)
1757     FActiveGameplayEffectsContainer ActiveGameplayEffects;
1758
1759     /** List of all active gameplay cues, including ones applied manually */
1760     UPROPERTY(Replicated)
1761     FActiveGameplayCueContainer ActiveGameplayCues;
1762
1763     /** Replicated gameplaycues when in minimal replication mode. These are cues that would come normally come from ActiveGameplayEffects */
1764     UPROPERTY(Replicated)
1765     FActiveGameplayCueContainer MinimalReplicationGameplayCues;
1766
1767 
```

One important future of `GameplayEffect` that is worth mentioning, is `AnimMontage Playing in Replication`:

```
1722 UE_DEPRECATED(4.26, "This will be made private in future engine versions. Use SetRepAnimMontageInfo, GetRepAnimMontageInfo, or GetRepAnimMontageInfo_Mutable instead.")
1723 /* Data structure for replicating montage info to simulated clients */
1724 UPROPERTY(ReplicatedUsing=OnRep_ReplicatedAnimMontage)
1725 FGameplayAbilityRepAnimMontage RepAnimMontageInfo;
1726
```

It's done with the `FGameplayAbilityLocalAnimMontage` structure (the return type in the picture is a struct).



This `PlayMontageAndWait` actually uses the ability system component to play and replicate the montage.

Specially to the simulated proxies because the abilities won't directly execute there.

```

1768     /** Abilities with these tags are not able to be activated */
1769     FGameplayTagCountContainer BlockedAbilityTags;
1770
1771     UE_DEPRECATED(4.26, "This will be made private in future engine versions. Use SetBlockedAbilityBindings, GetBlockedAbilityBindings, or GetBlockedAbilityBindings_Mutable instead.")
1772     /** Tracks abilities that are blocked based on input binding. An ability is blocked if BlockedAbilityBindings[InputID] > 0 */
1773     UPROPERTY(Transient, Replicated)
1774     TArray<uint8> BlockedAbilityBindings;
1775
1776     void SetBlockedAbilityBindings(const TArray<uint8>& NewBlockedAbilityBindings);
1777     TArray<uint8> GetBlockedAbilityBindings_Mutable();
1778     const TArray<uint8> GetBlockedAbilityBindings() const;
1779
1780     void DebugCyclicAggregatorBroadcasts(struct FAggregator* Aggregator);
1781
1782     /** Acceleration map for all gameplay tags (OwnedGameplayTags from GEs and explicit GameplayCueTags) */
1783     FGameplayTagCountContainer GameplayTagCountContainer;
1784
1785     UE_DEPRECATED(4.26, "This will be made private in future engine versions. Use SetMinimalReplicationTags, GetMinimalReplicationTags, or GetMinimalReplicationTags_Mutable instead.")
1786     UPROPERTY(Replicated)
1787     #MinimalReplicationTagCountMap MinimalReplicationTags;

```

The **AbilitySystemComponent** is used to store applied **Gameplay Tags** which they're also **replicated**.

3. Gameplay Ability

GameplayAbility is **thrived** from the **UObject**.

The way we handle an Ability is defined from a set of Policies:

```

142     /** Returns how the ability is instanced when executed. This limits what an ability can do in its implementation. */
143     EGameplayAbilityInstancingPolicy::Type GetInstancingPolicy() const
144     {
145         return InstancingPolicy;
146     }
147
148     /** How an ability replicates state/events to everyone on the network */
149     EGameplayAbilityReplicationPolicy::Type GetReplicationPolicy() const
150     {
151         return ReplicationPolicy;
152     }
153
154     /** Where does an ability execute on the network? Does a client "ask and predict", "ask and wait", "don't ask (just do it)" */
155     EGameplayAbilityNetExecutionPolicy::Type GetNetExecutionPolicy() const
156     {
157         return NetExecutionPolicy;
158     }
159
160     /** Where should an ability execute on the network? Provides protection from clients attempting to execute restricted abilities. */
161     EGameplayAbilityNetSecurityPolicy::Type GetNetSecurityPolicy() const
162     {
163         return NetSecurityPolicy;
164     }

```

1. GameplayAbilityInstancingPolicy:

```

GameplayAbilityTypes.h ➔ X GameplayAbility.h          GameplayAbilityTypes.cpp      AbilitySystemComponent.h      Test1_THOPCharacter.h      Test1_THOPCharacter.cpp
➔ TStructOpsTypeTraits<FGar... ➔ template<> struct TStructOpsTypeTraits<FGameplayAbilityRepAnimMontage> : public TStructOpsTypeTraitsBase2<FGameplayAbilityRepAnimMontag
UE5
33 #define ENABLE_ABILITYTASK_DEBUGMSG !(UE_BUILD_SHIPPING | UE_BUILD_TEST)
34
35
36
37 UENUM(BlueprintType)
38 namespace EGameplayAbilityInstancingPolicy
39 {
40     /**
41     * How the ability is instanced when executed. This limits what an ability can do in its implementation. For example, a NonInstanced
42     * Ability cannot have state. It is probably unsafe for an InstancedPerActor ability to have latent actions, etc.
43     */
44     enum Type
45     {
46         // This ability is never instanced. Anything that executes the ability is operating on the CDO.
47         NonInstanced,
48
49         // Each actor gets their own instance of this ability. State can be saved, replication is possible.
50         InstancedPerActor,
51
52         // We instance this ability each time it is executed. Replication possible but not recommended.
53         InstancedPerExecution,
54     };
55

```

It Defines **how** and **when** we want to create an instance of an Ability.

1. **non-instance**, we gonna use the CDO(class default object)
2. **InstancedPerActor**, will be created at a time we give this ability to an actor.
3. **InstancedPerExecution**, will be created at a time we try to activate an ability.

For **InstancedPerActor** we create it like this (these codes are in the **AbilitySystemComponent_Abilities.cpp**):

```
273
274     if (OwnedSpec.Ability->GetInstancingPolicy() == EGameplayAbilityInstancingPolicy::InstancedPerActor)
275     {
276         // Create the instance at creation time
277         CreateNewInstanceOfAbility(OwnedSpec, Spec.Ability);
278     }
279 
```

But for **non-instance** not even using any Instance Object, but directly calling function on the class default. This have some limitations because we can't change this object at run-time.

```
1633     else
1634     {
1635         Ability->CallActivateAbility(Handle, ActorInfo, ActivationInfo, OnGameplayAbilityEndedDelegate, TriggerEventData);
1636     }
1637
1638     else if (Ability->GetNetExecutionPolicy() == EGameplayAbilityNetExecutionPolicy::LocalPredicted) 
```

In case of **InstancedPerExecution** we'll create an Instance right before we try to activate an ability.

```
// Create instance of this ability if necessary
if (Ability->GetInstancingPolicy() == EGameplayAbilityInstancingPolicy::InstancedPerExecution)
{
    InstancedAbility = CreateNewInstanceOfAbility("Spec", Ability);
    InstancedAbility->CallActivateAbility(Handle, ActorInfo, ActivationInfo, OnGameplayAbilityEndedDelegate, TriggerEventData); 
```

2. NetExecutionPolicy:

```
UENUM(BlueprintType)
namespace EGameplayAbilityNetExecutionPolicy
{
    /** Where does an ability execute on the network. Does a client "ask and predict", "ask and wait", "don't ask (just do it)" */
    enum Type
    {
        // Part of this ability runs predictively on the local client if there is one
        LocalPredicted      UMETADATA(DisplayName = "Local Predicted"),

        // This ability will only run on the client or server that has local control
        LocalOnly           UMETADATA(DisplayName = "Local Only"),

        // This ability is initiated by the server, but will also run on the local client if one exists
        ServerInitiated     UMETADATA(DisplayName = "Server Initiated"),

        // This ability will only run on the server
        ServerOnly          UMETADATA(DisplayName = "Server Only"),
    };
} 
```

It Defines where the ability will Execute.

1. [LocalPredicted](#), first only on client then the server. [Client->Server](#)
2. [LocalOnly](#), or on client or on server. [Client\(Server\)](#)
3. [ServerInitiated](#), run on server first then on client. [Server->Client](#)
4. [ServerOnly](#), will run only on server. [Server](#)

Ability Execution can start in several different ways. For example we can use [TryActivateAbilitiesByTag](#) or [TryActivateAbilityByClass](#).

```
AbilitySystemComponent_Abilities.cpp  AbilitySystemComponent.h  GameplayAbilityTypes.h  GameplayAbility.h  GameplayAbilityTypes.cpp
→ UAbilitySystemComponent. → for (const FGameplayAbilitySpec& Spec : ActivatableAbilities.Items) → UAbilitySystemComponent → GetActivatableGameplay
UE5
1296     }
1297     }
1298     }
1299   }
1300 
1301   Ebool UAbilitySystemComponent::TryActivateAbilitiesByTag(const FGameplayTagContainer& GameplayTagContainer, bool bAllowRemoteActivation)
1302   {
1303     TArray<FGameplayAbilitySpec> AbilitiesToActivate;
1304     GetActivatableGameplayAbilitySpecsByAllMatchingTags(GameplayTagContainer, AbilitiesToActivate);
1305 
1306     bool bSuccess = false;
1307 
1308     for (auto GameplayAbilitySpec : AbilitiesToActivate)
1309     {
1310       bSuccess |= TryActivateAbility(GameplayAbilitySpec->Handle, bAllowRemoteActivation);
1311     }
1312 
1313     return bSuccess;
1314   }
1315 
1316   Ebool UAbilitySystemComponent::TryActivateAbilityByClass(TSubclassOf<UGameplayAbility> InAbilityToActivate, bool bAllowRemoteActivation)
1317   {
1318     bool bSuccess = false;
1319 
1320     const UGameplayAbility* const InAbilityCDO = InAbilityToActivate.GetDefaultObject();
1321 
1322     for (const FGameplayAbilitySpec& Spec : ActivatableAbilities.Items)
1323     {
1324       if (Spec.Ability == InAbilityCDO)
1325       {
1326         bSuccess |= TryActivateAbility(Spec.Handle, bAllowRemoteActivation);
1327         break;
1328       }
1329     }
1330 
1331     return bSuccess;
1332   }

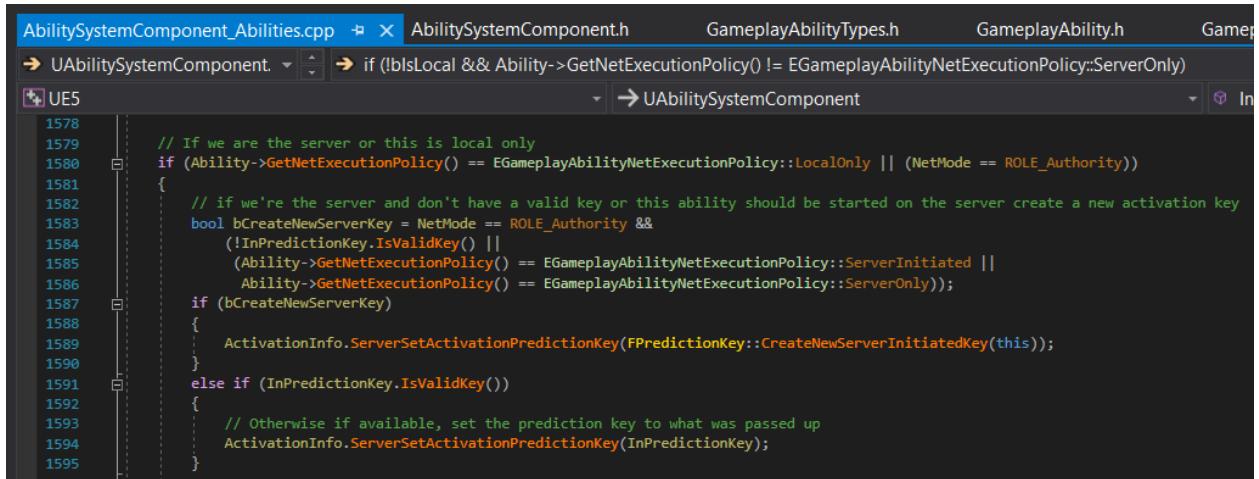
```

In a [special case](#) in which we [don't have an authority](#) but we start the ability for [ServerOnly](#) or [ServerInitiated](#). This can be possible if we set the [bAllowRemoteActivastion](#). In this case we'll directly move the ability activation to the server:

```
1383   if (NetMode != ROLE_Authority && (Ability->GetNetExecutionPolicy() == EGameplayAbilityNetExecutionPolicy::ServerOnly || Ability->GetNetExecutionPolicy() == EGameplayAbilityNetExecutionPolicy::ServerInitiated))
1384   {
1385     if (bAllowRemoteActivation)
1386     {
1387       if (Ability->CanActivateAbility(AbilityToActivate, ActorInfo, nullptr, nullptr, &FailureTags))
1388       {
1389         // No prediction key, server will assign a server-generated key
1390         CallServerTryActivateAbility(AbilityToActivate, Spet->InputPressed, FPredictionKey());
1391         return true;
1392       }
1393       else
1394       {
1395         NotifyAbilityFailed(AbilityToActivate, Ability, FailureTags);
1396         return false;
1397       }
1398     }
1399 
1400     ABILITY_LOG(Log, TEXT("Can't activate ServerOnly or ServerInitiated ability %s when not the server."), *Ability->GetName());
1401     return false;
1402   }
1403 
1404   return InternalTryActivateAbility(AbilityToActivate);
1405 }
```

And for all other cases we go to [InternalTryActivateAbility](#) (all of them are in the above picture)

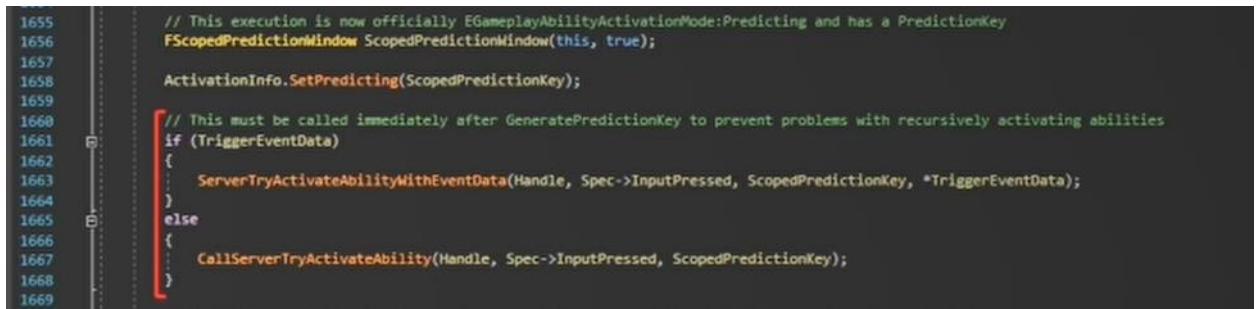
In case of [LocalOnly](#) or [Authority](#) we create and set the [Predictionkey](#).



```
1578 // If we are the server or this is local only
1579 if (Ability->GetNetExecutionPolicy() == EGameplayAbilityNetExecutionPolicy::LocalOnly || (NetMode == ROLE_Authority))
1580 {
1581     // if we're the server and don't have a valid key or this ability should be started on the server create a new activation key
1582     bool bCreateNewServerKey = NetMode == ROLE_Authority &&
1583         (!InPredictionKey.IsValidKey()) ||
1584         (Ability->GetNetExecutionPolicy() == EGameplayAbilityNetExecutionPolicy::ServerInitiated || 
1585          Ability->GetNetExecutionPolicy() == EGameplayAbilityNetExecutionPolicy::ServerOnly);
1586     if (bCreateNewServerKey)
1587     {
1588         ActivationInfo.ServerSetActivationPredictionKey(FPredictionKey::CreateNewServerInitiatedKey(this));
1589     }
1590     else if (InPredictionKey.IsValidKey())
1591     {
1592         // Otherwise if available, set the prediction key to what was passed up
1593         ActivationInfo.ServerSetActivationPredictionKey(InPredictionKey);
1594     }
1595 }
```

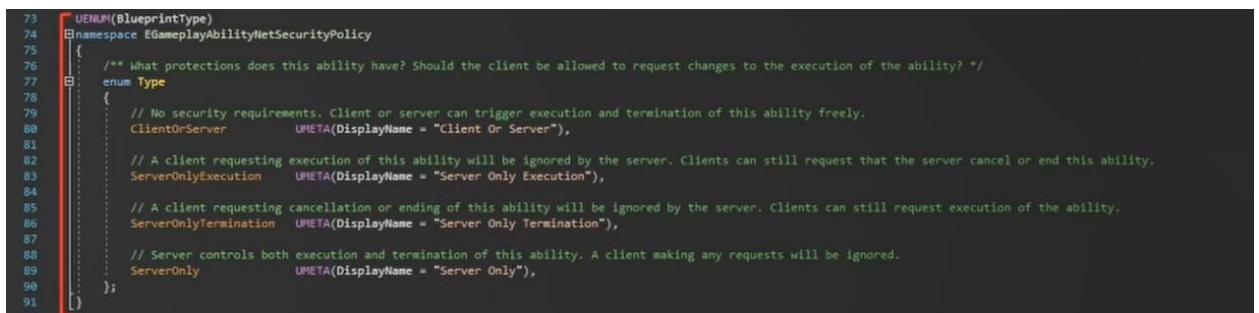
[PredictionKey](#) is an entity that will be associated with the predictive actions, and can be used to identify and roll back them if server decides.

In case of [LocalPredicted](#) we first create and set Prediction window and then activate the ability with [EventData](#) or without.



```
1655 // This execution is now officially EGameplayAbilityActivationMode::Predicting and has a PredictionKey
1656 FScopedPredictionWindow ScopedPredictionWindow(this, true);
1657
1658 ActivationInfo.SetPredicting(ScopedPredictionKey);
1659
1660 // This must be called immediately after GeneratePredictionKey to prevent problems with recursively activating abilities
1661 if (TriggerEventData)
1662 {
1663     ServerTryActivateAbilityWithEventData(Handle, Spec->InputPressed, ScopedPredictionKey, *TriggerEventData);
1664 }
1665 else
1666 {
1667     CallServerTryActivateAbility(Handle, Spec->InputPressed, ScopedPredictionKey);
1668 }
1669 }
```

3. NetSecurityPolicy:



```
73 UENUM(BlueprintType)
74 ENamespace EGameplayAbilityNetSecurityPolicy
75 {
76     /* What protections does this ability have? Should the client be allowed to request changes to the execution of the ability? */
77     enum Type
78     {
79         // No security requirements. Client or server can trigger execution and termination of this ability freely.
80         ClientOrServer   UMETADATA(DisplayName = "Client Or Server"),
81
82         // A client requesting execution of this ability will be ignored by the server. Clients can still request that the server cancel or end this ability.
83         ServerOnlyExecution   UMETADATA(DisplayName = "Server Only Execution"),
84
85         // A client requesting cancellation or ending of this ability will be ignored by the server. Clients can still request execution of the ability.
86         ServerOnlyTermination   UMETADATA(DisplayName = "Server Only Termination"),
87
88         // Server controls both execution and termination of this ability. A client making any requests will be ignored.
89         ServerOnly   UMETADATA(DisplayName = "Server Only"),
90     };
91 }
```

Is responsible to check that can restrict where the Ability [can execute](#).

4. ReplicationPolicy:

```
//
// 93 UENUM(BlueprintType)
// 94 EGameplayAbilityReplicationPolicy
// 95 {
// 96     /* How an ability replicates state/events to everyone on the network */
// 97     enum Type
// 98     {
// 99         // We don't replicate the instance of the ability to anyone.
//100        ReplicateNo      UPROPERTY(DisplayName = "Do Not Replicate"),
//101
//102        // We replicate the instance of the ability to the owner.
//103        ReplicateYes     UPROPERTY(DisplayName = "Replicate"),
//104    };
//105 }
//106
```

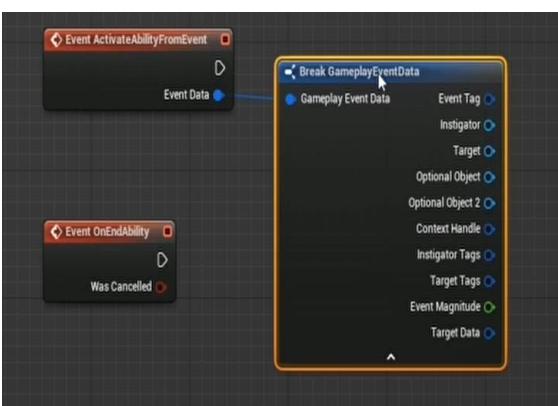
It Defines whether we replicate an instance of the ability to owner or not.

Some important functions:

```
//
// -----
// The important functions:
// -----
// CanActivateAbility() - const function to see if ability is activatable. Callable by UI etc
// TryActivateAbility() - Attempts to activate the ability. Calls CanActivateAbility(). Input events can call this directly.
// - Also handles instancing-per-execution logic and replication/prediction calls.
// CallActivateAbility() - Protected, non virtual function. Does some boilerplate 'pre activate' stuff, then calls ActivateAbility()
// ActivateAbility() - What the abilities *does*. This is what child classes want to override.
// CommitAbility() - Commits resources/cooldowns etc. ActivateAbility() must call this!
// CancelAbility() - Interrupts the ability (from an outside source).
// EndAbility() - The ability has ended. This is intended to be called by the ability to end itself.
// -----
// -----
// Accessors
// -----
```

The way we handle the ability execution flow. We can do that by overwriting some main function in child class. (The above functions)

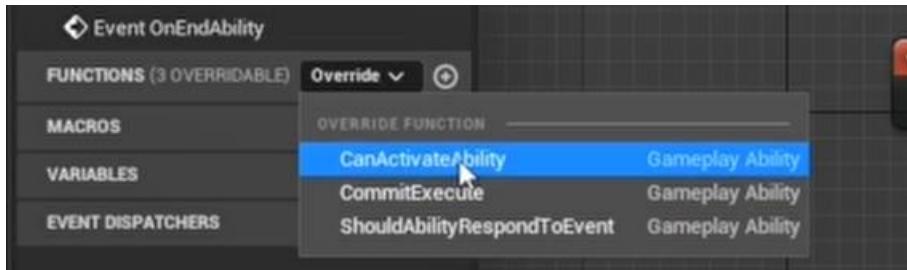
Also some of them are available as Events and some we can just overwrite to.



[ActivateAbilityFromEvent](#) provides us with the event data, that allows us to **pass additional data to the ability**.

By overwriting this function we will **write what the ability will be doing exactly**.(like playing the montage or spawning the magic spell)

Here we are overwriting [CanActivateAbility](#) Function and getting some data from actor info.

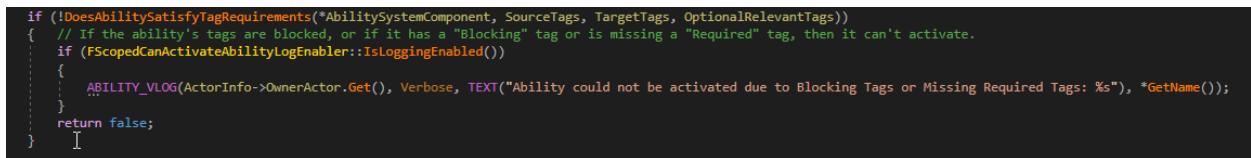


Ability relation with other abilities / the outside world is defined by Tags.

All of them are stored as [GamplayTagContainer](#).



They will be checked inside of the [DoesAbilitySatisfyTagRequirements](#) function which is inside of the [CanActivateAbility](#) function.



```

bool UGameplayAbility::CanActivateAbility(const FGameplayAbilitySpecHandle Handle, const FGameplayAbilityActorInfo* ActorInfo, const FGameplayTagContainer* SourceTags, const FGameplayAbilitySpec* TargetAbility)
{

```

Which is called before the activation.

And in the `DoesAbilitySatisfyTagRequirements` we form different checks to decide whether the ability is blocked from the activation or not(bellow pic is not the whole function)

```

193 EBool UGameplayAbility::DoesAbilitySatisfyTagRequirements(const UAbilitySystemComponent* AbilitySystemComponent, const FGameplayTagContainer* SourceTags, const FGameplayTagContainer* TargetTags, OUT FGameplayTagContainer* OptionalRelevantTags) const
194 {
195     bool bBlocked = false;
196     bool bMissing = false;
197
198     UAbilitySystemGlobals::AbilitySystemGlobals = UAbilitySystemGlobals::Get();
199     const FGameplayTag BlockedTag = AbilitySystemGlobals->ActivateAbilityBlockedTag;
200     const FGameplayTag MissingTag = AbilitySystemGlobals->ActivateAbilityMissingTag;
201
202     // Check if any of this ability's tags are currently blocked
203     if (AbilitySystemComponent->AreAbilityTagsBlocked(SourceTags))
204     {
205         bBlocked = true;
206     }
207
208     // Check to see the required/blocked tags for this ability
209     if (ActivationBlockedTags.Num() || ActivationRequiredTags.Num())
210     {
211         static FGameplayTagContainer AbilitySystemComponentTags;
212         AbilitySystemComponentTags.Reset();
213
214         AbilitySystemComponent->GetOwnedGameplayTags(AbilitySystemComponentTags);
215
216         if (AbilitySystemComponentTags.HasAny(ActivationBlockedTags))
217         {
218             bBlocked = true;
219         }
220
221         if (AbilitySystemComponentTags.HasAll(ActivationRequiredTags))
222         {
223             bMissing = true;
224         }
225     }
226 }

```

4. Ability Tasks

Are `UObjects` that can **execute logic inside of our ability**. The task can be written in cpp. The creation of a task from code works through a `static function` which can also be made accessible through Blueprint.

```

1 //FUNCTION(BlueprintCallable, Category="Ability|Tasks", meta = (DisplayName="PlayMontageAndWait",
2 HideIn= "OwningAbility", DefaultToSelf = "OwningAbility", BlueprintInternalUseOnly = "TRUE")
3 static UAbilityTask_PlayMontageAndWait* CreatePlayMontageAndWaitProxy(UGameplayAbility* OwningAbility,
4 FName TaskInstanceName, UAnimMontage* MontageToPlay, float Rate, FName StartSection = NAME_None, bool bStopWhenAbilityEnds = true, float AnimRootMotionTranslationScale = 1.f, float StartTimeSeconds = 0.f);

```

It will create and return the new instance of a task a preform some initial set-ups

```

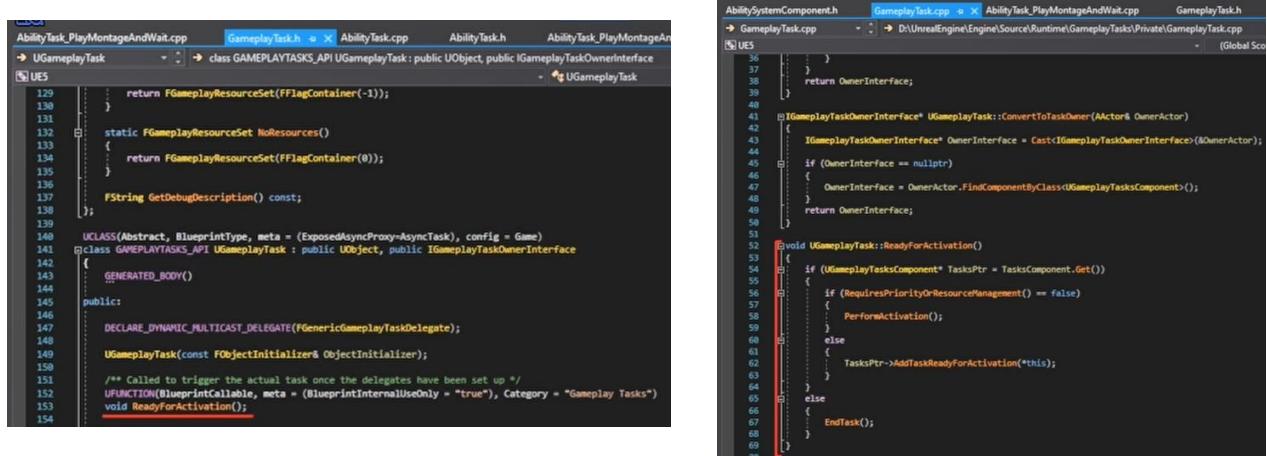
Ability_Task_PlayMontageAndWait.cpp  □  GameplayTask.h      AbilityTask.cpp    AbilityTask.h    Ability_Task_PlayMontageAndWait.h
AbilityTask_PlayMontageAndWait.h → D:\UnrealEngine\Engine\Plugins\Runtime\GameplayAbilities\Source\GameplayAbilities\Private\Abilities\Tasks\AbilityTask_PlayMontageAndWait.cpp
AbilityTask_PlayMontageAndWait.h → (Global Scope)
58     ...
59     ...
60 }
61
62 void UAbilityTask_PlayMontageAndWait::OnMontageEnded(UAnimMontage* Montage, bool bInterrupted)
63 {
64     if (!bInterrupted)
65     {
66         if (ShouldBroadcastAbilityTaskDelegates())
67         {
68             OnCompleted.Broadcast();
69         }
70     }
71
72     EndTask();
73 }
74
75 UAbilityTask_PlayMontageAndWait* UAbilityTask_PlayMontageAndWait::CreatePlayMontageAndWaitProxy(UGameplayAbility* OwningAbility,
76                                             FName TaskInstanceName, UAnimMontage* MontageToPlay, float Rate, FName StartSection, bool bStopWhenAbilityEnds, float AnimRootMotionTranslationScale, float StartTimeSeconds)
77 {
78
79     UAbilitySystemGlobals::NonShipping_ApplyGlobalAbilityScaler_Rate(Rate);
80
81     UAbilityTask_PlayMontageAndWait* MyObj = NewAbilityTask<UAbilityTask_PlayMontageAndWait>(OwningAbility, TaskInstanceName);
82     MyObj->MontageID.Play = MontageID.Play;
83     MyObj->Rate = Rate;
84     MyObj->StartSection = StartSection;
85     MyObj->AnimRootMotionTranslationScale = AnimRootMotionTranslationScale;
86     MyObj->bStopWhenAbilityEnds = bStopWhenAbilityEnds;
87     MyObj->StartTimeSeconds = StartTimeSeconds;
88
89     return MyObj;
90 }

```

We can declare different delegates to send messages to the ability during task execution.

```
14 Eclass GAMEPLAYABILITIES_API UAbilityTask_PlayMontageAndWait : public UAbilityTask
15 {
16     GENERATED_UCLASS_BODY()
17
18     UPROPERTY(BlueprintAssignable)
19     FMontageWaitSimpleDelegate OnCompleted;
20
21     UPROPERTY(BlueprintAssignable)
22     FMontageWaitSimpleDelegate OnBlendOut;
23
24     UPROPERTY(BlueprintAssignable)
25     FMontageWaitSimpleDelegate OnInterrupted;
26
27     UPROPERTY(BlueprintAssignable)
28     FMontageWaitSimpleDelegate OnCancelled;
```

If we want to start a task from c++, we can create a similar static function to create an Instance and then call [ReadyForActivation](#).



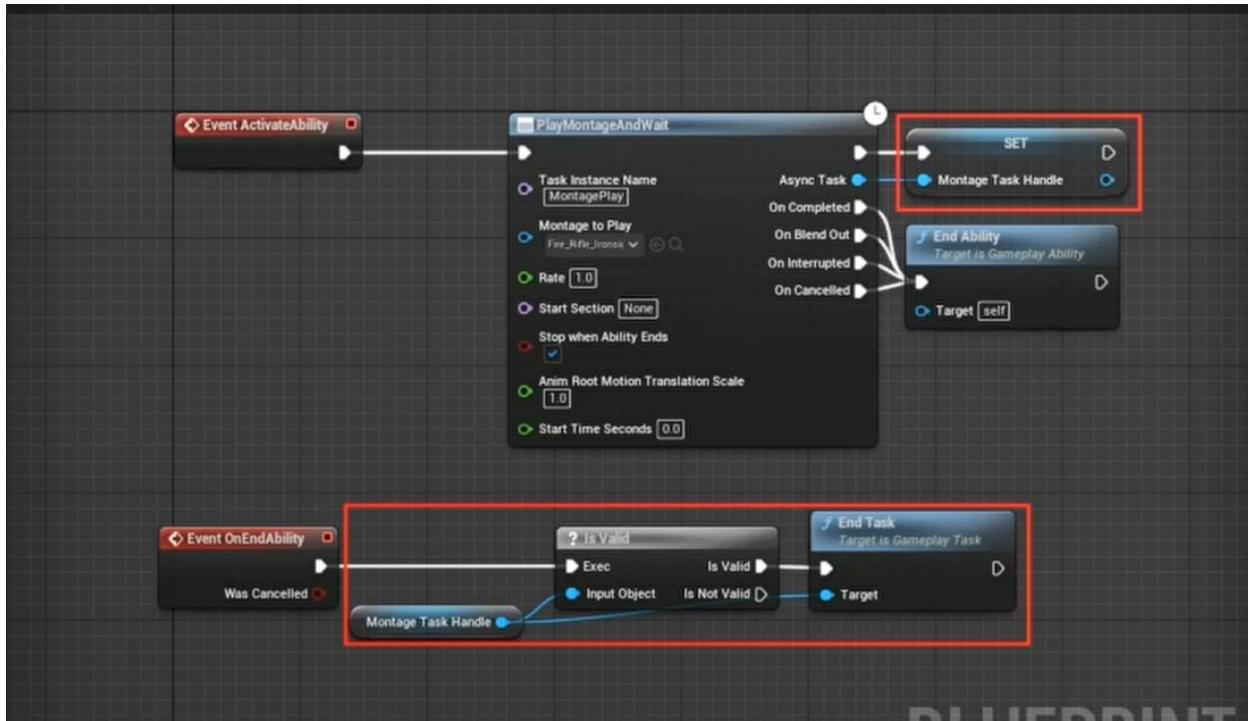
Inside of the `ReadyForActivation` function we'll see it refers to a `GameplayTaskComponent`.

`GameplayTaskComponent` is the `AbilitySystemComponent` direct parent before `ActorComponent`.

So we can say that the AbilitySystemComponent is also responsible for managing the AbilityTask.



It's considered good practice to keep a task handle and **properly end** and **clean the task when the ability ends.**



The most common example of the ability task is the Montage Play task. The montage it plays would be replicated by the ability system component because it's logic is directly supported by it.

This is important because the **Root motion replication is works with montages** so will be using it a lot.

```

GameplayTasksComponent.h          AbilitySystemComponent.h          GameplayTask.cpp          AbilityTask_PlayMontageAndWait.cpp          GameplayTask.h          AbilityTask.cpp          AbilityTask.h          AbilityTask_P.h
AbilityTask_PlayMontageAndWait.h  > D:\UnrealEngine\Engine\Plugins\Runtime\GameplayAbilities\Source\GameplayAbilities\Private\Abilities\Tasks\AbilityTask_PlayMontageAndWait.cpp
UE5
91 void UAbilityTask_PlayMontageAndWait::Activate()
92 {
93     if (Ability == nullptr)
94     {
95         return;
96     }
97
98     bPlayedMontage = false;
99
100    if (AbilitySystemComponent)
101    {
102        const FGameplayAbilityActorInfo* ActorInfo = Ability->GetCurrentActorInfo();
103        UAnimInstance* AnimInstance = ActorInfo->GetAnimInstance();
104        if (AnimInstance != nullptr)
105        {
106            if (AbilitySystemComponent->PlayMontage(Ability, Ability->GetCurrentActivationInfo(), MontageToPlay, Rate, StartSection, StartTimeSeconds) > 0.f)
107            {
108                // Playing a montage could potentially fire off a callback into game code which could kill this ability! Early out if we are pending kill.
109                if (ShouldBroadcastAbilityTaskDelegates() == false)
110                {
111                    return;
112                }
113            }
114
115            InterruptedHandle = Ability->OnGameplayAbilityCancelled.AddUObject(this, &UAbilityTask_PlayMontageAndWait::OnMontageInterrupted);
116
117            BlendingOutDelegate.BindUObject(this, &UAbilityTask_PlayMontageAndWait::OnMontageBlendingOut);
118            AnimInstance->Montage_SetBlendingOutDelegate(BlendingOutDelegate, MontageToPlay);
119
120            MontageEndedDelegate.BindUObject(this, &UAbilityTask_PlayMontageAndWait::OnMontageEnded);
121            AnimInstance->Montage_SetEndDelegate(MontageEndedDelegate, MontageToPlay);
122
123            ACharacter* Character = Cast<ACharacter>(GetAvatarActor());
124            if (Character && (Character->GetLocalRole() == ROLE_Authority ||
125                                (Character->GetLocalRole() == ROLE_AutonomousProxy && Ability->GetNetExecutionPolicy() == EGameplayAbilityNetExecutionPolicy::LocalPredicted)))
126            {
127                Character->SetAnimRootMotionTranslationScale(AnimRootMotionTranslationScale);
128            }
129
130            bPlayedMontage = true;
131        }
132    }
133
134    else
135    {
136        ABILITY_LOG(Warning, TEXT("UAbilityTask_PlayMontageAndWait call to PlayMontage failed!"));
137    }
138}

```

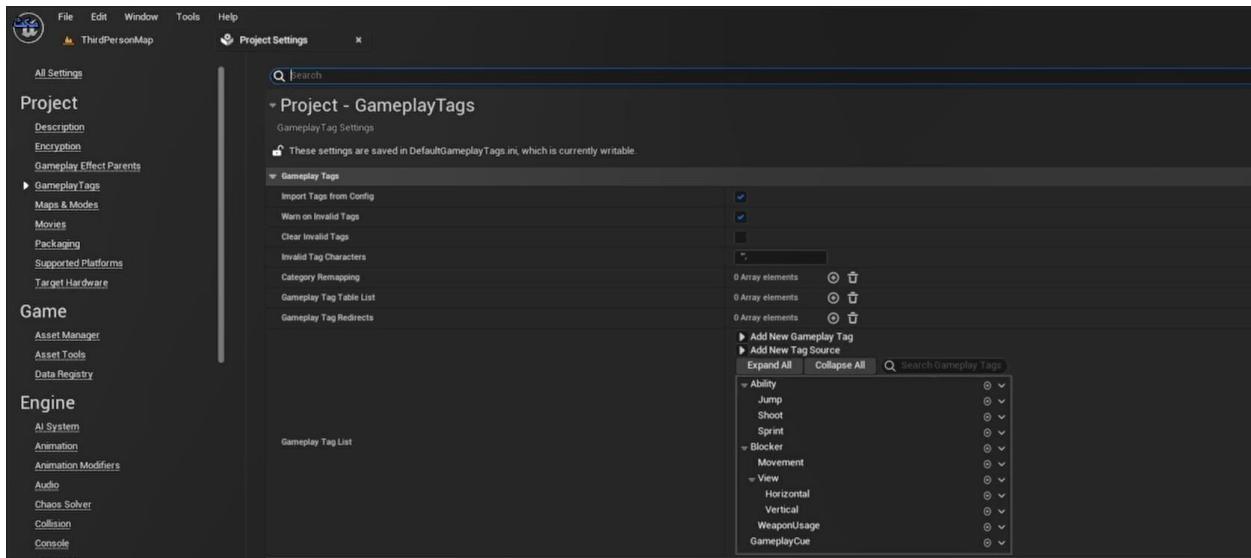
Warning: The `AbilitySystemComponent` was designed to handle one montage at a time. So if we have a complex montage playing structure and expect to your montage to end when the ability ends, you'll be having to care about its ending additionally on simulated proxies.

AbilityTask's can run a `synchronously` and `can Tick`. So they are useful if you need some logic to execute repeatedly. When you don't need Tick anymore, you can just stop the Task without ending the ability.

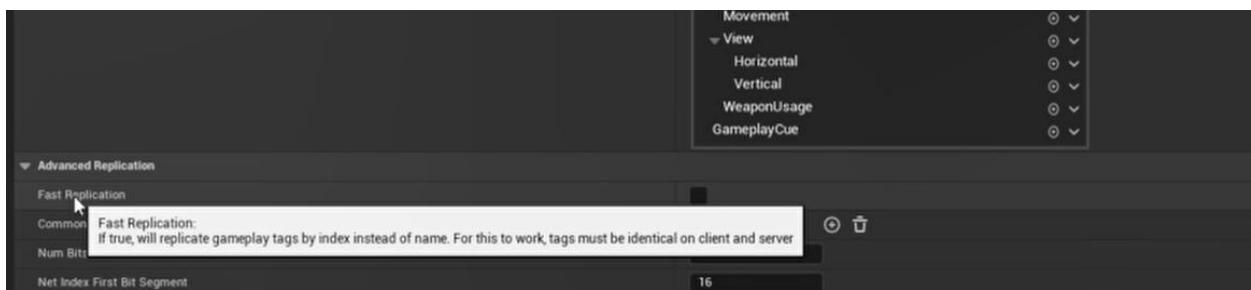
Tasks are also good for composition because you can reuse one task in many abilities.

5. Gameplay Tags

Are unique hierarchical keys that can be used for many purposes.



They look like strings, but because we do not intend to add or remove the Tags in runtime, we can replicate them with their indexes which makes replication lightweight.



After making that true we'll have:

The screenshot shows the Unreal Engine Editor interface. At the top, there's a search bar for "Search Gameplay Tags" and a list of tags: Ability, Blocker, and GameplayCue. Below this is the "Gameplay Tag List" panel, which is currently empty. Under the "Advanced Replication" section, there's a dropdown set to "True". The bottom half of the screen is a code editor showing the source code for `GameplayTagContainer.cpp`. The code handles serializing and deserializing `FGameplayTag` objects, specifically dealing with network replication logic.

```
bool FGameplayTagContainer::NetSerialize(FArchive& Ar, class UPackageMap* Map, bool& bOutSuccess)
{
    if (NetIndex != INVALID_TAGNETINDEX && ensure(NetFieldExportGroup.IsValid()) && ensure(NetIndex < NetFieldExportGroup->NetFieldExports.Num()))
    {
        UGameplayTagsManager* TagManager = UGameplayTagsManager::Get();

        if (TagManager.ShouldUseFastReplication())
        {
            FGameplayTagNetIndex NetIndex = INVALID_TAGNETINDEX;

            UPackageMapClient* PackageMapClient = Cast<UPackageMapClient>(Map);
            const bool bIsReplay = PackageMapClient && PackageMapClient->GetConnection() && PackageMapClient->GetConnection()->IsInternalAck();

            TSharedPtr<FNetFieldExportGroup> NetFieldExportGroup;
            if (!bIsReplay)
            {
                // For replays, use a net field export group to guarantee we can send the name reliably (without having to rely on the client having a deterministic NetworkGameplayTagNodeIndex array)
                const TCHAR* NetFieldExportGroupName = TEXT("NetworkGameplayTagNodeIndex");
                NetFieldExportGroup = PackageMapClient->GetNetFieldExportGroup(NetFieldExportGroupName);
            }

            if (Ar.IsSaving())
            {
                // If we didn't find it, we need to create it (only when saving though, it should be here on load since it was exported at save time)
                if (!NetFieldExportGroup.IsValid())
                {
                    NetFieldExportGroup = CreateNetfieldExportGroupForNetworkGameplayTags(TagManager, NetFieldExportGroupName);
                    PackageMapClient->AddNetfieldExportGroup(NetFieldExportGroupName, NetFieldExportGroup);
                }
            }

            NetIndex = TagManager.GetNetIndexFromTag(*this);

            if (NetIndex != TagManager.GetInvalidTagNetIndex() && NetIndex != INVALID_TAGNETINDEX)
            {
                PackageMapClient->TrackNetFieldExport(NetFieldExportGroup.Get(), NetIndex);
            }
            else
            {
                NetIndex = INVALID_TAGNETINDEX; // We can't save InvalidTagNetIndex, since the remote side could have a different value for this
            }
        }

        uint32 NetIndex32 = NetIndex;
        Ar.SerializeIntPacked(NetIndex32);
        NetIndex = NetIndex32;
    }
}
```

You can **add** `GameplayTags` from the [editor window](#) or load them from [data tables](#).

Tags can be referenced in code via the `GameplayTag` property. (we done this in our game `TPCharactor.h`):

The screenshot shows a code editor with a line of code containing `UPROPERTY(EditDefaultsOnly)` and `FGameplayTag MovementBlockingTag;`. This indicates that the `MovementBlockingTag` property is of type `FGameplayTag` and has the `EditDefaultsOnly` attribute applied.

```
97
98     UPROPERTY(EditDefaultsOnly)
99     FGameplayTag MovementBlockingTag;
100
101 protected:
```

or directly requested by name. (we done this in our game TPCharactor.cpp):

```
MovementBlockingTag = FGameplayTag::RequestGameplayTag(TEXT("Blocker.Movement"));
```

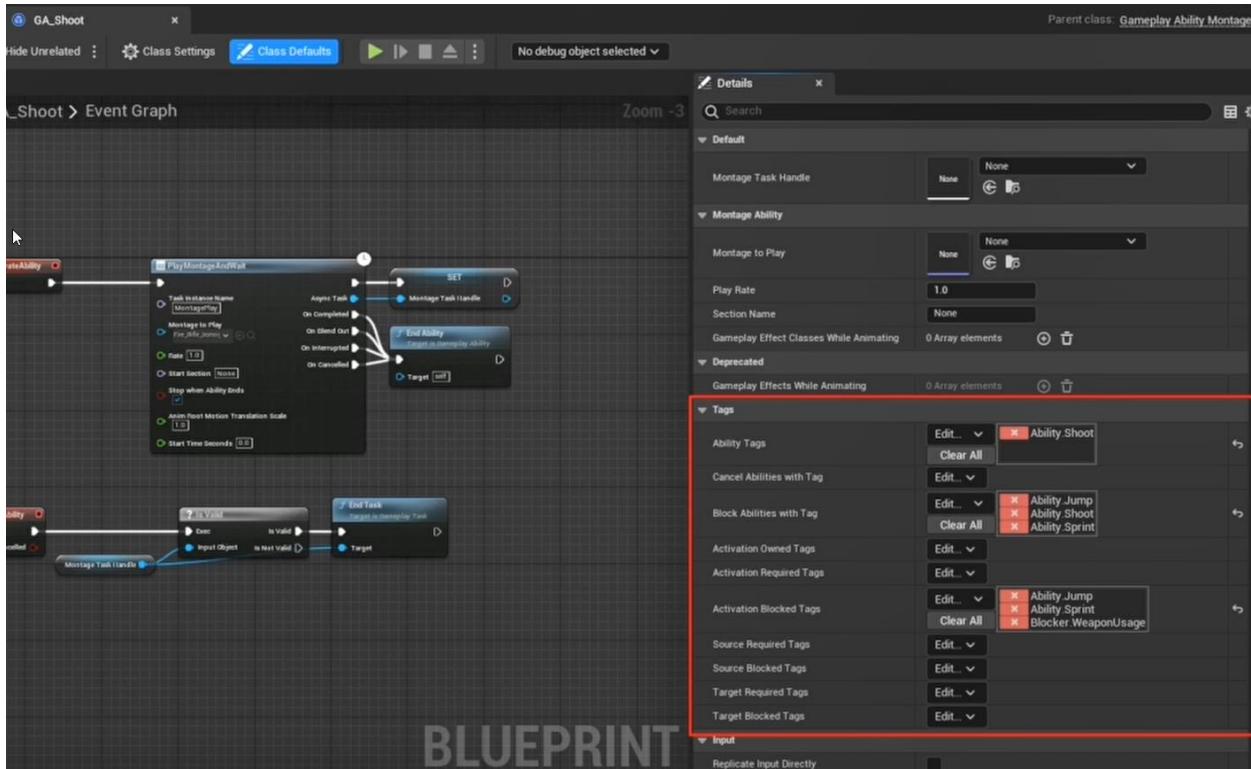
Direct referencing from code has one **disadvantage**. Editor **won't be able to check** if the Tag **is still referenced** if you will **try to delete it**.

A set of Tags can be stored inside of a [GamplayTagContainer](#).

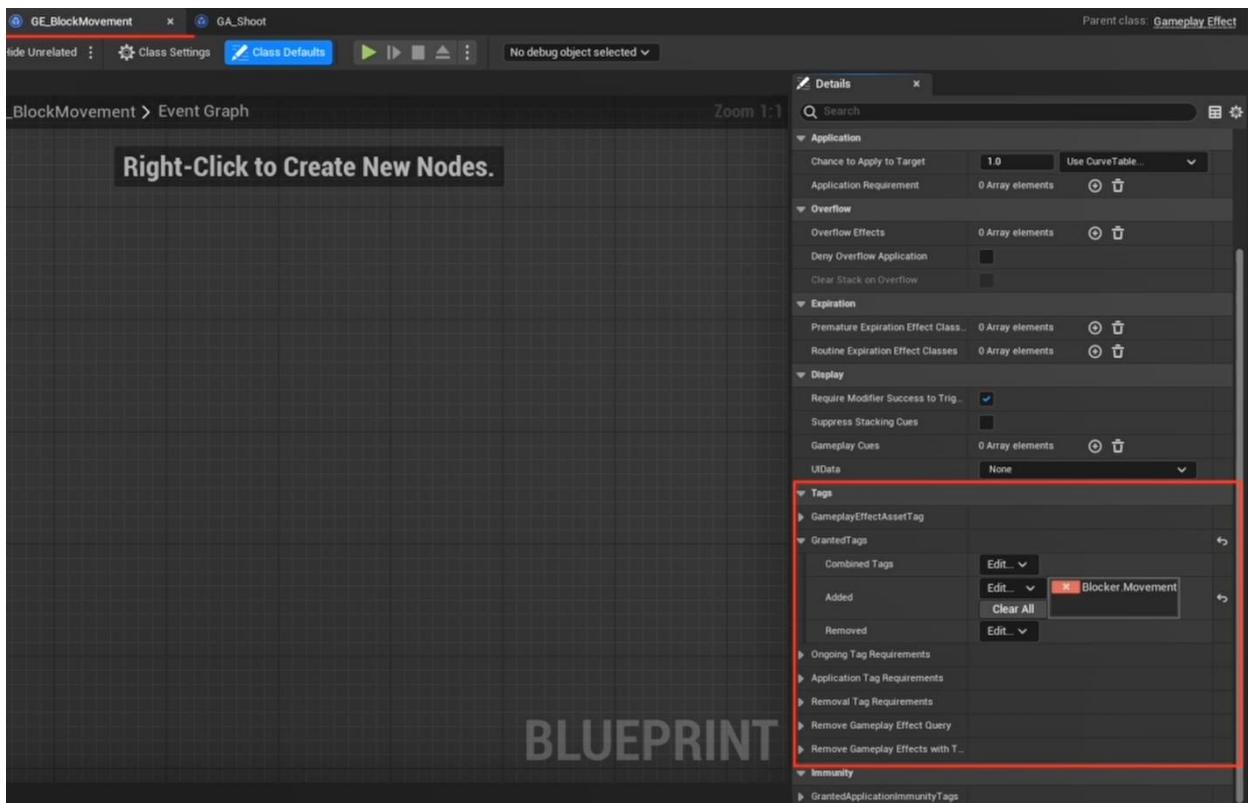
```
UPROPERTY(EditDefaultsOnly)
FGameplayTag MovementBlockingTag;

UPROPERTY(EditDefaultsOnly)
FGameplayTagContainer MovementBlockingTags;
```

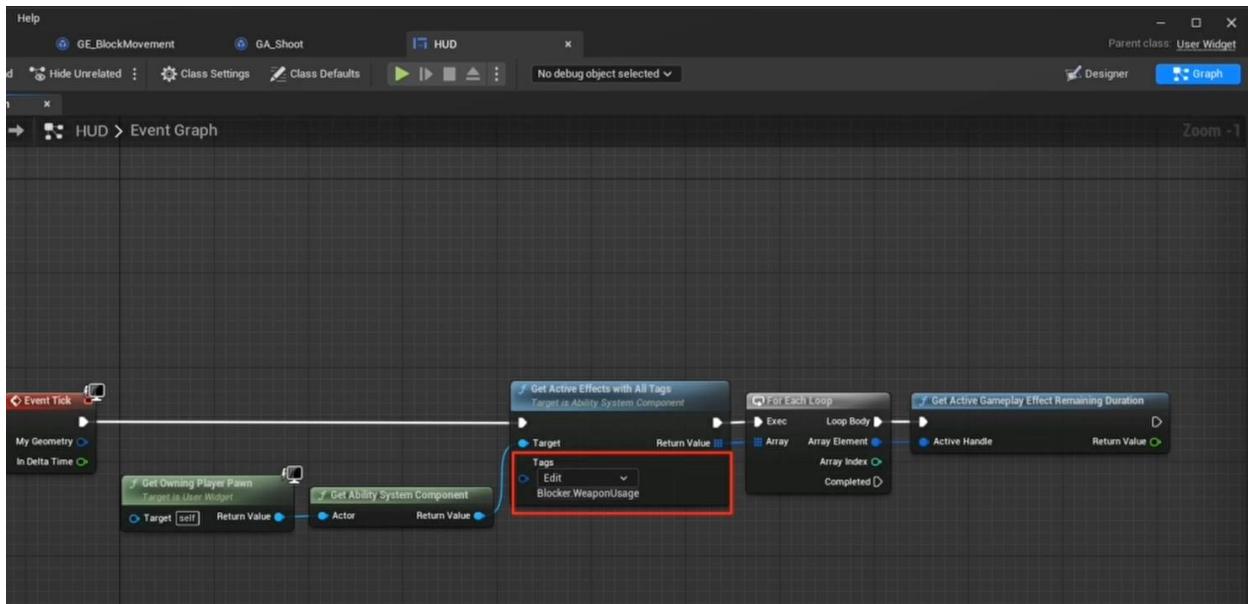
Ability can have tags, that will be **associated with** or tags it will directly apply to the owning ability system component. Or just use rules based on tags to define relations with other abilities.



Gameplay effect can also apply tags to the target. All tags applied to the target in both cases **will be automatically removed when ability ends or the effect is removed**.



Through the AbilitySystemComponent you can **directly check if some actor has some tags or effect with certain tags** and use this information in other systems if you need.



Worth mentioning that actor himself has its own FNameTag Array.

But we'll be not working with this.

```
12     uint8 ActorIsBeingConstructed : 1;
13
14 public:
15     /* Array of tags that can be used for grouping and categorizing. */
16     UPROPERTY(EditAnywhere, BlueprintReadWrite, AdvancedDisplay, Category=Actor)
17     TArray< FName> Tags;
```

One of the biggest advantage of applying tags we can track when some tag is added or removed via delegates.

```
59     AbilitySystemComponent->OnGameplayEffectAppliedEvent.AddToSelf(Event::OnGameplayEffectAppliedToSelf);
60     AbilitySystemComponent->OnGameplayEffectCancelledEvent.AddToSelf(Event::OnGameplayEffectCancelled);
61     AbilitySystemComponent->OnGameplayTagEvent.AddToSelf(Event::OnAbilityAttack);
62     AbilitySystemComponent->RegisterGameplayEvent(FGameplayEvent::RequestGameplayTag(Event::Ability_Reload), EGameplayEventType::OnRemoved).AddToSelf(this, BAUES_ThirdPersonCharacter::OnInteractionTagChanged);
```

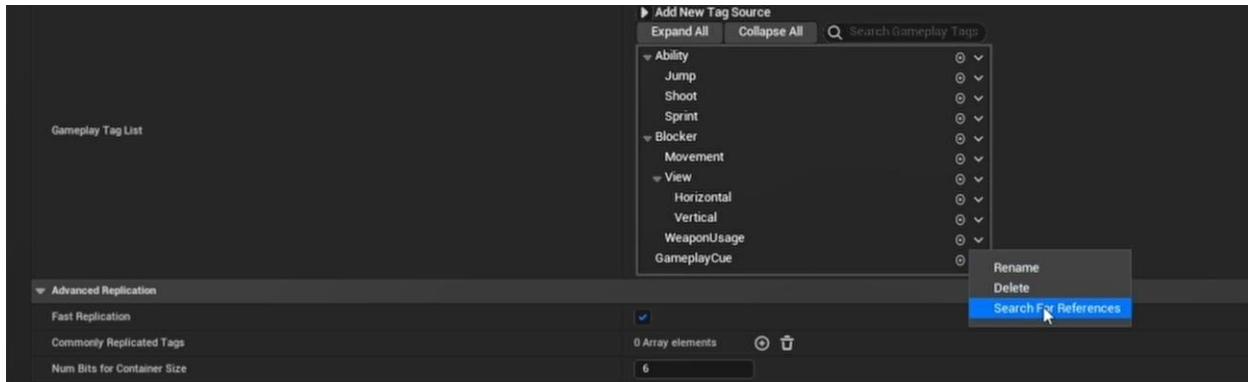
On the high level, tags are managed by [GameplayTagsManager](#).

```
GameplayTagsManager.h  X GameplayTagsManager.cpp  GameplayTagContainer.cpp  GameplayTagContainer.h  AbilitySystemComponent.cpp
→ GameplayTagsManager.h  → D:\UnrealEngine\Engine\Source\Runtime\GameplayTags\Classes\GameplayTagsManager.h
UE5  (Global Scope)

292  };
293
294  /** Holds data about the tag dictionary, is in a singleton UObject */
295  UCCLASS(config=Engine)
296  EClass GAMEPLAYTAGS_API UGameplayTagsManager : public UObject
297  {
298     GENERATED_UCLASS_BODY()
299
300     /** Destructor */
301     ~UGameplayTagsManager();
302
303     /** Returns the global UGameplayTagsManager manager */
304     FORCEINLINE static UGameplayTagsManager& Get()
305     {

```

You can always check where the Tag is referenced from inside the editor. But remember it won't work with the hard coded references in code.



7. Gameplay Attributes

Attribute are specific parameters that exist inside the AttributeSet. To use them, you must first override the base AttributeSet class with your own. And then, add your attributes as gameplay attribute data struck objects.

A screenshot of the Unreal Engine Editor showing the 'BaseCharacterAttributeSet.h' header file. The code defines a class 'UBaseCharacterAttributeSet' that inherits from 'UAttributeSet'. It contains properties for Health, Mana, and Stamina, each with replicated using functions like OnRep_Health, OnRep_MaxHealth, etc. The code also includes attribute accessors for each attribute and their respective regen rates.

Next add your AttributesSet as a [SubObject](#) to your actor that owns AbilitySystemComponent.

```

98 // Create the attribute set, this replicates by default
99 // Adding it as a subobject of the owning actor of an AbilitySystemComponent
100 // automatically registers the AttributeSet with the AbilitySystemComponent
101 BaseCharacterAttributeSet = CreateDefaultSubobject<UBaseCharacterAttributeSet>(TEXT("BaseCharacterAttributeSet"));
102
103 HealthChangedDelegateHandle = AbilitySystemComponent->GetGameplayAttributeValueChangedDelegate(BaseCharacterAttributeSet->GetHealthAttribute()).AddObject(this, &ABaseCharacter::OnHealthChanged);
104 MaxWalkSpeedChangedDelegateHandle = AbilitySystemComponent->GetGameplayAttributeValueChangedDelegate(BaseCharacterAttributeSet->GetMaxMovementSpeedAttribute()).AddObject(this, &ABaseCharacter::OnMaxWalkSpeedChanged);
105
106

```

The great thing about attributes, they can be modified implicitly via the gameplay effects.

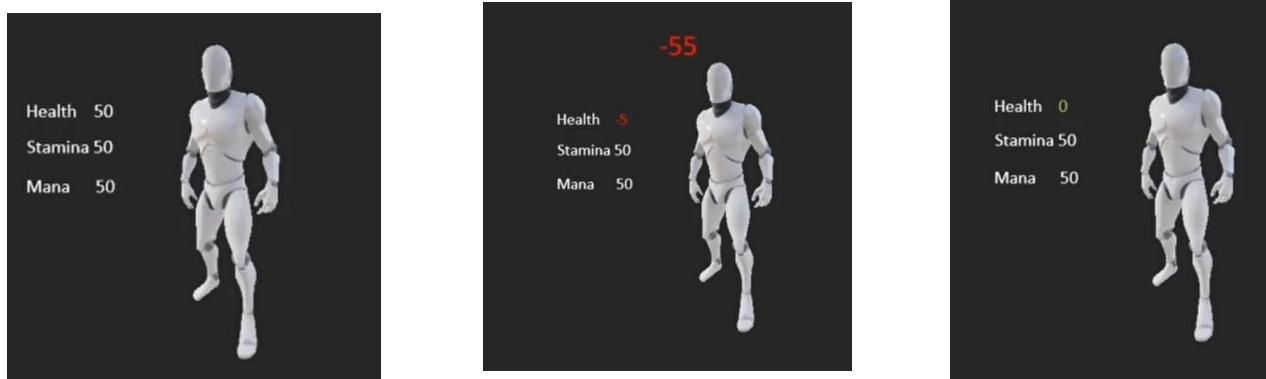
We can also have functions to track the attribute change or apply additional modifications.

```

98 // Create the attribute set, this replicates by default
99 // Adding it as a subobject of the owning actor of an AbilitySystemComponent
100 // automatically registers the AttributeSet with the AbilitySystemComponent
101 BaseCharacterAttributeSet = CreateDefaultSubobject<UBaseCharacterAttributeSet>(TEXT("BaseCharacterAttributeSet"));
102
103 HealthChangedDelegateHandle = AbilitySystemComponent->GetGameplayAttributeValueChangedDelegate(BaseCharacterAttributeSet->GetHealthAttribute()).AddObject(this, &ABaseCharacter::OnHealthChanged);
104 MaxWalkSpeedChangedDelegateHandle = AbilitySystemComponent->GetGameplayAttributeValueChangedDelegate(BaseCharacterAttributeSet->GetMaxMovementSpeedAttribute()).AddObject(this, &ABaseCharacter::OnMaxWalkSpeedChanged);
105
106

```

For example, we can modify the health attribute with the Damage gameplay effect and inside their attribute claimed the value between zero and maximum health.



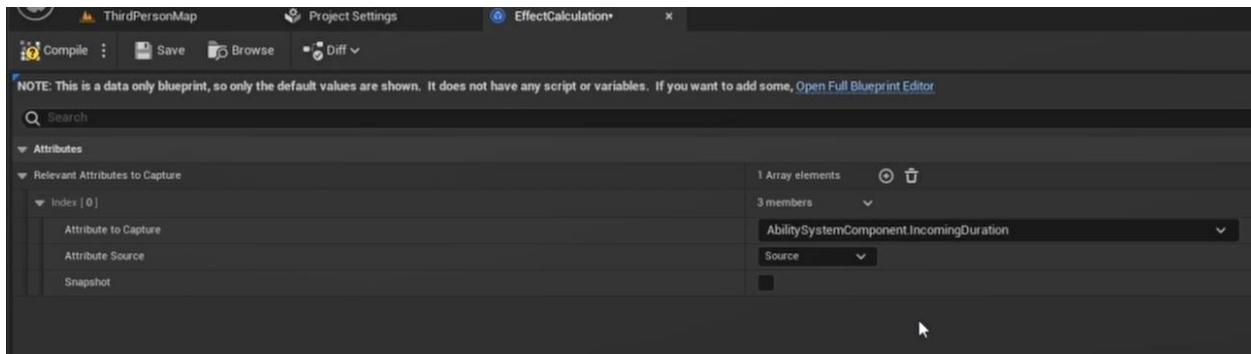
```

void UBaseCharacterAttributeSet::PostGameplayEffectExecute(const FGameplayEffectModCallbackData& Data)
{
    Super::PostGameplayEffectExecute(Data);

    if (Data.EvaluatedData.Attribute == GetHealthAttribute())
    {
        // Handle other health changes.
        // Health loss should go through Damage.
        SetHealth(FMath::Clamp(GetHealth(), 0.0f, GetMaxHealth()));
    } // Health
    else if (Data.EvaluatedData.Attribute == GetManaAttribute())
    {
        // Handle mana changes.
        SetMana(FMath::Clamp(GetMana(), 0.0f, GetMaxMana()));
    } // Mana
    else if (Data.EvaluatedData.Attribute == GetStaminaAttribute())
    {
        // Handle stamina changes.
        SetStamina(FMath::Clamp(GetStamina(), 0.0f, GetMaxStamina()));
    }
    else if (Data.EvaluatedData.Attribute == GetMaxMovementSpeedAttribute())
    {
        if (ACharacter* OwningCharacter = Cast<ACharacter>(GetOwningActor()))
        {
            OwningCharacter->GetCharacterMovement()->MaxWalkSpeed = GetMaxMovementSpeed();
        }
    }
}

```

Attributes can also be accessed from the outside to take part in gameplay effect calculations.

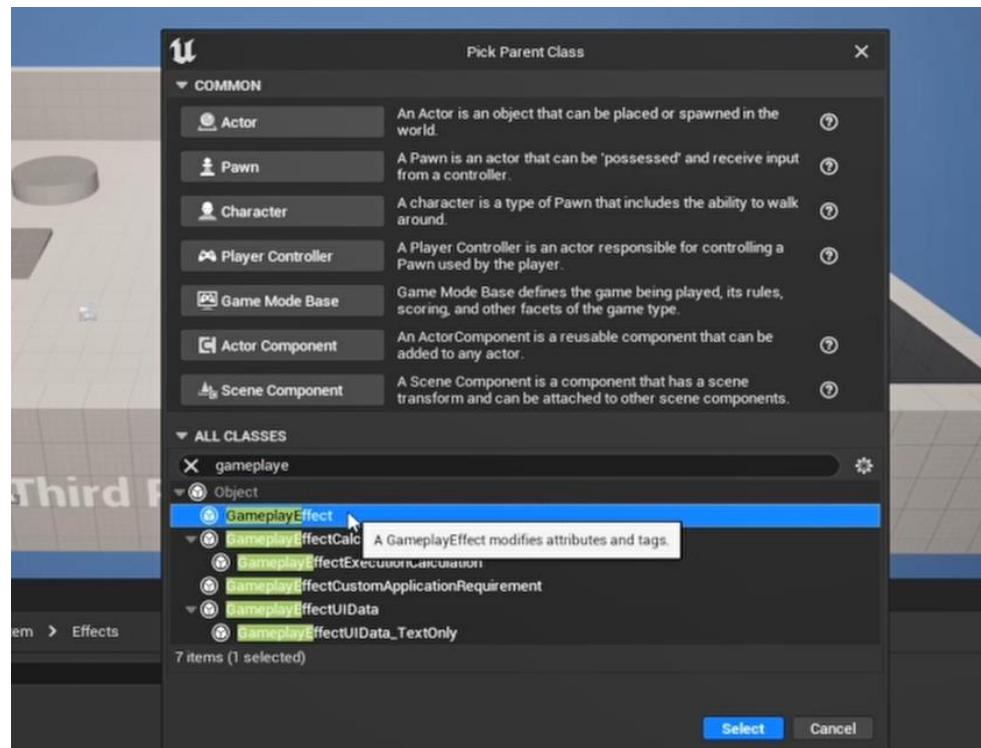


8. Gameplay Effects

GameplayEffects Are UObject mostly used to [change attributes](#), [Apply tags](#) and [gameplay cues](#).

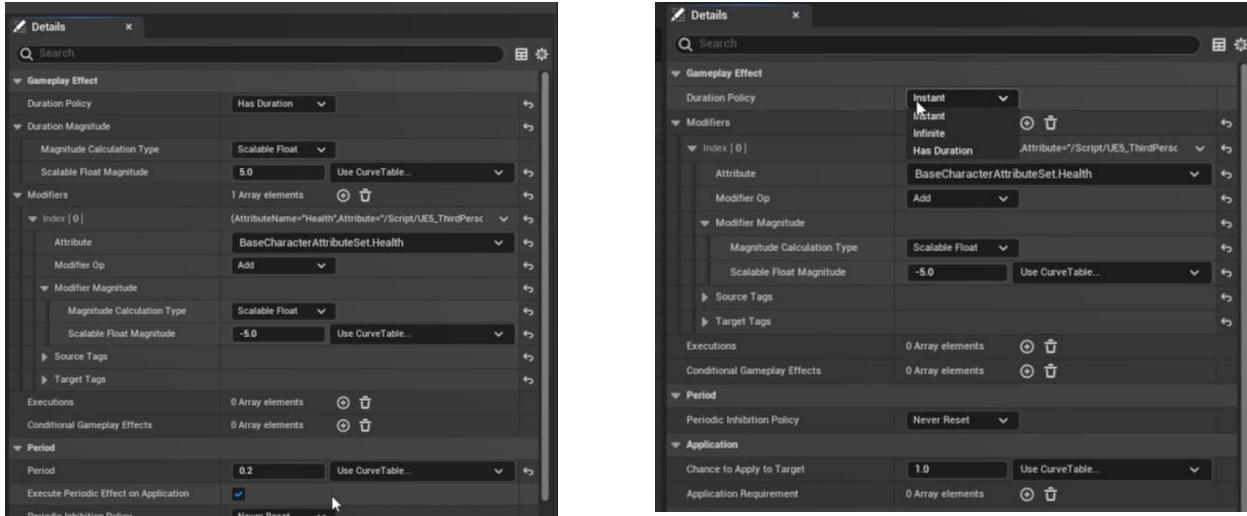
They're **not designed to run logic**, so we should [work with](#) them just with their [exposed properties](#).

Most likely you won't need to use C++ to create your own derived class for them, but it's possible. Usually you just have to create [UGamePlayEffect](#) arrived blueprint class.



GameplayEffects can have [immediate](#) effect, [permanent](#), or have a [duration](#).

If you choose [duration](#), we can also specify at what period we can apply the effect.



The Effect application starts with a [ApplyGameplayEffectSpecToSelf](#) function.

(in AbilitySystemComponent.h)

```
299  /** Applies a previously created gameplay effect spec to a target */
300  #FUNCTION(BlueprintCallable, Category = GameplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToTarget", ScriptName = "ApplyGameplayEffectSpecToTarget"))
301  FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpecHandle SpecHandle, UAbilitySystemComponent* Target);
302
303  virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpec& GameplayEffect, UAbilitySystemComponent *Target, FPredictionKey PredictionKey=FPredictionKey());
304
305  /** Applies a previously created gameplay effect spec to this component */
306  #FUNCTION(BlueprintCallable, Category = GameplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToSelf", ScriptName = "ApplyGameplayEffectSpecToSelf"))
307  FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpecHandle& SpecHandle);
308
309  virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpec& GameplayEffect, FPredictionKey PredictionKey = FPredictionKey());
```

One important thing you should also remember, after applying the effect, we can have its handle. Which can be useful to **try in specific instance of it**. Which might be useful sometimes.

```
299  /** Applies a previously created gameplay effect spec to a target */
300  #FUNCTION(BlueprintCallable, Category = GameplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToTarget", ScriptName = "ApplyGameplayEffectSpecToTarget"))
301  FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpecHandle SpecHandle, UAbilitySystemComponent* Target);
302
303  virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpec& GameplayEffect, UAbilitySystemComponent *Target, FPredictionKey PredictionKey=FPredictionKey());
304
305  /** Applies a previously created gameplay effect spec to this component */
306  #FUNCTION(BlueprintCallable, Category = GameplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToSelf", ScriptName = "ApplyGameplayEffectSpecToSelf"))
307  FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpecHandle& SpecHandle);
308
309  virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpec& GameplayEffect, FPredictionKey PredictionKey = FPredictionKey());
```

Inside of this function we are also checking if we are immune to this effect application (pic 1).

Also Every effect has a chance to be applied. Here we check (pic 2) if the current effect will be applied. 1.0 means 100% chance. (both pics are in the [ApplyGameplayEffectSpecToSelf](#))

Pic 1:

AbilitySystemComponent.h AbilitySystemComponent.cpp → GameplayEffect.h

UAbilitySystemComponent → UAbilitySystemComponent

```
683 FActiveGameplayEffectHandle UAbilitySystemComponent::ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpec &Spec, FPredictionKey PredictionKey)
684 {
685 #if WITH_SERVER_CODE
686     SCOPE_CYCLE_COUNTER(STAT_AbilitySystemComp_ApplyGameplayEffectSpecToSelf);
687 #endif
688
689     // Scope lock the container after the addition has taken place to prevent the new effect from potentially getting mangled during the remainder
690     // of the add operation
691     FScopedActiveGameplayEffectLock ScopeLock(ActiveGameplayEffects);
692
693     FScopeCurrentGameplayEffectBeingApplied ScopedGEApplication(&Spec, this);
694
695     const bool bIsNetAuthority = IsOwnerActorAuthoritative();
696
697     // Check Network Authority
698     if (!HasNetworkAuthorityToApplyGameplayEffect(PredictionKey))
699     {
700         return FActiveGameplayEffectHandle();
701     }
702
703     // Don't allow prediction of periodic effects
704     if (PredictionKey.IsValidKey() && Spec.GetPeriod() > 0.f)
705     {
706         if (IsOwnerActorAuthoritative())
707         {
708             // Server continue with invalid prediction key
709             PredictionKey = FPredictionKey();
710         }
711         else
712         {
713             // Client just return now
714             return FActiveGameplayEffectHandle();
715         }
716     }
717
718     // Are we currently immune to this? (ApplicationImmunity)
719     const FActiveGameplayEffect* ImmunityGE=nullptr;
720     if (ActiveGameplayEffects.HasApplicationImmunityToSpec(Spec, ImmunityGE))
721     {
722         OnImmunityBlockGameplayEffect(Spec, ImmunityGE);
723         return FActiveGameplayEffectHandle();
724     }
725
726 }
```

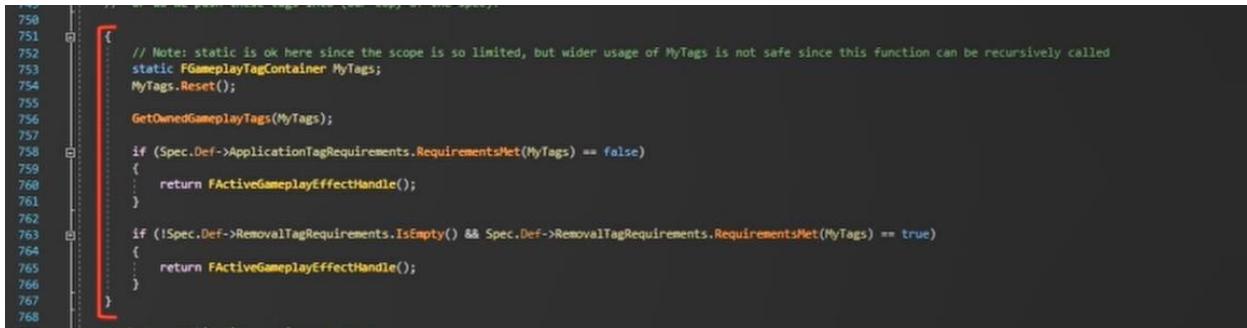
Pic 2:

AbilitySystemComponent.h AbilitySystemComponent.cpp → GameplayEffect.h

UAbilitySystemComponent → FActiveGameplayEffectHandle UAbilitySystemComponent::ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpec &Spec, FPredictionKey PredictionKey)

```
717 }
718
719     // Are we currently immune to this? (ApplicationImmunity)
720     const FActiveGameplayEffect* ImmunityGE=nullptr;
721     if (ActiveGameplayEffects.HasApplicationImmunityToSpec(Spec, ImmunityGE))
722     {
723         OnImmunityBlockGameplayEffect(Spec, ImmunityGE);
724         return FActiveGameplayEffectHandle();
725     }
726
727     // Check AttributeSet requirements: make sure all attributes are valid
728     // We may want to cache this off in some way to make the runtime check quicker.
729     // We also need to handle things in the execution list
730     for (const FGameplayModifierInfo& Mod : Spec.Def->Modifiers)
731     {
732         if (!Mod.Attribute.IsValid())
733         {
734             ABILITY_LOG(Warning, TEXT("%s has a null modifier attribute."), *Spec.Def->GetPathName());
735             return FActiveGameplayEffectHandle();
736         }
737     }
738
739     // check if the effect being applied actually succeeds
740     float ChanceToApply = Spec.GetChanceToApplyToTarget();
741     if ((ChanceToApply < 1.f - SMALL_NUMBER) && (FMath::FRand() > ChanceToApply))
742     {
743         return FActiveGameplayEffectHandle();
744     }
745 }
```

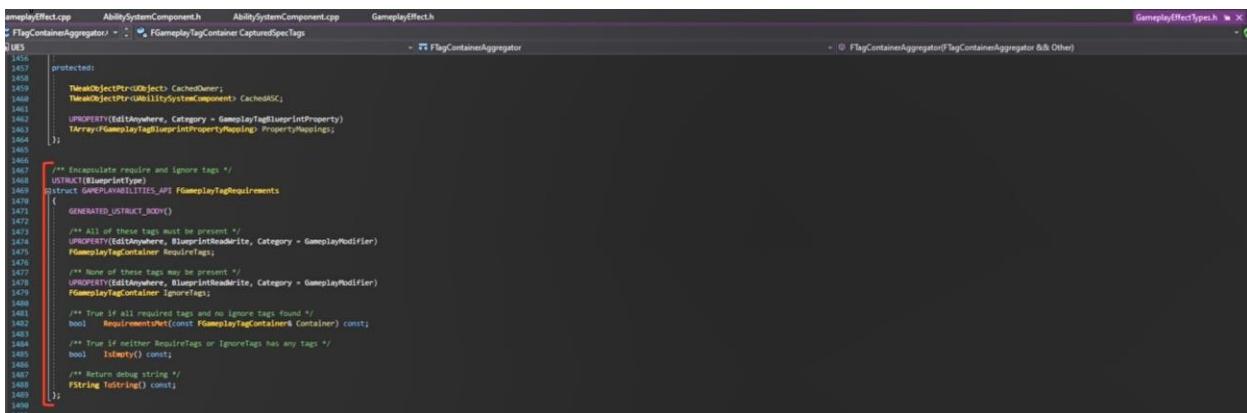
Next we have to check the application Tag Requirements.



```
750
751     {
752         // Note: static is ok here since the scope is so limited, but wider usage of MyTags is not safe since this function can be recursively called
753         static FGameplayTagContainer MyTags;
754         MyTags.Reset();
755
756         GetOwnedGameplayTags(MyTags);
757
758         if (Spec.Def->ApplicationTagRequirements.RequirementsMet(MyTags) == false)
759         {
760             return FActiveGameplayEffectHandle();
761         }
762
763         if (!Spec.Def->RemovalTagRequirements.IsEmpty() && Spec.Def->RemovalTagRequirements.RequirementsMet(MyTags) == true)
764         {
765             return FActiveGameplayEffectHandle();
766         }
767     }
768 }
```

Base rule is **RequiredTags** must be **presented**, **IgnoreTags** should **not be presented**.

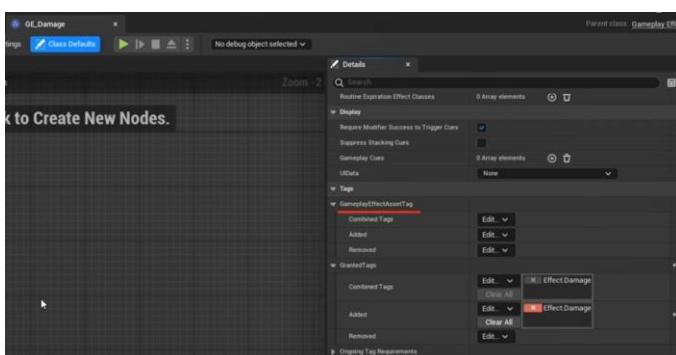
If at least one tag from ignored tack is presented, **we will not apply the effect**.



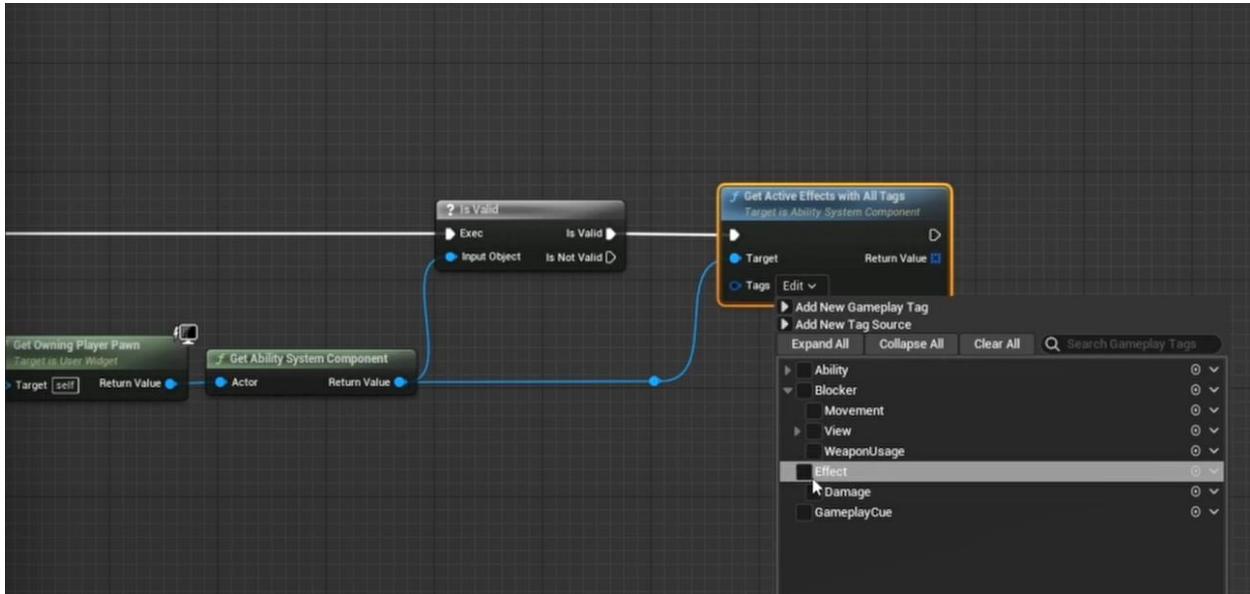
```
1456
1457     protected:
1458
1459         TWeakObjectPtr< UObject> CachedOwner;
1460         TWeakObjectPtr< UAbilitySystemComponent> CachedASC;
1461
1462         UPROPERTY(EditAnywhere, Category = GameplayTagBlueprintProperty)
1463         TArray< FGameplayTagBlueprintPropertyParams> PropertyMappings;
1464
1465
1466
1467     /* Encapsulate require and ignore tags */
1468     USTRUCT(BlueprintType)
1469     struct GAMEPLAY_ABILITIES_API FGameplayTagRequirements
1470     {
1471         GENERATED_USTRUCT_BODY()
1472
1473         /* All of these tags must be present */
1474         UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayModifier)
1475         FGameplayTagContainer Requirements;
1476
1477         /* None of these tags may be present */
1478         UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayModifier)
1479         FGameplayTagContainer IgnoreTags;
1480
1481         /* True if all required tags and no ignore tags found */
1482         bool RequirementsMet() const;
1483
1484         /* True if neither Requirements nor IgnoreTags has any tags */
1485         bool IsEmpty() const;
1486
1487         /* Return debug string */
1488         FString ToString() const;
1489     };
1490
1491 }
```

GameplayEffects are connected to Tags using which we can access or modify certain effects. Was no need to directly know their class. For example, if all of our debuff effects have the tag debuff. We can remove all those effects by removing all gameplay effects with the tag debuff.

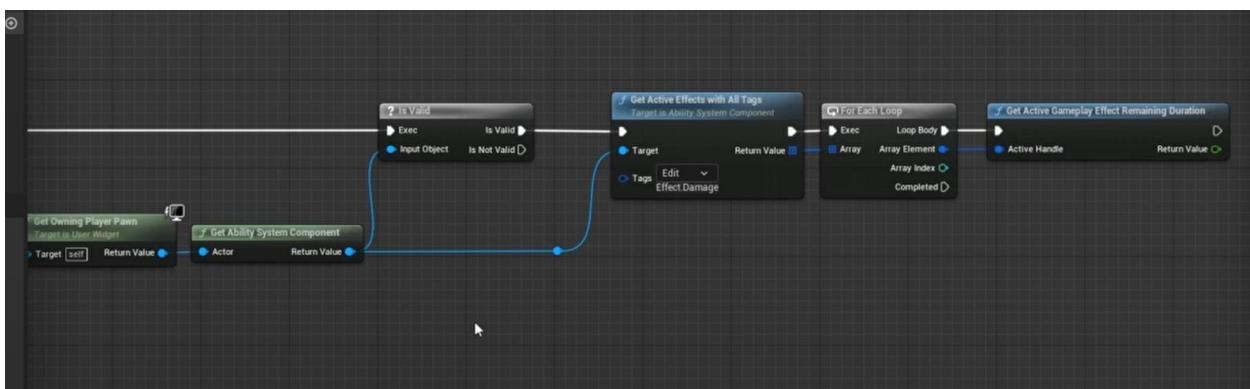
Tags over here will be associated with the Effect itself.



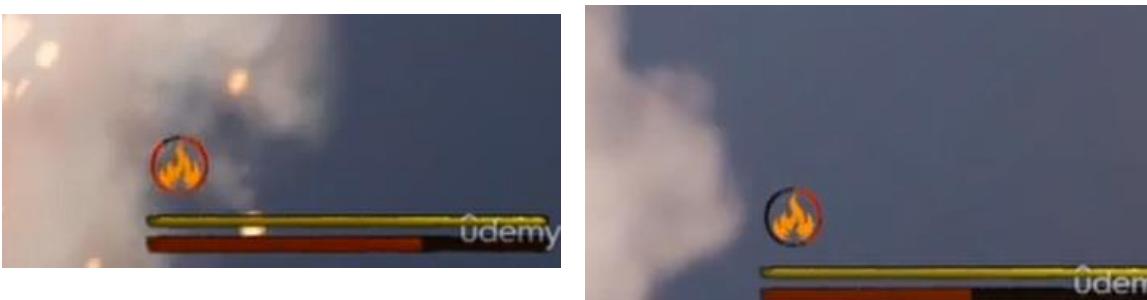
Using them we can explicitly track what effects are applied to our character.



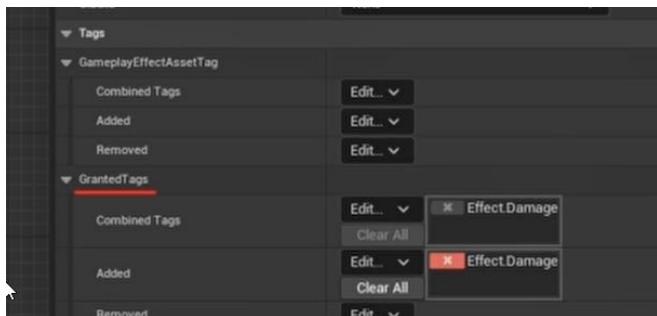
And even get some runtime parameters from them like the remaining time.



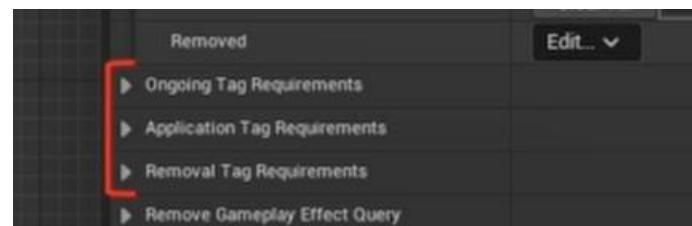
We can use this for example to show the remaining time of some Effects like burning effect or poison.



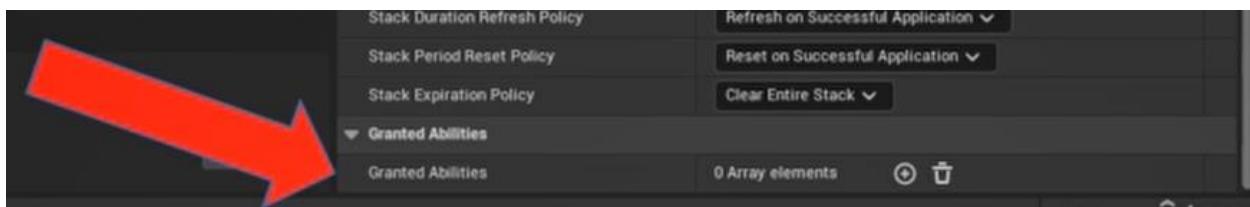
GrantedTags will be applied to the owner of the effect.



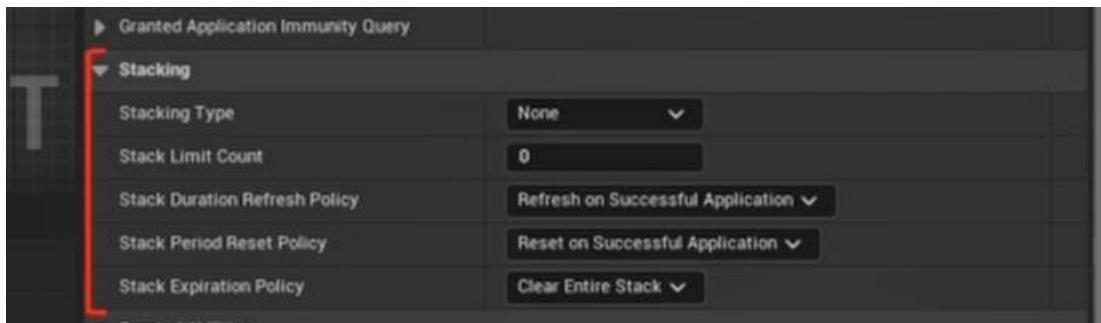
Next is common set of requirements.



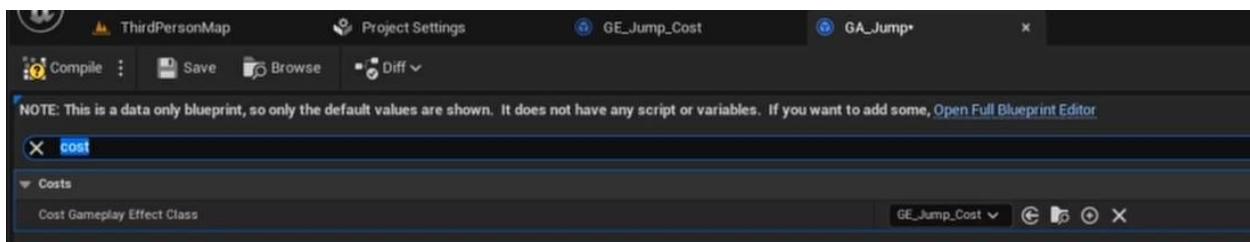
One interesting ability of an effect is to [grant abilities to its target](#). This can be used, for example, by a magic potion to give an ability to fly.



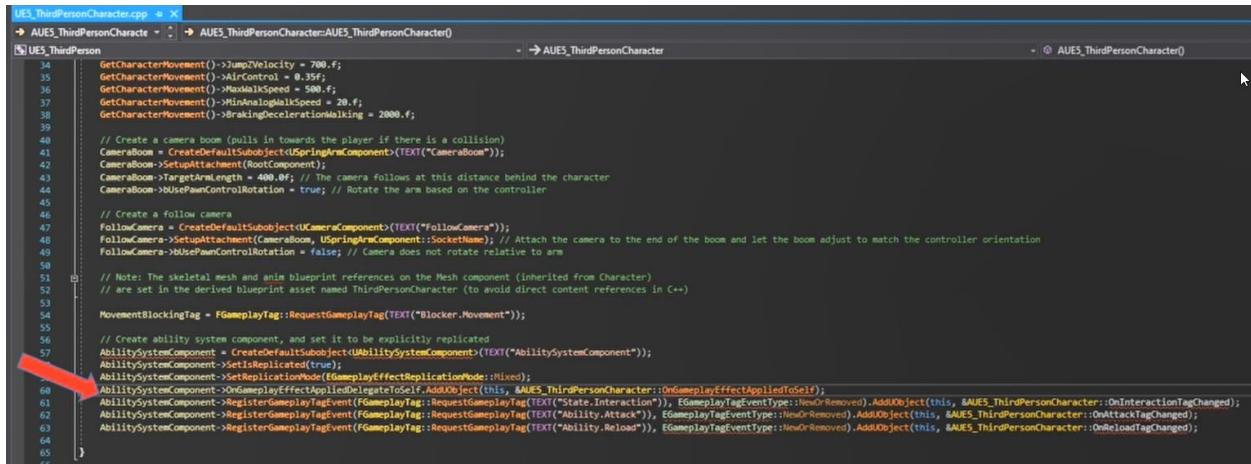
As we can apply one effect several times, we also have some rules about staking.



Because the Effects can represent attribute change, they can be used as a [cost](#) for running and ability. For example, you can have a cost effect that will reduce Manna for jumping.



The great thing about effects, we can **track their Replication** and trigger some logic if it's necessary.

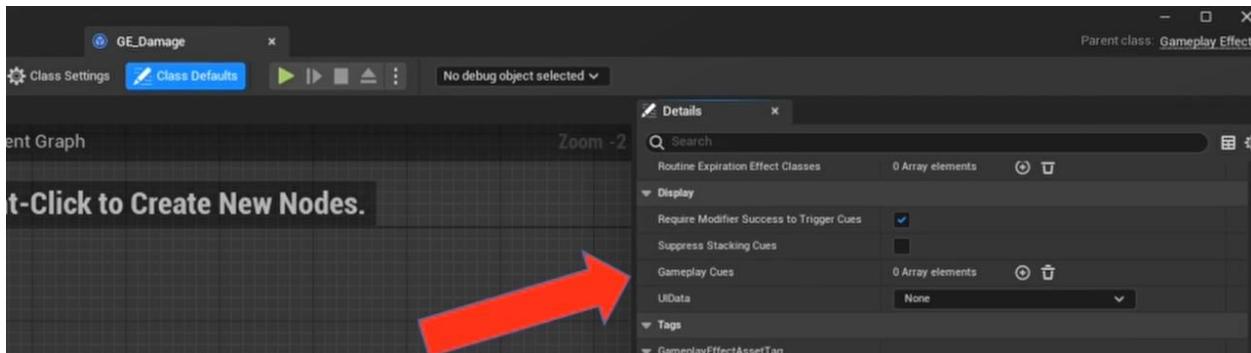


```

UES_ThirdPersonCharacter.cpp
AUE5_ThirdPersonCharacter->AbilitySystemComponent = CreateDefaultSubobject(UAbilitySystemComponent)(TEXT("AbilitySystemComponent"));
AbilitySystemComponent->SetIsReplicated(true);
AbilitySystemComponent->OnGameplayEffectAppliedDelegateToSelf.AddObject(this, &AUE5_ThirdPersonCharacter::OnInteractionTagChanged);
AbilitySystemComponent->RegisterGameplayEvent(FGameplayTag::RequestGameplayTag(TEXT("State.Interaction")), EGameplayEventType::NewOrRemoved).AddObject(this, &AUE5_ThirdPersonCharacter::OnInteractionTagChanged);
AbilitySystemComponent->RegisterGameplayEvent(FGameplayTag::RequestGameplayTag(TEXT("Ability.Attack")), EGameplayEventType::NewOrRemoved).AddObject(this, &AUE5_ThirdPersonCharacter::OnAttackTagChanged);
AbilitySystemComponent->RegisterGameplayEvent(FGameplayTag::RequestGameplayTag(TEXT("Ability.Reload")), EGameplayEventType::NewOrRemoved).AddObject(this, &AUE5_ThirdPersonCharacter::OnReloadTagChanged);

```

Effects can be used also to run **GameplayCues**.



9. Gameplay Cues

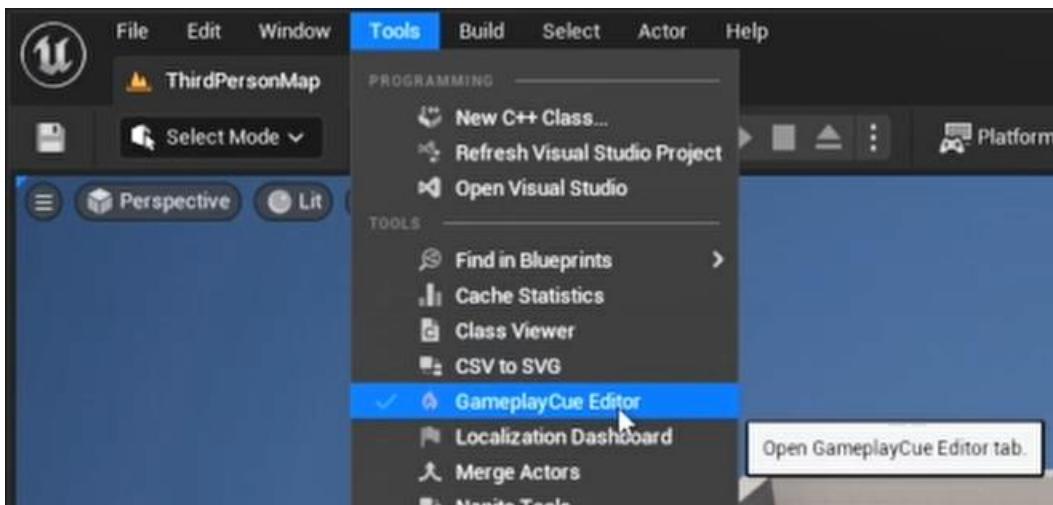
GameplayCues can be used, for example, to display any graphical asset on your character during the gameplay effect.

GameplayCues Execution will be replicated out-of-the-box for us in this case.

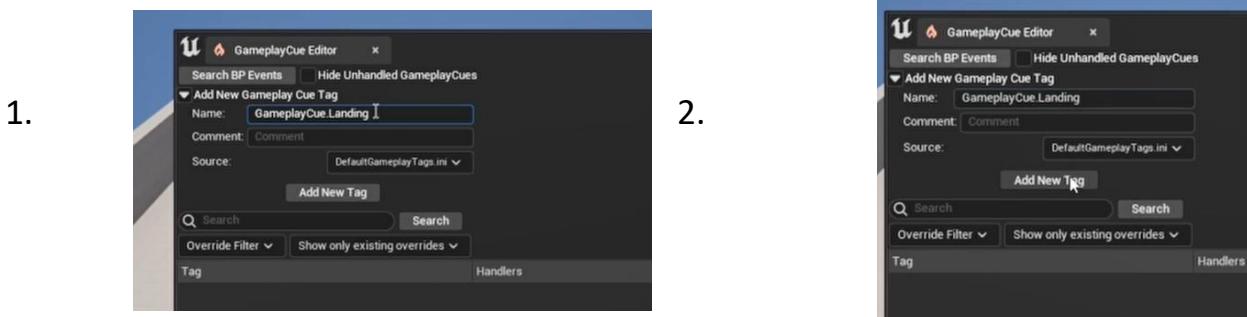
GameplayCues can be a **static** or **actor base**.

- **Static Cues** will have **only one object in memory** and will **not keep or modify a per object data** during the execution because it will **operate on class default object**.
- An **Actor based** will be spawned just as an actor and **will inherit all its capabilities**.

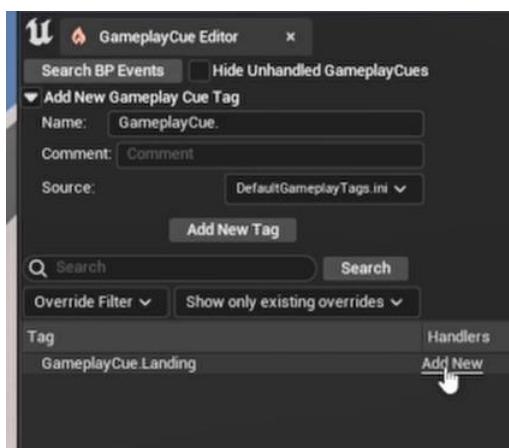
GameplayCues have their own section inside Tag Editor. And we should use this specific window to add them.



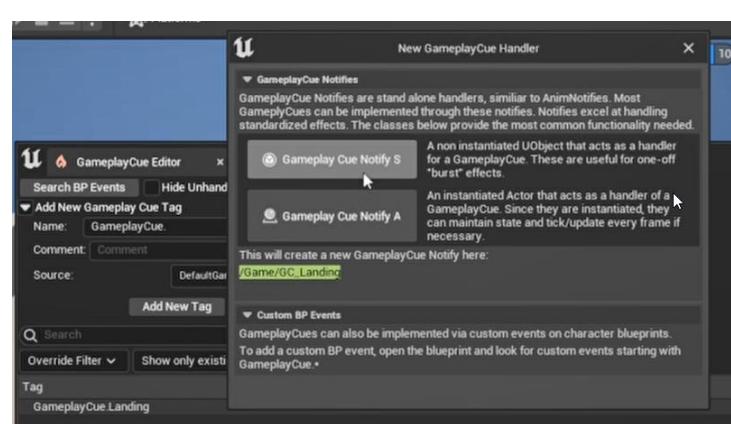
Spawn VFX for character superhero landing and can be made with a Static Cue because we just need to spawn the particle system once and we don't care about tracking its instance afterwards.



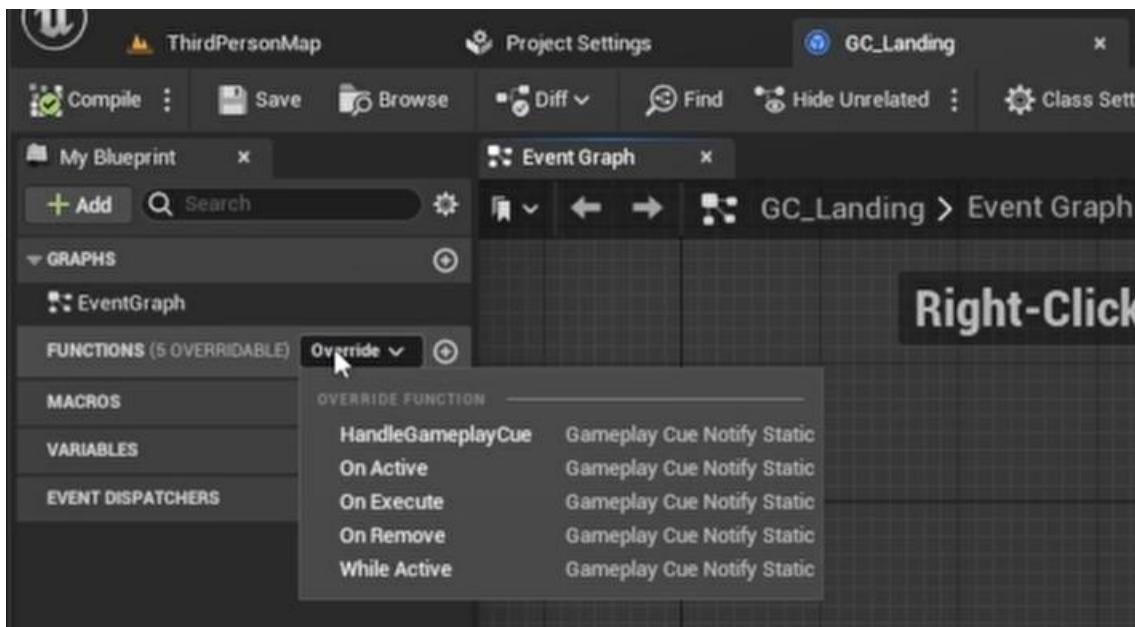
3.



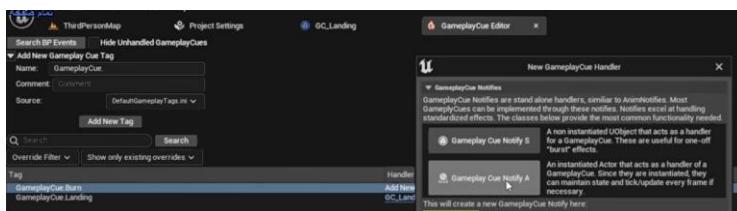
4.



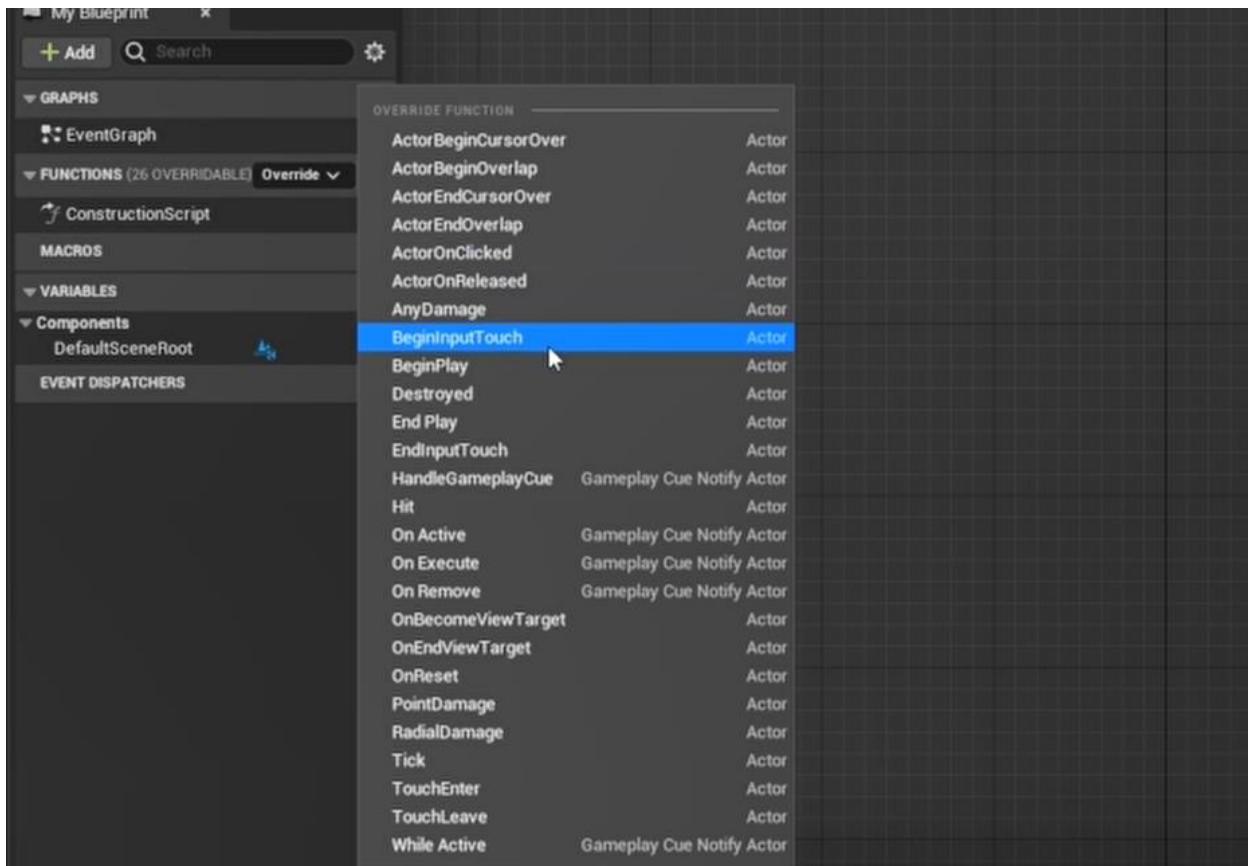
Here we can override some functions that will be triggered during the Cue execution.



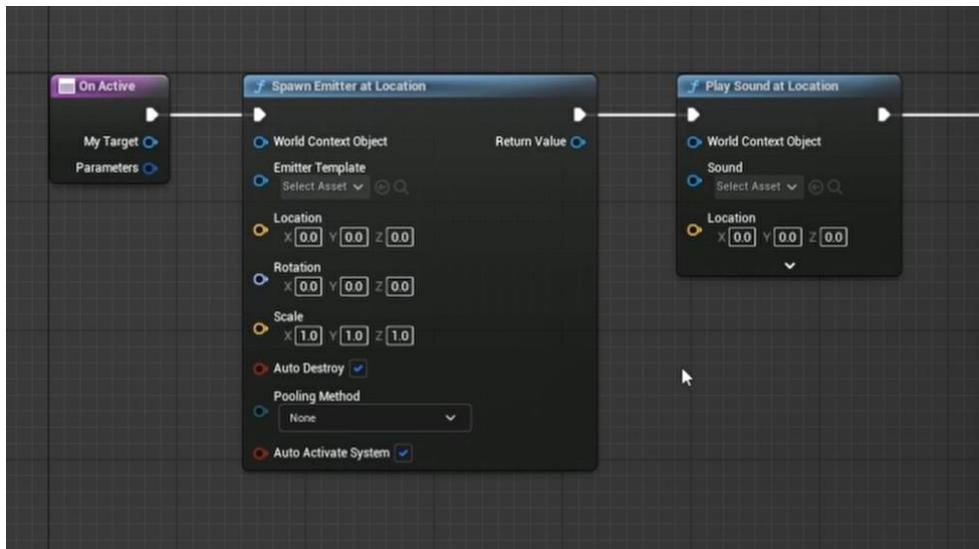
Applying a Burning effect and removing it directly on end by the reference to the particle system can be done via a non-static actor based GamplayCue. because you might want to **save the reference to the particle system** for every character that burns.



GameplayCue's **allows you to write any kind of logic you want** literally, even if you need to do something with the user interface during the effect or modify some customs scene component with character visuals.

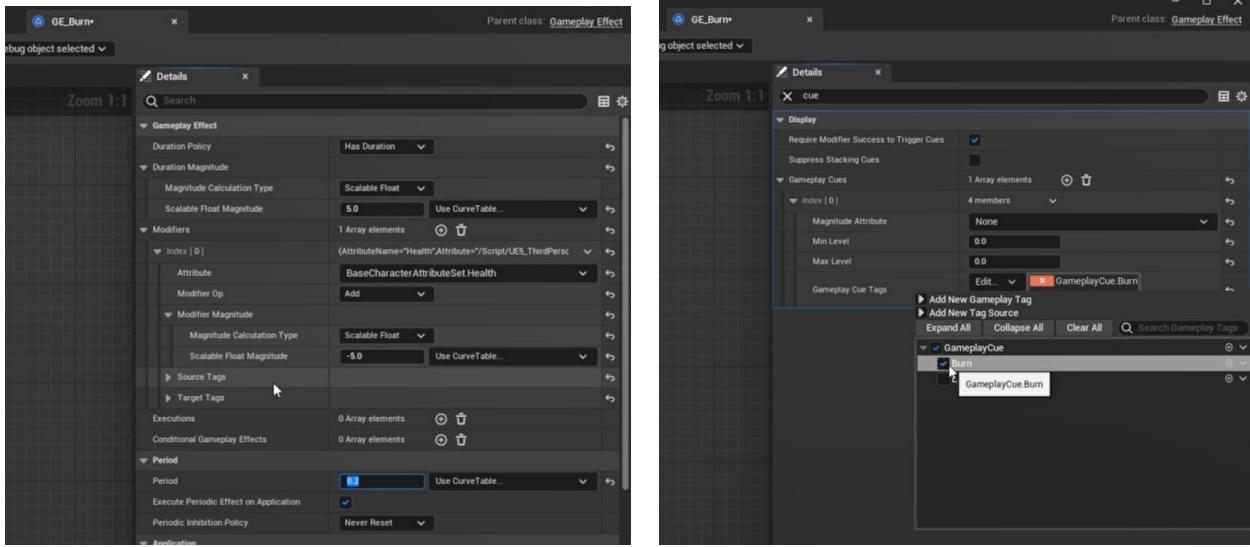


As you can see right here we're trying to spawn some particle system and play sound on the Cue will on the start executing.



Cues can be triggered implicitly by a [GameplayEffect](#). In this case, the events on the Cue will be triggered accordingly to the effect lifetime.

For example, in case of a burning effect, you might want to have an effect with a duration that will reduce health attribute with some low period.



To display Cue for play a sound and the VFX, you will have to add a `GameplayCue` record was the correct tag.

The GameplayEffect will play this Cue when being executed or we can trigger Cues directly.

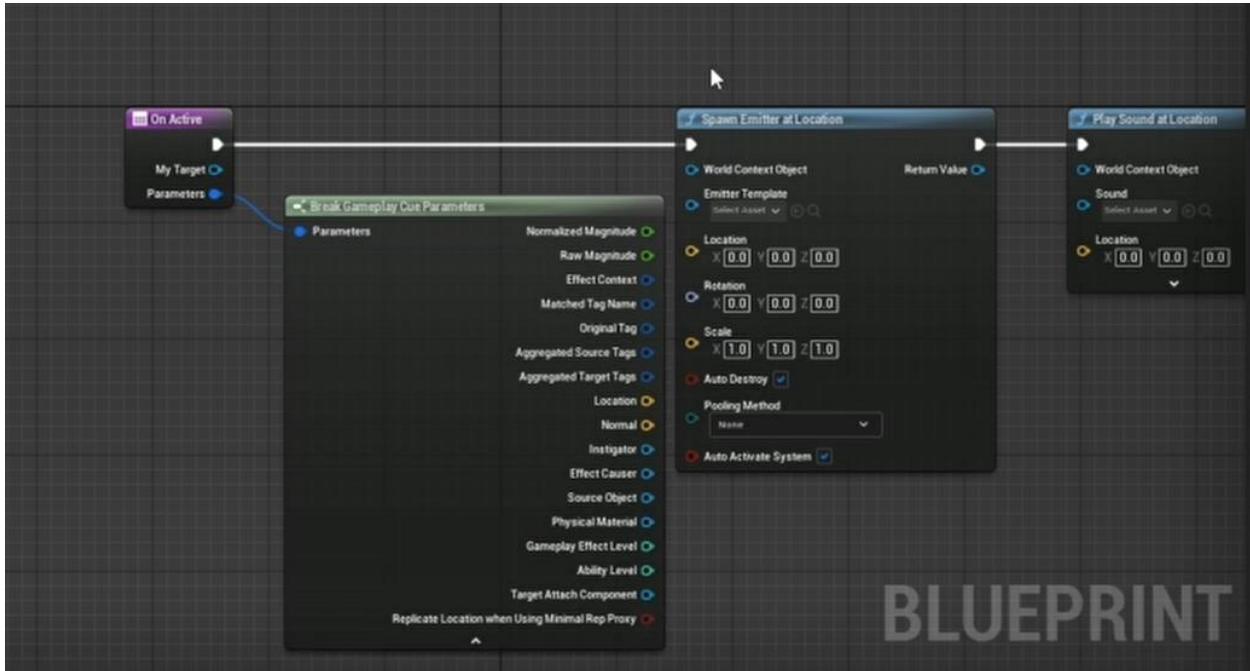
```
2675 // 
2676 // Invoke GameplayCue events...
2677 // 
2678 // If there are no modifiers or we don't require modifier success to trigger, we apply the GameplayCue.
2679 const bool bHasModifiers = SpecType.Def.Modifiers.Num() > 0;
2700 const bool bHasExecutionSpecs = SpecType.Def.Execution.Num() > 0;
2701 const bool bHasModifiersOrExecutionSpecs = bHasModifiers || bHasExecutionSpecs;
2702 // 
2703 // If there are no modifiers or we don't require modifier success to trigger, we apply the GameplayCue.
2704 bool InvokeGameplayCueExecute = !bHasModifiersOrExecutionSpecs || !SpecType.Def.bRequiresModifierSuccessToTrigger;
2705 // 
2706 if (bHasModifiersOrExecutionSpecs && ModifierSuccessfullyExecuted)
2707 {
2708     InvokeGameplayCueExecute = true;
2709 }
2710 // Don't trigger gameplay cues if one of the executions says it manually handled them
2711 if (GameplayCueIsManuallyHandled)
2712 {
2713     InvokeGameplayCueExecute = false;
2714 }
2715 // 
2716 if (InvokeGameplayCueExecute && SpecType.Def.bGameplayCues.bAllow)
2717 {
2718     // TODO: check replication policy. Right now we will replicate every execute via a multicast RPC
2719     ABILITY_LOGGING(TEXT("Invoking Execute GameplayCue for %s"), *SpecType.Def.DisplayName);
2720     (AbilitySystemGlobals::Get().GetGameplayCueManager())->InvokeGameplayCueExecuted(Frontrunner::Owner, *TargetSpec);
2721 }
2722 // Apply any conditional lined effects
2723 for (const FGameplayEffectSpecHandle& TargetSpec : ConditionalEffectSpecs)
2724 {
2725     if (TargetSpec.IsValid())
2726     {
2727         Owner->ApplyGameplayEffectSpecToSelf(*TargetSpec.Data.Get(), PredictionKey);
2728     }
2729 }
2730 }
2731 }
```

We can even trigger them from the **AnimNotify** if we need to.

```
AnimNotify_GameplayCue.h  o  x  AbilitySystemComponent.h          GameplayEffect.cpp          AbilitySystemComponent_Abilities.cpp          AbilitySystemComponent.cpp          GameplayCueNotify_Actor.h          GameplayCueNotify_Static.h
→ GameplayCueInterface.h  +  h  D:\UE_5.0\Engine\Plugins\Runtime\GameplayAbilities\Source\GameplayAbilities\Public\GameplayCueInterface.h
UE
18  UCLASS(editinlinenew, Const, hideCategories = Object, collapseCategories, MinimalAPI, Meta = { DisplayName = "GameplayCue (Burst)" })
19  class UAnimNotify_GameplayCue : public UAnimNotify
20  {
21      GENERATED_BODY()
22
23  public:
24
25      UAnimNotify_GameplayCue();
26
27      UE_DEPRECATED(5.0, "Please use the other Notify Function instead")
28      virtual void Notify(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation) override;
29
30      virtual void Notify(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, const FAnimNotifyEventReference& EventReference) override;
31
32      FString GetNotifyName_Implementation() const override;
33
34 #if WITH_EDITOR
35      virtual bool CanBePlaced(UAnimSequenceBase* Animation) const override;
36 #endif // #if WITH_EDITOR
37
38 protected:
39
40     // GameplayCue tag to invoke.
41     UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = GameplayCue, meta = { Categories = "GameplayCue" })
42     FGameplayCueTag GameplayCue;
43 }
44
45
46 /**
47 * UAnimNotify_GameplayCueState
48 * - An animation notify state used for duration based gameplay cues (Looping).
49 */
50 UCLASS(editinlinenew, Const, hideCategories = Object, collapseCategories, MinimalAPI, Meta = { DisplayName = "GameplayCue (Looping)" })
51 class UAnimNotify_GameplayCueState : public UAnimNotifyState
52 {
53     GENERATED_BODY()
54
55  public:
56
57      UAnimNotify_GameplayCueState();
58      UE_DEPRECATED(5.0, "Please use the other NotifyBegin Function instead")
59      virtual void Begin(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float TotalDuration) override;
60
61      UE_DEPRECATED(5.0, "Please use the other NotifyTick Function instead")
62      virtual void NotifyTick(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float FrameDeltaTime) override;
63
64      UE_DEPRECATED(5.0, "Please use the other NotifyEnd Function instead")
65      virtual void End(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation) override;
66
67      virtual void NotifyBegin(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float TotalDuration, const FAnimNotifyEventReference& EventReference) override;
68      virtual void NotifyTick(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float FrameDeltaTime, const FAnimNotifyEventReference& EventReference) override;
69      virtual void NotifyEnd(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, const FAnimNotifyEventReference& EventReference) override;
70
71      FString GetNotifyName_Implementation() const override;
72
73 #if WITH_EDITOR
74      virtual bool CanBePlaced(UAnimSequenceBase* Animation) const override;
75 #endif // #if WITH_EDITOR
76
77 protected:
```

```
AnimNotify_GameplayCue.cpp  o  x  AnimNotify_GameplayCue.h          AbilitySystemComponent.h          GameplayEffect.cpp          AbilitySystemComponent_Abilities.cpp          AbilitySystemComponent.cpp          GameplayCueNotify_Actor.h          GameplayCueNotify_Static.h
→ AnimNotify_GameplayCue  +  h  D:\UE_5.0\Engine\Plugins\Runtime\GameplayAbilities\Source\GameplayAbilities\Private\AnimNotify_GameplayCue.cpp
UE
76  {
77
78  void UAnimNotify_GameplayCue::Notify(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, const FAnimNotifyEventReference& EventReference)
79  {
80      Super::Notify(MeshComp, Animation, EventReference);
81
82      ProcessGameplayCue(AUGameplayCueManager::ExecuteGameplayCue_NonReplicated, MeshComp, GameplayCue.GameplayCueTag, Animation);
83  }
84
85
86  FString UAnimNotify_GameplayCue::GetNotifyName_Implementation()
87  {
88      FString DisplayName = TEXT("GameplayCue");
89
90      if (GameplayCue.GameplayCueTag.IsValid())
91      {
92          DisplayName = GameplayCue.GameplayCueTag.ToString();
93          DisplayName += TEXT("( Burst )");
94      }
95
96      return DisplayName;
97  }
98
99 #if WITH_EDITOR
100     bool UAnimNotify_GameplayCue::CanBePlaced(UAnimSequenceBase* Animation) const
101     {
102         return (Animation && Animation->IsA(UAnimMontage::StaticClass()));
103     }
104 #endif // WITH_EDITOR
105
106
107 /**
108  * UAnimNotify_GameplayCueState
109  */
110 UAnimNotify_GameplayCueState::UAnimNotify_GameplayCueState()
111 {
112 }
113
114 void UAnimNotify_GameplayCueState::NotifyBegin(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float TotalDuration)
115 {
116 }
117
118 void UAnimNotify_GameplayCueState::NotifyBegin(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float TotalDuration, const FAnimNotifyEventReference& EventReference)
119 {
120     Super::NotifyBegin(MeshComp, Animation, TotalDuration, EventReference);
121
122     ProcessGameplayCue(AUGameplayCueManager::AddGameplayCue_NonReplicated, MeshComp, GameplayCue.GameplayCueTag, Animation);
123
124 #if WITH_EDITORONLY_DATA
125     // Grab proxy tick delegate if someone registered it.
126     if (UGameplayCueManager::PreviewProxyTick.IsBound())
127     {
128         PreviewProxyTick = UGameplayCueManager::PreviewProxyTick;
129         UGameplayCueManager::PreviewProxyTick.Unbind();
130     }
131 #endif // #if WITH_EDITORONLY_DATA
132
133
134 void UAnimNotify_GameplayCueState::NotifyTick(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float FrameDeltaTime)
135 {
136 }
```

From inside the Cue, we can get access to some useful data through the parameters passed to the function.



6. Gameplay Events

[GameplayEvents](#) is the way you can communicate with AbilitySystem, technically [GameplayEvent](#) is just a **combination**

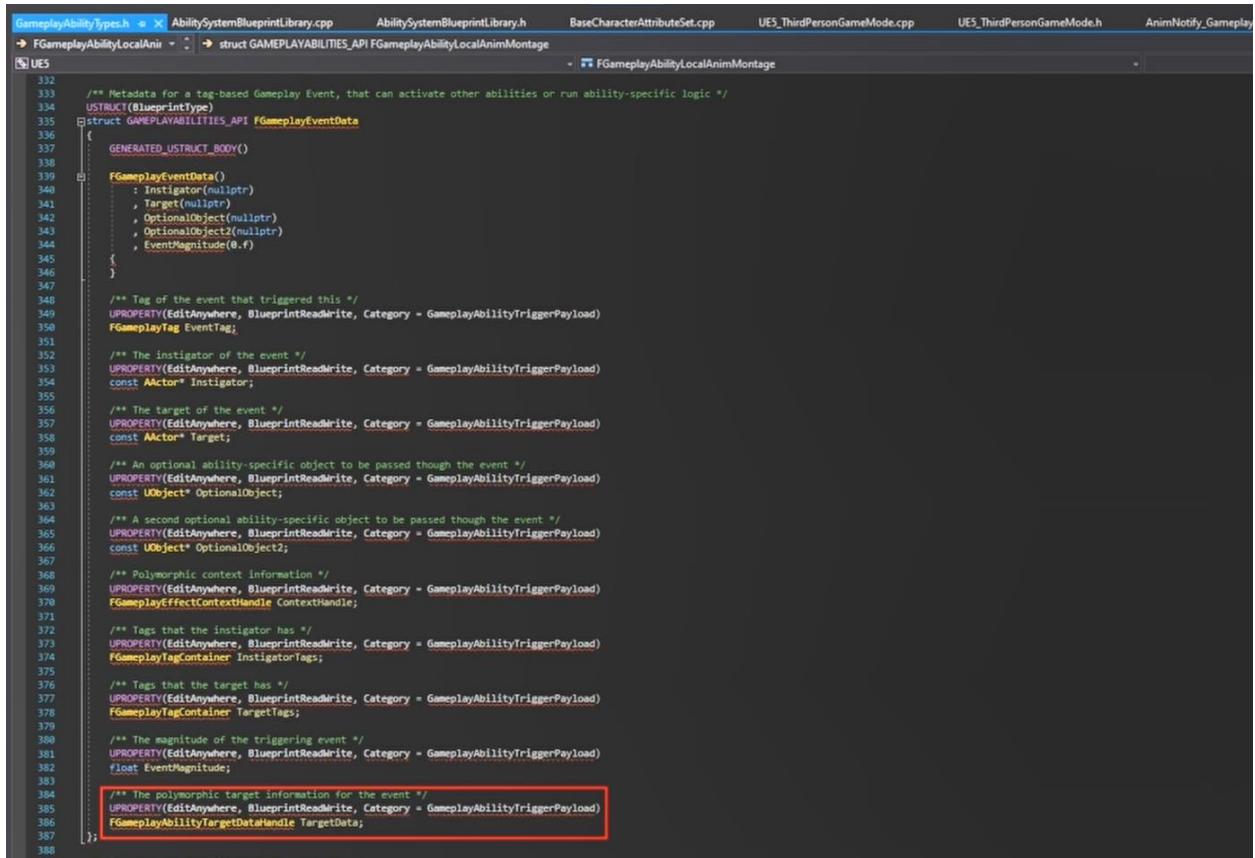
of a [Tag](#) and the [GameplayEventData](#) payload.

A screenshot of the Unreal Engine code editor. The file 'AbilitySystemBlueprintLibrary.h' is open, showing the implementation of the `SendGameplayEventToActor` function. The code is as follows:

```
32 /**
33 * This function can be used to trigger an ability on the actor in question with useful payload data.
34 * NOTE: GetAbilitySystemComponent is called on the actor to find a good component, and if the component isn't
35 * found, the event will not be sent.
36 */
37
38 UFUNCTION(BlueprintCallable, Category = Ability, Meta = (Tooltip = "This function can be used to trigger an ability on the actor in question with useful payload data."))
39 static void SendGameplayEventToActor(AActor* Actor, FGameplayTag EventTag, FGameplayEventData Payload);
40 
```

The code editor interface shows tabs for 'AbilitySystemBlueprintLibrary.h', 'UES_ThirdPersonCharacter.cpp', 'BaseCharacterAttributeSet.cpp', 'UES_ThirdPersonGameMode.cpp', 'UES_ThirdPersonGameMode.h', and 'AnimNotify_GameplayCue.cpp'. The code is color-coded with syntax highlighting.

This Struct `GameplayEventData` has many useful fields. You can use its field called `TargetData` to pass the game playability target data derived class with your own custom data.

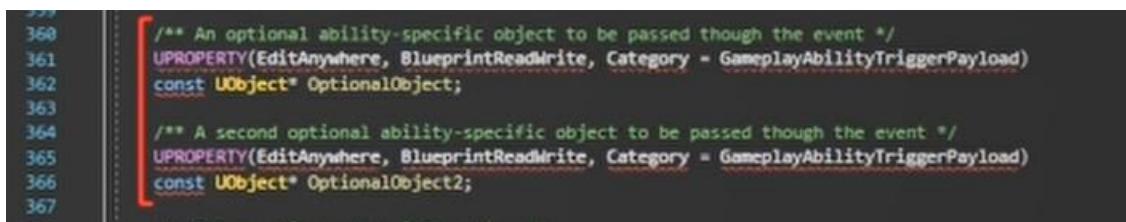


```

333     /** Metadata for a tag-based Gameplay Event, that can activate other abilities or run ability-specific logic */
334     USTRUCT(BlueprintType)
335     struct GAMEPLAYABILITIES_API FGameplayEventData
336     {
337         GENERATED_USTRUCT_BODY()
338
339         FGameplayEventData()
340         : Instigator(nullptr)
341         , Target(nullptr)
342         , OptionalObject(nullptr)
343         , OptionalObject2(nullptr)
344         , EventMagnitude(0.f)
345     };
346
347
348     /** Tag of the event that triggered this */
349     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
350     FGameplayTag EventTag;
351
352     /** The instigator of the event */
353     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
354     const Actor* Instigator;
355
356     /** The target of the event */
357     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
358     const Actor* Target;
359
360     /** An optional ability-specific object to be passed though the event */
361     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
362     const UObject* OptionalObject;
363
364     /** A second optional ability-specific object to be passed though the event */
365     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
366     const UObject* OptionalObject2;
367
368     /** Polymorphic context information */
369     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
370     FGameplayEffectContextHandle ContextHandle;
371
372     /** Tags that the instigator has */
373     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
374     FGameplayTagContainer InstigatorTags;
375
376     /** Tags that the target has */
377     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
378     FGameplayTagContainer TargetTags;
379
380     /** The magnitude of the triggering event */
381     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
382     float EventMagnitude;
383
384     /** The polymorphic target information for the event */
385     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
386     FGameplayAbilityTargetDataHandle TargetData;
387
388     ...
389 };

```

Worth mentioning that you **should be careful passing the `OptionalObject` and `OptionalObject2` in multiplayer**, because the object you will reference there should be replicated **otherwise** it will be received as **null PTR**. (it's in the above picture too)

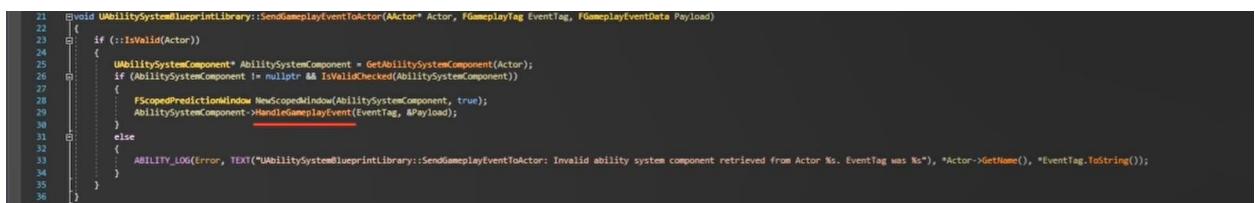


```

360     /** An optional ability-specific object to be passed though the event */
361     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
362     const UObject* OptionalObject;
363
364     /** A second optional ability-specific object to be passed though the event */
365     UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
366     const UObject* OptionalObject2;

```

But let's go back to `SendGameplayToActor` function. Here we can see we wrote the `GameplayEvent` to the `AbilitySystemComponent`.

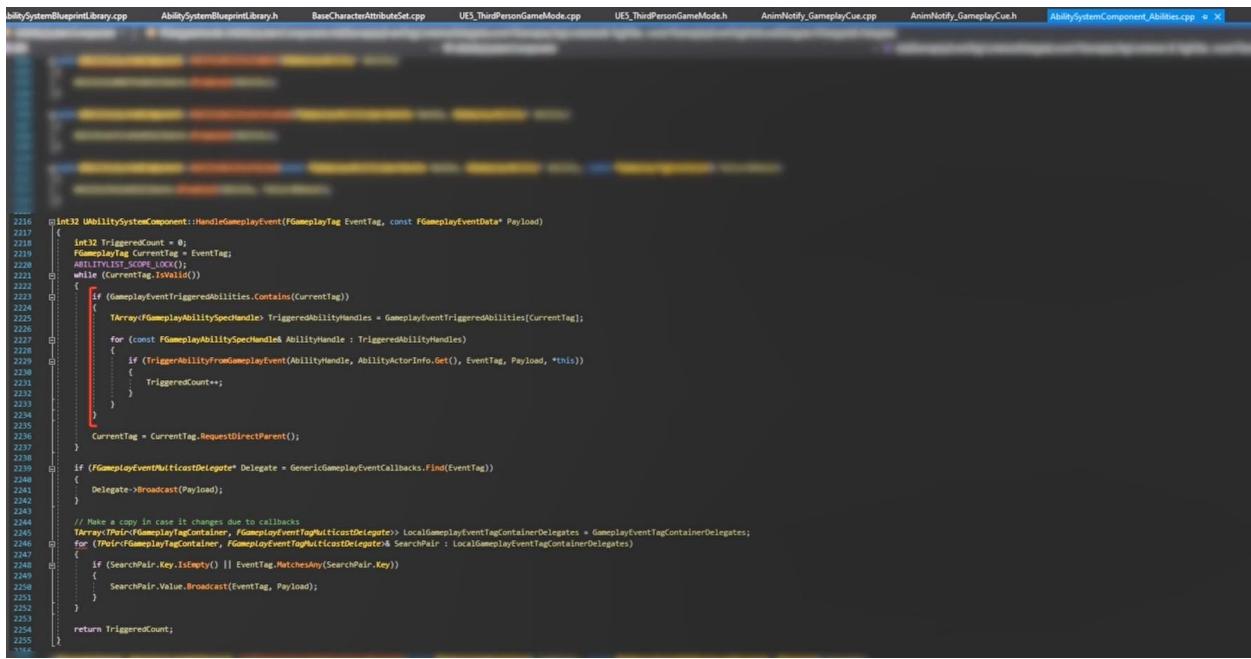


```

21     void UAbilitySystemBlueprintLibrary::SendGameplayEventToActor(Actor* Actor, FGameplayTag EventTag, FGameplayEventData Payload)
22     {
23         if (!IsValid(Actor))
24         {
25             UAbilitySystemComponent* AbilitySystemComponent = GetAbilitySystemComponent(Actor);
26             if (AbilitySystemComponent != nullptr && IsValidChecked(AbilitySystemComponent))
27             {
28                 FScopedPredictionWindow NewScopedPredictionWindow(AbilitySystemComponent, true);
29                 AbilitySystemComponent->HandleGameplayEvent(EventTag, &Payload);
30             }
31         }
32         else
33         {
34             ABILITY_LOG(Error, TEXT("UAbilitySystemBlueprintLibrary::SendGameplayEventToActor: Invalid ability system component retrieved from Actor %s. EventTag was %s"), *Actor->GetName(), *EventTag.ToString());
35         }
36     }

```

Inside the handle GameplayEvent function we conceded gameplay events can be used to activate the abilities.

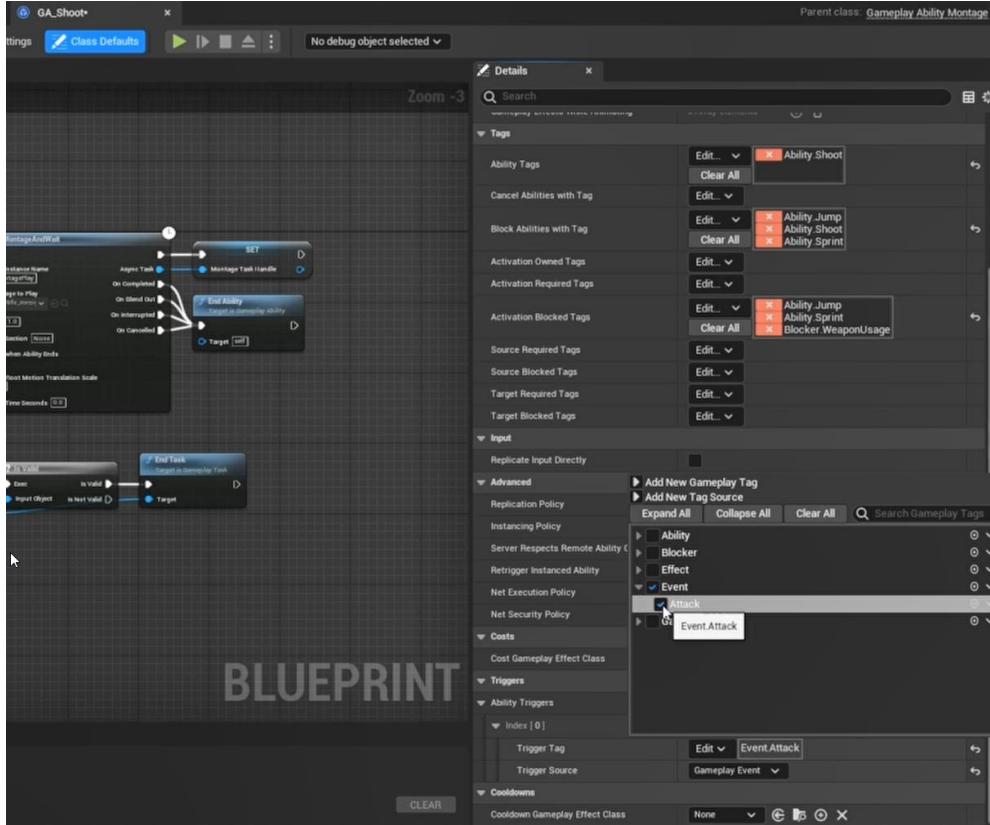


```

2216 int32 UAbilitySystemComponent::HandleGameplayEvent(FGameplayTag EventTag, const FGameplayEventData* Payload)
2217 {
2218     int32 TriggeredCount = 0;
2219     FGameplayTag CurrentTag = EventTag;
2220     ABILITYLIST_SCOPE_LOCK();
2221     while (CurrentTag.IsValid())
2222     {
2223         if (GameplayEventTriggeredAbilities.Contains(CurrentTag))
2224         {
2225             TArray<FGameplayAbilitySpecHandle> TriggeredAbilityHandles = GameplayEventTriggeredAbilities[CurrentTag];
2226 
2227             for (const FGameplayAbilitySpecHandle AbilityHandle : TriggeredAbilityHandles)
2228             {
2229                 if (TriggerAbilityFromGameplayEvent(AbilityHandle, AbilityActorInfo.Get(), EventTag, Payload, *this))
2230                 {
2231                     TriggeredCount++;
2232                 }
2233             }
2234         }
2235 
2236         CurrentTag = CurrentTag.RequestIDirectParent();
2237     }
2238 
2239     if (FGameplayEventMulticastDelegate* Delegate = GenericGameplayEventCallbacks.Find(EventTag))
2240     {
2241         Delegate->Broadcast(Payload);
2242     }
2243 
2244     // Make a copy in case it changes due to callbacks
2245     TArray<Pair<FGameplayTagContainer, FGameplayEventTagMulticastDelegate*>> LocalGameplayEventTagContainerDelegates = GameplayEventTagContainerDelegates;
2246     for (Pair<FGameplayTagContainer, FGameplayEventTagMulticastDelegate*> SearchPair : LocalGameplayEventTagContainerDelegates)
2247     {
2248         if (SearchPair.Key.IsEmpty() || EventTag.MatchesAny(SearchPair.Key))
2249         {
2250             SearchPair.Value.Broadcast(EventTag, Payload);
2251         }
2252     }
2253 
2254     return TriggeredCount;
2255 }

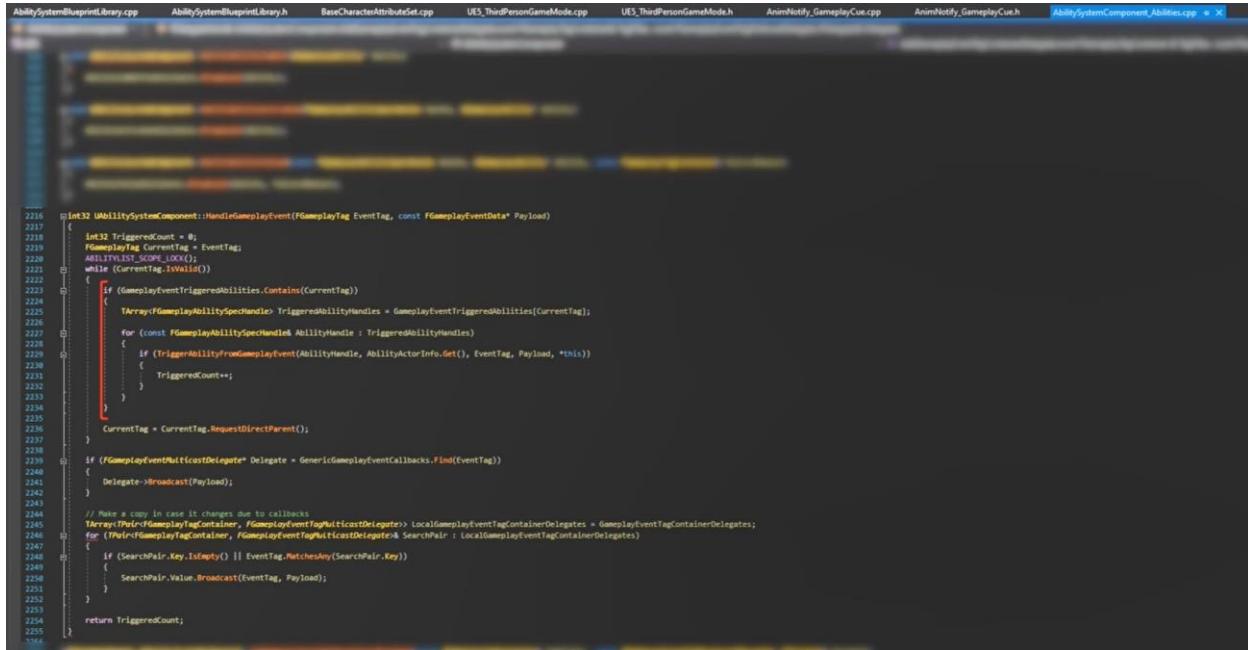
```

You can set the Ability to react to a specific event.



And when receiving an event, AbilitySystemComponent will iterate over the Abilities that should be triggered by the sent event and will try to activate them.

With this approach we can send additional data through the payload.

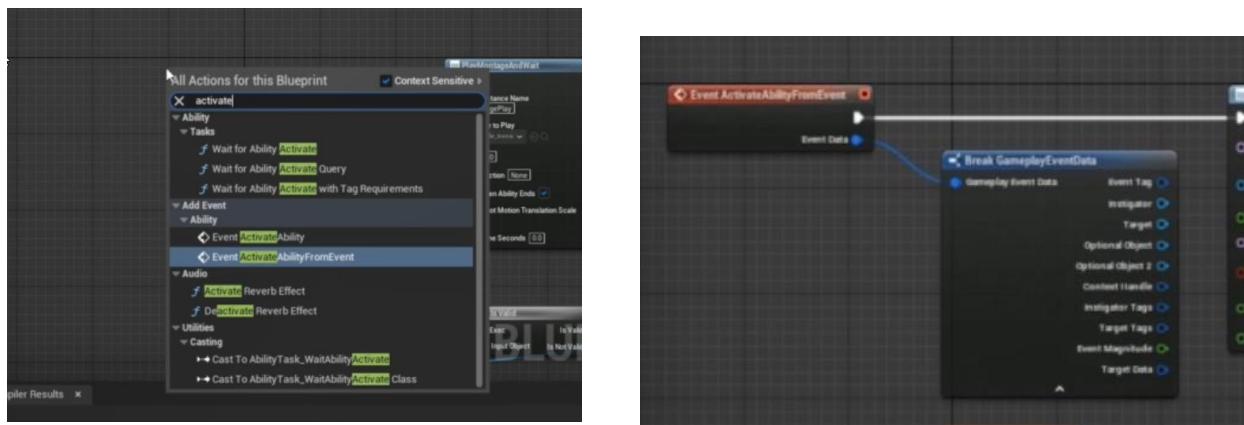


```

2237 int32 UAbilitySystemComponent::HandleGameplayEvent(FGameplayTag EventTag, const FGameplayEventData* Payload)
2238 {
2239     int32 TriggeredCount = 0;
2240     FGameplayTag CurrentTag = EventTag;
2241     ABILITYLIST_SCOPE_LOCK();
2242     while (CurrentTag.IsValid())
2243     {
2244         if (GameplayEventTriggeredAbilities.Contains(CurrentTag))
2245         {
2246             TArray<FGameplayAbilityHandle> TriggeredAbilityHandles = GameplayEventTriggeredAbilities[CurrentTag];
2247             for (const FGameplayAbilityHandle& AbilityHandle : TriggeredAbilityHandles)
2248             {
2249                 if (TriggerAbilityFromGameplayEvent(AbilityHandle, AbilityActorInfo.Get(), EventTag, Payload, *this))
2250                 {
2251                     TriggeredCount++;
2252                 }
2253             }
2254         }
2255         CurrentTag = CurrentTag.RequestDirectParent();
2256     }
2257     if (FGameplayEventMulticastDelegate* Delegate = GenericGameplayEventCallbacks.Find(EventTag))
2258     {
2259         Delegate->Broadcast(Payload);
2260     }
2261     // Make a copy in case it changes due to callbacks
2262     TArray<FGameplayEventTagContainer> LocalGameplayEventTagContainerDelegates = GameplayEventTagContainerDelegates;
2263     for (TPair<FGameplayEventContainer, FGameplayEventTagContainerDelegate> SearchPair : LocalGameplayEventTagContainerDelegates)
2264     {
2265         if (SearchPair.Key.IsEmpty() || EventTag.MatchesAny(SearchPair.Key))
2266         {
2267             SearchPair.Value.Broadcast(EventTag, Payload);
2268         }
2269     }
2270     return TriggeredCount;
2271 }

```

Inside the ability that will be triggered by an event, we should override trigger ability from event from which we can get our send payload as event data.



Here we will trigger delegates to notify everyone who's subscribed to any event received or some specific one.

```

116 int32 UAbilitySystemComponent::HandleGameplayEvent(FGameplayTag EventTag, const FGameplayEventData* Payload)
117 {
118     int32 TriggeredCount = 0;
119     FGameplayTagContainer CurrentTag;
120     ABILITYLIST_SCOPE_LOCK();
121     while (CurrentTag.IsValid())
122     {
123         if (GameplayEventTriggersAbilities.Contains(CurrentTag))
124         {
125             TArray<UAbility> TriggeredAbilityHandles = GameplayEventTriggeredAbilities[CurrentTag];
126             for (const UAbility* AbilityHandle : TriggeredAbilityHandles)
127             {
128                 if (TriggerableIfFromGameplayEvent(AbilityHandle, AbilityActorInfo.Get(), EventTag, Payload, *this))
129                 {
130                     TriggeredCount++;
131                 }
132             }
133         }
134         CurrentTag = CurrentTag.RequestDirectParent();
135     }
136     if (FGameplayEventNotificaitonDelegate* Delegate = GenericGameplayEventCallbacks.Find(EventTag))
137     {
138         Delegate->Broadcast(Payload);
139     }
140     // Note: A copy is made here to eliminate
141     // memory from GameplayEventContainer, FGameplayTagContainerDelegate, LocalGameplayEventTagContainerDelegate, & GameplayEventTagContainerDelegate
142     for (TPair<UAbility, FGameplayEventNotificaitonDelegate*> SearchPair : LocalGameplayEventTagContainerDelegates)
143     {
144         if (SearchPair.Key == EventTag) || EventTag.Neighbors(SearchPair.Key))
145         {
146             SearchPair.Value.Broadcast(EventTag, Payload);
147         }
148     }
149     return TriggeredCount;
150 }

```

We can also have a specific task called `WaitGameplayEvent` which can a synchronously wait for some `GameplayEvent` and trigger a delegate when the event will be received in.

```

// Copyright Epic Games, Inc. All Rights Reserved.
#include "AbilitySystem/AbilityAsync_WaitGameplayEvent.h"
#include "AbilitySystem/Ability.h"
#include "AbilitySystemComponent.h"

UAbilitySync_WaitGameplayEvent::UAbilitySync_WaitGameplayEvent() : Super()
{
    UAbilitySync_WaitGameplayEvent* MyObj = NewObject(UAbilitySync_WaitGameplayEvent());
    MyObj->Tag = EventTag;
    MyObj->OnlyTriggerOnce = OnlyTriggerOnce;
    MyObj->OnlyMatchExact = OnlyMatchExact;
    return MyObj;
}

void UAbilitySync_WaitGameplayEvent::Activate()
{
    Super::Activate();
    UAbilitySystemComponent* ASC = GetAbilitySystemComponent();
    if (ASC)
    {
        if (OnlyMatchExact)
        {
            MyHandle = ASC->GenericGameplayEventCallbacks.FindOrAdd(Tag).AddObject(this, UAbilitySync_WaitGameplayEvent::GameplayEventCallback);
        }
        else
        {
            MyHandle = ASC->AddGameplayEventContainerDelegate(FGameplayTagContainer(Tag), FGameplayEventTagNotifyDelegate::FDelegate::CreateObject(this, UAbilitySync_WaitGameplayEvent::GameplayEventContainerCallback));
        }
    }
    else
    {
        EndAction();
    }
}

void UAbilitySync_WaitGameplayEvent::GameplayEventCallback(const FGameplayEventData* Payload)
{
    GameplayEventContainerCallback(Tag, Payload);
}

void UAbilitySync_WaitGameplayEvent::GameplayEventContainerCallback(FGameplayTag MatchingTag, const FGameplayEventData* Payload)
{
    if (ShouldEndOnDelegates())
    {
        FGameplayEventData TempPayload = *Payload;
        TempPayload.EventTag = MatchingTag;
        EndAction(TempPayload);
    }
    if (OnlyTriggerOnce)
    {
        EndAction();
    }
}

```

Inside the task you can see have a subscribe to the mentioned earlier delegates on ability system component.

```

void UAbilitySync_WaitGameplayEvent::Activate()
{
    Super::Activate();
    UAbilitySystemComponent* ASC = GetAbilitySystemComponent();
    if (ASC)
    {
        if (OnlyMatchExact)
        {
            MyHandle = ASC->GenericGameplayEventCallbacks.FindOrAdd(Tag).AddObject(this, UAbilitySync_WaitGameplayEvent::GameplayEventCallback);
        }
        else
        {
            MyHandle = ASC->AddGameplayEventContainerDelegate(FGameplayTagContainer(Tag), FGameplayEventTagNotifyDelegate::FDelegate::CreateObject(this, UAbilitySync_WaitGameplayEvent::GameplayEventContainerCallback));
        }
    }
    else
    {
        EndAction();
    }
}

```