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,LOD,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.

```
bool UAbilitySystemComponent::ReplicateSubobjects(class UActorChannel *Channel, class FOutBunch *Bunch, FReplicationFlags *RepFlags)
{
    bool WroteSomething = Super::ReplicateSubobjects(Channel, Bunch, RepFlags);

    for (const UAttributeSet* Set : GetSpawnedAttributes())
    {
        if (IsValid(Set))
        {
            WroteSomething |= Channel->ReplicateSubobject(const_cast<UAttributeSet*>(Set), *Bunch, *RepFlags);
        }
    }

    for (UGameplayAbility* Ability : AllReplicatedInstancedAbilities)
    {
        if (IsValid(Ability))
        {
            WroteSomething |= Channel->ReplicateSubobject(Ability, *Bunch, *RepFlags);
        }
    }

    return WroteSomething;
}
```

AllReplicatedInstanceAbilities 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 UObjects.** 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 GameplayEffects with the ApplyGameplayEffectSpecToSelf funcdtion.

```
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
UE5 | UAbilitySystemComponent

//  GameplayEffects: Primary outward facing API for other systems

/** Applies a previously created gameplay effect spec to a target */
UFUNCTION(BlueprintCallable, Category = GameplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToTarget", ScriptName = "ApplyGameplayEffectSpecToTarget"))
FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpecHandle& SpecHandle, UAbilitySystemComponent* Target);

virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpec& GameplayEffect, UAbilitySystemComponent *Target, FPredictionKey PredictionKey=FP

/** Applies a previously created gameplay effect spec to this component */
UFUNCTION(BlueprintCallable, Category = GameplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToSelf", ScriptName = "ApplyGameplayEffectSpecToSelf"))
FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpecHandle& SpecHandle);

virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpec& GameplayEffect, FPredictionKey PredictionKey = FPredictionKey());
```

If the GameplayEffects have a non-instance effect, we will add it to the ActiveGameplayEffects container

```
    FGameplayEffectSpec* OurCopyOfSpec = nullptr;
    TSharedPtr<FGameplayEffectSpec> StackSpec;
    {
        if (Spec.Def->DurationPolicy != EGameplayEffectDurationType::Instant || bTreatAsInfiniteDuration)
        {
            AppliedEffect = ActiveGameplayEffects.ApplyGameplayEffectSpec(Spec, PredictionKey, bFoundExistingStackableGE);
            if (!AppliedEffect)
            {
                return FActiveGameplayEffectHandle();
            }
```

And that container among Cue containers are marked as Replicated.

```
/** Contains all of the gameplay effects that are currently active on this component */
UPROPERTY(Replicated)
FActiveGameplayEffectsContainer ActiveGameplayEffects;

/** List of all active gameplay cues, including ones applied manually */
UPROPERTY(Replicated)
FActiveGameplayCueContainer ActiveGameplayCues;

/** Replicated gameplaycues when in minimal replication mode. These are cues that would come normally come from ActiveGameplayEffects */
UPROPERTY(Replicated)
FActiveGameplayCueContainer MinimalReplicationGameplayCues;
```

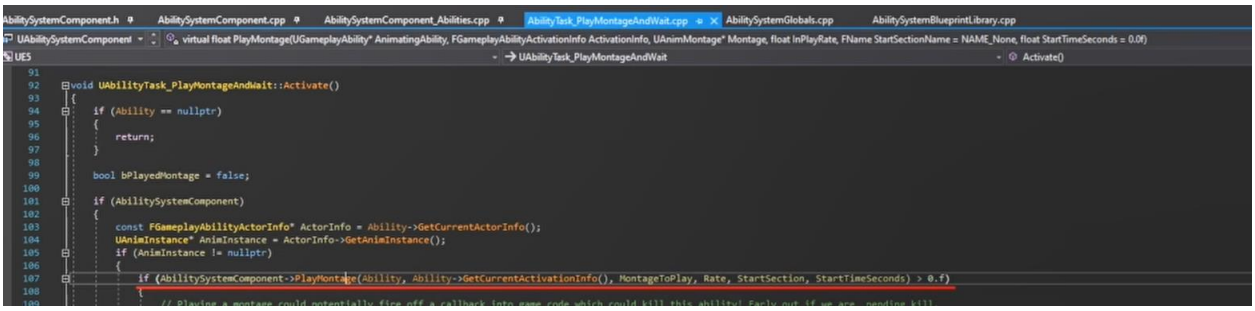
One important future of GameplayEffect that is worth mentioning, is AnimMontage Playing in Replication:

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

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.



```
/** Abilities with these tags are not able to be activated */
FGameplayTagCountContainer BlockedAbilityTags;

UE_DEPRECATED(4.26, "This will be made private in future engine versions. Use SetBlockedAbilityBindings, GetBlockedAbilityBindings, or GetBlockedAbilityBindings_Mutable instead.")
/** Tracks abilities that are blocked based on input binding. An ability is blocked if BlockedAbilityBindings[InputID] > 0 */
UPROPERTY(Transient, Replicated)
TArray<uint8> BlockedAbilityBindings;

void SetBlockedAbilityBindings(const TArray<uint8>& NewBlockedAbilityBindings);
TArray<uint8>& GetBlockedAbilityBindings_Mutable();
const TArray<uint8>& GetBlockedAbilityBindings() const;

void DebugCyclicAggregatorBroadcasts(struct FAggregator* Aggregator);

/** Acceleration map for all gameplay tags (OwnedGameplayTags from GEs and explicit GameplayCueTags) */
FGameplayTagCountContainer GameplayTagCountContainer;

UE_DEPRECATED(4.26, "This will be made private in future engine versions. Use SetMinimalReplicationTags, GetMinimalReplicationTags, or GetMinimalReplicationTags_Mutable instead.")
UPROPERTY(Replicated)
FMinimalReplicationTagCountMap 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:

```
/** Returns how the ability is instanced when executed. This limits what an ability can do in its implementation. */
EGameplayAbilityInstancingPolicy::Type GetInstancingPolicy() const
{
    return InstancingPolicy;
}

/** How an ability replicates state/events to everyone on the network */
EGameplayAbilityReplicationPolicy::Type GetReplicationPolicy() const
{
    return ReplicationPolicy;
}

/** Where does an ability execute on the network? Does a client "ask and predict", "ask and wait", "don't ask (just do it)" */
EGameplayAbilityNetExecutionPolicy::Type GetNetExecutionPolicy() const
{
    return NetExecutionPolicy;
}

/** Where should an ability execute on the network? Provides protection from clients attempting to execute restricted abilities. */
EGameplayAbilityNetSecurityPolicy::Type GetNetSecurityPolicy() const
{
    return NetSecurityPolicy;
}
```

1.GameplayAbilityInstacingPolicy:

GameplayAbilityTypes.h | GameplayAbility.h | GameplayAbilityTypes.cpp | AbilitySystemComponent.h | Test1_THOPCharacter.h | Test1_THOPCharacter.cpp

TStructOpsTypeTraits<FGar → template<> struct TStructOpsTypeTraits<FGameplayAbilityRepAnimMontage> : public TStructOpsTypeTraitsBase2<FGameplayAbilityRepAnimMontag

UE5 → TStructOpsTypeTraits<FGameplayAbilityRepAnimMontage>::__unn

```
#define ENABLE_ABILITYTASK_DEBUGMSG !(UE_BUILD_SHIPPING | UE_BUILD_TEST)


UENUM(BlueprintType)
namespace EGameplayAbilityInstancingPolicy
{
    /**
     *  How the ability is instanced when executed. This limits what an ability can do in its implementation. For example, a NonInstanced
     *  Ability cannot have state. It is probably unsafe for an InstancedPerActor ability to have latent actions, etc.
     */
    enum Type
    {
        // This ability is never instanced. Anything that executes the ability is operating on the CDO.
        NonInstanced,

        // Each actor gets their own instance of this ability. State can be saved, replication is possible.
        InstancedPerActor,

        // We instance this ability each time it is executed. Replication possible but not recommended.
        InstancedPerExecution,
    };
}
```

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):

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

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    }
```

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);
}
```

---

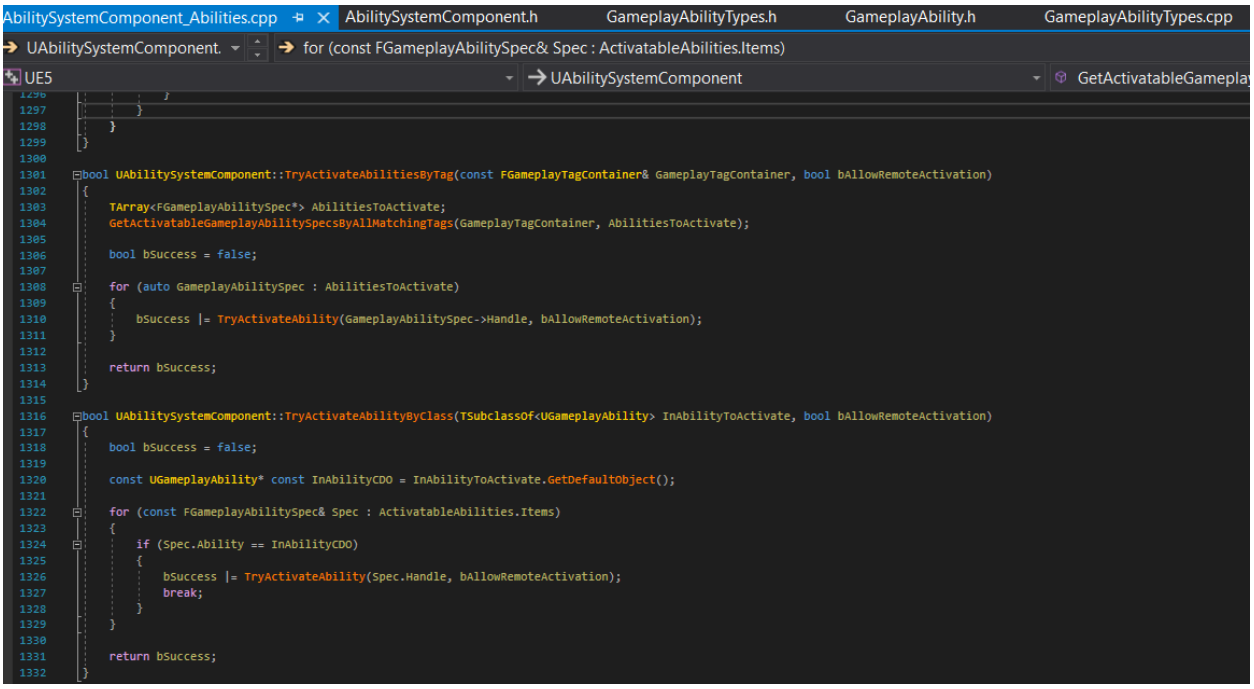
2. NetExecutionPolicy:

```
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      UMETA(DisplayName = "Local Predicted"),

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

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

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

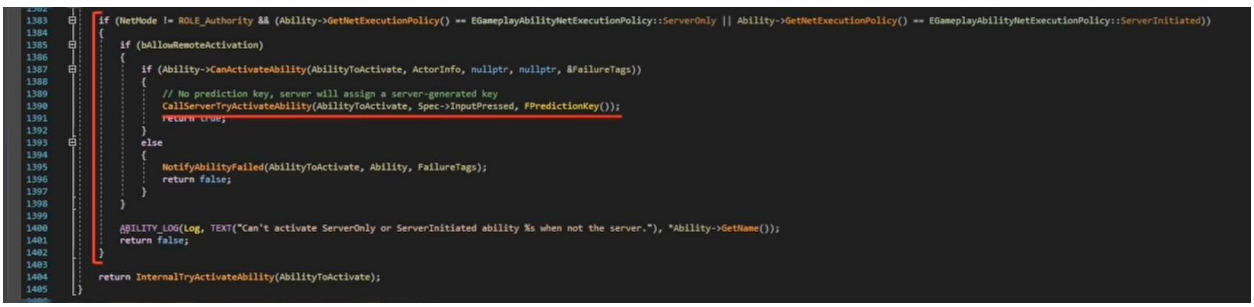
It Defines where the ability will Execute.

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.

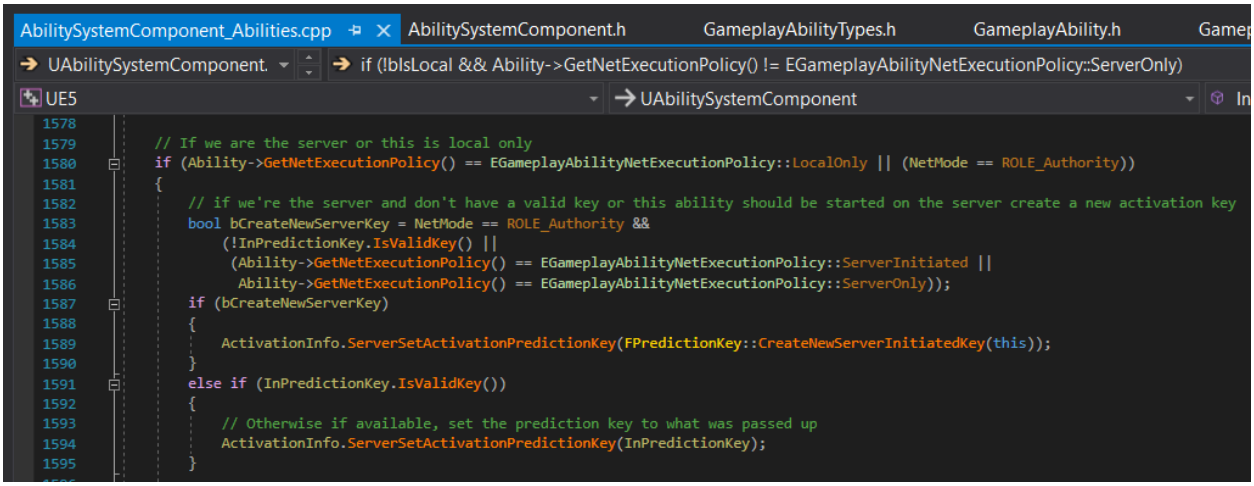


In a special case in which we don't have an authority but we start the ability for ServerOnly or ServerInitiated. This can be possible if we set the bAllowRemoteActivastion. In this case we'll directly move the ability activation to the server:



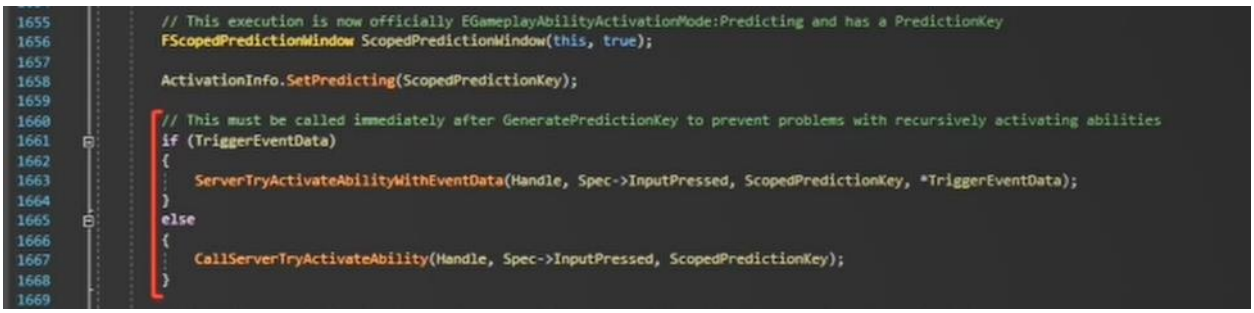
And for all other cases we go to InternalTryActivateAbility (all of them are in the above picture)

In case of LocalOnly or Authority we create and set the Predictionkey.



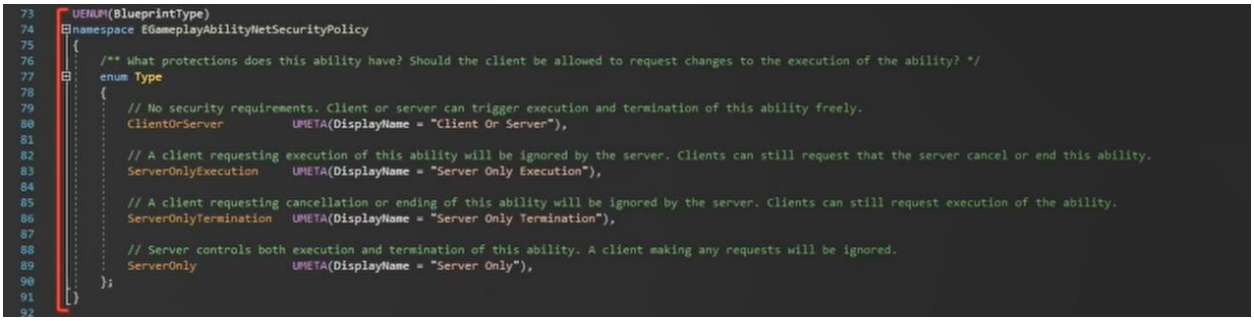
PredictionKey is an entity that will be associated with the predictive actions, and can be used to identify and roll back them if server decides.

In case of LocalPredicted we first create and set Prediction window and then activate the ability with EvenData or without.



---

3. NetSecurityPolicy:



Is responsible to check that can restrict where the Ability **can execute**.

4. ReplicationPolicy:

```
93    UENUM(BlueprintType)
94    namespace 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         UMETA(DisplayName = "Do Not Replicate"),
101
102           // We replicate the instance of the ability to the owner.
103           ReplicateYes        UMETA(DisplayName = "Replicate"),
104       };
105   }
```

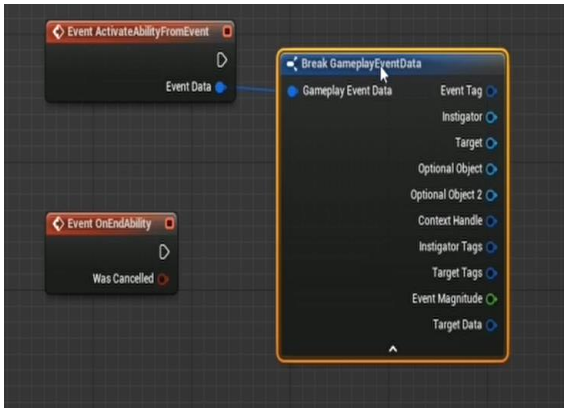
It Defines whether we replicate an instance of the ability to owner or not.

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 reources/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)

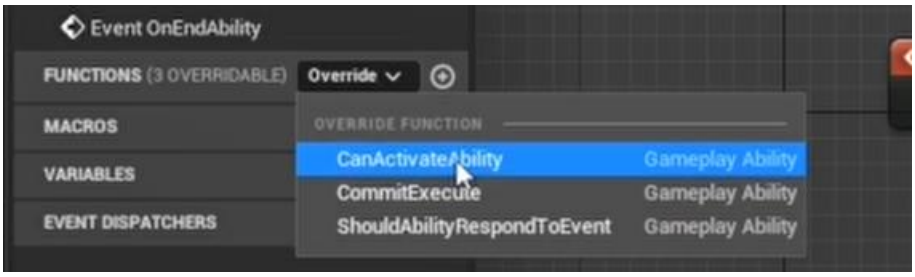
Also some of them are available as Events and some we can just overwrite to.



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.

```
493     /** This ability has these tags */
494     UPROPERTY(EditDefaultsOnly, Category = Tags, meta=(Categories="AbilityTagCategory"))
495     FGameplayTagContainer AbilityTags;
496
```

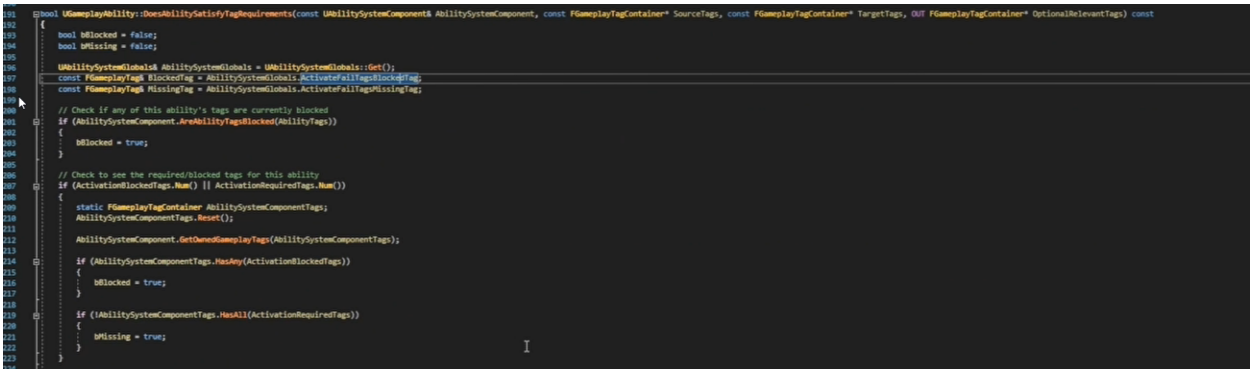
They will be check inside of the DoesAbilitySatisfyTagRequirements function which is inside of the CanActivateAbility function.

```
if (!DoesAbilitySatisfyTagRequirements(*AbilitySystemComponent, SourceTags, TargetTags, OptionalRelevantTags))
{   // If the ability's tags are blocked, or if it has a "Blocking" tag or is missing a "Required" tag, then it can't activate.
    if (FScopedCanActivateAbilityLogEnabler::IsLoggingEnabled())
    {
        ABILITY_VLOG(ActorInfo->OwnerActor.Get(), Verbose, TEXT("Ability could not be activated due to Blocking Tags or Missing Required Tags: %s"), *GetName());
    }
    return false;
}
```

```
bool UGameplayAbility::CanActivateAbility(const FGameplayAbilitySpecHandle Handle, const FGameplayAbilityActorInfo* ActorInfo, const FGameplayTagContainer* SourceTags, const FGa
{
```

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)

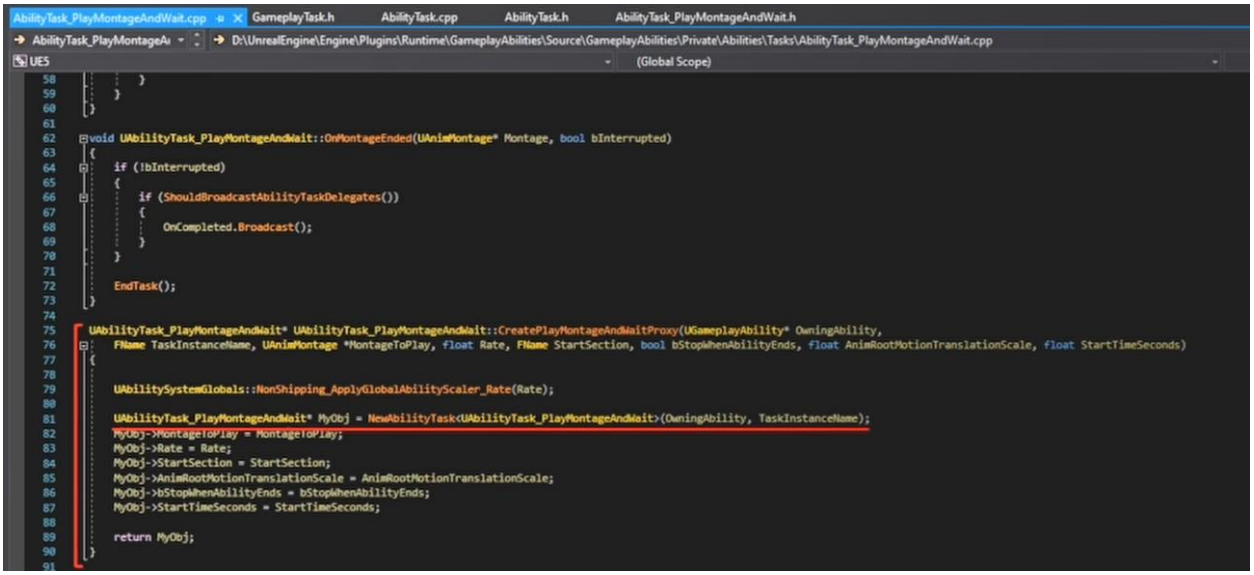


---

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.

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



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

```cpp
class GAMEPLAYABILITIES_API UAbilityTask_PlayMontageAndWait : public UAbilityTask
{
    GENERATED_UCLASS_BODY()

    UPROPERTY(BlueprintAssignable)
    FMontageWaitSimpleDelegate  OnCompleted;

    UPROPERTY(BlueprintAssignable)
    FMontageWaitSimpleDelegate  OnBlendOut;

    UPROPERTY(BlueprintAssignable)
    FMontageWaitSimpleDelegate  OnInterrupted;

    UPROPERTY(BlueprintAssignable)
    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.

```cpp
        return FGameplayResourceSet(FFlagContainer(-1));
    }

    static FGameplayResourceSet NoResources()
    {
        return FGameplayResourceSet(FFlagContainer(0));
    }

    FString GetDebugDescription() const;
};

UCLASS(Abstract, BlueprintType, meta = (ExposedAsyncProxy=AsyncTask), config = Game)
class GAMEPLAYTASKS_API UGameplayTask : public UObject, public IGameplayTaskOwnerInterface
{
    GENERATED_BODY()

public:

    DECLARE_DYNAMIC_MULTICAST_DELEGATE(FGenericGameplayTaskDelegate);

    UGameplayTask(const FObjectInitializer& ObjectInitializer);

    /** Called to trigger the actual task once the delegates have been set up */
    UFUNCTION(BlueprintCallable, meta = (BlueprintInternalUseOnly = "true"), Category = "Gameplay Tasks")
    void ReadyForActivation();
```

```cpp
    }
    return OwnerInterface;
}

IGameplayTaskOwnerInterface* UGameplayTask::ConvertToTaskOwner(AActor& OwnerActor)
{
    IGameplayTaskOwnerInterface* OwnerInterface = Cast<IGameplayTaskOwnerInterface>(&OwnerActor);

    if (OwnerInterface == nullptr)
    {
        OwnerInterface = OwnerActor.FindComponentByClass<UGameplayTasksComponent>();
    }
    return OwnerInterface;
}

void UGameplayTask::ReadyForActivation()
{
    if (UGameplayTasksComponent* TasksPtr = TasksComponent.Get())
    {
        if (RequiresPriorityOrResourceManagement() == false)
        {
            PerformActivation();
        }
        else
        {
            TasksPtr->AddTaskReadyForActivation(*this);
        }
    }
    else
    {
        EndTask();
    }
}
```

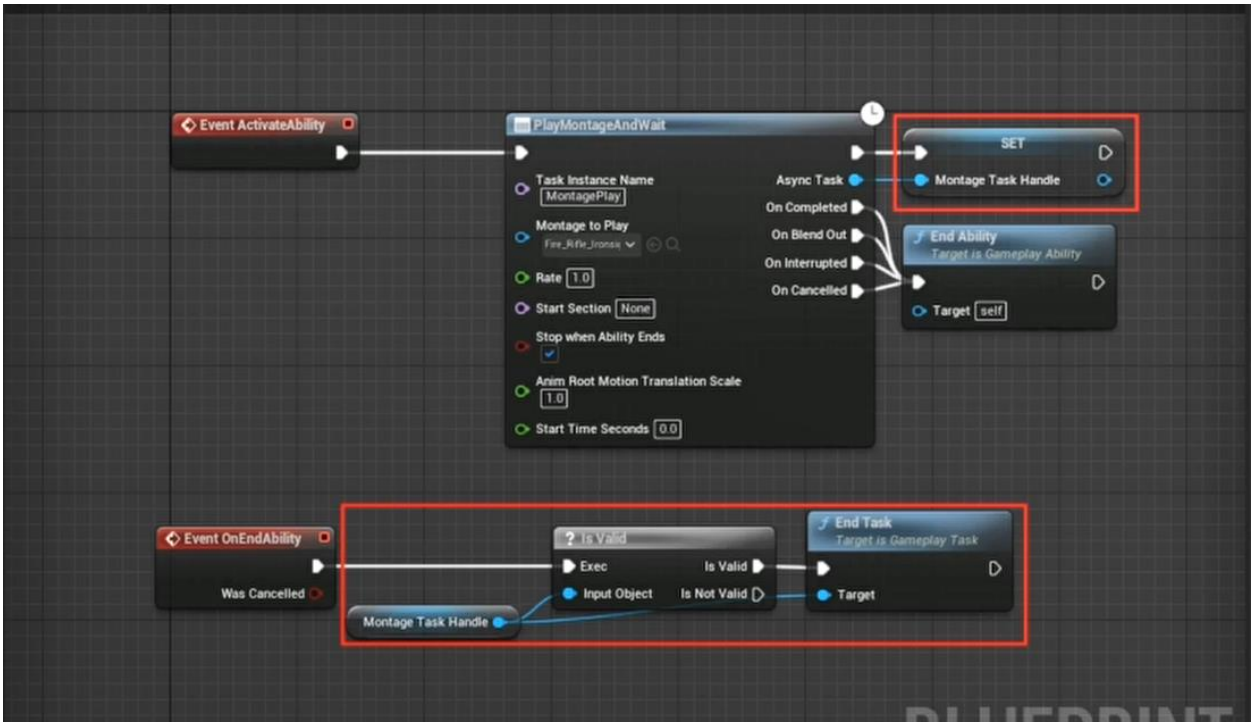
Inside of the ReadyForActivation function we'll see it refers to a GameplayTaskComponent.

GameplayTaskComponent is the AbilitySystemComponent direct parent before ActorComponent.

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

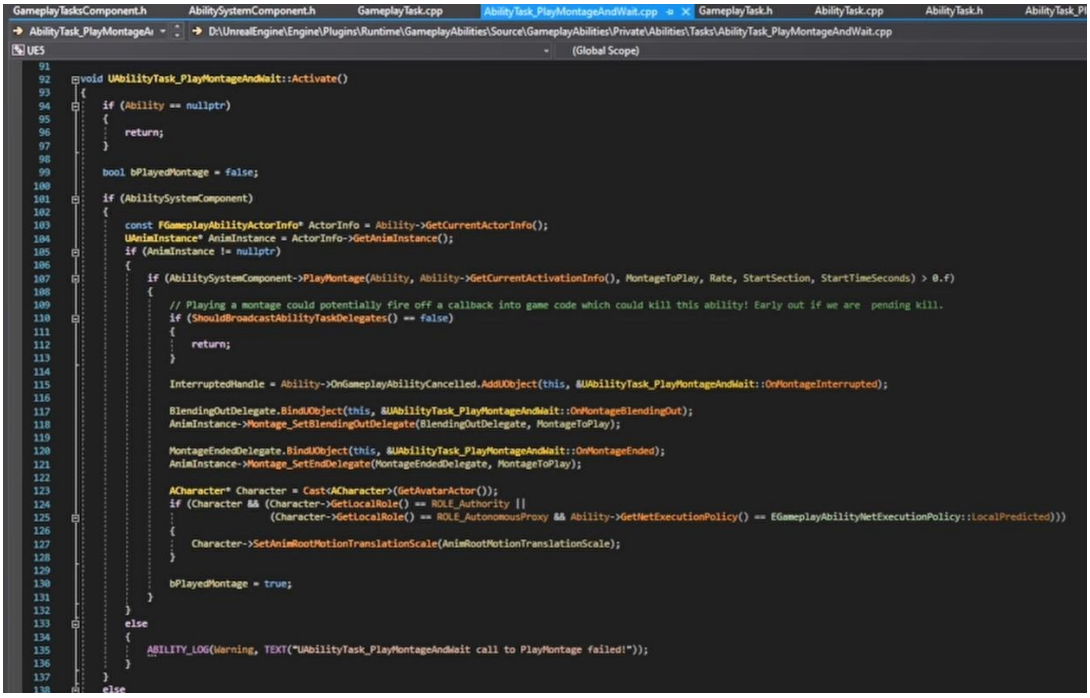
```cpp
    Full,
};

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

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.



```
void UAbiliityTask_PlayMontageAndWait::Activate()
{
    if (Ability == nullptr)
    {
        return;
    }

    bool bPlayedMontage = false;

    if (AbilitySystemComponent)
    {
        const FGameplayAbilityActorInfo* ActorInfo = Ability->GetCurrentActorInfo();
        UAnimInstance* AnimInstance = ActorInfo->GetAnimInstance();
        if (AnimInstance != nullptr)
        {
            if (AbilitySystemComponent->PlayMontage(Ability, Ability->GetCurrentActivationInfo(), MontageToPlay, Rate, StartSection, StartTimeSeconds) > 0.f)
            {
                // Playing a montage could potentially fire off a callback into game code which could kill this ability! Early out if we are  pending kill.
                if (ShouldBroadcastAbilityTaskDelegates() == false)
                {
                    return;
                }

                InterruptedHandle = Ability->OnGameplayAbilityCancelled.AddUObject(this, &UAbilityTask_PlayMontageAndWait::OnMontageInterrupted);

                BlendingOutDelegate.BindUObject(this, &UAbilityTask_PlayMontageAndWait::OnMontageBlendingOut);
                AnimInstance->Montage_SetBlendingOutDelegate(BlendingOutDelegate, MontageToPlay);

                MontageEndedDelegate.BindUObject(this, &UAbilityTask_PlayMontageAndWait::OnMontageEnded);
                AnimInstance->Montage_SetEndDelegate(MontageEndedDelegate, MontageToPlay);

                ACharacter* Character = Cast<ACharacter>(GetAvatarActor());
                if (Character && (Character->GetLocalRole() == ROLE_Authority ||
                                  (Character->GetLocalRole() == ROLE_AutonomousProxy && Ability->GetNetExecutionPolicy() == EGameplayAbilityNetExecutionPolicy::LocalPredicted)))
                {
                    Character->SetAnimRootMotionTranslationScale(AnimRootMotionTranslationScale);
                }

                bPlayedMontage = true;
            }
        }
        else
        {
            ABILITY_LOG(Warning, TEXT("UAbilityTask_PlayMontageAndWait call to PlayMontage failed!"));
        }
    }
    else
```

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.

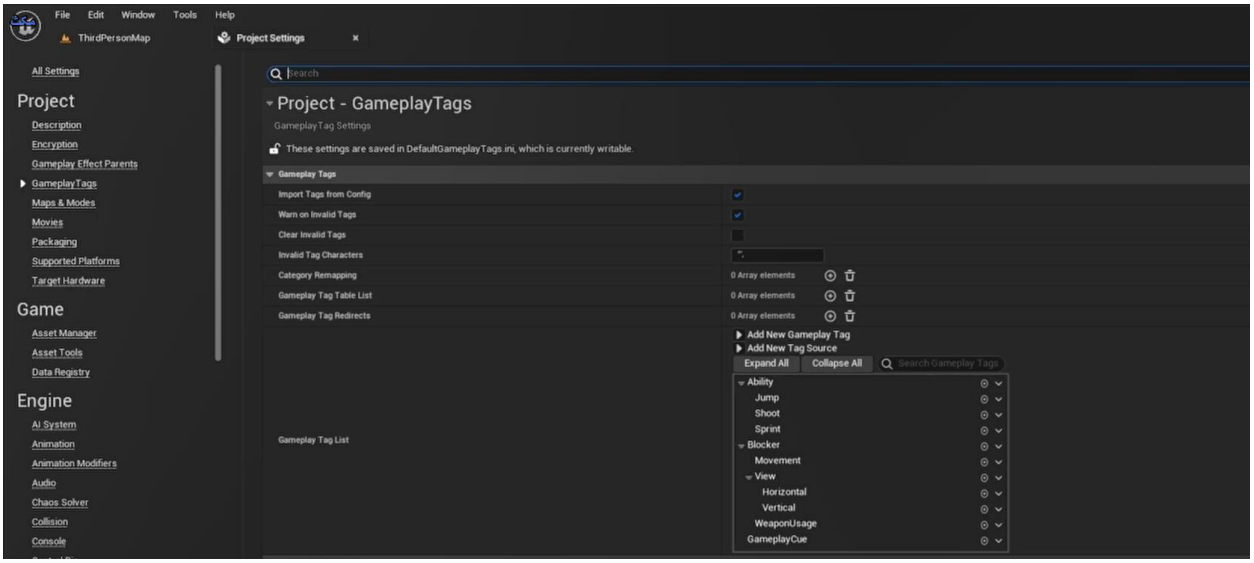
AbilityTask's can run a synchronously and can Tick. So they are useful if you need **some logic to execute repeatedly**. When you don't need Tick anymore, you can just stop the Task without ending the ability.

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

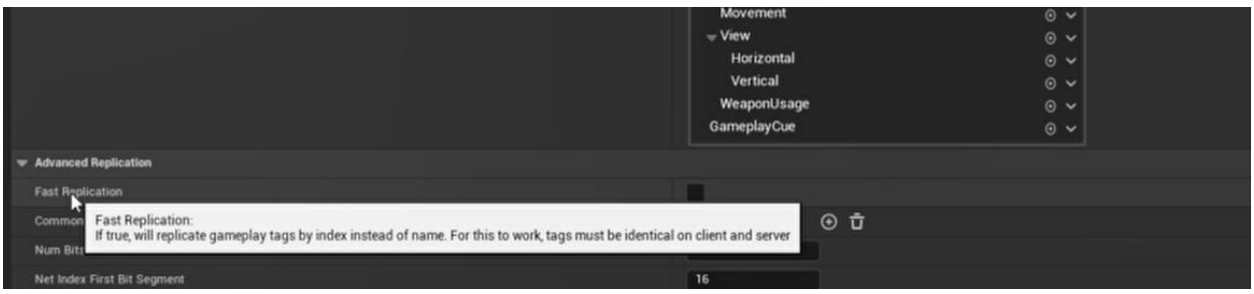
---

5. Gameplay Tags

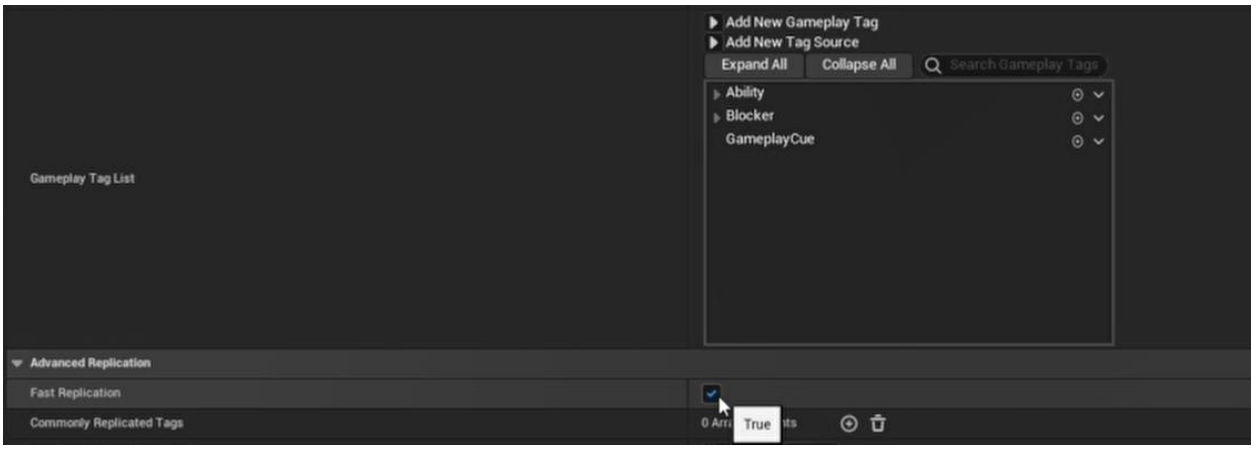
Are unique hierarchical keys that can be used for many purposes.

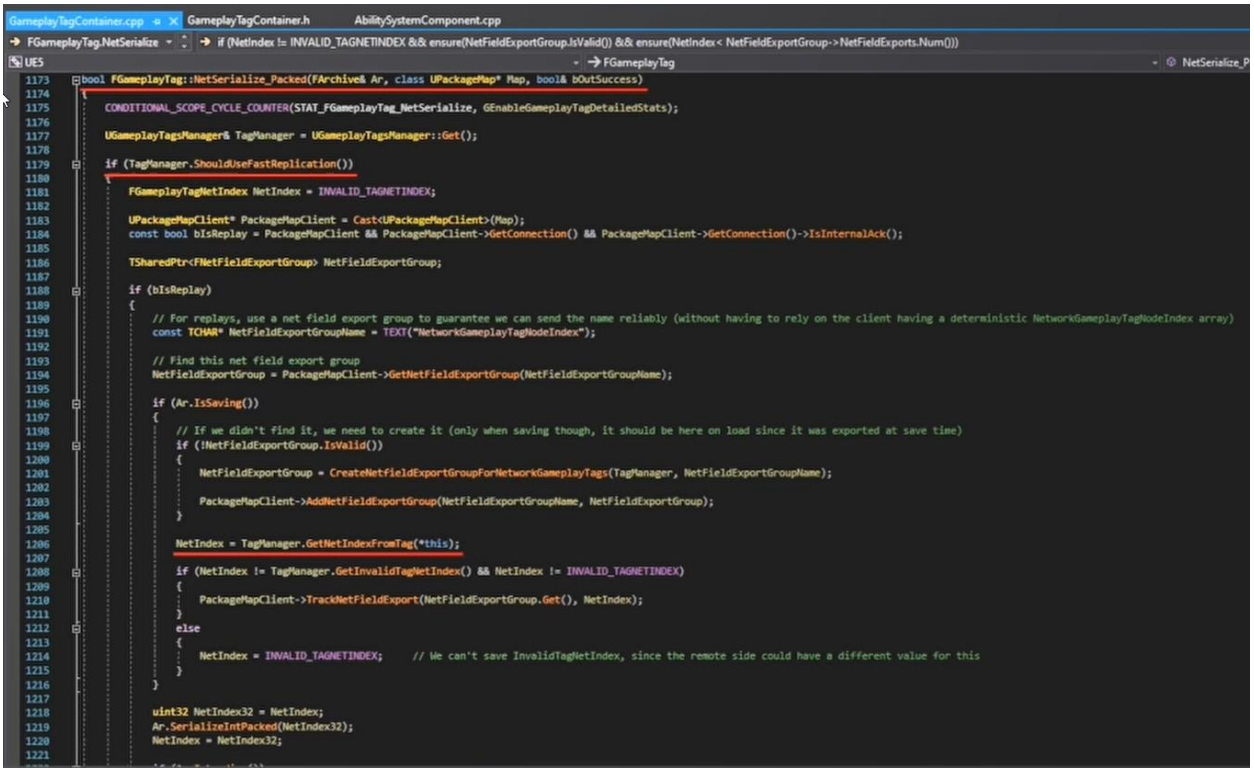


They look like strings, but because we do not intend to add or remove the Tags in runtime, we can replicate them with their indexes which makes replication lightweight.



After making that true we'll have:





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):

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

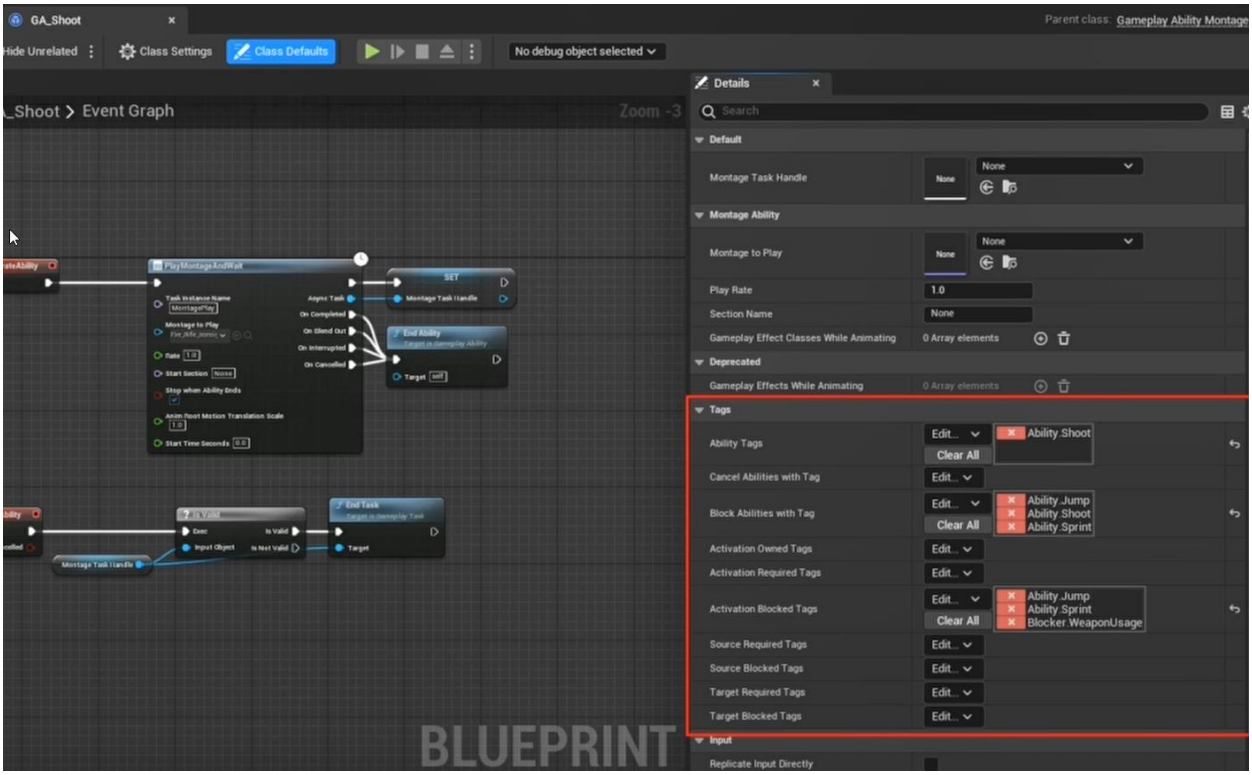
```
MovementBlockingTag = FGameplayTag::RequestGameplayTag(TEXT("Blocker.Movement"));
```

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.

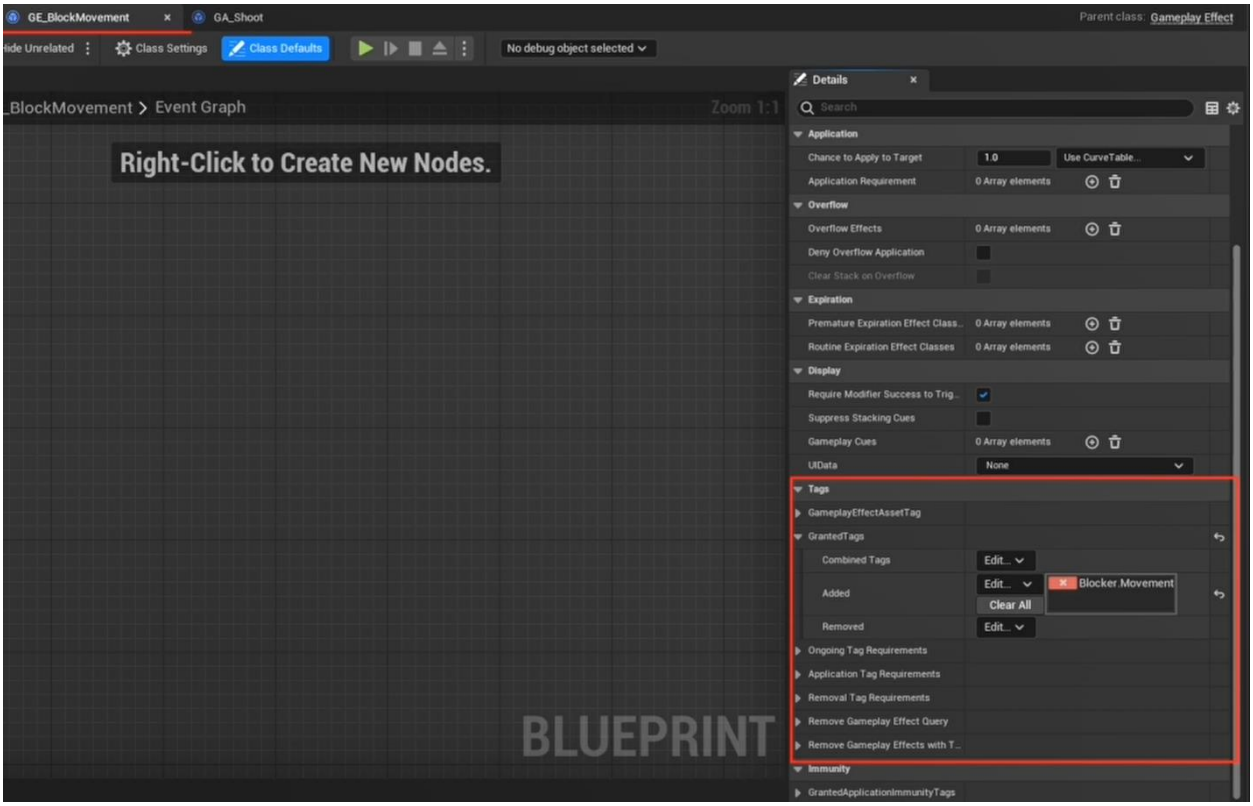
A set of Tags can be stored inside of a GamplayTagContainer.

```
UPROPERTY(EditDefaultsOnly)
FGameplayTag MovementBlockingTag;

UPROPERTY(EditDefaultsOnly)
FGameplayTagContainer MovementBlockingTags;
```

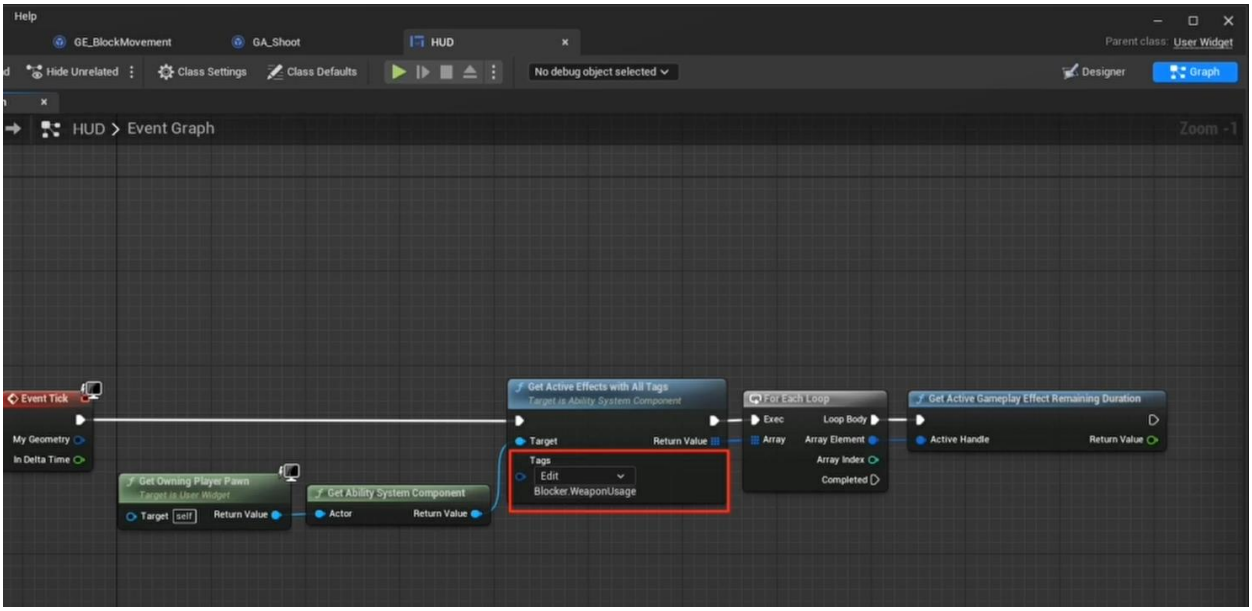
Ability can have tags, that will be **associated with** or tags it will directly apply to the owning ability system component. Or just use rules based on tags to define relations with other abilities.



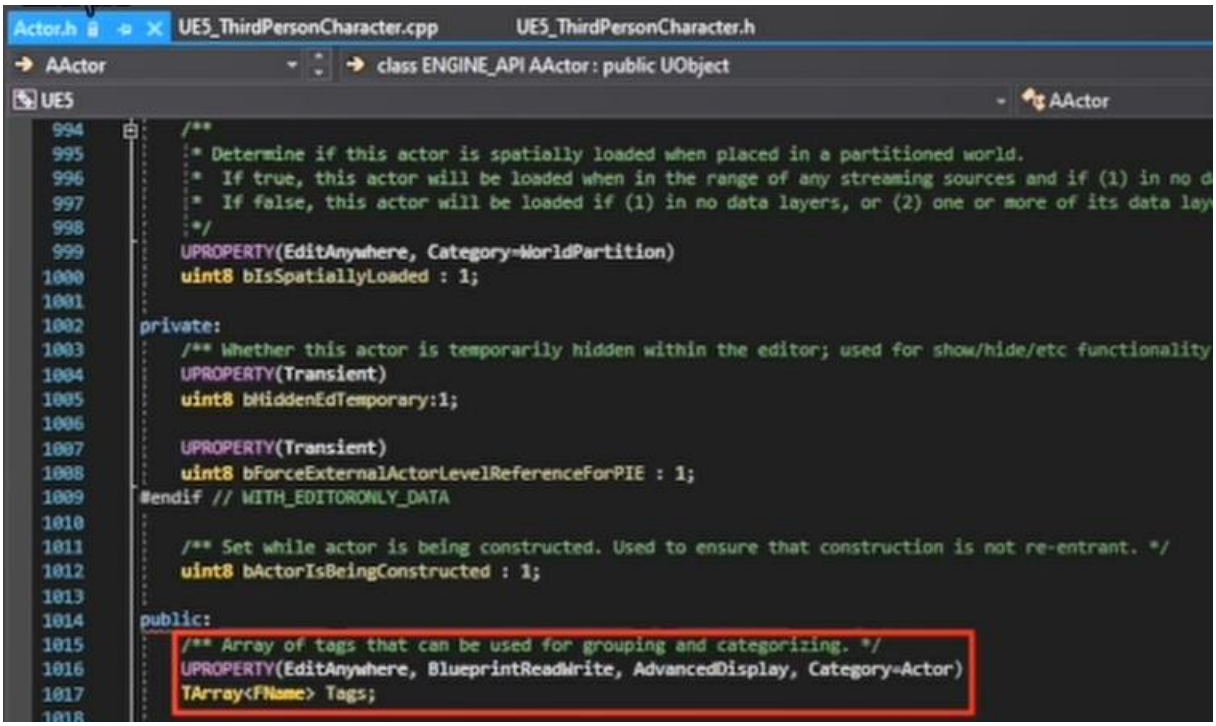
Gameplay effect can also apply tags to the target. All tags applied to the target in both cases will be automatically removed **when ability ends** or the **effect is removed**.



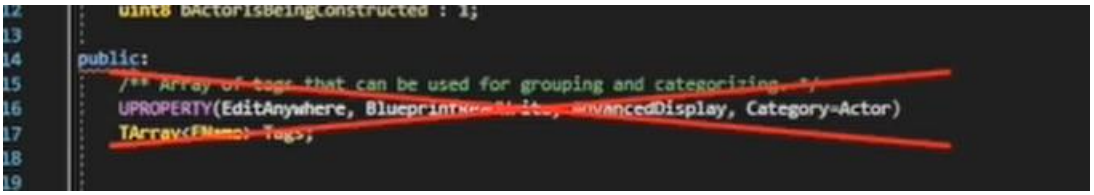
Through the AbilitySystemComponent you can **directly check** if some **actor has some tags** or **effect with certain tags** and use this information in other systems if you need.



Worth mentioning that actor himself has its own FNameTag Array.

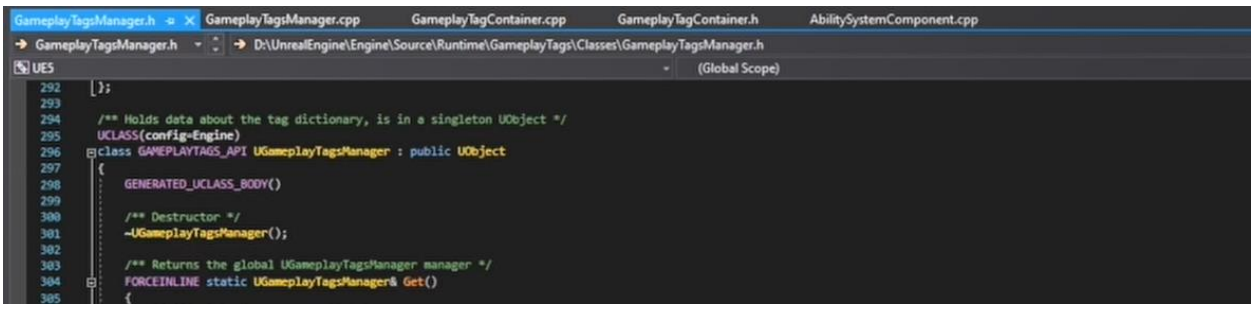


But we'll be not working with this.

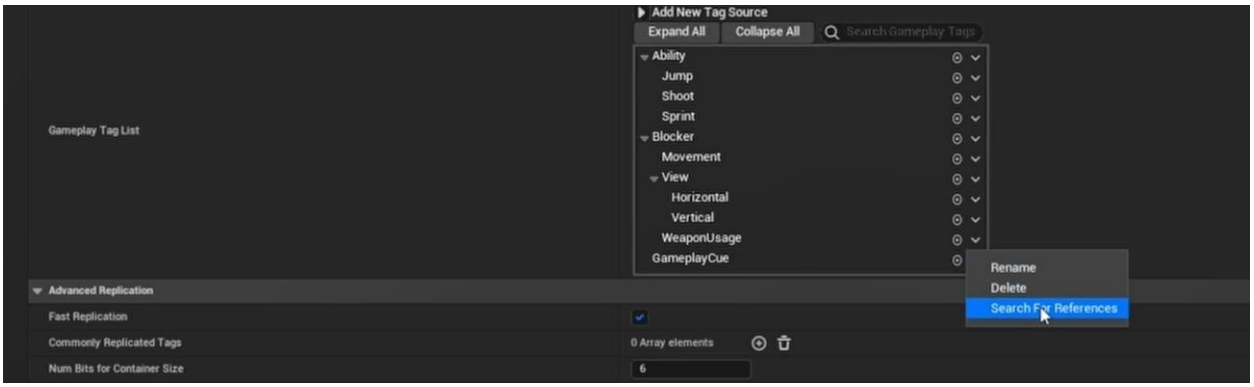


One of the biggest advantage of applying tags we can track when some tag is added or removed via delegates.

On the high level, tags are managed by GameplayTagsManager.



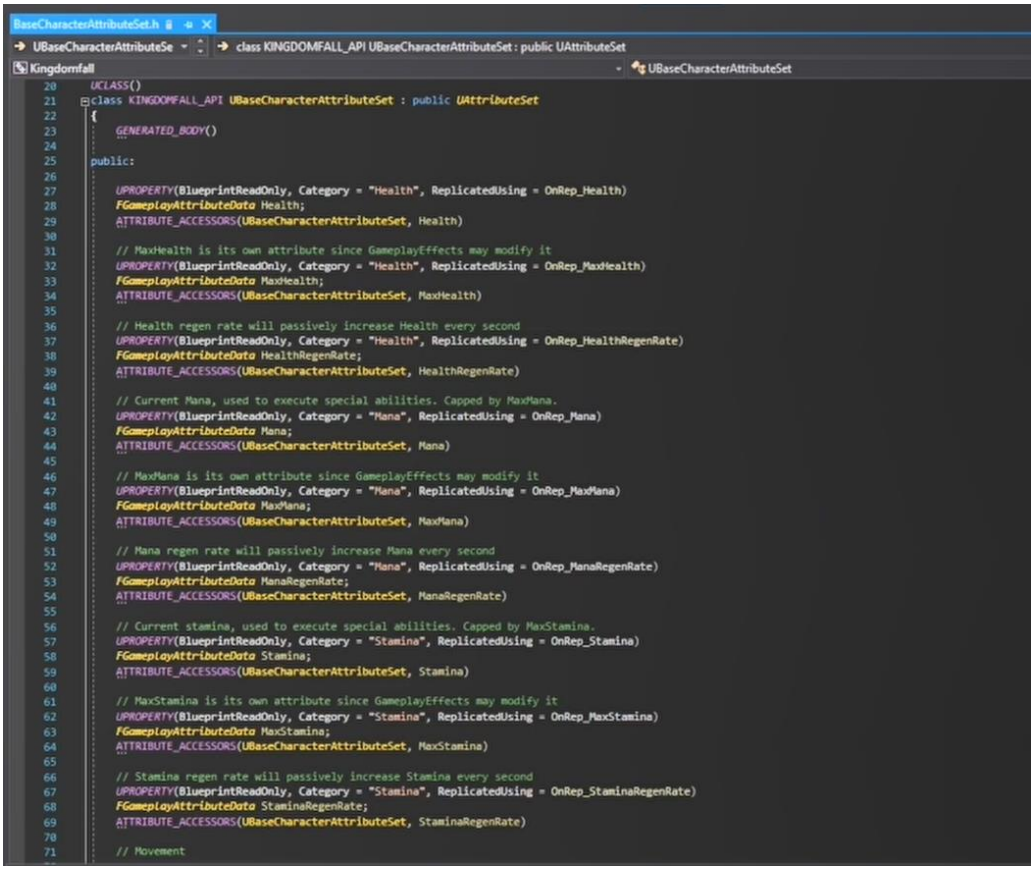
You can always check **where the Tag is referenced from inside the editor**. But remember it won't work with the hard coded references in code.



---

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.



Next **add your AttributesSet** as a SubObject to your actor that owns AbilitySystemComponent.

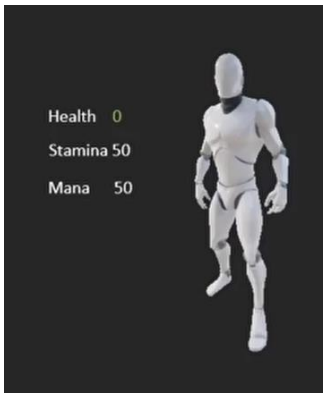
```
// Create the attribute set, this replicates by default
// Adding it as a subobject of the owning actor of an AbilitySystemComponent
// automatically registers the AttributeSet with the AbilitySystemComponent
BaseCharacterAttributeSet = CreateDefaultSubobject<UBaseCharacterAttributeSet>(TEXT("BaseCharacterAttributeSet"));

HealthChangedDelegateHandle = AbilitySystemComponent->GetGameplayAttributeValueChangeDelegate(BaseCharacterAttributeSet->GetHealthAttribute()).AddUObject(this, &ABaseCharacter::OnHealthChanged);
MaxWalkSpeedChangedDelegateHandle = AbilitySystemComponent->GetGameplayAttributeValueChangeDelegate(BaseCharacterAttributeSet->GetMaxMovementSpeedAttribute()).AddUObject(this, &ABaseCharacter::OnMaxWalkSpeedChanged);
```

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.

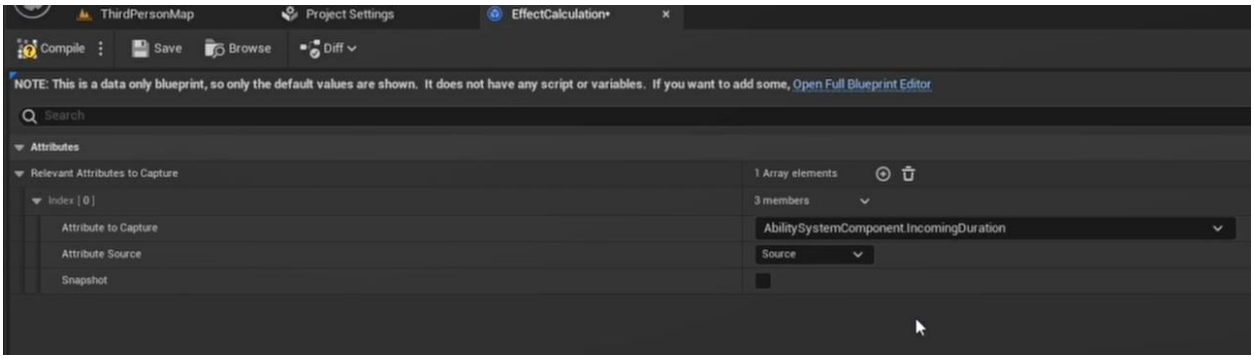
```
// Create the attribute set, this replicates by default
// Adding it as a subobject of the owning actor of an AbilitySystemComponent
// automatically registers the AttributeSet with the AbilitySystemComponent
BaseCharacterAttributeSet = CreateDefaultSubobject<UBaseCharacterAttributeSet>(TEXT("BaseCharacterAttributeSet"));

HealthChangedDelegateHandle = AbilitySystemComponent->GetGameplayAttributeValueChangeDelegate(BaseCharacterAttributeSet->GetHealthAttribute()).AddUObject(this, &ABaseCharacter::OnHealthChanged);
MaxWalkSpeedChangedDelegateHandle = AbilitySystemComponent->GetGameplayAttributeValueChangeDelegate(BaseCharacterAttributeSet->GetMaxMovementSpeedAttribute()).AddUObject(this, &ABaseCharacter::OnMaxWalkSpeedChanged);
```

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.



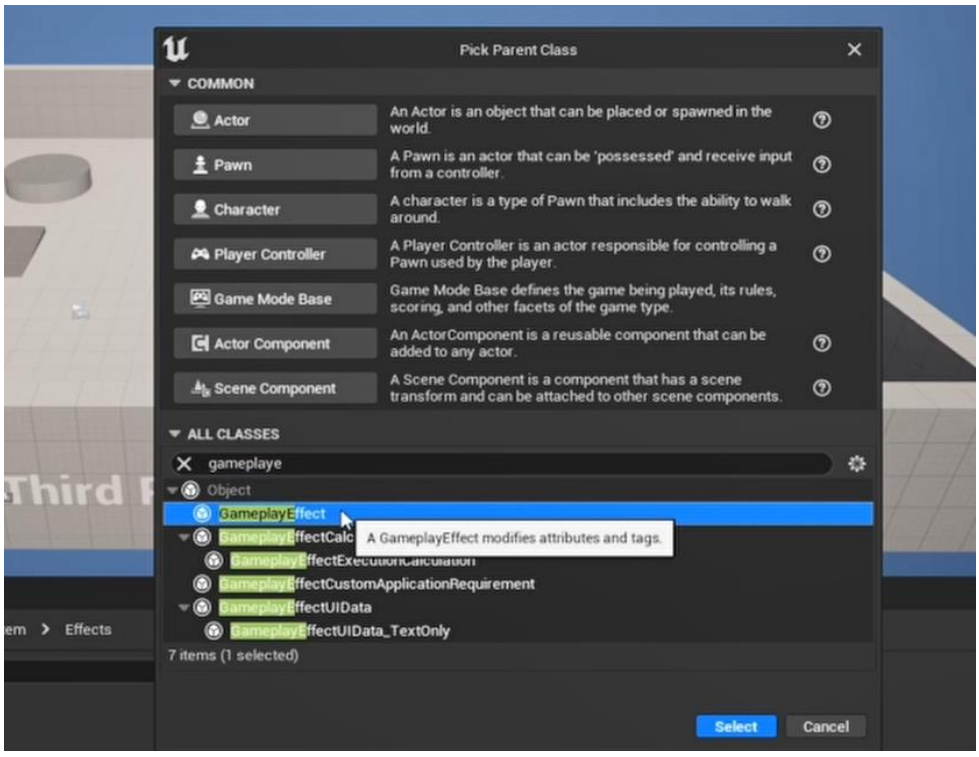
---

8. Gameplay Effects

GameplayEffects Are UObjects mostly used to change attributes, Apply tags and gameplay cues.

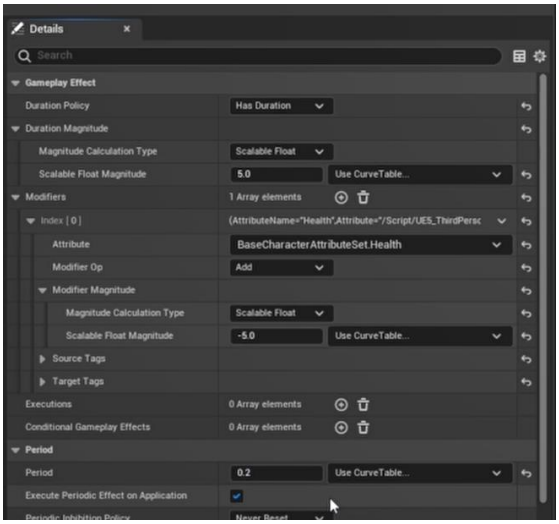
They're not designed to run logic, so we should work with them just with they **exposed properties**.

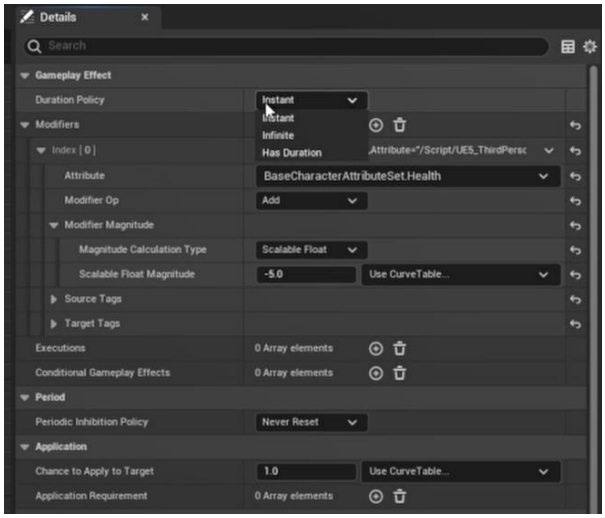
Most likely you won't need to use C++ to create your own derived class for them, but it's possible. Usually you just have to create UGamePlayEffect arrived blueprint class.



GamplayEffects 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)

```
/** Applies a previously created gameplay effect spec to a target */
UFUNCTION(BlueprintCallable, Category = GameplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToTarget", ScriptName = "ApplyGameplayEffectSpecToTarget"))
FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpecHandle& SpecHandle, UAbilitySystemComponent* Target);

virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpec& GameplayEffect, UAbilitySystemComponent *Target, FPredictionKey PredictionKey=FPredictionKey());

/** Applies a previously created gameplay effect spec to this component */
UFUNCTION(BlueprintCallable, Category = GameplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToSelf", ScriptName = "ApplyGameplayEffectSpecToSelf"))
FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpecHandle& SpecHandle);

virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpec& GameplayEffect, FPredictionKey PredictionKey = FPredictionKey());
```

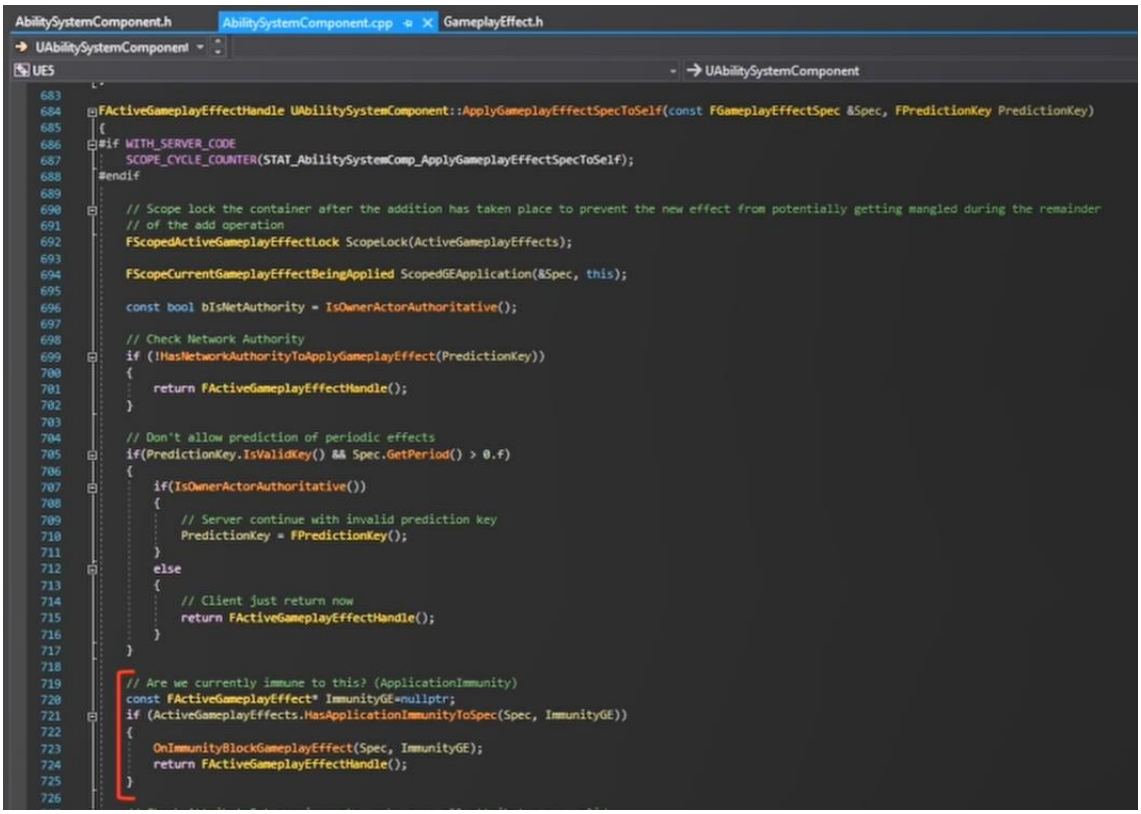
One important thing you should also remember, after applying the effect, we can have its handle. Which can be useful to **try in specific instance of it**. Which might be useful sometimes.

```
/** Applies a previously created gameplay effect spec to a target */
UFUNCTION(BlueprintCallable, Category = GameplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToTarget", ScriptName = "ApplyGameplayEffectSpecToTarget"))
FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpecHandle& SpecHandle, UAbilitySystemComponent* Target);

virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToTarget(const FGameplayEffectSpec& GameplayEffect, UAbilitySystemComponent *Target, FPredictionKey PredictionKey=FPredictionKey());

/** Applies a previously created gameplay effect spec to this component */
UFUNCTION(BlueprintCallable, Category = GameplayEffects, meta = (DisplayName = "ApplyGameplayEffectSpecToSelf", ScriptName = "ApplyGameplayEffectSpecToSelf"))
FActiveGameplayEffectHandle BP_ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpecHandle& SpecHandle);

virtual FActiveGameplayEffectHandle ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpec& GameplayEffect, FPredictionKey PredictionKey = FPredictionKey());
```

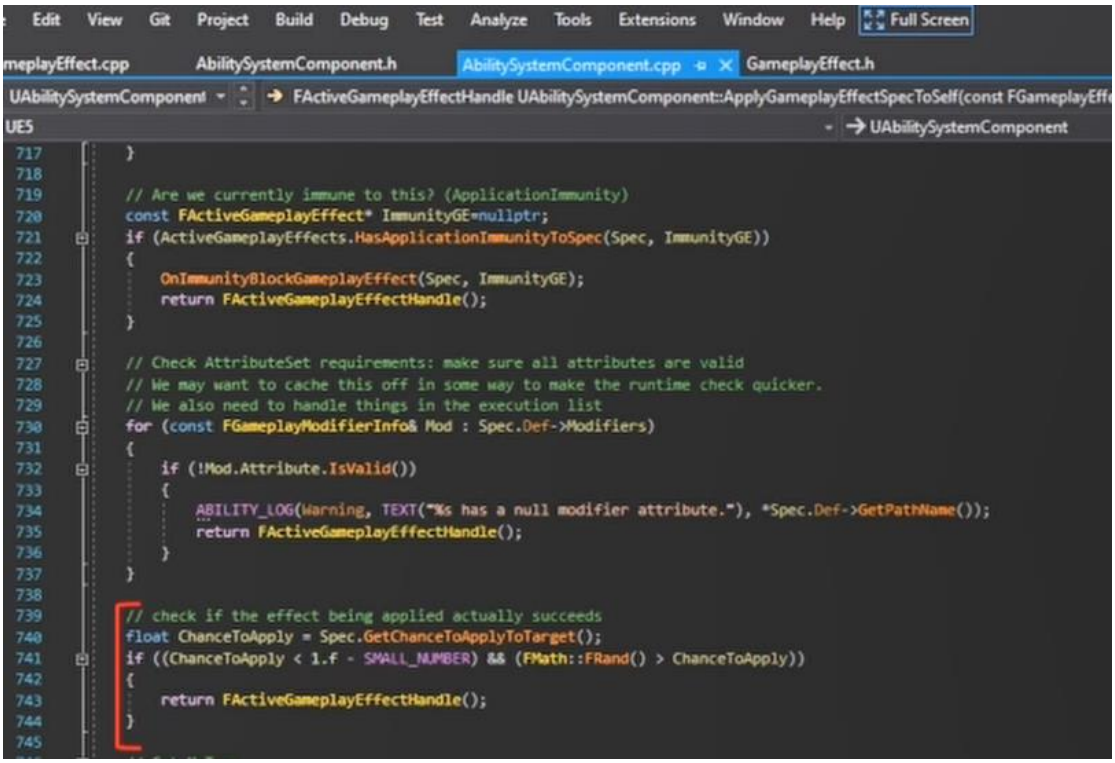
Inside of this function we are also checking if we are immune to this effect application (pic 1).

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
UE5
UAbilitySystemComponent
683
684 FActiveGameplayEffectHandle UAbilitySystemComponent::ApplyGameplayEffectSpecToSelf(const FGameplayEffectSpec &Spec, FPredictionKey PredictionKey)
685 {
686 #if WITH_SERVER_CODE
687 SCOPE_CYCLE_COUNTER(STAT_AbilitySystemComp_ApplyGameplayEffectSpecToSelf);
688 #endif
689
690 // Scope lock the container after the addition has taken place to prevent the new effect from potentially getting mangled during the remainder
691 // of the add operation
692 FScopedActiveGameplayEffectLock ScopeLock(ActiveGameplayEffects);
693
694 FScopeCurrentGameplayEffectBeingApplied ScopedGEApplication(&Spec, this);
695
696 const bool bIsNetAuthority = IsOwnerActorAuthoritative();
697
698 // Check Network Authority
699 if (!HasNetworkAuthorityToApplyGameplayEffect(PredictionKey))
700 {
701 return FActiveGameplayEffectHandle();
702 }
703
704 // Don't allow prediction of periodic effects
705 if(PredictionKey.IsValidKey() && Spec.GetPeriod() > 0.f)
706 {
707 if(IsOwnerActorAuthoritative())
708 {
709 // Server continue with invalid prediction key
710 PredictionKey = FPredictionKey();
711 }
712 else
713 {
714 // Client just return now
715 return FActiveGameplayEffectHandle();
716 }
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

Pic 2:

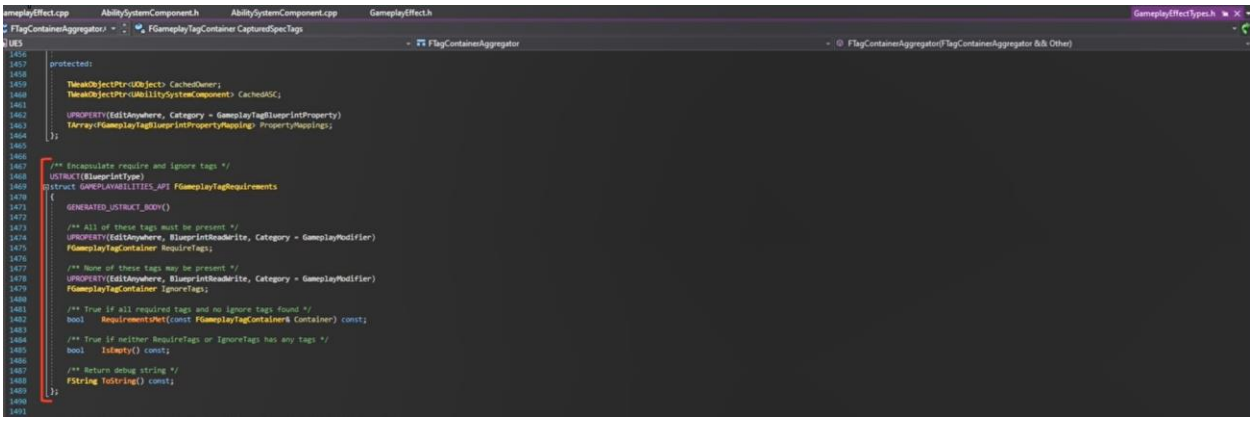

Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Full Screen
meplayEffect.cpp
AbilitySystemComponent.h
AbilitySystemComponent.cpp
GameplayEffect.h
UAbilitySystemComponent
FActiveGameplayEffectHandle UAbilitySystemComponent::ApplyGameplayEffectSpecToSelf(const FGameplayEffe
UE5
UAbilitySystemComponent
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.

```
{
    // 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
    static FGameplayTagContainer MyTags;
    MyTags.Reset();

    GetOwnedGameplayTags(MyTags);

    if (Spec.Def->ApplicationTagRequirements.RequirementsMet(MyTags) == false)
    {
        return FActiveGameplayEffectHandle();
    }

    if (!Spec.Def->RemovalTagRequirements.IsEmpty() && Spec.Def->RemovalTagRequirements.RequirementsMet(MyTags) == true)
    {
        return FActiveGameplayEffectHandle();
    }
}
```

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

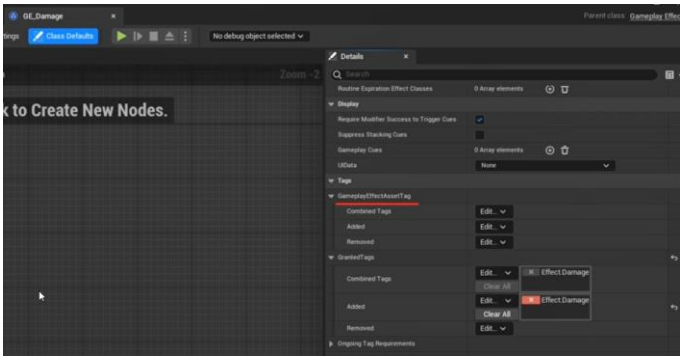
If at least one tag from ignored tack is presented, we will not apply the effect.



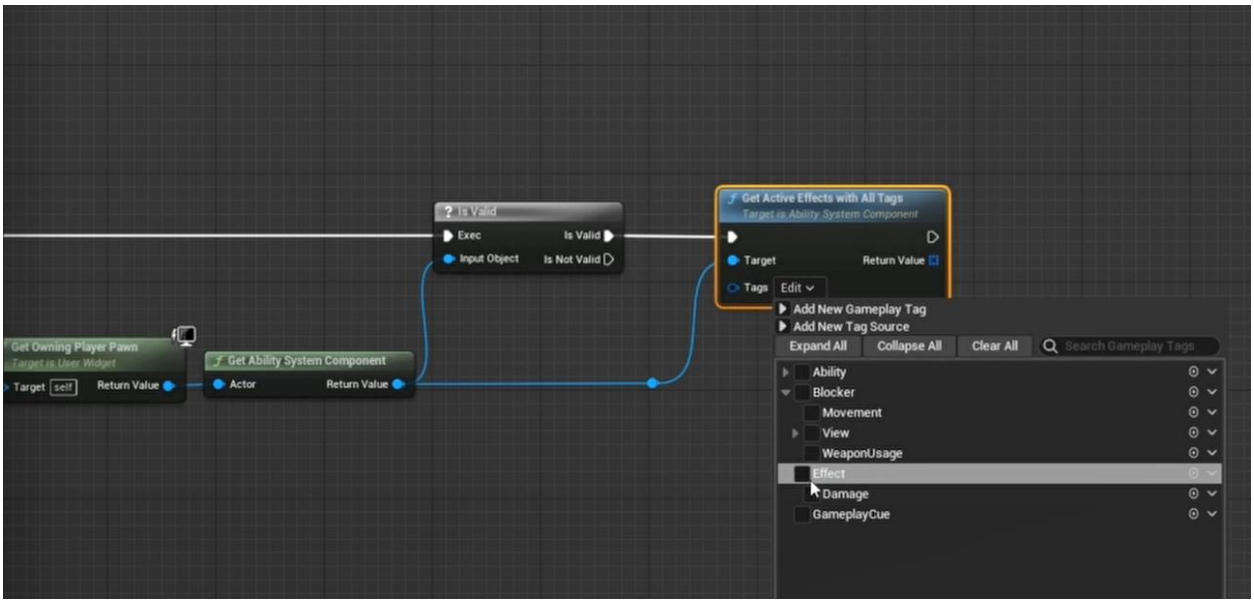
```
protected:

    TWeakObjectPtr<UObject> CachedOwner;
    TWeakObjectPtr<UAbilitySystemComponent> CachedASC;

    UPROPERTY(EditAnywhere, Category = GameplayTagBlueprintProperty)
    TArray<FGameplayTagBlueprintPropertyMapping> PropertyMappings;
};

/** Encapsulate require and ignore tags */
USTRUCT(BlueprintType)
struct GAMEPLAYABILITIES_API FGameplayTagRequirements
{
    GENERATED_USTRUCT_BODY()

    /** All of these tags must be present */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayModifier)
    FGameplayTagContainer RequireTags;

    /** None of these tags may be present */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayModifier)
    FGameplayTagContainer IgnoreTags;

    /** True if all required tags and no ignore tags found */
    bool    RequirementsMet(const FGameplayTagContainer& Container) const;

    /** True if neither RequireTags or IgnoreTags has any tags */
    bool    IsEmpty() const;

    /** Return debug string */
    FString ToString() const;
};
```

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.

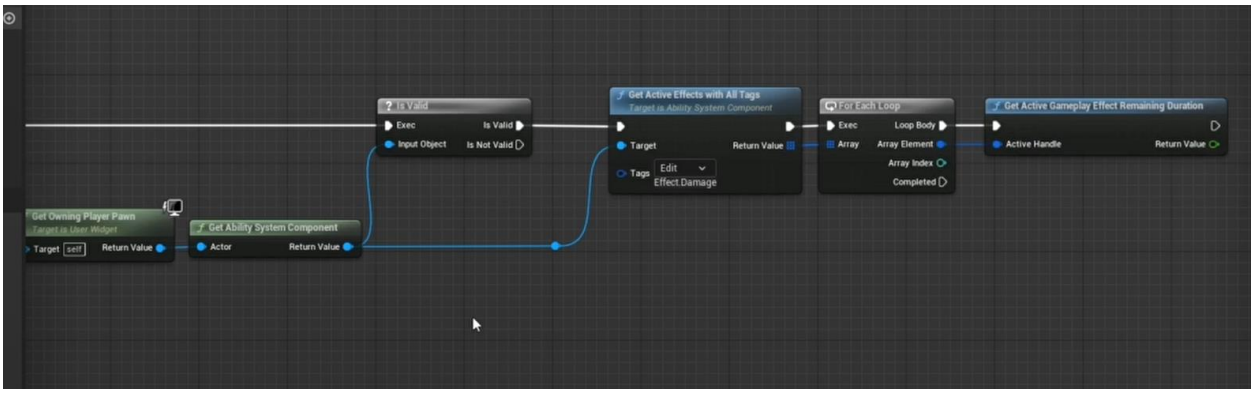
Tags over here will be associated with the Effect itself.



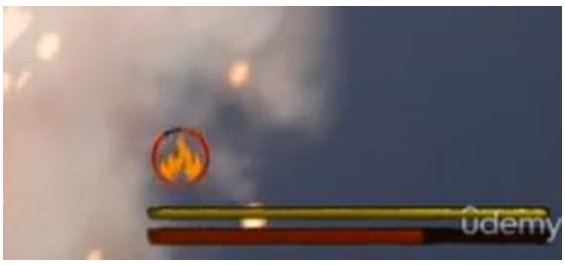
Using them we can explicitly track what effects are applied to our character.



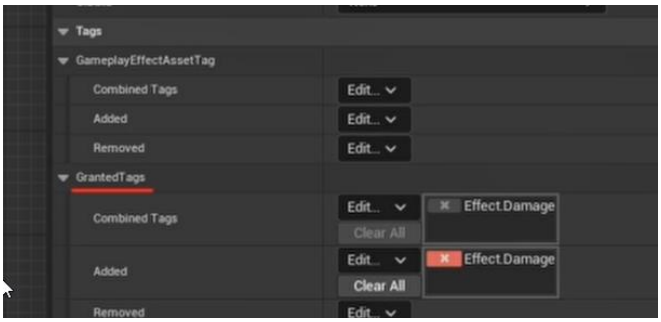
And even get some runtime parameters from them like the remaining time.



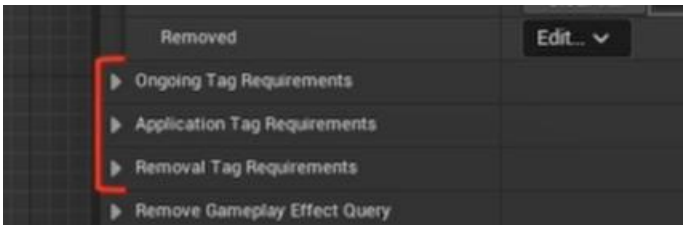
We can use this for example to show the remaining time of some Effects like burning effect or poison.



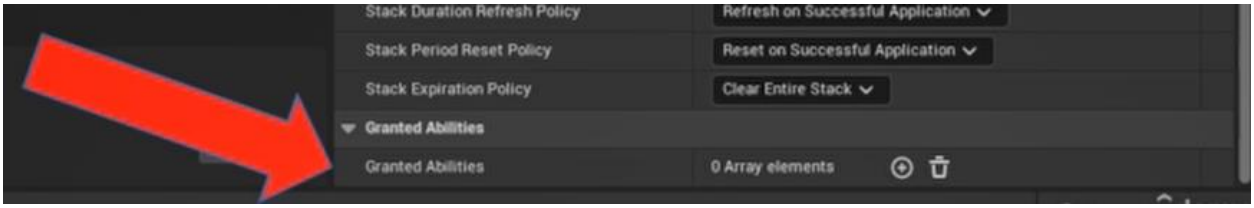
GrantedTags will be applied to the owner of the effect.



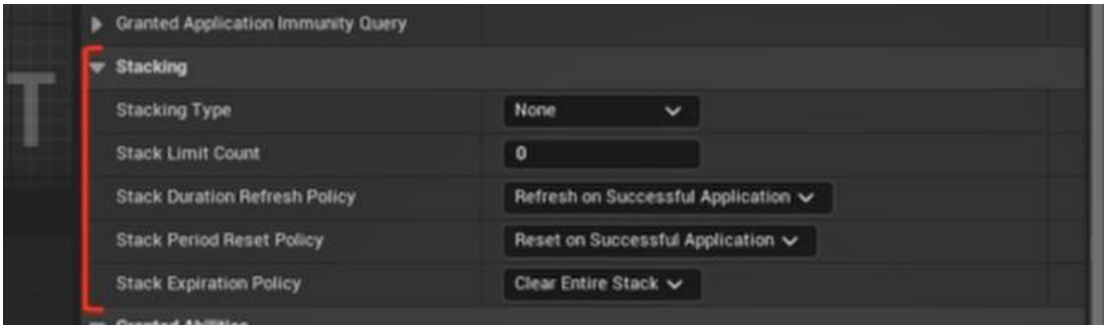
Next is common set of requirements.



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.

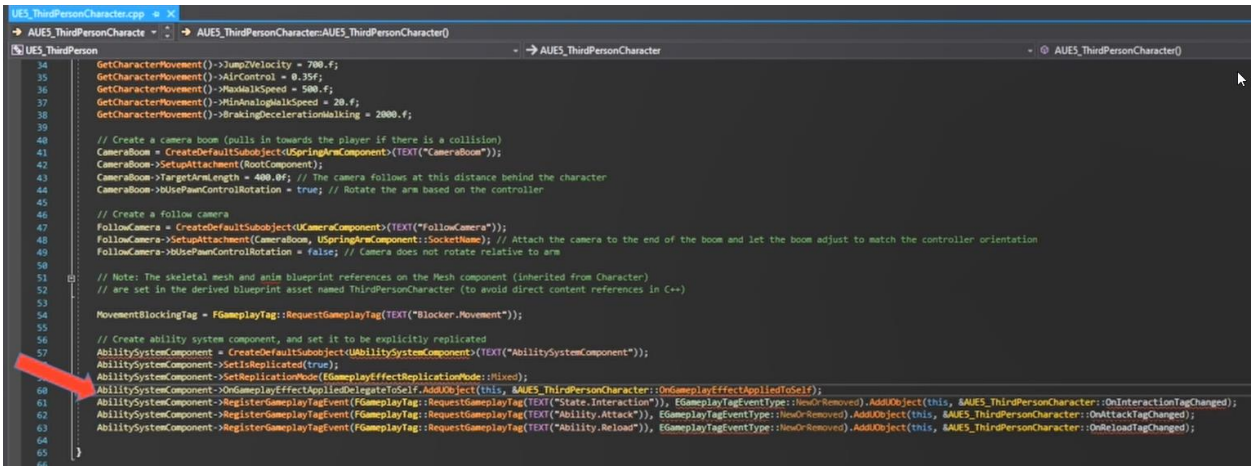


As we can apply one effect several times, we also have some rules about staking.

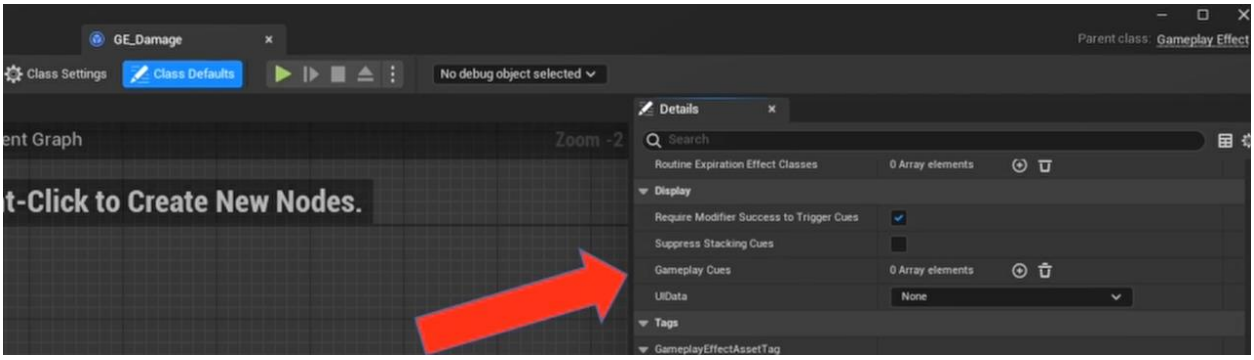


Because the Effects can represent attribute change, they can be used as a cost for running and ability. For example, you can have a cost effect that will reduce Manna for jumping.

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



Effects can be used also to run GameplayCues.



---

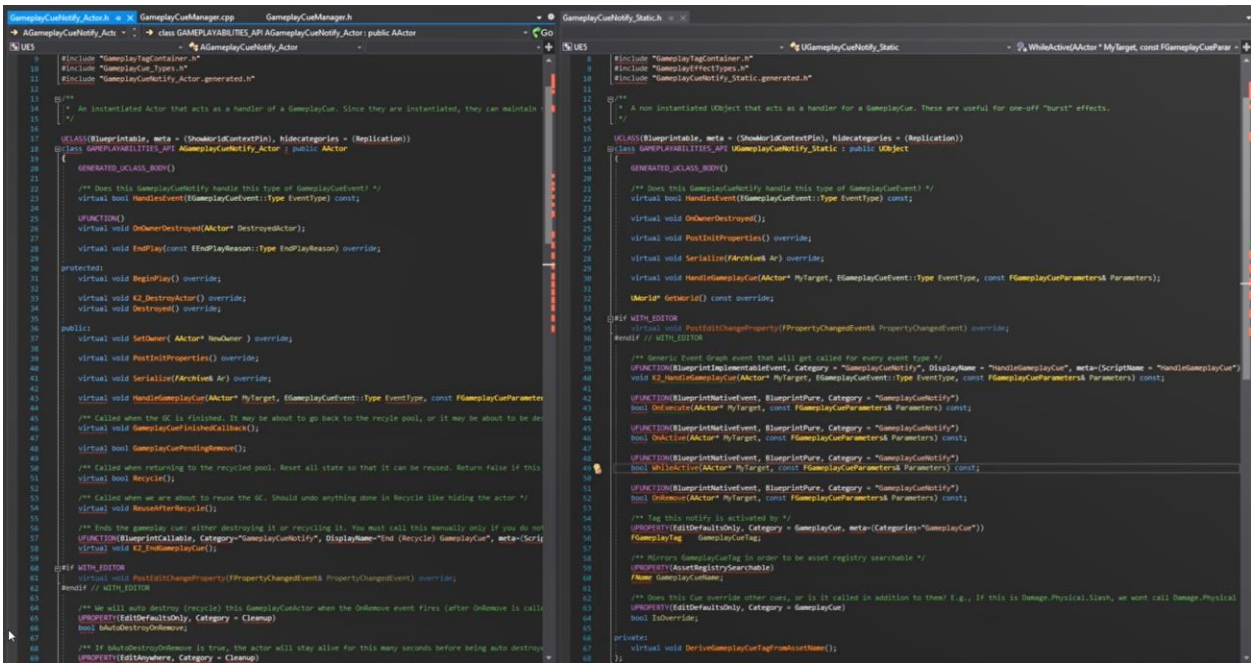
9. Gameplay Cues

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

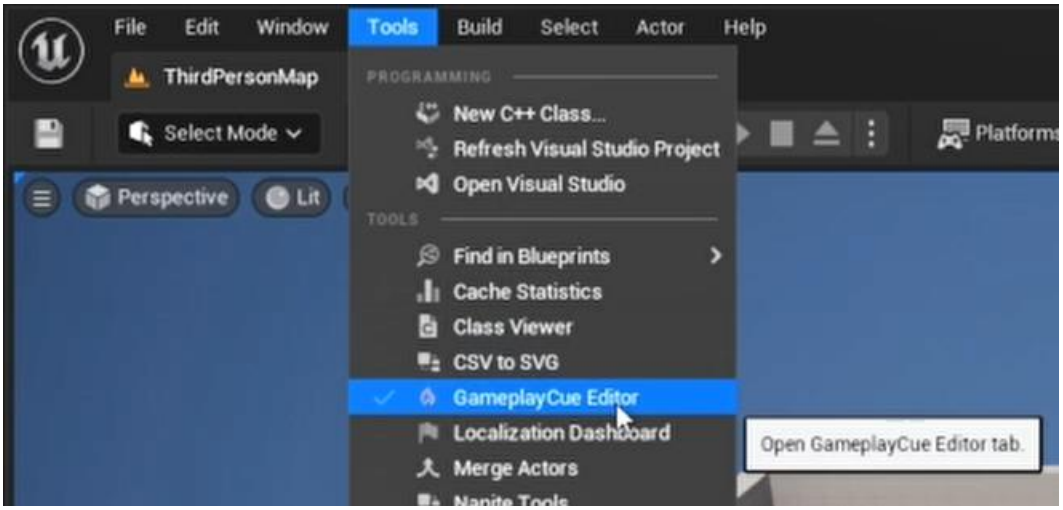
GameplayCues Execution will be replicated out-of-the-box for us in this case.

GameplayCues can be a static or actor base.

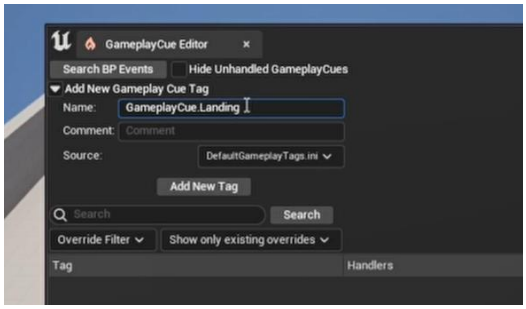
- Static Cues will have **only one object in memory** and will not keep or modify a per object data during the execution because it will **operate on class default object**.
- An Actor based will be spawned just as an actor and **will inherit all its capabilities**.

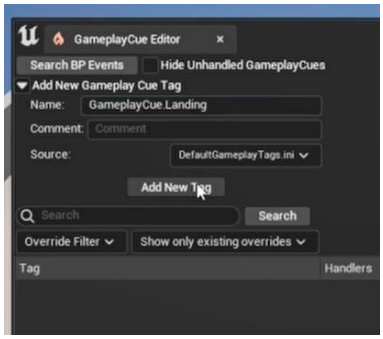


GameplayCues have their own section inside Tag Editor. And we should use this specific window to add them.

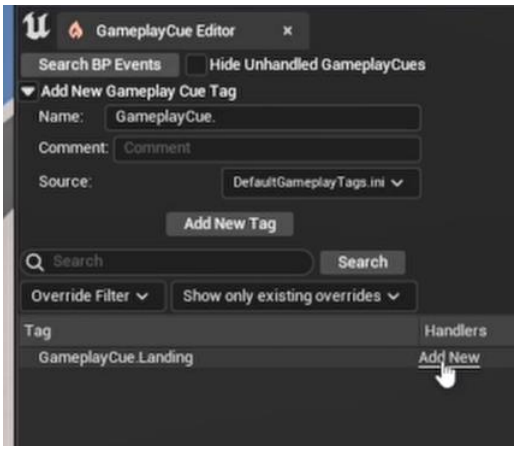


Spawn VFX for character superhero landing and can be made with a Static Cue because we just need to spawn the particle system once and we don't care about tracking its instance afterwards.

1. 

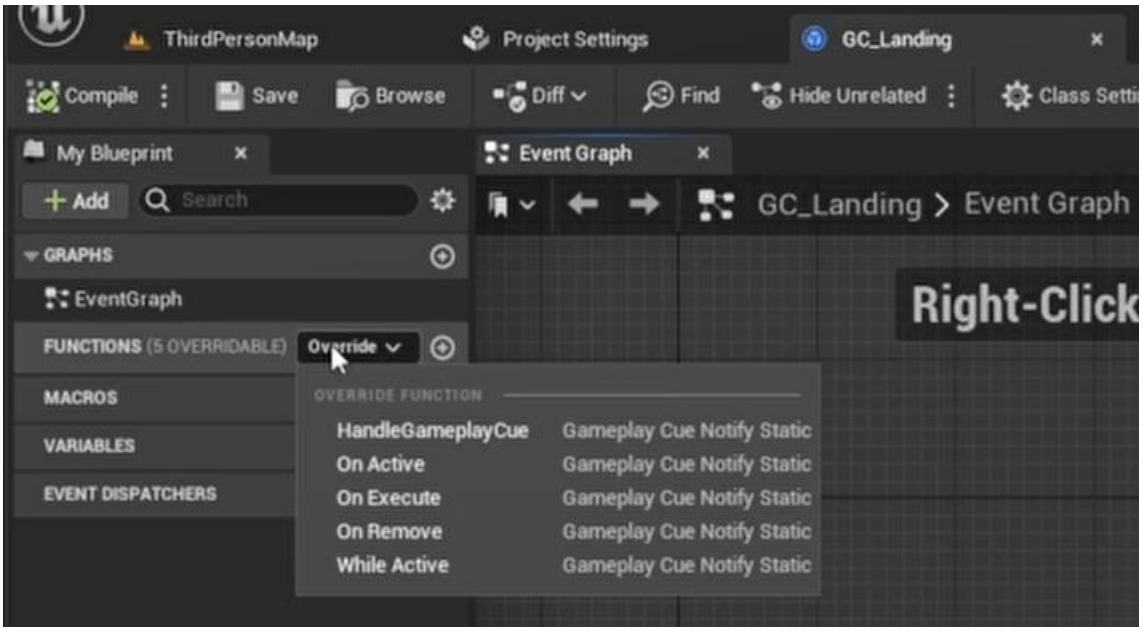
2. 

3.

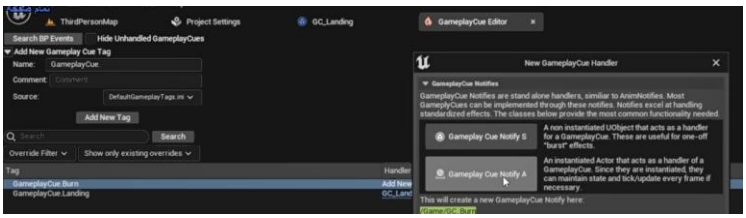


4.

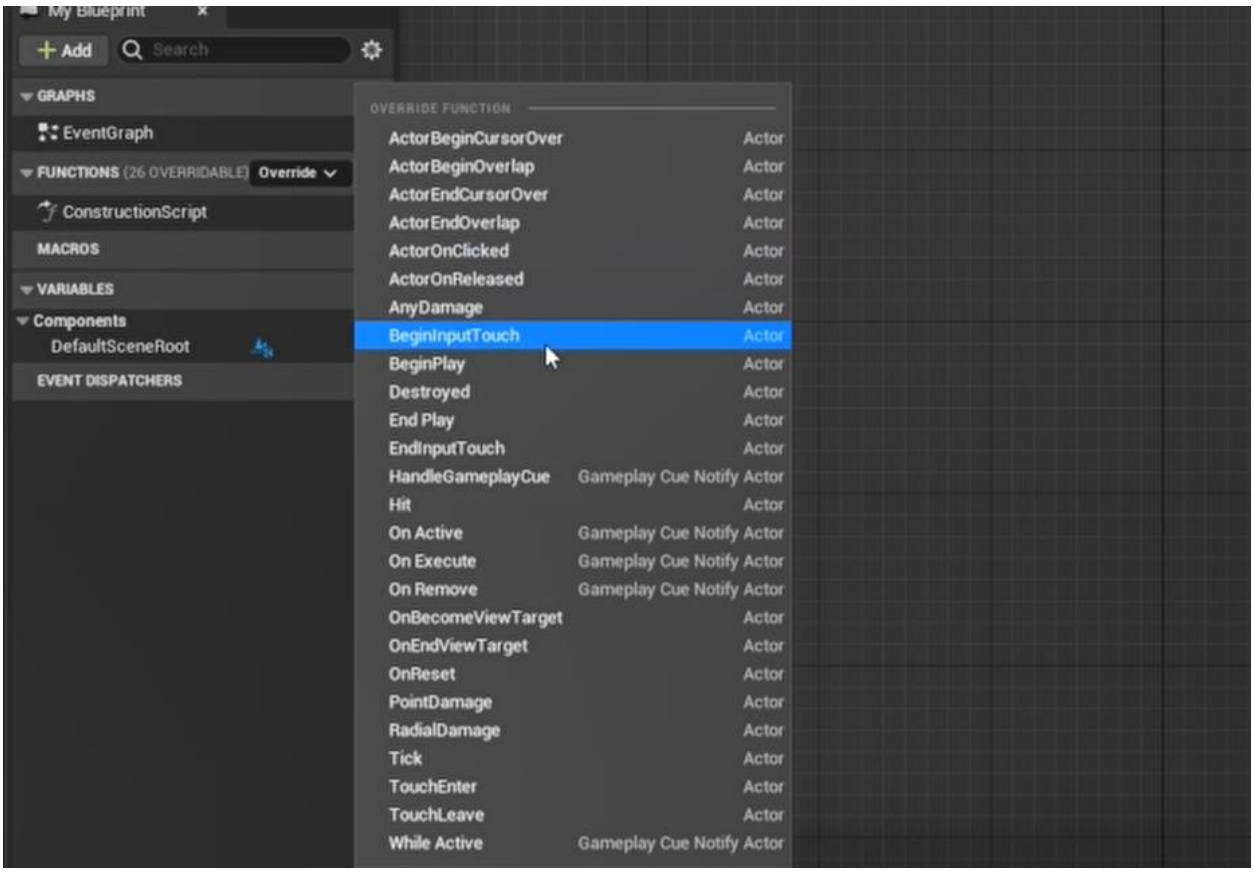
Here we can override some functions that will be triggered during the Cue execution.



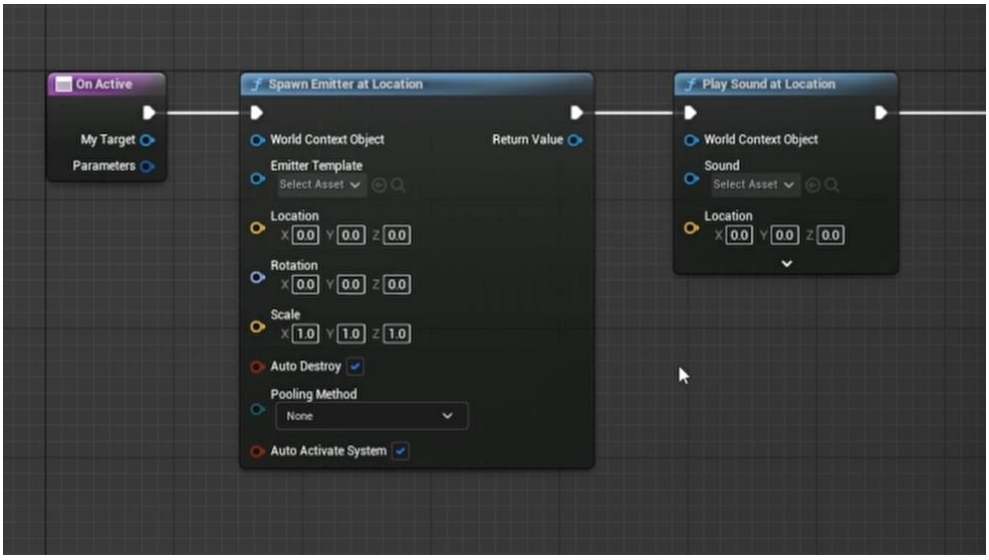
Applying a Burning effect and removing it directly on end by the reference to the particle system can be done via a non-static actor based GamplayCue. because you might want to **save the reference to the particle system** for every character that burns.



GamplayCue'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.

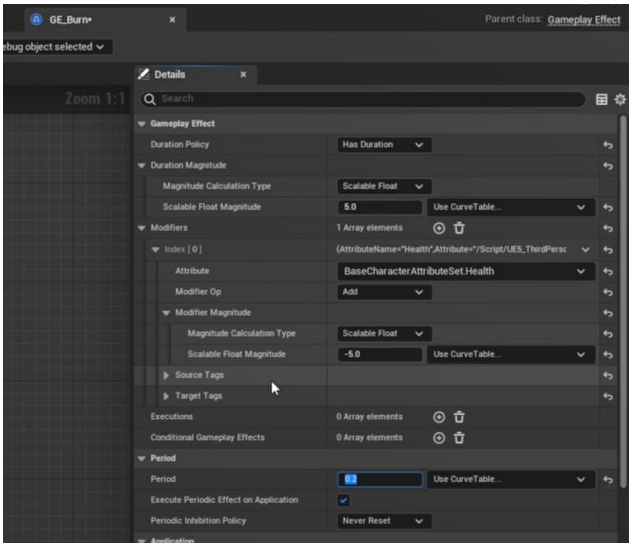


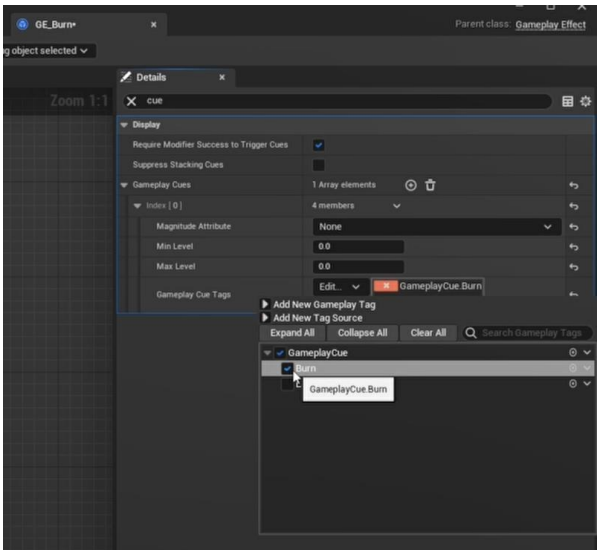
As you can see right here we're trying to spawn some particle system and play sound on the Cue will on the start executing.



Cues can **be triggered implicitly by a GameplayEffect**. In this case, the events on the Cue will be **triggered accordingly to the effect lifetime**.

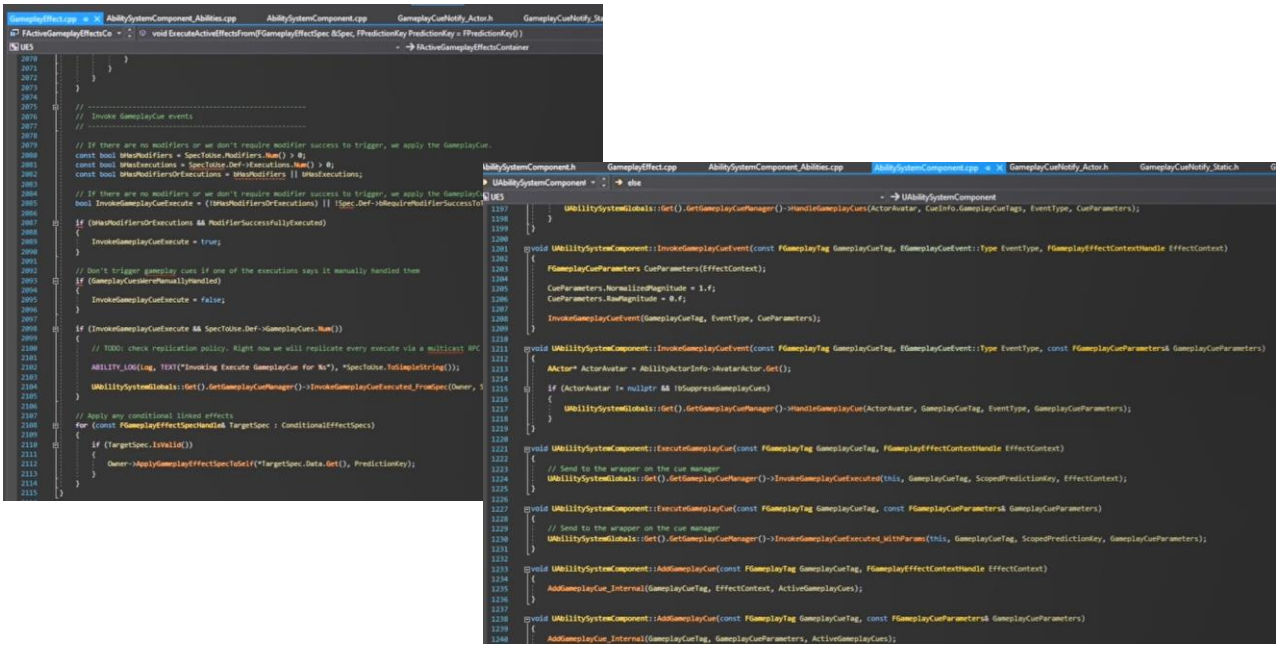
For example, in case of a burning effect, you might want to have an effect with a duration that will reduce health attribute with some low period.



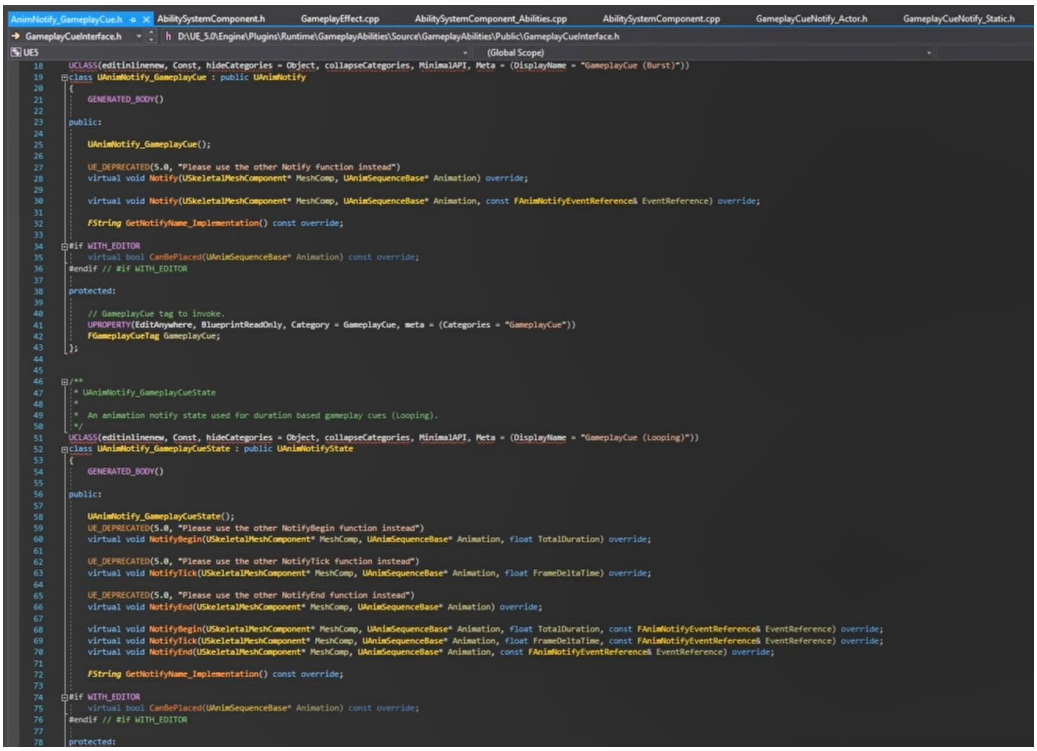


To display Cue for play a sound and the VFX, you will have to add a GameplayCue record was the correct tag.

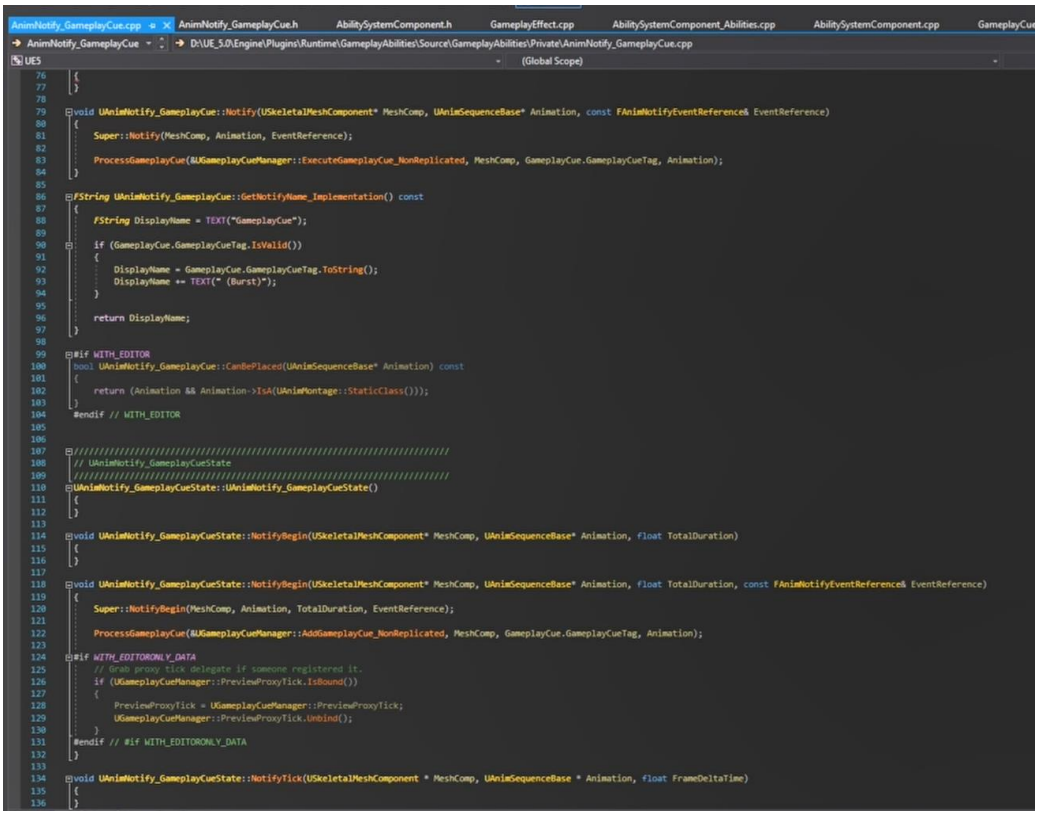
The GameplayEffect will play this Cue when being executed or we can trigger Cues directly.



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

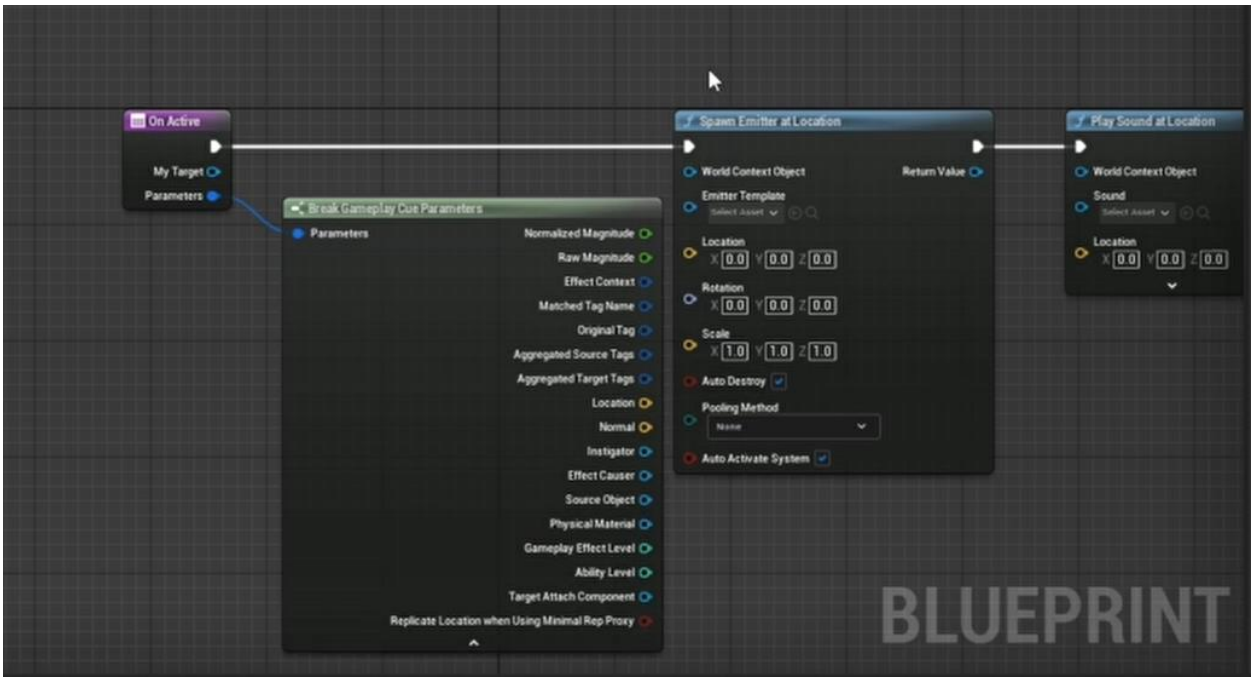


```
AnimNotify_GameplayCue.h | AbilitySystemComponent.h | GameplayEffect.cpp | AbilitySystemComponent_Abilities.cpp | AbilitySystemComponent.cpp | GameplayCueNotify_Actor.h | GameplayCueNotify_Static.h
GameplayCueInterface.h | D:\UE_5.0\Engine\Plugins\Runtime\GameplayAbilities\Source\GameplayAbilities\Public\GameplayCueInterface.h
UE5 | (Global Scope)
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   *
49   *  An animation notify state used for duration based gameplay cues (Looping).
50   */
51  UCLASS(editinlinenew, Const, hideCategories = Object, collapseCategories, MinimalAPI, Meta = (DisplayName = "GameplayCue (Looping)"))
52  class UAnimNotify_GameplayCueState : public UAnimNotifyState
53  {
54      GENERATED_BODY()
55
56  public:
57
58      UAnimNotify_GameplayCueState();
59      UE_DEPRECATED(5.0, "Please use the other NotifyBegin function instead")
60      virtual void NotifyBegin(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float TotalDuration) override;
61
62      UE_DEPRECATED(5.0, "Please use the other NotifyTick function instead")
63      virtual void NotifyTick(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float FrameDeltaTime) override;
64
65      UE_DEPRECATED(5.0, "Please use the other NotifyEnd function instead")
66      virtual void NotifyEnd(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation) override;
67
68      virtual void NotifyBegin(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float TotalDuration, const FAnimNotifyEventReference& EventReference) override;
69      virtual void NotifyTick(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, float FrameDeltaTime, const FAnimNotifyEventReference& EventReference) override;
70      virtual void NotifyEnd(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, const FAnimNotifyEventReference& EventReference) override;
71
72      FString GetNotifyName_Implementation() const override;
73
74  #if WITH_EDITOR
75      virtual bool CanBePlaced(UAnimSequenceBase* Animation) const override;
76  #endif // #if WITH_EDITOR
77
78  protected:
```



```
AnimNotify_GameplayCue.cpp | AnimNotify_GameplayCue.h | AbilitySystemComponent.h | GameplayEffect.cpp | AbilitySystemComponent_Abilities.cpp | AbilitySystemComponent.cpp | GameplayCue
AnimNotify_GameplayCue | D:\UE_5.0\Engine\Plugins\Runtime\GameplayAbilities\Source\GameplayAbilities\Private\AnimNotify_GameplayCue.cpp
UE5 | (Global Scope)
76  {
77  }
78
79  void UAnimNotify_GameplayCue::Notify(USkeletalMeshComponent* MeshComp, UAnimSequenceBase* Animation, const FAnimNotifyEventReference& EventReference)
80  {
81      Super::Notify(MeshComp, Animation, EventReference);
82
83      ProcessGameplayCue(&UGameplayCueManager::ExecuteGameplayCue_NonReplicated, MeshComp, GameplayCue.GameplayCueTag, Animation);
84  }
85
86  FString UAnimNotify_GameplayCue::GetNotifyName_Implementation() const
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(&UGameplayCueManager::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.

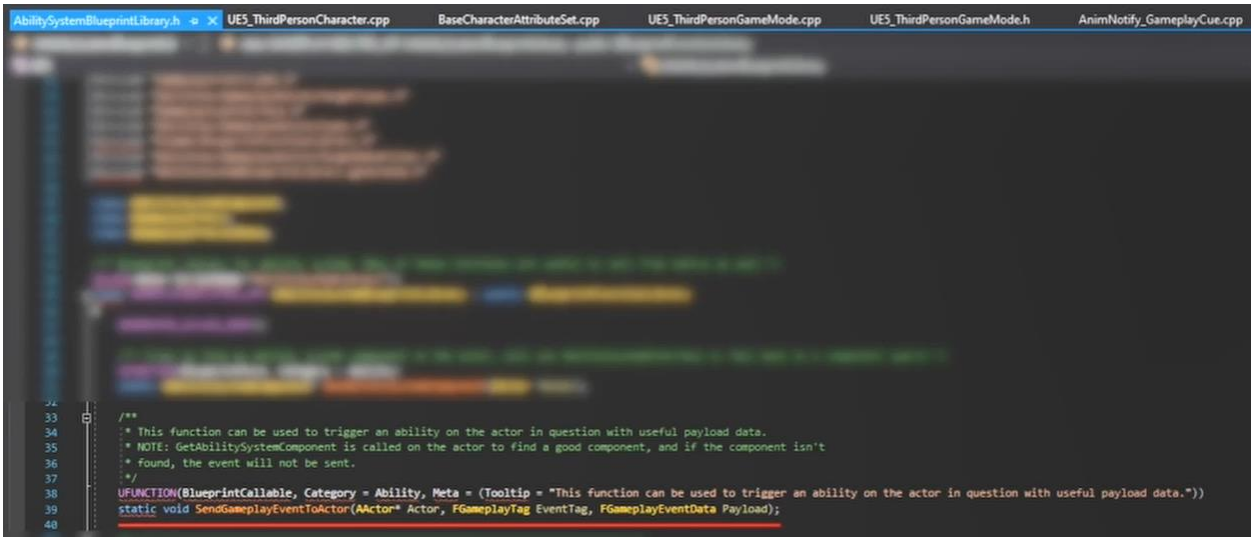


---

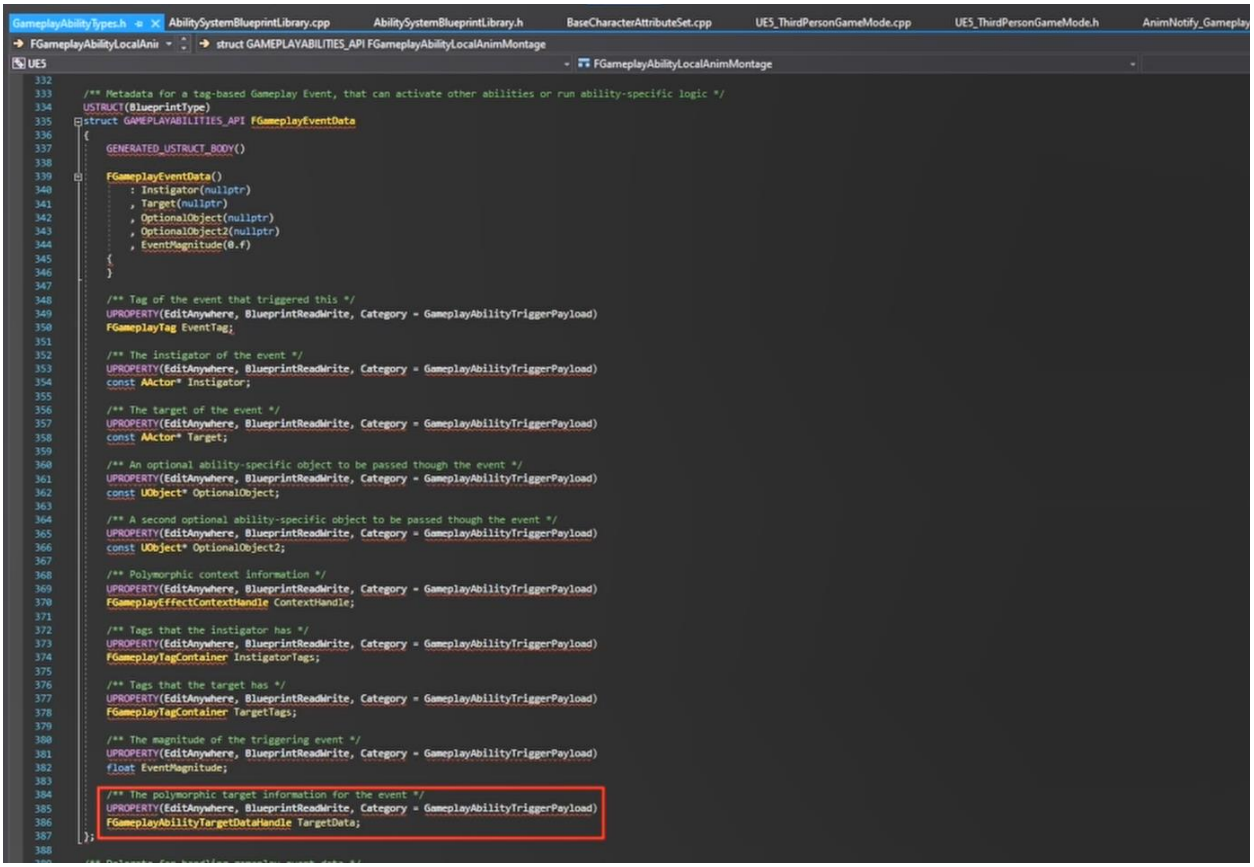
6. Gameplay Events

GameplayEvents is the way you can communicate with AbilitySystem, technically GameplayEvent is just a **combination**

of a Tag and the GameplayEventData payload.

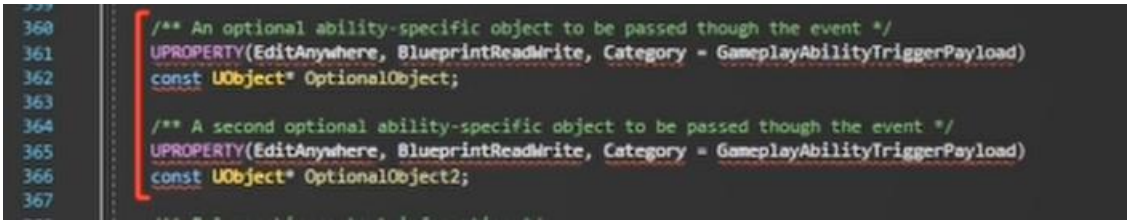


This Struct GameplayEventData has many useful fields. You can use its field called TargetData to **pass the game playability target data derived class with your own custom data**.



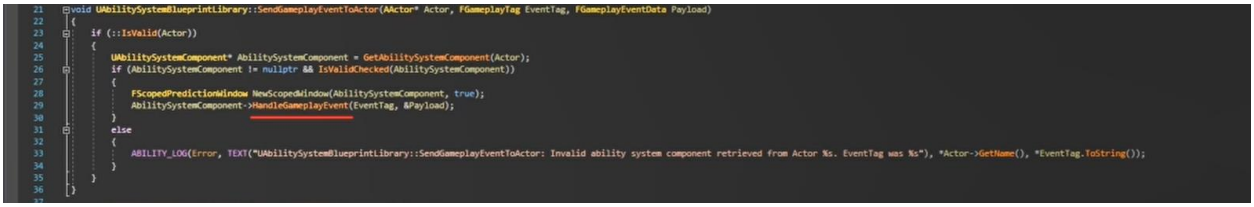
```
/** Metadata for a tag-based Gameplay Event, that can activate other abilities or run ability-specific logic */
USTRUCT(BlueprintType)
struct GAMEPLAYABILITIES_API FGameplayEventData
{
    GENERATED_USTRUCT_BODY()

    FGameplayEventData()
        : Instigator(nullptr)
        , Target(nullptr)
        , OptionalObject(nullptr)
        , OptionalObject2(nullptr)
        , EventMagnitude(0.f)
    {
    }

    /** Tag of the event that triggered this */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
    FGameplayTag EventTag;

    /** The instigator of the event */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
    const AActor* Instigator;

    /** The target of the event */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
    const AActor* Target;

    /** An optional ability-specific object to be passed though the event */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
    const UObject* OptionalObject;

    /** A second optional ability-specific object to be passed though the event */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
    const UObject* OptionalObject2;

    /** Polymorphic context information */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
    FGameplayEffectContextHandle ContextHandle;

    /** Tags that the instigator has */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
    FGameplayTagContainer InstigatorTags;

    /** Tags that the target has */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
    FGameplayTagContainer TargetTags;

    /** The magnitude of the triggering event */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
    float EventMagnitude;

    /** The polymorphic target information for the event */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
    FGameplayAbilityTargetDataHandle TargetData;
};
```

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)



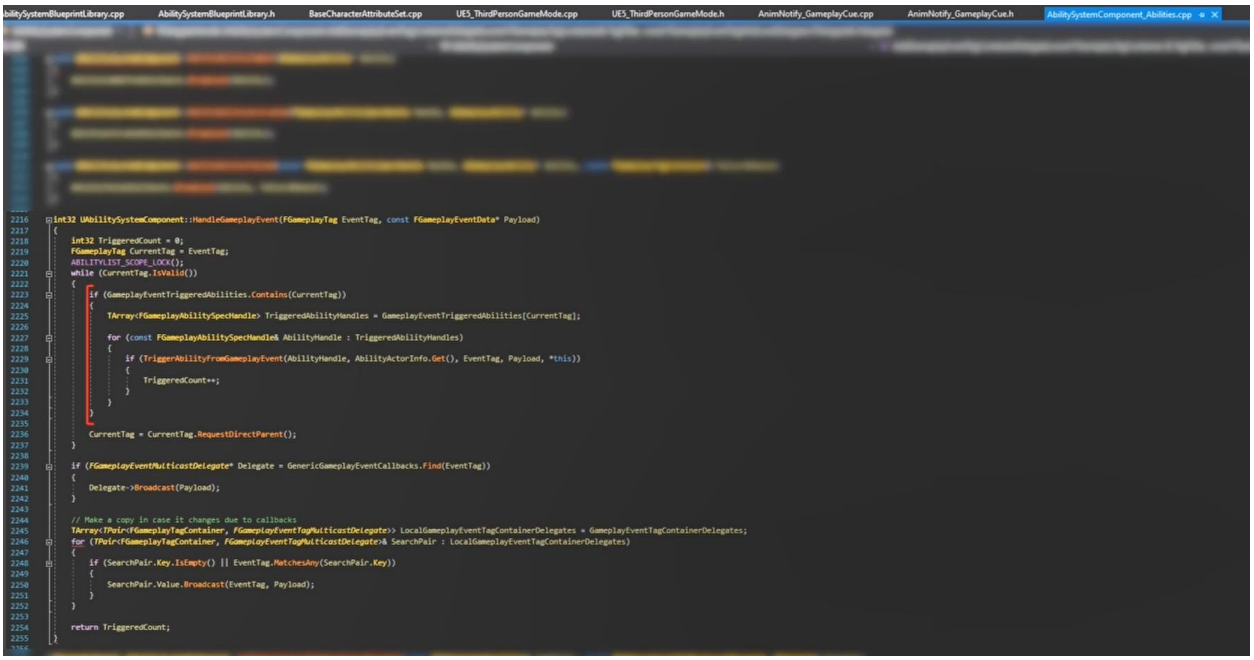
```
/** An optional ability-specific object to be passed though the event */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
const UObject* OptionalObject;

/** A second optional ability-specific object to be passed though the event */
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = GameplayAbilityTriggerPayload)
const UObject* OptionalObject2;
```

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

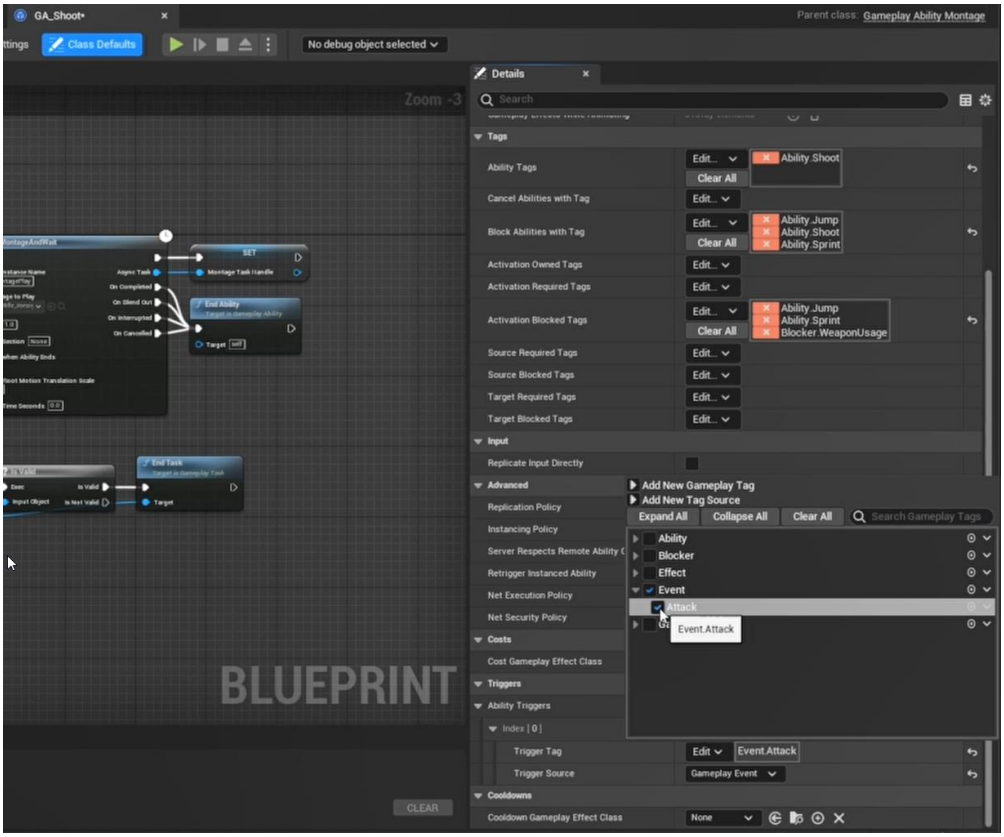


```
void UAbilitySystemBlueprintLibrary::SendGameplayEventToActor(AActor* Actor, FGameplayTag EventTag, FGameplayEventData Payload)
{
    if (::IsValid(Actor))
    {
        UAbilitySystemComponent* AbilitySystemComponent = GetAbilitySystemComponent(Actor);
        if (AbilitySystemComponent != nullptr && IsValidChecked(AbilitySystemComponent))
        {
            FScopedPredictionWindow NewScopedWindow(AbilitySystemComponent, true);
            AbilitySystemComponent->HandleGameplayEvent(EventTag, &Payload);
        }
        else
        {
            ABILITY_LOG(Error, TEXT("UAbilitySystemBlueprintLibrary::SendGameplayEventToActor: Invalid ability system component retrieved from Actor %s. EventTag was %s"), *Actor->GetName(), *EventTag.ToString());
        }
    }
}
```

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

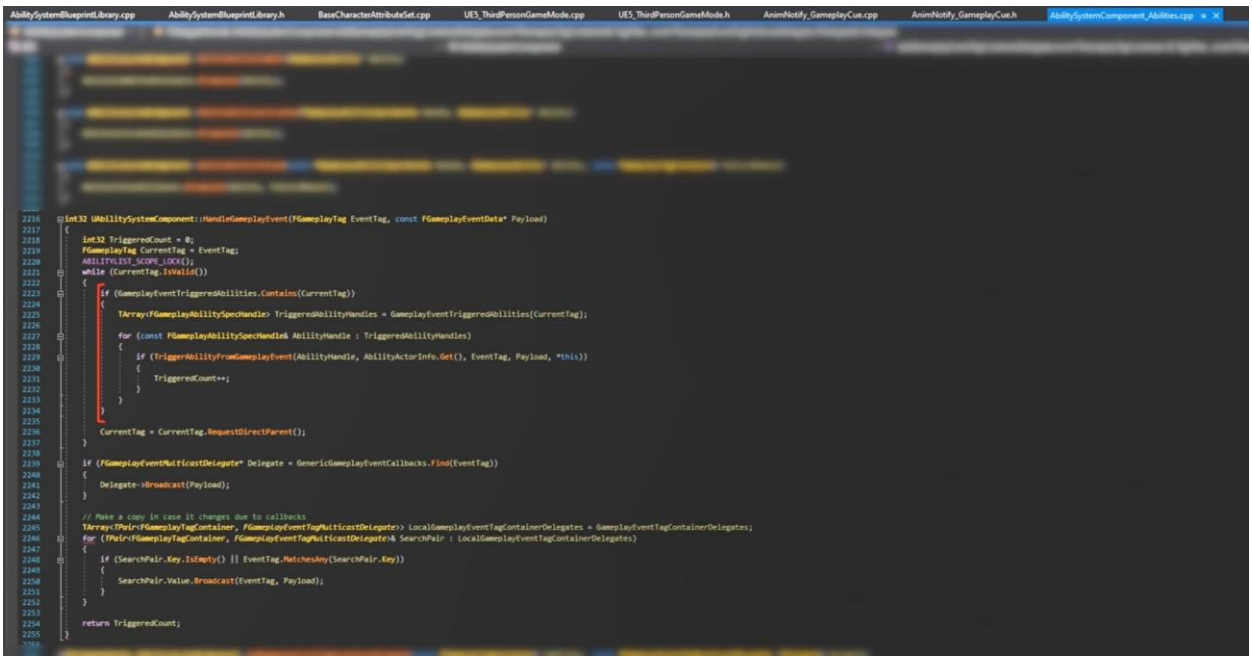


You can set the Ability to react to a specific event.



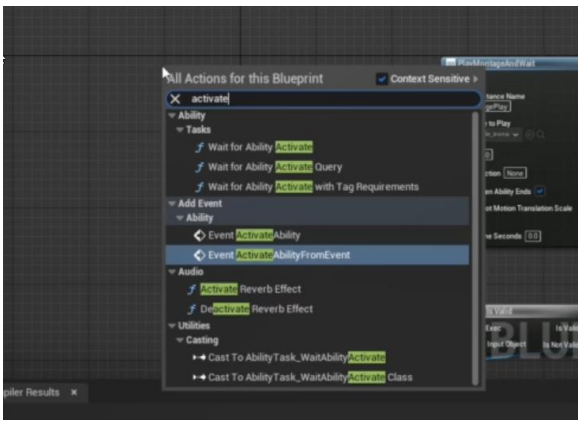
And when receiving an event, AbilitySystemComponent will iterate over the Abilities that should be triggered by the sent event and will try to activate them.

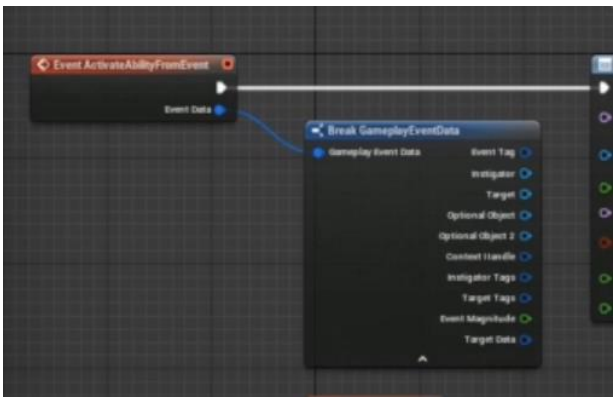
With this approach we can send additional data through the payload.



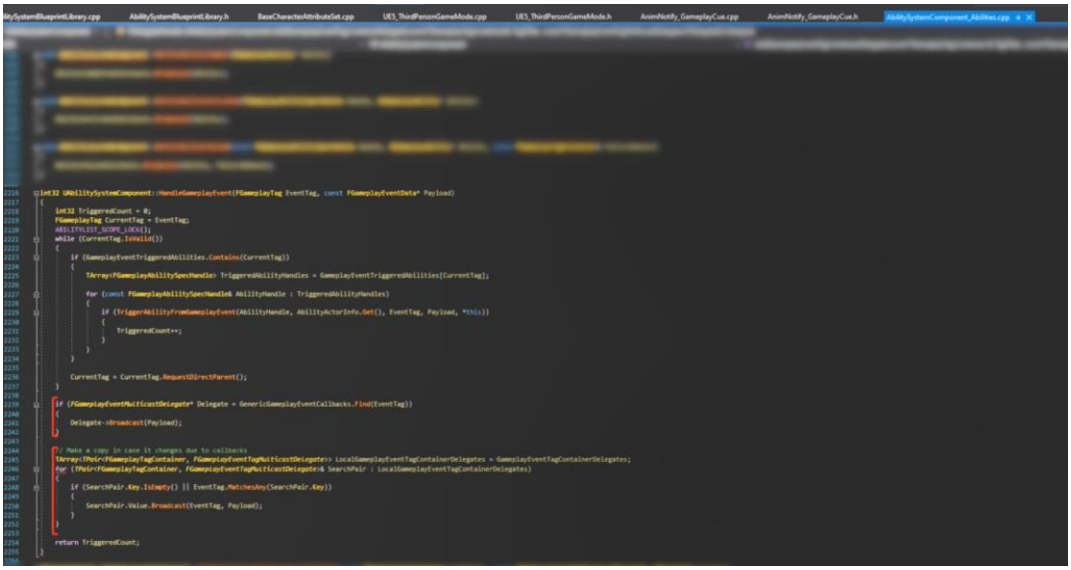
```
int32 UAbilitySystemComponent::HandleGameplayEvent(FGameplayTag EventTag, const FGameplayEventData* Payload)
{
    int32 TriggeredCount = 0;
    FGameplayTag CurrentTag = EventTag;
    ABILITYLIST_SCOPE_LOCK();
    while (CurrentTag.IsValid())
    {
        if (GameplayEventTriggeredAbilities.Contains(CurrentTag))
        {
            TArray<FGameplayAbilitySpecHandle> TriggeredAbilityHandles = GameplayEventTriggeredAbilities[CurrentTag];

            for (const FGameplayAbilitySpecHandle& AbilityHandle : TriggeredAbilityHandles)
            {
                if (TriggerAbilityFromGameplayEvent(AbilityHandle, AbilityActorInfo.Get(), EventTag, Payload, *this))
                {
                    TriggeredCount++;
                }
            }
        }
        CurrentTag = CurrentTag.RequestDirectParent();
    }

    if (FGameplayEventMulticastDelegate* Delegate = GenericGameplayEventCallbacks.Find(EventTag))
    {
        Delegate->Broadcast(Payload);
    }

    // Make a copy in case it changes due to callbacks
    TArray<TPair<FGameplayTagContainer, FGameplayEventTagMulticastDelegate>> LocalGameplayEventTagContainerDelegates = GameplayEventTagContainerDelegates;
    for (TPair<FGameplayTagContainer, FGameplayEventTagMulticastDelegate>& SearchPair : LocalGameplayEventTagContainerDelegates)
    {
        if (SearchPair.Key.IsEmpty() || EventTag.MatchesAny(SearchPair.Key))
        {
            SearchPair.Value.Broadcast(EventTag, Payload);
        }
    }

    return TriggeredCount;
}
```

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.

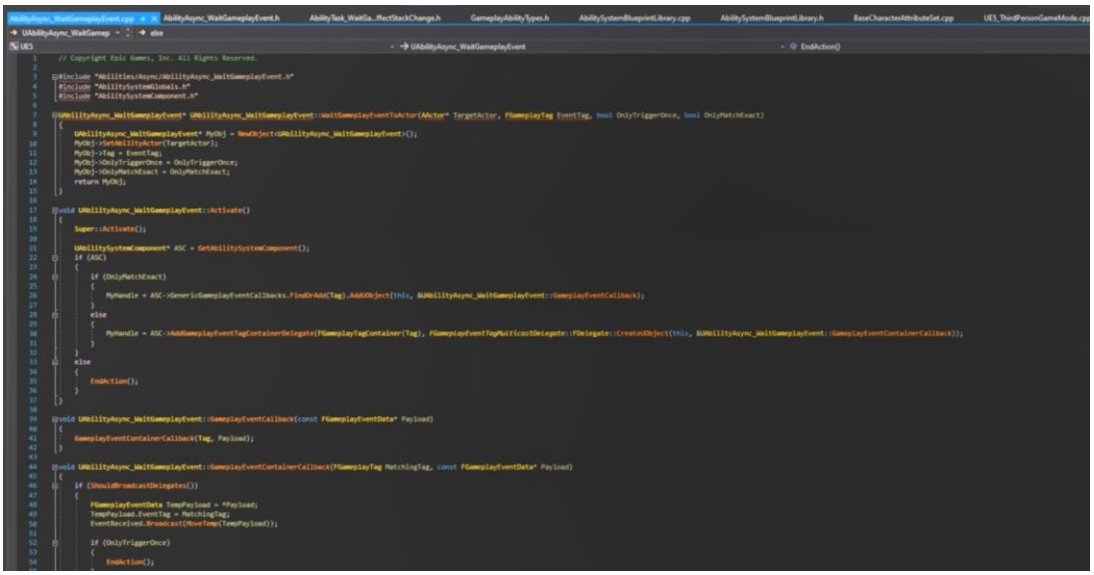




Here we will trigger delegates to notify everyone who's subscribed to any event received or some specific one.



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.



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