How to start with the GAS template project ?

If you found this document without any additional information. You should know that there is a ***template project on my github with a guide*** which explains my view of the GAS.

I highly suggest that you ***read the "GAS Beginner Guide" prior and then alongside this "Step by Step Guide"***. These docs contain detailed information that will help you on your way to create games with the GAS. ***They are meant to be used in coherence to get the full learning experience***.

1.) GAS Beginner Guide
2.) GAS Components
3.) Step by Step Guide

You should definitely check out the attribution of sources, read through demo project source code and watch the Unreal Engine talks to dive deeper into the GAS.

Good luck.

TheBitFossil

Index

GAS Components

Component		Description
Ability	● brain of the system ● self contained	An Ability will be only executed in **one frame**.
Tasks	● asynchronous ● listen to Anim Notifies ● plays montages	Tasks help make changes, or **listen** for events, while the current effect is **executing**.
Gameplay Tags	● Build into UE ● Backbone of the ASC ● Hierarchical ordering	Essential building block when using GAS. Defines **conditions** for how the Abilities will be applied.
Gameplay Effects	● can apply gameplay tags ● changes attributes ● Duration Policy (instant, duration, infinite)	Use it to **change** the **Attributes** of the actor.
Gameplay Attributes	● Data ● States	The **values** of specific data and states: health, mana, speed, etc
Attribute Set	● Attributes ● One or Multiple	Putting **Attributes together** makes an AttributeSet. Those values affect the gameplay directly.
Gameplay Cues	● Visual Effects ● Not essential	When an **effect** is running, you can add a gameplay cue, which will take care of its **visual representation**.

How to set up a basic GAS template project?

Chapter 1 : Initializing the Project

□ AssetManager.h

We need this entry to make a call to the *Ability System Globals*.
Find the file *DefaultEngine.ini* and add the Entry:

```
AssetManagerClassName=/Script/ProjectNameHere.GAssetManager
```

Use this Category:

```
[/Script/Engine.Engine]
```

□ Private Module Dependencies

To make use of the GAS we add these inside our *ProjectName.Build.cs*

```
PrivateDependencyModuleNames.AddRange(new string[] {
"GameplayAbilities", "GameplayTags", "GameplayTasks" });
```

□ Macros for the AttributeSetBase.h

These are *predefined* and will help to *Get, Set* and *Init* our Values.

```
// Uses macros from AttributeSet.h

#define ATTRIBUTE_ACCESSORS(ClassName, PropertyName) \

   GAMEPLAYATTRIBUTE_PROPERTY_GETTER(ClassName, PropertyName) \

   GAMEPLAYATTRIBUTE_VALUE_GETTER(PropertyName) \

   GAMEPLAYATTRIBUTE_VALUE_SETTER(PropertyName) \

   GAMEPLAYATTRIBUTE_VALUE_INITTER(PropertyName)
```

Starting with the template project

Chapter 2 : Creating the GAS building blocks

□ Adding Attributes and Replication

Template for all future Attributes. Change the ***values*** you need to fit your projects needs ***AttributeSetBase.h***.

```
UPROPERTY(BlueprintReadOnly, Category = "Health", ReplicatedUsing
= OnRep_Health)
FGameplayAttributeData Health;
ATTRIBUTE_ACCESSORS(UCharacterAttributeSetBase, Health)
```

Add the ***callback*** function, so we can act when the value is ***replicated***.

```
protected:

UFUNCTION()
virtual void OnRep_Health(const FGameplayAttributeData& OldHealth);
```

□ Creating the Character Ability System Component

Inside the UE create a ***new C++ Class*** inheriting from ***AbilitySystemComponent***.
At least put these two bools inside:

```
bool bCharacterAbilitiesGiven = false;
bool bStartupEffectsApplied = false;
```

□ Creating a blank GameplayAbilities class

This will be ***used*** inside the Engine ***to create new Abilities*** later on, when we are finished with the Project Setup.

```
class GAMEPLAYSYSTEM_API UGGameplayAbility : public UGameplayAbility
```

Add those two fields, so we can ***assign*** the correct ***inputs*** and for our ***passive Abilities***.

```
// ID which is tied to the corresponding Input
UPROPERTY(BlueprintReadOnly, EditAnywhere ,Category ="GAS|Abilities")
GameplayAbilityID AbilityInputID = GameplayAbilityID::None;

// Not tied to any slot. Passive abilities are not activated via input
UPROPERTY(BlueprintReadOnly, EditAnywhere, Category="GAS|Abilities")
GameplayAbilityID AbilityID = GameplayAbilityID::None;
```

□ Where and how to store the Input for our Abilities

Abilities can either be ***passive or active.*** To know which input is bound to active abilities we ***declare an enum*** inside the ***ProjectName.h***.

```
UENUM(BlueprintType)

enum class GameplayAbilityID: uint8

{

   None UMETA(DisplayName = "None"),

   Confirm UMETA(Displayname = "Confirm"),

   Cancel UMETA(DisplayName = "Cancel"),

   Ability1 UMETA(DisplayName ="Ability1")

};
```

Ready to put the components together

Chapter 3 : A base Character class for humanoid Actors

□ **Base character: CharacterBase.h**

Our *CharacterBase* has *access* to the *AttributeSetBase* and AbilitySystemComponent.

Create a *new C++ Class* inside the Unreal Engine, that *inherits* from *Character*.
This will serve as a template for the Player and can also be used for the AI.

Exposing the Attributes through Getters:

```
UFUNCTION(BlueprintCallable,
Category="GAS|Character|Attributes")
float GetMana() const;
```

Set the Field for *DefaultAttributes*:

```
// This is an instant GE that overrides the values for attributes
that get reset on spawn/respawn.

UPROPERTY(BlueprintReadOnly, EditAnywhere, Category ="GAS|Abilities")
TSubclassOf<class UGameplayEffect> DefaultAttributes;
```

Init the an *Array* for possible *Startup Effects*:

```
UPROPERTY(BlueprintReadOnly, EditAnywhere, Category="GAS|Abilities")
TArray<TSubclassOf<class UGameplayEffect>> StartupEffects;
```

Granting Abilities: make sure that you have a second Array to hold our Abilities.

```
UPROPERTY(BlueprintReadOnly, EditAnywhere,
Category="GAS|Abilities")
TArray<TSubclassOf<class UGGameplayAbility>> CharacterAbilities;
```

The *CharacterBase* also contains other *important* Methods like:

```
// Getters for our AttributeSet Values
UFUNCTION(BlueprintCallable, Category="GAS|Character|Attributes")
float GetHealth() const;
```

```
// Server grants Attributes, can be run on client for faster init
virtual void InitAttributes();
```

```
// Server grants Abilities
virtual void AddCharacterAbilities();
```

```
// Remove the individual ability outgoing from the Server
virtual void RemoveCharacterAbilities();
```

```
UFUNCTION(BlueprintCallable, Category ="GAS|Character")
virtual int32 GetCharacterLevel() const;
```

- □ Also think about adding an Interface to the CharacterBase.h

The *Interface* will help us get *Access to* the *ASC*:

```
virtual UAbilitySystemComponent* GetAbilitySystemComponent()
const override;
```

- □ Double check that you have set inside the Constructor:

```
bAlwaysRelevant = true;
```

- □ AbilitySystemComponent

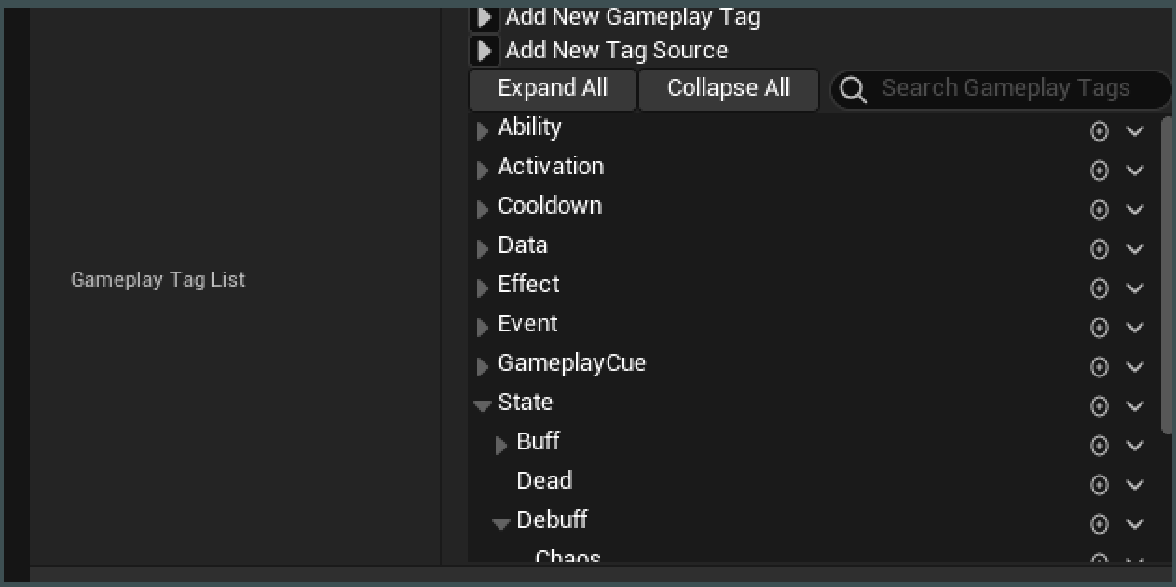
```
if(bActivateAbilityOnGranted){
ActorInfo->AbilitySystemComponent->TryActivateAbility(Spec.Handl
e, false);}
```

☐ Gameplay Tags

Gameplay tags ***serve as conditional triggers*** that control the behavior of the system. They allow you to ***define actions*** or ***behaviors*** based on whether specific tags are active or not, enabling dynamic and adaptive gameplay mechanics.

Gameplay tags are ***organized in a hierarchical structure***, making them ***easy to set up and manage***.

Project Settings-> Gameplay Tags -> Add a new Tag



Abilities after Respawn and Multiplayer

Chapter 4: Player State as owner of the ASC

□ Player State

If you want that ***Abilities persist after respawn***, you should put them into the ***Player State***. Otherwise it's totally fine to add the ASC to the Character directly.

□ Implement the IAbilitySytemInferface to the Player State

The ***PS*** will be the ***Owner of the ASC*** this is why we add the Interface.

```
class GAMEPLAYSYSTEM_API AGPlayerState : public APlayerState,
public IAbilitySystemInterface
```

```
// Implement AbilitySystemInterface
virtual class UAbilitySystemComponent*
GetAbilitySystemComponent() const override;
```

We are basically building ***similar*** Methods to the ones of our ***CharacterBase.h***.

If we have an ASC we also want to change Attributes.

```
class UCharacterAttributeSetBase* GetAttributeSetBase() const;
```

So add as many ***methods*** as needed ***for*** the ***available Attributes.***

```
/** Getters & Setters from the AttributeSetBase (CurrentValues)**/
UFUNCTION(BlueprintCallable,Category="GAS|PlayerState|Attributes
")
float GetHealth() const
```

□ Add Pointers to our own AbilitySystemComponent and AttributeSet

We are also accessing our ***own versions*** of the ***ASC*** and ***AttributeSet.***

```
protected:
// Our own created ASC, that we put on the PlayerCharacter
UPROPERTY()
class UCharacterAbilitySystemComponent* AbilitySystemComponent;
UPROPERTY()
class UCharacterAttributeSetBase* AttributeSetBase;
```

The Player that we are controlling

Chapter 5: Finally we put the parts together

- □ **Player Avatar the one we are using and can see or control during the gameplay**

Create a *new Blueprint class* from our *CharacterBase C++ class* that will be the *PlayerAvatar*. Inside its Constructor set the basic camera, collision and movement options.

- □ **Input section will bind our Enums to Input**

```
void AGPlayerAgent::SetupPlayerInputComponent(UInputComponent*
PlayerInputComponent)
```

```
// Bind ASC and Player Input : OnRep_PlayerState
BindASCInput();
```

- □ **Possessed By**

Here the server gets our *ASC* and calls the *InitAbilityActorInfo(ownerActor, AvatarActor)* and gets the *AttributeSetBase* from our *PlayerState.*

Now we can *InitializeAttributes* for the first time and *set default values*, *startup effects* and *character abilities.*

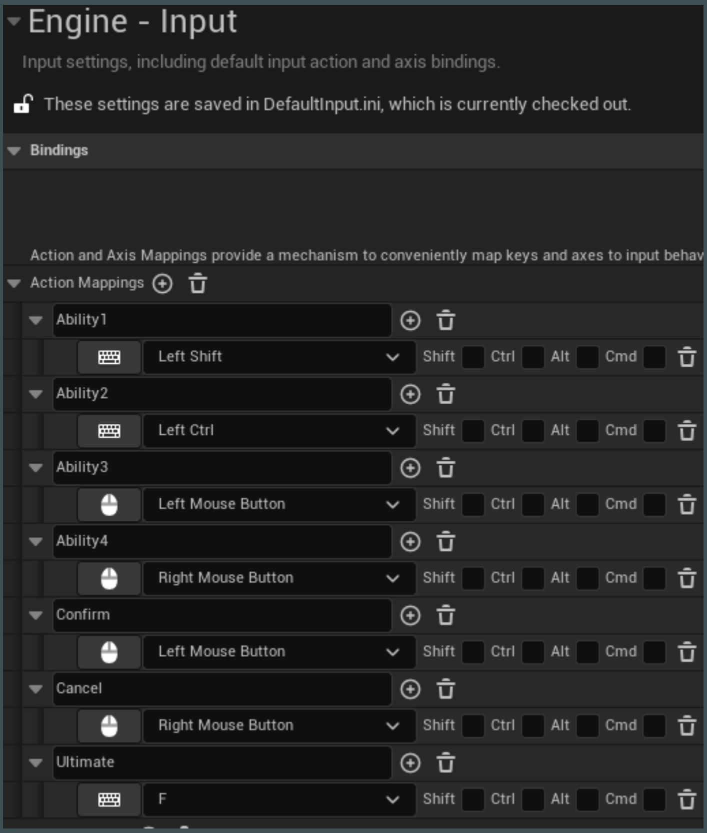
- □ **OnRep_PlayerState**

Is very similar to the PossessedBy but it's *on the client side*. Where the first Method is on the server side.

Add the *Inputbinding again*, for safety with *race conditions*. Sometimes the client is faster and sometimes the server. It does not hurt to bind the input here again.

```
// InputBinds also from SetupPlayerInputComponent.
// Race Condition safety
BindASCInput();
```

☐ Add the Inputs inside the Unreal Engine to call our Abilities



Engine - Input

Input settings, including default input action and axis bindings.

These settings are saved in DefaultInput.ini, which is currently checked out.

Bindings

Action and Axis Mappings provide a mechanism to conveniently map keys and axes to input behav

Action Mappings

Action	Key	Shift	Ctrl	Alt	Cmd
Ability1	Left Shift	☐	☐	☐	☐
Ability2	Left Ctrl	☐	☐	☐	☐
Ability3	Left Mouse Button	☐	☐	☐	☐
Ability4	Right Mouse Button	☐	☐	☐	☐
Confirm	Left Mouse Button	☐	☐	☐	☐
Cancel	Right Mouse Button	☐	☐	☐	☐
Ultimate	F	☐	☐	☐	☐

☐ The base version of a GAS is now ready to be explored

Finally let's test our template project

We will ***add our first Gameplay Effect*** which basically will ***drain*** our ***health and*** stop our ability to move at all. It's a quick way to check that the basic system is working as expected.

Inside the source code, there are conditions to stop reacting when the PlayerAvatar has run out of health.

HINT: This is also true, when you start the game and are ***missing the DefaultAttributes.***

Optional

Chapter 6: Adding UI which is controlled via the Player Controller, Player State and UIWidget

If you make a ***Network*** game, ***each Player*** will have ***his own UI*** and it will only be on his client. The controller is responsible for updating the UI. This step is optional and not necessary for a functioning base version of the GAS.

□ PlayerState.h: Access to the current Attribute values:

```
UFUNCTION(BlueprintCallable, Category =
"GASDocumentation|GDPlayerState|Attributes")
float GetStamina() const;

UFUNCTION(BlueprintCallable, Category =
"GASDocumentation|GDPlayerState|Attributes")
float GetMaxStamina() const;

UFUNCTION(BlueprintCallable, Category =
"GASDocumentation|GDPlayerState|Attributes")
float GetStaminaRegenRate() const;
```

□ PlayerState.h: Delegate handles

Identification of the Delegate/ Function pair.

```
FDelegateHandle StaminaChangedDelegateHandle;
FDelegateHandle MaxStaminaChangedDelegateHandle;
FDelegateHandle StaminaRegenRateChangedDelegateHandle;
```

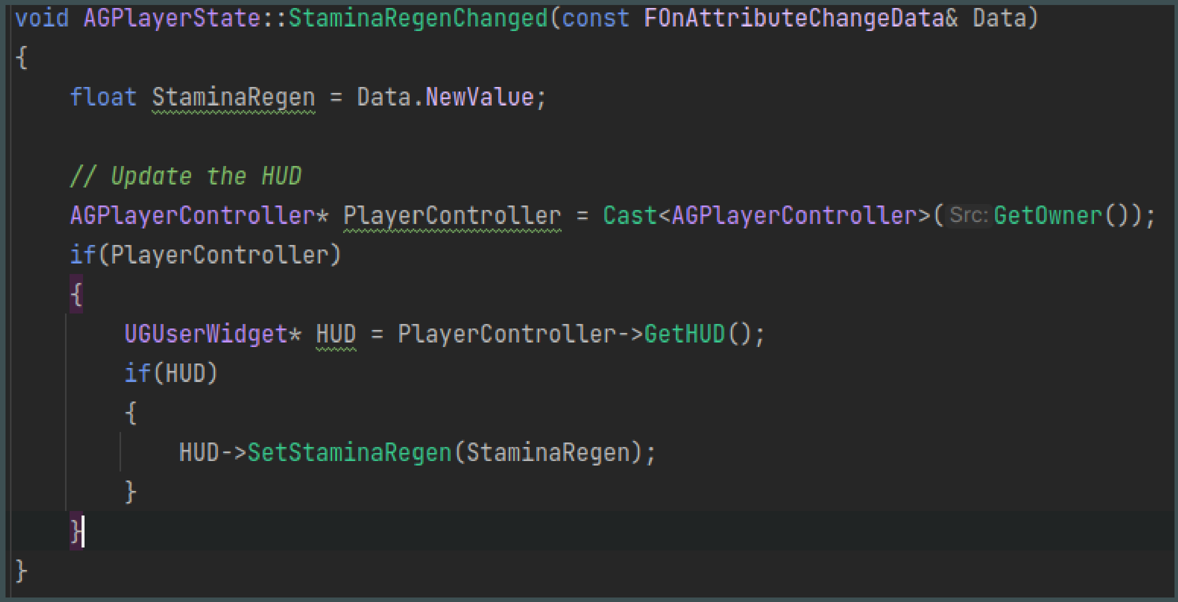
□ PlayerState.h: And the Method callbacks

```cpp
virtual void StaminaChanged(const FOnAttributeChangeData& Data);
virtual void MaxStaminaChanged(const FOnAttributeChangeData& Data);
virtual void StaminaRegenRateChanged(const FOnAttributeChangeData& Data);
```

□ PlayerState.cpp: Callbacks when Attributes are changed

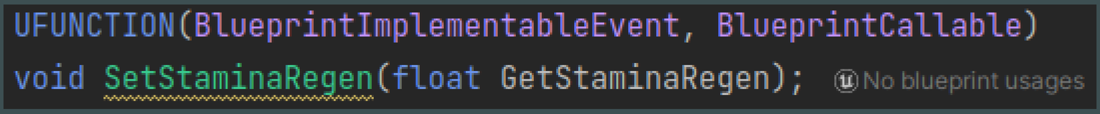
The ***Begin Play connects*** the ***DelegateHandles*** with our Methods. Please read the source code for the correct implementation.

The UI gets its Data from the PlayerState, which gets the values from the PlayerController.



```cpp
void AGPlayerState::StaminaRegenChanged(const FOnAttributeChangeData& Data)
{
    float StaminaRegen = Data.NewValue;

    // Update the HUD
    AGPlayerController* PlayerController = Cast<AGPlayerController>( Src: GetOwner());
    if(PlayerController)
    {
        UGUserWidget* HUD = PlayerController->GetHUD();
        if(HUD)
        {
            HUD->SetStaminaRegen(StaminaRegen);
        }
    }
}
```

□ GUserWidget.h



```
UFUNCTION(BlueprintImplementableEvent, BlueprintCallable)
void SetStaminaRegen(float GetStaminaRegen);
```

□ GUserWidget.cpp

Initializing the Attributes for the first time, when the Hud is created.

```
HUDWidget->SetStamina(PlayerState->GetStamina());
HUDWidget->SetMaxStamina(PlayerState->GetMaxStamina());
HUDWidget->SetMaxStamina(PlayerState->GetStaminaRegen());
```

How to add attributes to the GAS

1. AttributeSet

First we are defining the new ***attribute*** inside the ***AttributeSetBase.h***

```
UPROPERTY(BlueprintReadOnly, Category="Character Level",
ReplicatedUsing = OnRep_Stamina)
FGameplayAttributeData Stamina;
ATTRIBUTE_ACCESSORS(UCharacterAttributeSetBase, Stamina);

UPROPERTY(BlueprintReadOnly, Category="Character Level",
ReplicatedUsing = OnRep_MaxStamina)
FGameplayAttributeData MaxStamina;
ATTRIBUTE_ACCESSORS(UCharacterAttributeSetBase, MaxStamina);
```

Also adding the callback functions.

```
UFUNCTION()
virtual void OnRep_Stamina(const FGameplayAttributeData&
OldStamina);

UFUNCTION()
virtual void OnRep_MaxStamina(const FGameplayAttributeData&
OldMaxStamina);
```

Inside the ***AttributeSetBase.cpp*** start from top:

```
// If we ADD a new Attribute, we also have to ADD it here
void UCharacterAttributeSetBase::GetLifetimeReplicatedProps
(TArray<FLifetimeProperty>& OutLifetimeProps) const
```

```
DOREPLIFETIME_CONDITION_NOTIFY(UCharacterAttributeSetBase,
Stamina, COND_None, REPNOTIFY_Always);

DOREPLIFETIME_CONDITION_NOTIFY(UCharacterAttributeSetBase,
MaxStamina, COND_None, REPNOTIFY_Always);
```

Then fill out the methods.

```
void UCharacterAttributeSetBase::OnRep_Stamina(const
FGameplayAttributeData& OldStamina){

GAMEPLAYATTRIBUTE_REPNOTIFY(UCharacterAttributeSetBase, Stamina,
OldStamina); }

void UCharacterAttributeSetBase::OnRep_MaxStamina(const
FGameplayAttributeData& OldMaxStamina){

GAMEPLAYATTRIBUTE_REPNOTIFY(UCharacterAttributeSetBase,
MaxStamina, OldMaxStamina); }
```

2. CharacterBase.h

Our ***PlayerAvatar*** will ***access these values*** on respawn, add some ***getters***

```
UFUNCTION(BlueprintCallable,Category="GAS|Character|Attributes")
float GetStamina() const;

UFUNCTION(BlueprintCallable,Category="GAS|Character|Attributes")
float GetMaxStamina() const;
```

and a ***setter***.

```
virtual void SetStamina(float NewStamina);
```

CharacterBase.cpp method implementation

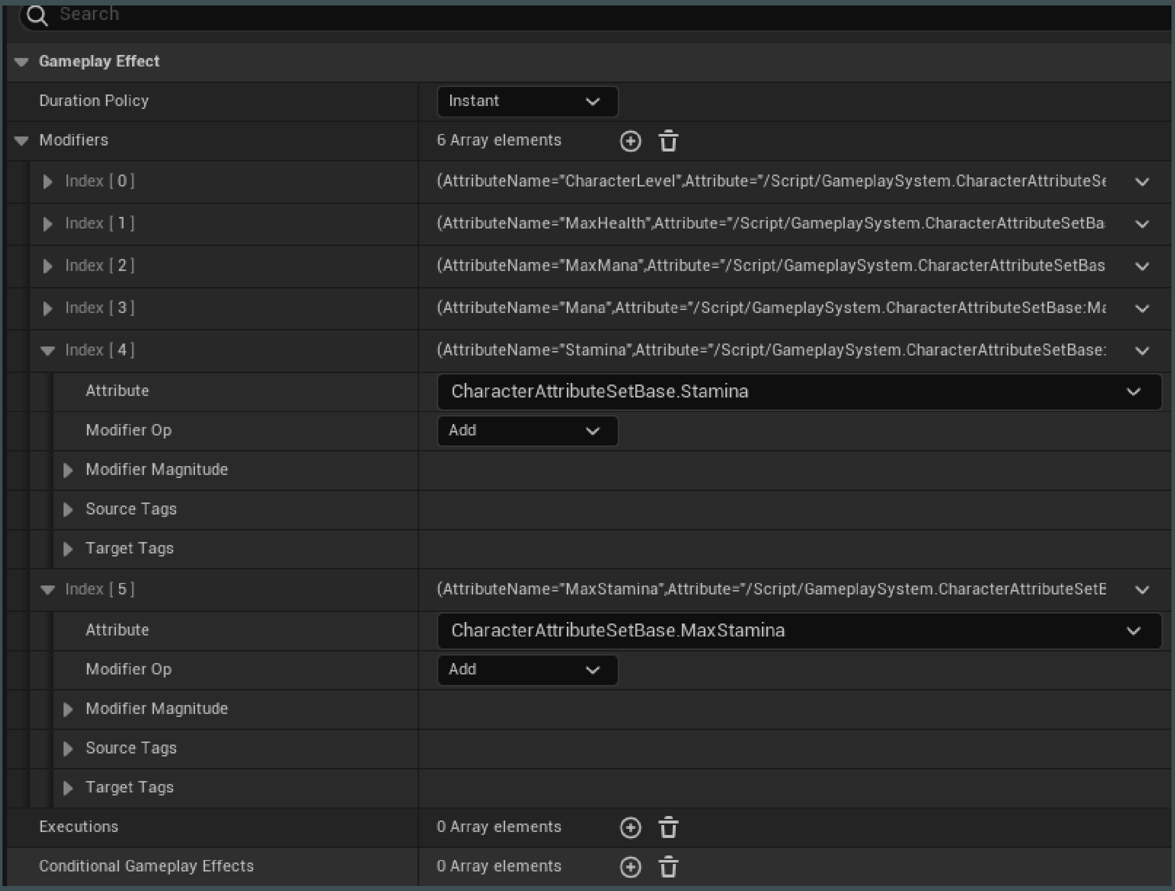
```
float ACharacterBase::GetStamina() const{
   if(AttributeSetBase.IsValid()){
      return AttributeSetBase->GetStamina(); }
     return 0.0f;}

float ACharacterBase::GetMaxStamina() const{
     if(AttributeSetBase.IsValid()){
      return AttributeSetBase->GetMaxStamina(); }

   return 0.0f; }

void ACharacterBase::SetStamina(float NewStamina){
     if(AttributeSetBase.IsValid()){
      AttributeSetBase->SetStamina(NewStamina); }

}
```

3. PlayerAvatar.h

Inside the ***OnPossessed*** we are initializing our values.

```
void AGPlayerAgent::PossessedBy(AController* NewController)
{
/** Respawning  **/
SetHealth(GetMaxHealth());
SetMana(GetMana());
SetStamina(GetMaxStamina());
/** End of Respawning **/
}
```

4. Your new Attributes are ready to be used.



Gameplay Effect	
Duration Policy	Instant
Modifiers	6 Array elements
Index [0]	(AttributeName="CharacterLevel",Attribute="/Script/GameplaySystem.CharacterAttributeSe
Index [1]	(AttributeName="MaxHealth",Attribute="/Script/GameplaySystem.CharacterAttributeSetBa
Index [2]	(AttributeName="MaxMana",Attribute="/Script/GameplaySystem.CharacterAttributeSetBas
Index [3]	(AttributeName="Mana",Attribute="/Script/GameplaySystem.CharacterAttributeSetBase:Ma
Index [4]	(AttributeName="Stamina",Attribute="/Script/GameplaySystem.CharacterAttributeSetBase:
Attribute	CharacterAttributeSetBase.Stamina
Modifier Op	Add
Modifier Magnitude	
Source Tags	
Target Tags	
Index [5]	(AttributeName="MaxStamina",Attribute="/Script/GameplaySystem.CharacterAttributeSetE
Attribute	CharacterAttributeSetBase.MaxStamina
Modifier Op	Add
Modifier Magnitude	
Source Tags	
Target Tags	
Executions	0 Array elements
Conditional Gameplay Effects	0 Array elements

Attribution of sources

Very important links to get an understanding of the GAS.

Source Code & Tutorials:
GASDocumentation
GASContent #Pantong51
ActionRPG v4.27 Demo Project

Talks:
Guided Tour
Exploring GAS
GAS Year One