GD3P01 - ExpBoost Plugin

End User Document

Aiden Storey

Table of Contents

1. Overview

ExpBoost is a plugin that helps to give developers easy access to experience and attribute systems within the Unreal Engine. It utilises a component based system to give *Actors* an experience dispatcher/receiver system to replicate an enemy/player setup. It has been designed with blueprints in mind, but can easily be used as C++ within Unreal Engine. However due to the limitations of Unreal Engine, any overriding functionalities provided must be implemented through blueprints.

1.1 Using this document

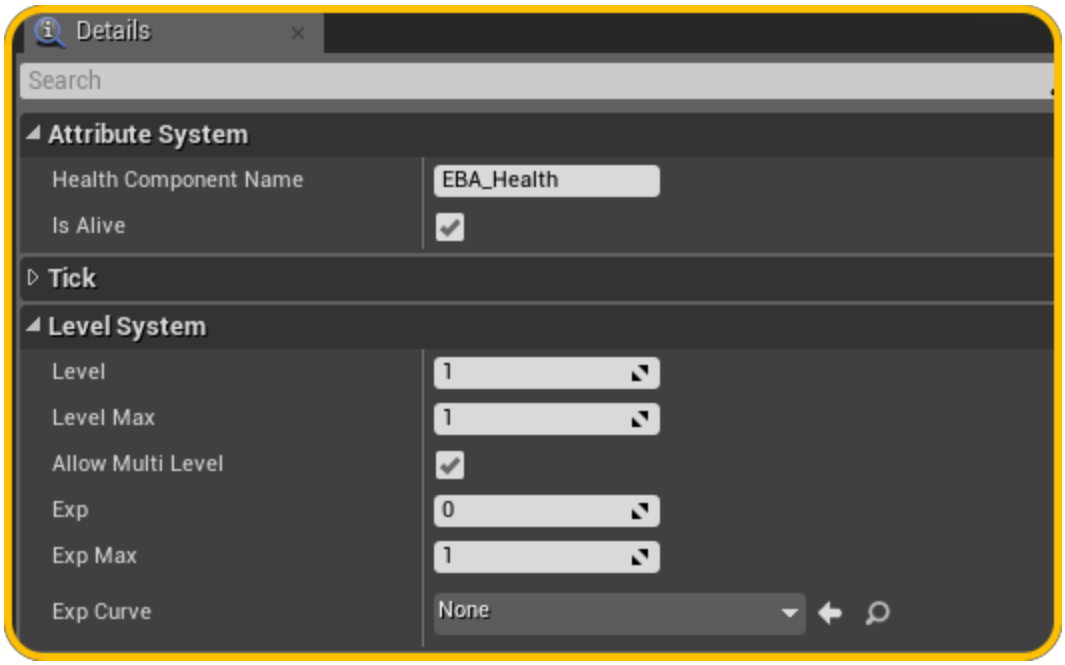
This document is aimed at blueprint developers. Each section has an overview of all the features of the component it relates to, followed by a step by step guide on how to setup the component for use within blueprint (exception being *EBAttribute* which cannot exist on its own). Some sections also contain information regarding room for developer extensions and how this may be done.

A supporting document for this is the Class Definition document which details all of the individual variables and functions of the *ExpBoost* plugin.

2. Components

Within the *ExpBoost* plugin there are four main components that are utilised to deliver the functionality of a core experience and attribute system. These components are designed in such a way that they all work with one another, but can be added individually and still get the intended functionality. It is recommended that the *EBComponent* is always present however.

2.1. EBComponent

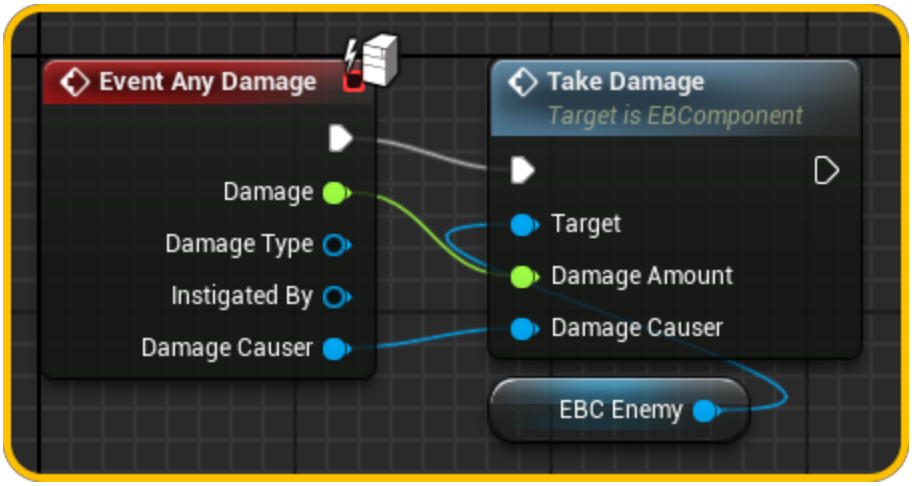


The *EBComponent* handles the core functionality of the experience and attribute system provided by the *ExpBoost* plugin. The main intended functionality is to hold the current state of the level and experience of an *Actor* and control any attached *EBDispatchers*, *EBReceivers*, *EBAttributes* appropriately.

The *Health Component Name* of the *EBComponent* is the name of the *EBAttributes* attached to the same *Actor* that represents the health value. When the *EBComponent* takes any damage* it will get the health *EBAttributes* and deduct the amount from its *Value* . If this process were to reduce the *Value* to zero the *EBComponent* would then check if an *EBDispatchers* was attached

to the *Actor* and get it to dispatch experience accordingly. At which time it will also set the *Is Alive* boolean to false.

* So that the *EBComponent* is able to receive any damage, the developer will need to forward the damage events of the *Actor* to the component. To ensure that the developer is still able to apply damage in their own way, damage forwarding occurs as shown below.



The *EBComponent* has a *ReceiveExp* function that handles any incoming experience from an *EBReceiver* attached to the *Actor* . The function adds the experience to the *Exp* value and processes the experience depending on the setup of the *EBComponent* . The incoming experience goes through a *CalculateModifiedExp* function that is able to be overridden by the developer to add their own modifications before applying the experience.

In the event of the incoming experience makes *Exp* more than *Exp Max* the *EBComponent* will increment the *Level* (provided *Level* isn't already at *Level Max*), then remove the *Exp Max* value from *Exp*. It will then use the attached *Exp Curve* to get the new value for *Exp Max* and repeat the process until either *Exp* is less than *Exp Max* of the new *Level*; or *Level* reaches *Level Max*. If the latter then *Exp* will be set to zero. The process will occur only once if the *Allow Multi Level* boolean is set to false, in which case after the first level up Exp will be set to the lower value of Exp or Exp Max minus one.

The *Exp Curve* attached to the *EBComponent* is a *CurveFloat* that is user defined to represent the amount of *Exp* required at a given *Level*.

To setup the *EBComponent* :

- Create a new *EBComponent* in the content browser of your Unreal project.
- Create a new *CurveFloat* in the content browser of your Unreal project to represent the required *Exp* of the *EBComponent* .
- Open the *EBComponent* blueprint and select the *CurveFloat* you created earlier from the dropdown list for *Exp Curve* in the *Details* panel. (This step can be completed after attaching to an *Actor* if you want the curve to be specific to the *Actor*.)
- Open the blueprint of the *Actor* that you want to attach the *EBComponent* to.
- Use the *Add Component* button to search for the name of the *EBComponent* that you created earlier and add it to the *Actor*.
- Alter the variables of the *EBComponent* for the desired setup of the *Actor* and you are done.

2.2. EBDispatcher

The *EBDispatcher* handles any outgoing experience for an *Actor*. The main intended functionality of the *EBDispatcher* is to be attached to an *Actor* that also has an *EBComponent*. This means that when the *EBComponent* has 'Died' it will call the dispatchers *DispatchExp* function, at which point it will dispatch the *Dispatch Amount* of experience based on the current *Dispatch Type*.

There are 3 *Dispatch Types* that are provided with the *EBDispatcher*.

- *Individual*: This gets the *Actor* that last caused damage to the *EBComponent* and directly rewards it the *Dispatch Amount*.
- World: This users a reliable server function to give the *Dispatch Amount* to all available *EBReceivers* within the level.
- *Other*: This is a function that is intended to be overridden by the developer to implement their own required functionality for the dispatcher.

To setup the *EBDispatcher*:

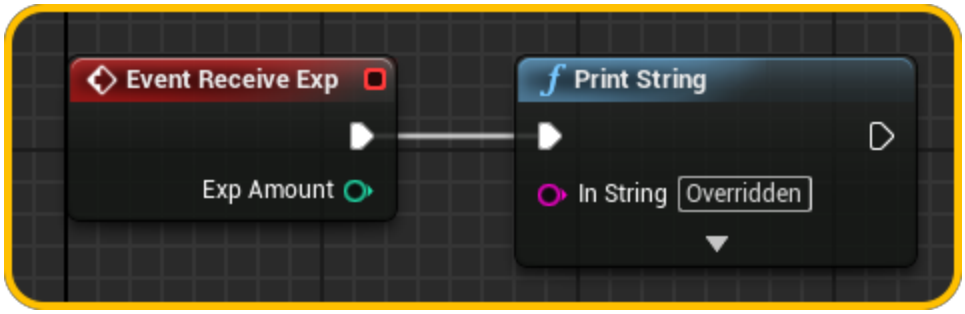
- Create a new *EBDispatcher* in the content browser of your Unreal project.
- Open the blueprint of the *Actor* that you want to attach the *EBDispatcher* to.
- Use the *Add Component* button to search for the name of the *EBDispatcher* that you created earlier and add it to the *Actor*.
- Set the *Dispatch Type* and *Dispatch Amount* that you would like for the *EBDispatcher*, at which point the setup is complete.

2.3. EBReceiver

The *EBReceiver* handles the incoming for an *Actor*. The main intended functionality of the *EBReceiver* is to be attached to an *Actor* that also has an *EBComponent*. This means that when an *EBDispatcher* sends an experience amount to the *EBReceiver* (through its *ReceiveExp* function), it passes that experience on to the attached *EBComponent*.

The *ReceiveExp* function of the *EBReceiver* was designed so that it could be overridden by the developer so that they would be able to change the behaviour that occurred when receiving experience. An example of this would be if the developer wanted experience to be shared within an in game party; they would be able to override this function and give out the exp accordingly.

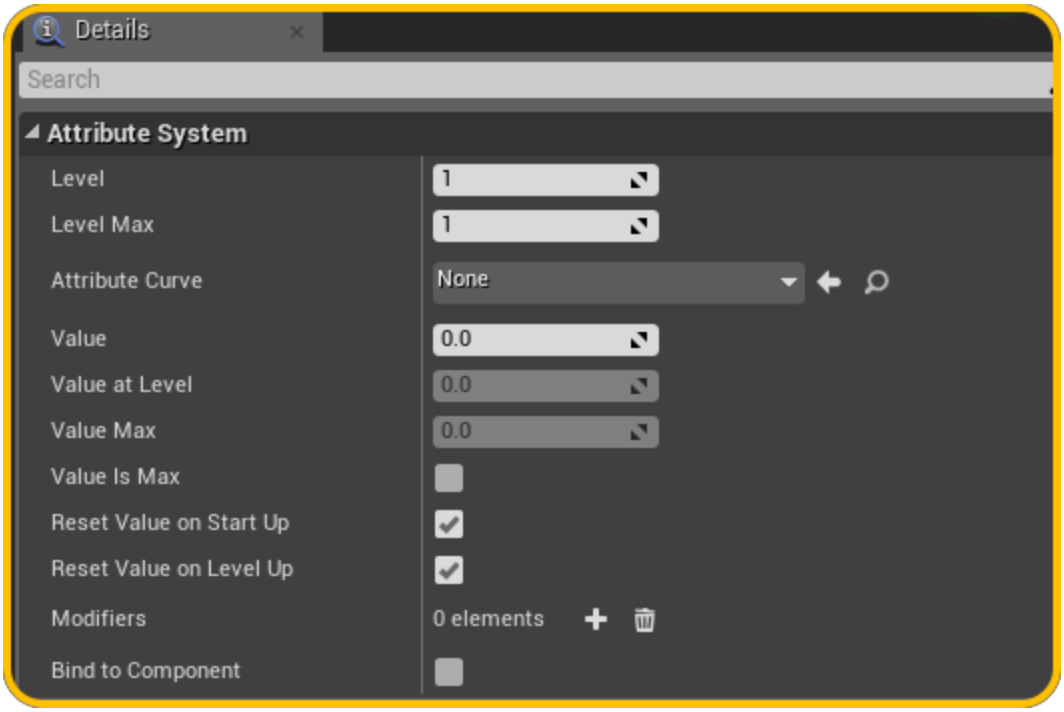
To override the *ReceiveExp* function in Blueprints simply do as shown below within the *EBReceiver*, with the intended functionality replacing the Print String.



To setup the *EBReceiver*:

- Create a new *EBReceiver* in the content browser of your Unreal project.
- Open the blueprint of the *Actor* that you want to attach the *EBReceiver* to.
- Use the *Add Component* button to search for the name of the *EBReceiver* that you created earlier and add it to the *Actor*.
- Optional: Override the *ReceiveExp* function if you would like to differ from the intended *ReceiveExp* functionality.

2.4. EBAttribute



The EBAttribute handles the definition and state of attributes for an *Actor*. The main intended functionality of the EBAttribute is to provide a simple way of defining dynamic attributes that an *Actor* can utilise for gameplay features.

The *Attribute Curve* for the component is a developer defined *CurveFloat* which will represent the *Value at Level* for any given *Level* of the *EBAttribute*. The component will take the value for the given *Level* from the *Attribute Curve* and then apply any *EBAttributeModifiers* attached to the component to calculate the *Value Max*, which is the max amount that the *Value* of the *EBAttribute* can be. (See *EBAttributeModifiers* section for more on modifiers.)

Setting the *Value Is Max* boolean determines if *Value* should constantly be at *Value Max* or not. This is so that the *Value* variable is always the main variable to look at for the state of the *EBAttribute.* This is useful in instances where the *EBAttribute* represents a static variable like ‘Strength’ instead of a dynamic like ‘Health’.

Setting the *Reset Value on Start Up* and *Reset Value on Level Up* booleans determine if *Value* should be reset back to *Value Max* when the game starts or when the *EBAttribute* levels up.

Setting the *Bind to Component* boolean determines if the *EBAttribute* gets its *Level* and *Level Max* from an *EBComponent* attached to the same Actor. When it is bound to an *EBComponent* the component will continually update the *Level* of the *EBAttribute* when it changes *Level* itself. In conjunction with the *Attribute Curve*, this means that the Value Max of the *EBAttribute* is able to be dynamically based on the actors *Level*.

To setup the *EBAttribute*:

- Create a new *EBAttribute* in the content browser of your Unreal project.
- Create a new *CurveFloat* in the content browser of your Unreal project to represent the value of the *EBAttribute*.
- Open the *EBAttribute* blueprint and select the *CurveFloat* you created earlier from the dropdown list for *Attribute Curve* in the *Details* panel. (This step can be completed after attaching to an *Actor* if you want the curve to be specific to the *Actor*.)
- Open the blueprint of the *Actor* that you want to attach the *EBAttribute* to.
- Use the *Add Component* button to search for the name of the *EBAttribute* that you created earlier and add it to the *Actor*.
- Alter the variables of the *EBAttribute* for the desired setup of the *Actor* and you are done.

3. Additional Classes

To extend the functionality within the attribute system of the *ExpBoost* plugin, and let the *EBAttributes* be more dynamic; two helper classes have been developed.

3.1. EBAttributeModifier

The *EBAttributeModifier* handles the definition of the modification to an *EBAttribute*. The main intended functionality of the *EBAttributeModifier* is to be able to modify *EBAttributes* easily without having to alter their actual value.

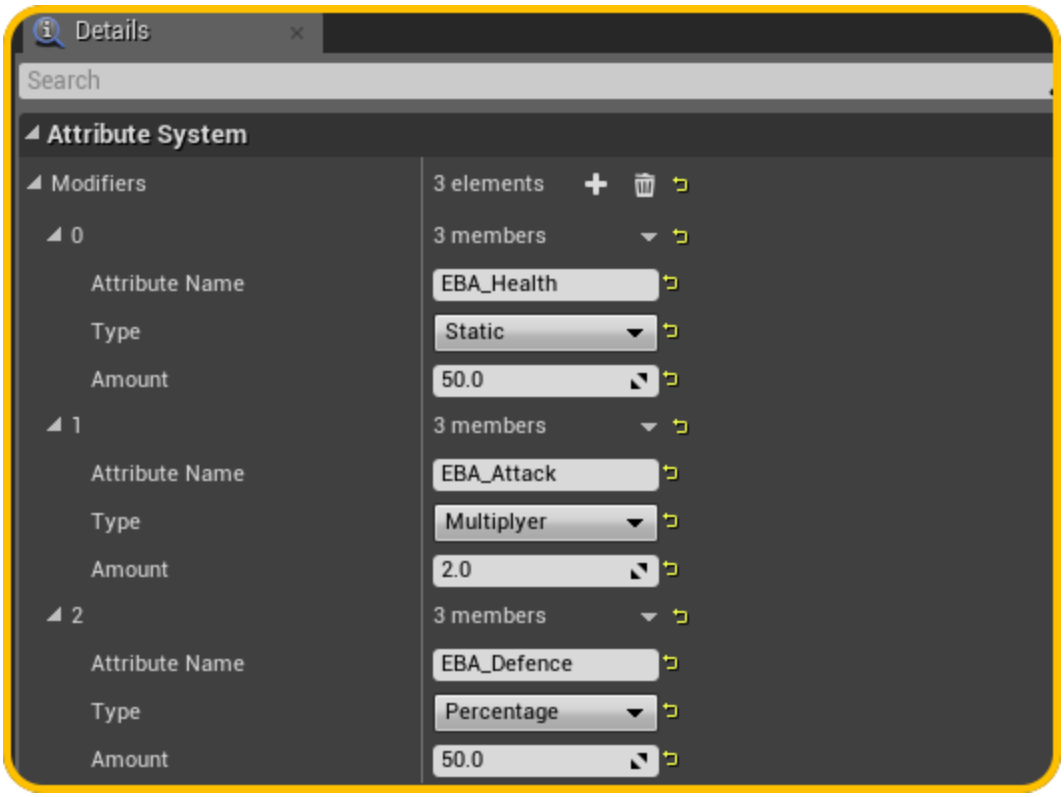
Attribute Name is the actual name of the *EBAttribute* that will be modified. This is not required if the *EBAttributeModifier* is directly on the *EBAttribute*.

There are three types of modification that can happen to the *EBAttribute*.

- *Static*, where Amount is applied to the *Value at Level* of the *EBAttribute*.
- *Multiplier*, where the *Value at Level* plus *Static* modification is multiplied by *Amount* then added to *Level plus Static*.
- *Percentage*, where the *Value at Level* plus *Static* modification is multiplied by *Amount* divided by one hundred then added to *Level* plus *Static*.

An *EBAttributeModifier* cannot exist on its own. It must be created either on the *EBAttribute* or through the use of an *EBItem*. (See *EBItem* section for more on items.)

3.2. EBItem



The *EBItem* handles the attaching/detaching of a collection of *EBAttributeModifiers* to an *EBAttribute*. The main intended functionality of the *EBItem* is to give the developer an easy way to create an organised set of *EBAttributeModifiers* that can be attached to an *EBAttribute*.

The *EBItem* is an abstract class at its core as it is created with the intent of being used alongside other elements of gameplay to create things such as weapons or buffing skills.

The *EBItem* has an *AttachItem* and *DetachItem* function that iterates over the *EBAttribute* components of the passed *Actor* and binds any of the matching *EBAttributeModifiers*.

To setup the *EBItem* it is recommended that it is used as an inherited class to extend the functionality of the class requiring it. When needed call the *AttachItem* and *DetachItem* to have *EBAttributeModifiers* be applied to corresponding *EBAttributes*.

4. Delegates:

Delegates have been setup to broadcast at the main interaction points of each element of the *ExpBoost* plugin to allow the developer to be notified for either debug or functional purposes.

4.1. EBComponent

- *OnDeath*: Broadcast occurs when the attached health *EBAttributes* value reaches zero.
- *OnGainExp*: Broadcast occurs when the *ReceiveExp* function is called.
- *OnLevelUp*: Broadcast occurs when *Exp* is more than the current *Levels*, *Exp Max*.

4.2. EBDispatcher

- *OnDispatchExp*: Broadcast occurs when the *DispatchExp* function is called.
- *OnDispatchIndividualExp*: Broadcast occurs during the *DispatchExp* function if Dispatch Type is *Individual*. (Occurs as well as *OnDispatchExp*.)
- *OnDispatchWorldExp*: Broadcast occurs during the *DispatchExp* function if Dispatch Type is *World*. (Occurs as well as *OnDispatchExp*.)
- *OnDispatchOtherExp*: Broadcast occurs during the *DispatchExp* function if Dispatch Type is *Other*. (Occurs as well as *OnDispatchExp*.)

4.3. EBReceiver

- *OnReceiveExp*: Broadcast occurs when the *ReceiveExp* function is called.

4.4. EBItem

- *OnAttachModifier*: Broadcast occurs each time an *EBAttributeModifier* is successfully attached to an *EBAttribute*.
- *OnDetachModifier*: Broadcast occurs each time an *EBAttributeModifier* is successfully detached from an *EBAttribute*.