# OpenRPG: Stats

*Release 1.0*

**Mar 04, 2021**

# Contents:

# CHAPTER 1

# Overview

Welcome to the OpenRPG Stats system documentation! This **powerful, modular, flexible** and **efficient** stats system will allow you to create not only stats with **buffs** and **effects**, but **also attributes** and **custom leveling system**, very common in most RPG games. You will be able to **extend any actor functionality** in your game with the features of this system **easily** and getting it implemented very quickly!

It has been structured to be **easy to understand and follow**, so everyone is capable of understanding how the whole system works, and with this documentation every aspect of it will be carefully explained. Even though im a completely beginner on documenting, i have tried to structure it from a **general overview** of the folder structure and its content to a **deeper look** at each part of the whole system. On each part i will show you some comprehensive guides that will clarify how to expand the system to adapt every aspect of it to your needs, as well as a look at the most important blueprints that handles the overall system functionality, which are the actor components. Then you will learn how to migrate and integrate the system into your own project.

**Hint:** If you like it dont forget to leave a star on GitHub and also it would be a huge help if you give me all your feedback about errors you find or some features that you would like to be implemented!

## 1.1 Features

### 1.1.1 Stats

- Create your own stats like health, mana, hunger, fire resistance, shield power, critical chance, etc. You can also make your own sets of stats!
- All of them managed via datatable, you dont have to touch code to create your custom character stats!
- You can also configure if you want to track the current value of any stat. This is useful in some stats like health, when you have the current and the maximum health.

### 1.1.2 Attributes

- Create your own attributes like intelligence, strenght, vitality, etc. As said with stats, you can also create your own sets of attributes.
- Managed via datatable.
- Add your custom bonuses to stats when any of the attributes values are modified!
- Receive attribute points and spend them in the attributes you want.

### 1.1.3 Leveling System

- Create your own per level needed experience values to level up, manually or by formula.
- Everything handled by datatable, extremely easy to configure and use.
- Add your custom functionality when leveling up (give rewards, show notification, etc).
- Scale stats and attributes when leveling up.

### 1.1.4 Buffs

- Dynamic bonuses to stats and/or attributes for a specific time duration or even permanent.
- Possitive or negative bonuses.
- Possibility to display icon on UI showing the remaining time.

### 1.1.5 Effects

- Over a specific time duration (or permanent), you can set a tick interval and create your own funcionality when the tick is executed.
- Also displayable as buffs.

## 1.2 Folder Structure

This section will cover the project folder structure and a brief description of the main files on it. When you download the project from GitHub and you open it with Unreal Engine, the first thing you will see at the root of the project folder are the `Demo` and `OpenRPG_Stats` folders. Lets quickly see what each one contains:

### 1.2.1 Demo

This folder contains blueprints, icons, widgets and stuff from the *Third Person Template* and other ones created just to test and explore the system. Lets focus on the blueprints and widgets folders:

#### Blueprints

- `BP_DemoCharacter`: Used to test the system when you run the project. It has all the system related functionality in a separated `StatsSystem` graph, which contains components initialization, keyboard controls, enemy selection with the mouse and interaction with the saving / loading functionality.

- `BP_DemoEnemy`: Just to see what the system is capable of, i have included an enemy that can be selected with the mouse and implementats stats functionality, as i think its something very common in RPG games and helps you see the flexibility of the system components, which arent limited to be implemented only in the character for example.
- `BP_DemoEnemyStats`: The implementation of the stats functionality for the enemies. The only thing you need to know by now is that it has all the system stats functionality but with a very little modification to handle the widget that displays the health of the selected enemies.
- `GI_DemoStats`: This is a `GameInstance` object. As you will see later, the system can save and load the data using a file in the hard drive with the `.sav` extension, or by using the `GameInstance`, which is commonly used to keep data between game levels without needing to use the hard disk `.sav` file, so its much faster and easy to manage. Its functionality is very simple and you can get it implemented in your own game instance in just a few minutes. When the game starts, this class will dump all the saved data in the file disk into itself.

#### Widgets

- `WB_DemoStatsWindow`: Simple widget to watch the system values updated when you use keyboard controls, you can also check my weak artist concepts :(
- `WB_DemoMain`: Its the widget added to the viewport and displayed in the game, which contains all the other widgets.
- `WB_DemoEnemy`: The widget that displays the health of the selected enemy.

**Note:** Its a good practice to create a main HUD widget as the WB_DemoMain, that contains all your widgets, and then add it to the viewport instead of adding all of them individually.

### 1.2.2 OpenRPG_Stats

It contains all the system functionality. Lets see exactly what each folder contains:

#### Blueprints

This is by far **the most important folder, because it contains the system core functionality**. This folder has the blueprint function library `BPL_Stats`, which includes utility functions that we can call anywhere, and the `BP_StatsSaveObject`, which is the container for the saved data in your hard disk, so it will keep the saved data even if the player leaves the game. This folder also contains another 2 subfolders:

#### Components

These components will provide any actor they are added to with stats functionality with buffs and effects (`BPC_Stats`), attributes (`BPC_Attributes`) and customizable leveling system (`BPC_Leveling`). These components are independent from each other, so you can just implement the ones you want on your actors, but they can work together perfectly! Each one will have its own section in the documentation as they implement most of the overall system functionality.

### Interfaces

They are used mainly to implement the interaction between the components and the widgets. They allow the components to be independent of the widgets, so you can use your own widgets and just implement the interface on them to be able to receive components data! There are 5 interfaces with this purpose:

- `BPI_Stats`: Used by `BPC_Stats` to notify the widgets when a stat value is modified.
- `BPI_Attributes`: Used by `BPC_Attributes` to notify the widgets when an attribute value or the attribute points are modified.
- `BPI_Leveling`: Used by `BPC_Leveling` to notify the widgets when some experience is received or the actor levels up.
- `BPI_Buffs`: Used by `BPC_Stats` to notify the widgets when a buff is applied or removed.
- `BPI_Effects`: Used by `BPC_Stats` to notify the widgets when an effect is applied, removed or ticks.

There is also an extra interface used by the built-in save / load system:

- `BPI_SaveLoad`: It is responsible of the communication between the components and the saving containers (`BP_StatsSaveObject` and `GameInstance`). Its useful to make the components independent from the saving containers, they will just send and receive data using the interface, but they dont care about the type of container the data is going or coming from.

**Note:** If you dont know how interfaces work here you have the link of the official documentacion.

### DataTables

This folder contains all the tables where you configure your own default system data. When you create a new row and fill its data, you can just use that row name putting it in the corresponding component, so it will load the data you filled in that row, which allows you to create your own sets of attributes and stats.

**Note:** This tables will be accesed by their respective components **only** when there is no saved data to load, so they take the tables data as **default**.

### DT_Stats

- `Stats`: Here you configure stats and the default values they are going to have when `BPC_Stats` loads them.
- `CurrentStats`: The default values of the stats that have current value. For example health, mana, hunger, stamina and so on are stats that have a value between 0 and the maximum value they can get, that value is called current stat. So all the stats you add in this variable will have a current stat value.
- `StatsPerLevel`: If an actor implementes leveling system and stats system, when it levels up, the component `BPC_Leveling` will tell `BPC_Stats` to add the stats you configure here.

### DT_Attributes

- `Attributes`: Here you configure attributes and the default values they are going to have when `BPC_Attributes` loads them.

- `AttributesBonusesToStats`: When an attribute is modified, here you can configure what bonuses are going to be applied to a set of stats. The best way to see this is with an example. Lets say you add the *Strenght* attribute, and then you say that when it is increased, the stat *AttackValue* is going to be increased by 1%. You can do it with as many stats as you want, either flat or percentage bonuses as seen in the example.
- `AttributesPerLevel`: If an actor implements leveling system and attributes system, when it levels up, the component `BPC_Leveling` will tell `BPC_Attributes` to add the attribute points you configure here.

### DT_Leveling

- `StartingLevel`: The starting level the actor is going to have.
- `MaxLevel`: The maximum level the actor can reach. When this level is reached and actor receives more experience points, they will be ignored.
- `UseCustomExperienceFormula?`: There are some RPG games where the experience per level is defined by a formula. So if you set this to true, system will use the formula specified by `ExperienceFormulaType`. If you leave it to false, you need to set the `NeededExperiencePerLevel` map variable manually.
- `ExperienceFormulaType`: The formula you want to apply. I have included 2 examples, which are implemented in `BPC_Leveling`. If you add new formulas make sure you implement them there!
- `NeededExperiencePerLevel`: As said before, if you dont use a custom formula, you need to manually set it up. The key is a level, and the value is the needed experience to reach next level. If custom formula is used, it will be constructed from `StartingLevel` to `MaxLevel` using the formula. Even if you use a formula, you can put default values that can be useful for some types of formulas. For example, in the *Type2* formula that i have implemented, the needed experience for each level is the needed experience in the previous level + 10% of that experience, but what do we do for level 1? I have simply added a level 0 entry, so when the data is constructed, for level 1 it will see that entry :)
- `LevelUpRewards`: The reward given to the player when it levels up, you can customize it as you want. Each key represents the reached level.

### DT_Buffs

- `DisplayedName`: The name you want to assing to that buff when displayed (for example if you have a tooltip or something like that).
- `Description`: The buff description (can be used in tooltips as mentioned before).
- `Icon`: The icon to display when buff is applied.
- `Displayable`: Controls if the buff should be displayed or not when applied.

### DT_Effects

- `DisplayedName`: The name you want to assing to that effect when displayed (for example if you have a tooltip or something like that).
- `Description`: The effect description (can be used in tooltips as mentioned before). In this case, if you write *{value}* in it, it will be parsed with the value the effect has.
- `Icon`: The icon to display when effect is applied.
- `Displayable`: Controls if the effect should be displayed or not when applied.

As you can see `DT_Buffs` and `DT_Effects` are almost identical, but i prefered to have them in separate data tables so if you want to add new data for buffs or i decide to include new features on them, it can be done without interfere in the effects and viceversa. Moreover, these tables are not used by components as `DT_Stats`, `DT_Attributes` or `DT_Leveling`, in this case we use them to refer to a buff or effect, very useful to retrieve neccessary data on demand (in widgets for example).

**Tip:** You can create your own data tables if you want to organize them differently. For example, you can have one datatable for the sets of stats of your enemies, and other one for your each of your character types of your game. Here you have a video explaining data tables.

## Enums

All the enums used in the system:

- `e_Stat`: Contains all the stats that you have in your game.
- `e_Attribute` : Contains all the attributes that you have in your game.
- `e_ExperienceFormulaType`: The leveling experience formulas that you use in your game.

## Structs

All the structs used in the system by the components, data tables, save / load system, etc.

- `s_Attribute`: It is used by `BPC_Attributes` to keep your attributes bonuses and final values correctly calculated.
- `s_AttributeBonus`: Used to store the flat and percentage bonuses that can be applied to attributes.
- `s_AttributesData`: Specifies the data each row has on *DT_Attributes*.
- `s_AttributesSaveData`: You can see it as a data packet that contains all the attributes saved data for a particular actor.
- `s_Buff`: Contains the stats and attributes bonuses, duration and so on that you configure when applying a buff.
- `s_BuffStatic`: Buff data which is always the same for that buff (used in *DT_Buffs*).
- `s_DynamicBuffData`: Used internally by `BPC_Stats` to keep track of the buffs timing.
- `s_Effect`: Contains the effect value, duration and so on that you configure when applying a effect.
- `s_EffectStatic`: Effect data which is always the same for that effect (used in *DT_Effects*).
- `s_DynamicEffectData`: Used internally by `BPC_Stats` to keep track of the effects ticks and timing.
- `s_LevelingData`: Specifies the data each row has on *DT_Leveling*.
- `s_LevelUpReward`: Here you can configure the reward you want to give to an actor when it levels up (applied also by yourself).
- `s_LevelingSaveData`: Contains all the leveling data to be saved, like level, experience, etc.
- `s_Stat`: It is used by `BPC_Stats` to keep your stats bonues and final values correctly calculated.
- `s_StatBonus`: Used to store the flat and percentage bonuses that can be applied to stats.
- `s_StatsData`: Specifies the data each row has on *DT_Stats*.
- `s_StatsSaveData`: All the stats data used to save and load from it.

#### UI

This folder contain a set of widgets to show how your own widget could be implemented, if you dont have you own widgets design yet, you can use this ones! The only thing you need to do for your custom widgets is adding the *Interfaces* you want depending on what data is going to be displayed in that widget. For example you can see that the `WB_Stat` implements the `BPI_Stats` interface, and the `WB_Bar` implements both `BPI_Buffs` and `BPI_Effects` interfaces to display buffs and effects on it.

As they are only 4 widgets with very little code, i suggest you to give a look at them and the variables they have. It wont take you much time and it will help you later when adding new stats and attributes ;)

## 1.3 System Core: Components

When we refer to the system functionality, the majority is implemented in `BPC_Stats`, `BPC_Attributes` and `BPC_Leveling`. However, when we refer to the system core, that is the `BPC_Stats` component, because `BPC_Attributes` and `BPC_Leveling` can be considered as extensions to the main purpose of the whole system, which is giving stats functionality to an actor.

**Note:** Each component work as isolated as they can, they just communicate the minimum to have as less dependencies between them as possible.

Lets say you want to use `BPC_Attributes` and `BPC_Stats` in an actor. When `BPC_Attributes` gets one attribute modified, it will search if the **owning actor** has a `BPC_Stats` component. If not, it wont do anything related to the stats, but if it is found, it will tell the `BPC_Stats` component about that attribute modification (attributes can affect stats). For example if you get +1 point of *Strenght*, you can set it to increase *Attack Value* by +15.

### 1.3.1 General Behaviour

In this section we will see that even though the components have different functionality, there are a few aspects that all of them share.

#### Settings Variables

Every component has a few variables to configure it. The names of the variables will vary a bit depending on the component, but the functionality is the same for all of them:

- `DataTable`: The datatable from which to extract the data.
- `TableDataRow`: The name of the row data in `DataTable`
- `DebugEnabled?`: If true, the component will print messages on the screen showing possible errors and warnings when its initialized.

These settings are enough to setup the component, because all the data is in the assigned datatable, so you only need to write the name of the row in `TableDataRow` and the component will load the data from there.

#### Widgets Refreshing

When communicating with widgets to send them data, all the components uses some of the interfaces showed in the `Blueprints` section, depending on the data it is going to send. Also, all components have a `Widgets` variable, which contains the widgets that the component is going to send data when some events happen.. There are also 2 common functions:

- AddCustomWidgets: Before any component is initialized, we need to pass it the widgets it has to refresh as said before, so this function just adds widgets to the Widgets variable mentioned before. For example, the BPC_Stats component in BP_DemoCharacter will refresh the WB_DemoDebugStats and its stats subwidgets, but it doesnt refresh the attributes or the current experience widgets. Its a simple way to link a component with the widgets you want.
- RefreshWidgets: Its called when the component has loaded the neccessary data, either from saved before or by datatable, to refresh its widgets with all the data it has. For example, the BPC_Attributes will refresh every attribute and the available attribute points.

### Save / Load

All the components have built-in save / load functionality. They can save and load data either by GameInstance or using a SaveGame class.

**Note:** If you dont know about GameInstance and SaveGame classes, here you have a nice YouTube video from Mathew Wadstein explaining the GameInstance class and this other link explains perfectly how the SaveGame class work.

There are 4 functions used for this purpose:

- CreateSaveData: It gets the current component data and creates the proper struct (you can see it as a packet) with all the data, depending on the component: s_AttributesSaveData , s_StatsSaveData or s_LevelingSaveData.
- SaveData: It will send the data created by CreateSaveData to the corresponding container that will store the data with an "identifier" of that data, given by the SaveName input. The container will save the data with the value from the SaveName variable, and it will be different depending on the ToGameInstance boolean input. If it is true, it will get the game's GameInstance object to send the data there. Remember that this class is persistent across levels, so if we go from one level to another, we can send the data to that object and retrieve it in the new level. If the boolean is false, then it will get the .sav file on disk (if it doesnt exists it will create it) to save the data there.
- LoadData: Exactly the inverse of the SaveData, it will get the data that has the label given by the input SaveName assigned to it, also depending on the container it will ask for that data to GameInstance or .sav file. When it gets the data it will update the component variables with it, and it will also set the boolean HasLoadBeenSuccessful? to true (false if no saved data was found). And if no data was found, what do we do? Here the LoadTableData function comes to the rescue!
- LoadTableData: It will be called in both BPC_Stats and BPC_Attributes **only** when the component hasnt received saved data before. If so, the function will take the value of TableDataRow mentioned before to extract the data from the assigned datatable, just like default data. In BPC_Leveling it will be called always **but** the value of HasBeenLoadSuccessful? will branch inside the function. Thats because we need to construct the needed experience for each level, because that data is not saved by the SaveData function.

**Note:** Typically you would only save data to the .sav file disk when the user clicks on some menu button to save the current game state or if you have a menu button that closes the game, you can save there to the disk before closing the game. Anyway, it depends on how have you configured the saving and loading options in your game.

### Consistency

Every component has a function called CheckConsistency. This function will tell you if there is any unconsistency in the component settings. For example if the variable DataTable doesnt have a valid value, or if the row assigned in `TableDataRow` doesnt exist in the datatable if you have assigned a valid value to it. It will check more consistency errors depending on the component, because each of them has its own consistency settings. This function is also callable from the editor, that means that you can call it without running the game. To try it just go to the `BP_DemoCharacter` blueprint, select one of the system components in the left side and you will see at the component details that you have a button with the same name as the function, just click it.

#### Initialization Pipeline

At this point, you have seen a lot of functions and what they do, so the last thing to clarify is when some of them are called and in what order. Typically the initialization process is done in the event BeginPlay of your character or player controller. In the demo content, it has been implemented in the character. The steps would be the following ones:

1. Get the component and call the function AddCustomWidgets and pass it an array of the widgets it has to refresh.
2. Call the functin LoadData and give it a SaveName value, for example if you have a main menu and the user can give a name to the character, that would be a perfect value to put here.
3. Call the Initialize event of the component.

Every component has an event called `Initialize`. The pipeline of this event is the following one:

1. Searches for other system components in the owning actor. For example `BPC_Leveling` will search for `BPC_Attributes` and `BPC_Stats` to perform some actions when the owning actor levels up. If the component hasnt found other components, nothing happens.
2. Calls the function `CheckConsistency` if the variable `DebugEnabled?` is true, which will print some strings in the screen if there is any configuration error.
3. Calls `LoadTableData` if `HasLoadBeenSuccessful?` is false (with the special case of the `BPC_Leveling` component mentioned before) to load the default data from the proper datatable.
4. Refresh all the widgets with the loaded data.

**Tip:** The previous process is the same for all the components.

Now that you have understood the main idea about how the components work, lets see the details of each component in the next sections.

## 1.4 Stats

In this section firstly im going to show you how easy is to add new stats and sets of stats to your actors like enemies or characters, as well as how to add new buffs and effects to your game. After that, as all functionality related to stats is implemented in `BPC_Stats`, i will explain its variables, functions and dispatchers so you can understand how it works and what can you do with it.

### 1.4.1 Adding new stats

Adding new stats is a very straightforward process:

1. Go to `e_Stat` under the *Enums* folder and add your new stat.
2. Then in your widget that displays all the stats, add a new `WB_Stat`, select it and you will see in the *Details* panel that it has some exposed variables. Fill the enum variable with your new stat and configure the rest of variables as you want.

3. Make sure you add that new widget to the widgets that the component has to refresh with the **AddCustomWidgets** function.

**Tip:** To remove a stat the process is exactly the inverse, just remove the widget and the entry in the enum.

### 1.4.2 Creating sets of stats

Once you have created your desired stats, then im sure you want to create a set of stats to, for example, assign a different one per character race in your game. This is also extremely easy:

1. Go to *DT_Stats* and create a new row, you can call it *Ninja* for example.
2. Then just fill the stats you want to have for that particular set.
3. Go to your class that has the stats component you want to load that data (lets suppose you have a *BP_Ninja* class).
4. Click on the stats component and in the *Details* panel you will see the `StatsTableRow` variable under the *Settings* category, just put the row name you created before (in this case it would be *Ninja*).

**Note:** I have used `DT_Stats` as the datatable for the example, if you have created your own one then its perfect, just make sure you put that table in the `StatsTable` of the component.

**Done!** Now when your *BP_Ninja* class is initialized, it will load the set of stats you have configured in the datatable (remember it will only happen if there is no previous saved data).

### 1.4.3 Adding new buffs

Buffs have two parts of data. One of them is the static data, which means it will never change (like the icon, the description, etc). The other one is the dynamic data, which can be the duration, the bonuses to stats and attributes, if it is going to be permanent, etc. Lets create the static data first:

1. Go to *DT_Buffs* and create a new row (lets call it *FireBall* for example).
2. Fill the required data for that particular buff.

With that now you are able to apply a buff. When you apply a buff using the *ApplyBuff* function in `BPC_Stats`, it will ask you for the row name you have created (the static data, *FireBall* in the example) and the dynamic data of the buff in that moment. This last one is completely up to you as it will may change depending on the situation and your own game requirements.

### 1.4.4 Adding new effects

The process is almost the same as buffs. The difference is that the implementation of each effect is completely up to you.

1. Go to *DT_Effects* and create a new row (lets call it *Poison* for example).
2. Fill the required data for that particular effect.

Now, as mentioned, the way each effect is implemented is completely up to you using the dispatcher **OnEffectTick** in the actors that implement the stats component. For example the *Poison* effect of the example can remove 3% of the actor's *Health* each time it ticks. You can see the `BP_DemoCharacter` to check some examples.

**Note:** Buffs and effects have the same static data, but they have been kept separated to make it more flexible, so if you expand the buffs static data, it doesnt interfer with the effects data.

### 1.4.5 BPC_Stats

#### Variables

- `UseAttributeBonusForPercentageCalculations?`: If true, the system will treat the stat base value **and** the bonuses from attributes for the percentage bonus calculations. If false, it will only consider the stat base value.
- `Stats`: The updated values of the stats set. It also keeps the bonuses.
- `CurrentStats`: The value of the stats that have current value, for example health, mana, hunger, etc.
- `Buffs`: The currently applied buffs and its dynamic data.
- `BuffsTimer`: This timer will keep track of the buffs remaining time and will fire an event to remove the one whose duration has expired.
- `NextShortestBuff`: The buff with the shortest remaining time, it will be the next one to be removed when **BuffsTimer** expires.
- `Effects`: The currently applied effects and its dynamic data.
- `EffectsTimer`: This timer will keep track of the effects remaining tick time and will fire an event to remove and tick them when needed.
- `NextEffectToTick`: The effect with the shortest remaining tick time, which will get its ticks count increased by 1 when **EffectsTimer** expires.

#### Functions

#### Stats

- `GetStatValue`: Returns the final value of the stat passed as input, with all the bonuses applied and calculated. If that stat is also a current stat, you will get the proper value in the *Current Stat* output pin.
- `AddStatBonuses`: It receives flat and percentage bonuses and sums them to the **Stats** component variable, then it calls **CalculateStatsFinalValues**.
- `RemoveStatBonuses`: Same as the previous one, but it will substract the bonuses instead of summing them.
- `CalculateStatsFinalValues`: Recalculates the values of the stats it receives as input. It will also take care of the current stat values to clamp them properly.
- `ModifyCurrentStatValue`: Modifies the stat passed as input with a value. It will make sure to clamp it between 0 and the final value that the stat has at the moment (it would be a problem if you can have more health than the maximum health).
- `AddStatsPerLevel`: It wil be called by **BPC_Leveling** if the owning actor implements leveling system and it levels up, adding the values you have set up in the **StatsPerLevel** on **StatsTable**.
- `IsCurrentStat`: It will return true if the stat passed as input is a current stat.
- `GetStatsData`: Used to get the data in **StatsTable** with the row you have set up in the variable **Table-DataRow**.

### Buffs

- `ApplyBuff`: Applies the buff passed as input. If the buff is already applied, it will just remove the current one to apply the new one.
- `RemoveBuff`: Removes the buff passed as input if it is already applied.
- `IsBuffActive?`: Returns true whether a buff is applied.
- `UpdateBuffsRemainingTime`: It will update the remaining time of each applied buff.
- `GetShortestBuffTime`: When the buffs remaining time is updated, this function will return the buff with the shortest remaining time, that is, the next buff to be removed when the **BuffsTimer** expires.
- `IsBuffPermanent?`: Returns true if buff is applied and permanent.

### Effects

- `ApplyEffect`: Applies the buff passed as input. If the effect is already applied, it will just remove the current one to apply the new one.
- `RemoveEffect`: Removes the effect passed as input.
- `IsEffectActive?`: Returns true whether an effect is applied.
- `UpdateEffectRemainingIntervalTime`: It will update the remaining tick time of each applied effect.
- `IncrementEffectTicks`: When an effect tick is executed, this function will increment its ticks count and if that updated count is equal to the ticks it was going to do, it will call **RemoveEffect** (except when its permanent).
- `GetShortestEffectTime`: When the effects remaining tick time is updated, this function will return the effect with the shortest remaining tick time, that is, the next effect to tick when the **EffectsTimer** expires.
- `IsEffectPermanent?`: Returns true if effect is applied and permanent.

### Dispatchers

The component has a set of dispatchers that will be called when certain events happen, so you can override them in the actor that owns the component. The names are very descriptive so im just going to mention them:

- `OnBuffApplied`
- `OnBuffRemoved`
- `OnEffectApplied`
- `OnEffectRemoved`
- `OnEffectTick`
- `OnStatFinalValueChanged`
- `OnStatCurrentValueChanged`

## 1.5 Attributes

In this section i will follow the same structure as in the *Stats* section, as the process is very similar. First you will learn how to create your own attributes and also your own sets of them. Then i will explain the `BPC_Attributes` to make sure you understand its functionality.

### 1.5.1 Adding new attributes

Adding new attributes, as with stats, is very easy:

1. Go to `e_Attribute` under the *Enums* folder and add your new attribute.
2. Then in your widget that displays all the attributes, add a new `WB_Attribute`, select it and you will see in the *Details* panel that it has some exposed variables. Fill the enum variable with your new attribute and configure the rest of variables as you want.
3. Make sure you add that new widget to the widgets that the component has to refresh with the **AddCustomWidgets** function.

**Tip:** To remove an attribute the process is exactly the inverse, just remove the widget and the entry in the enum.

### 1.5.2 Creating sets of attributes

Once you have created your desired attributes, the process is the same as with stats:

1. Go to *DT_Attributes* and create a new row, you can call it *Warrior* for example.
2. Then just fill the attributes you want to have for that particular set.
3. Go to your class that has the attributes component you want to load that data (lets suppose you have a *BP_Warrior* class).
4. Click on the attributes component and in the *Details* panel you will see the `AttributesTableRow` variable under the *Settings* category, just put the row name you created before (in this case it would be *Warrior*).

**Note:** I have used `DT_Attributes` as the datatable for the example, if you have created your own one then its perfect, just make sure you put that table in the `AttributesTable` of the component.

**Done!** Now when your *BP_Warrior* class is initialized, it will load the set of attributes you have configured in the datatable (remember it will only happen if there is no previous saved data).

### 1.5.3 BPC_Attributes

#### Variables

- `Attributes`: The updated values of the attributes set. It also keeps the bonuses.
- `AvailableAttributePoints`: The attribute points that the owning actor can spend on improving attributes.
- `SpentAttributePoints`: The attribute points that the owning actor has spent already.

#### Functions

- `GetAttributeValue`: Returns the final value of the attribute passed as input, with all the bonuses applied and calculated.
- `AddAttributeBonuses`: It receives flat and percentage bonuses and sums them to the **Attributes** component variable, then it calls **CalculateAttributesFinalValues** to recalculate the final attributes values.

- `RemoveAttributeBonuses`: Same as the previous one, but it will substract the bonuses instead of summing them.
- `SpendAttributePoints`: It will substract the points passed as input to the **AvailableAttributePoints** variable, then it will call **AddPointsToAttribute**.
- `AddPointsToAttribute`: Adds the points passed as input to the attribute base value, then it recalculates the final value and also recalculates the stats bonuses this attribute can have.
- `ReceiveAttributePoints`: Just sum the points passed as input to the **AvailableAttributePoints** variable.
- `CalculateAttributesFinalValues`: Recalculates the final values of the attributes it receives as input.
- `AddAttributesPerLevel`: It will be called by **BPC_Leveling** if the owning actor implements leveling system and it levels up, adding the values you have set up in the **AttributesPerLevel** on **AttributesTable**.
- `ResetAttributes`: It resets the current attributes to the default attributes in the datatable row assigned. It will just reset and recalculate the base values. Bonuses will be still applied until they are removed by other functions.
- `AddAttributeBonusesToStats`: When an attribute is modified, this function will recalculate the stats bonuses if the attribute has any bonuses on stats.
- `GetAttributesData`: Used to get the data in **AttributesTable** with the row you have set up in the variable **AttributesDataRow**.

#### Dispatchers

- `OnAttributePointsReceived`
- `OnAttributesResetted`
- `OnAttributeFinalValueChanged`
- `OnAttributePointsSpent`

## 1.6 Leveling

This section will cover all the leveling system related functionality. In this case i will explain you how to create your own leveling profiles and your own experience formulas if you have a formula based experience assignation to each of your levels. For example you can have one leveling profile depending on your character type or the difficulty mode of your game. As all the leveling functionality is implemented in `BPC_Leveling`, you have a detailed explanation of its variables, functions and dispatchers at the end of this section.

### 1.6.1 Creating new leveling profiles

This process is also datatable oriented, which makes the process very easy to manage:

1. Go to *DT_Leveling* and create a new row name (lets call it *Test* for example).
2. Fill the leveling settings you want to have for that profile.
3. Go to your class that has the leveling component you want to load that data.
4. Click on the leveling component and in the *Details* panel, you will see the `LevelingTableRow` variable under the *Settings* category, just put the row name you created before (in this case it would be *Test*).

**Note:** If you dont use a formula, make sure you fill the `NeededExperiencePerLevel` variable manually from `StartingLevel` to `MaxLevel` in the datatable.

### 1.6.2 Adding new experience formulas

This process has 2 steps: creating the formula itself and its calculus implementation.

1. Go to `e_ExperienceFormulaType` under the *Enums* folder and add your new formula name.
2. In `BPC_Leveling`, you will see there is a function called **ApplyExperienceFormula**, just open it and now you will have a new pin available in the *Switch* node. You just have to implement your calculations there as you see in the examples i have provided. The function will give you the level that the experience is calculated for.

**Tip:** Unreal Engine has a dedicated node for math expressions as you have seen in the examples, if you want to get more information about it here you have a link to the official documentation.

### 1.6.3 BPC_Leveling

#### Variables

- `StartingLevel`: The actor starting level.
- `MaxLevel`: Maximum level the actor can reach. When reached, it wont accept any experience points.
- `CurrentLevel`: The current actor level.
- `CurrentExperience`: The current actor experience.
- `NeededExperience`: The experience that the actor needs to reach next level.
- `TotalExperience`: The total experience that the actor has received (from **StartingLevel** to **CurrentLevel** + **CurrentExperience**)
- `NeededExperiencePerLevel`: The levels that the actor can reach and the needed experience for each one. It will be constructed by formula or by datatable as said in the *DT_Leveling* section.

#### Functions

- `ReceiveExperience`: Takes some experience and sums it to the **TotalExperience** and **CurrentExperience** variables. If this one is greater than the **NeededExperience** it will loop leveling up and updating them until **CurrentExperience** is less than the **NeededExperience**. This is because the actor can get a huge amount of experience which can cause leveling a few levels at once with just 1 call to the function.
- `UpdateNeededExperience`: Just updates **NeededExperience** with the value in **NeededExperiencePerLevel** for the new reached level.
- `LevelUp`: Increases **CurrentLevel** by 1, then it calls **AddStatsPerLevel** and **AddAttributesPerLevel** if any **BPC_Stats** or **BPC_Attributes** was found on initialization, then it gets and send the level reward to be handled by the owning actor through the **OnReceiveLevelUpReward** dispatcher.
- `ApplyExperienceFormula`: Here you implement you own formulas calculations as explained in *Adding new experience formulas*.

- `CreateNeededExperienceByFormula`: Called in the initialization when using a formula to set the experience per level. It will loop from **StartingLevel** to **MaxLevel** applying the formula for each level and setting the values in the **NeededExperiencePerLevel** variable.
- `GetLevelingData`: Used to get the data in **LevelingTable** with the row you have set up in the variable **LevelingDataRow**.

#### Dispatchers

- `OnExperienceReceived`
- `OnLevelUp`
- `OnReceiveLevelUpReward`

## 1.7 Migration

In this section im going to show you how extremely easy it is to migrate the system into your own project. Typically, the migration process is very easy, but the problem is that usually, due to assets that references other ones, you get some assets that you didnt want to have in your project, and you need to remove them manually. While developing this system, i tried my best to isolate funtionality, making the systems as much modular as possible, which reduces the dependencies between them, so when migrating you will only get the most important assets.

As you have seen, the project has two folders: `Demo` and `OpenRPG_Stats`. You have 2 options when migrating:

### 1.7.1 Migrating system only

If you want to migrate the system to implement it by your own with no demo content, just follow these steps:

1. Right click the `OpenRPG_Stats` folder and in the context menu click on the **Migrate** option. The engine will show you a list with all the files that are going to be migrated.
2. Click **Ok** and you will now have to look for the *Content* folder of your project
3. Select it and confirm the migration.

When done, the system will be completely migrated and you can start implementing it into your project!

**Note:** If you noticed, in the migration files list, the *Icons* folder of the `Demo` folder will be migrated as well, because the data tables *DT_Buffs* and *DT_Effects* are using those icons. If you dont want to migrate them because they are horrible, then before starting the migration you just need to clear the icons for those data tables and save everything.

### 1.7.2 Migrating system and demo content

If you also want the demo implementation, the process is almost the same as before. The only difference is that you need to right click the `Demo` folder, not the `OpenRPG_Stats`.

If you have any problem just take a look at the migration documentation from the Unreal Engine official docs. Once you have migrated what you want, you are ready to go to the *Integration* section.

## 1.8 Integration

In the previous section you learned how to migrate the system into your own project. Now im going to show you the necessary steps to get it integrated into your project.

If you migrated the system with the demo content, the only thing you need to do is telling your project to use the demo classes already built. You can achieve it by setting the `BP_DemoGI` as your game instance in your project settings (just search game instance and select it) and in your **GameMode** set the `BP_DemoCharacter` as the **Pawn** class. With that, you have everything ready to go!

On the other hand, if you have migrated the system with no demo content, there are more things to do, but i promise they are easy and fast (just as done in the demo content). Lets go step by step.

### 1.8.1 Widgets setup

First you will need a widget that contains all of the other widgets you have created to show your stats, attributes and/or leveling data. In your widgets just make sure you implement the proper *Interfaces* that provides the data you want to display on them. If you dont have one main widget then just simply create a new widget blueprint (lets call it `WB_MainWidget`) and add the widgets into its canvas panel. Adjust them to fit your needs and the widgets setup will be completed!

### 1.8.2 Character setup

In your character blueprint add the components you want using the green button *Add Component* at the top left side. When done, click on each component and configure the variables under the *Settings* category in the details panel (check the explanation of each variable in the *Settings Variables*). Now go to the **BeginPlay** event (or whatever event you have that is called when game initializes), then you need to create a widget and that will be the `WB_MainWidget` we created before, then save it in a variable. Now for each component you have to set them up like explained in the *Initialization Pipeline* section. Remember that now you have access to all the widgets because we saved the main one in a variable. Then if you want to put some keyboard controls thats completely up to you, for example you can set the C key to save all the data.

**Note:** Its not 100% necessary to have a `WB_MainWidget` that has all of your other widgets. If you have added some widgets directly to the viewport in your project, its ok as long as you can give the proper component a reference to the widget you want, using the **AddCustomWidgets** function.

With that done, **everything is finished and you have completely integrated the system into your project!** Easy, right? Now its your turn to create your own stats, attributes, leveling profiles, buffs, effects and everything you need to suit your game requirements ;)