

# UE5 Specifiers Detailed Explanation

---

- **Author:** Jack Fu
- **Document version:** 1.1
- **Revision Date:** 2024/10/23
- **Applicable engine version:** UE5.4
- **GitHub:** <https://github.com/fjz13/UnrealSpecifiers>
- **Zhihu:** <https://www.zhihu.com/people/fjz13>
- **Email:** [fjz13@live.cn](mailto:fjz13@live.cn)
- **Copyright statement:** This document and the sample project are provided as open-source material for free reading and learning. All rights are reserved by the author (Jack Fu), and no one else is authorized to use it for commercial purposes.
- **Warning:** The English version of this document is translated by MACHINE, which may lead to POOR QUALITY or INCORRECT INFORMATION, please read with CAUTION!"

Greetings everyone, I am Jack Fu. Inspired by the frequent questions within the Unreal Engine community about specifiers and the use of various metas, and dissatisfied with the insufficient explanations provided in the official Unreal Engine documentation, I have compiled this document. It includes detailed explanations for over 100 specifiers and more than 300 metas. This document is continuously being maintained.

For the best experience, it is recommended to view the various markdown files on GitHub or clone them to benefit from the organized directory structure and linked navigation. For those with limited internet access, a PDF version of the e-book is also available. At the beginning of the document, there is a comprehensive table of specifiers, allowing for a quick overview of their functions. For specific specifiers, please utilize the search function to locate them.

The document is bound to have its share of omissions, errors, and inadequacies. I welcome pull requests or feedback to help improve this resource and benefit the wider Unreal Engine community.

## Specifier

---

Here are the links to the tables for each specifier:

- UCLASS
- UINTERFACE
- USTRUCT
- UENUM
- UFUNCTION
- UPARAM
- UPROPERTY

# Meta

---

Here is the link to the table for metas:

- Meta

## Flags

---

Here are the links to the tables for various Flags. These are included for reference as the internal workings of specifiers involve adding and removing Flags. They are provided here for your perusal without the need to delve deeply into the purpose of each flag.

- EClassFlags
- EStructFlags
- EEnumFlags
- EFunctionFlags
- EPropertyFlags

## UCLASS(specifier)

---

### UHT

---

Name	engine module	function description	frequency of use
NoExport	UHT	Specifies that UHT should not be used to automatically generate registration code, but should only perform lexical analysis to extract metadata.	
Intrinsic	UHT	Indicates that UHT will not generate any code for this class at all, and it must be written manually.	
Interface	UHT	Indicates that this class is an interface.	
UCLASS()	UHT	The default behavior of leaving it blank is that it cannot be inherited in blueprints, variables cannot be defined in blueprints, but it retains reflection capabilities.	★★★★★
Without_UCLASS	UHT	Functions as an ordinary C++ object without reflection capabilities.	★

<b>Name</b>	<b>engine module</b>	<b>function description</b>	<b>frequency of use</b>
CustomThunkTemplates	UHT	Specifies the struct that contains the CustomThunk implementations	
CustomConstructor	UHT	Prevents the automatic generation of constructor declarations.	
CustomFieldNotify	UHT	Prevents UHT from generating code related to FieldNotify for this class.	

## Blueprint

<b>Name</b>	<b>engine module</b>	<b>function description</b>	<b>frequency of use</b>
Blueprintable	Blueprint	Can be inherited in blueprints and can also be used as a variable type implicitly	★★★★★
NotBlueprintable	Blueprint	Cannot be inherited in blueprints and its implicit use cannot be as a variable	★★★★
BlueprintType	Blueprint	Can be used as a variable type	★★★★★
NotBlueprintType	Blueprint	Cannot be used as a variable type	★★★★
Abstract	Blueprint	Designates this class as an abstract base class, which can be inherited but objects cannot be instantiated from it.	★★★★★
Const	Blueprint	Indicates that the internal properties of this class are not modifiable in blueprints; they are read-only and not writable.	★★★

<b>Name</b>	<b>engine module</b>	<b>function description</b>	<b>frequency of use</b>
ShowFunctions	Blueprint	Re-enables certain functions in the subclass's function override list.	★★
HideFunctions	Blueprint	Hides certain functions in the subclass's function override list.	★★
SparseClassDataType	Blueprint	Allows some repetitive and unchanging data of Actors to be stored in a common structure to reduce content usage	★★★
NeedsDeferredDependencyLoading	Blueprint		☠

## DIIExport

---

<b>Name</b>	<b>engine module</b>	<b>function description</b>	<b>frequency of use</b>
MinimalAPI	DIIExport	Does not export the class's functions to the DLL, only exports type information as variables.	★★★

## Category

---

<b>Name</b>	<b>engine module</b>	<b>function description</b>	<b>frequency of use</b>
ClassGroup	Category	Specifies the grouping for components in the Actor's AddComponent panel and in the blueprint's right-click menu.	★★★
ShowCategories	Category	Displays certain Category properties in the class's ClassDefaults property panel.	★★★
HideCategories	Category	Hides certain Category properties in the class's ClassDefaults property panel.	★★★★

<b>Name</b>	<b>engine module</b>	<b>function description</b>	<b>frequency of use</b>
CollapseCategories	Category	Hides all properties with Category in the class's property panel, but only for properties with multiple nested Categories.	★★
DontCollapseCategories	Category	Invalidates the CollapseCategories specifier inherited from the base class.	★★
AutoExpandCategories	Category	Specifies the Category that should be automatically expanded for this class's objects in the details panel.	★
AutoCollapseCategories	Category	The AutoCollapseCategories specifier invalidates the effect of the AutoExpandCategories specifier for categories listed on the parent class.	★
DontAutoCollapseCategories	Category	Invalidates the AutoCollapseCategories specifier for the listed categories inherited from the parent class.	★
PrioritizeCategories	Category	Prioritizes the display of the specified attribute directory at the front of the details panel.	★★★
ComponentWrapperClass	Category	Designates this class as a simple encapsulation class, ignoring the Category-related settings of the subclass.	★★
AdvancedClassDisplay	Category	Displays all properties under this class in the advanced directory by default	★★★★

## TypePicker

<b>Name</b>	<b>engine module</b>	<b>function description</b>	<b>frequency of use</b>
HideDropDown	TypePicker	Hides this class in the class selector	★★

# Development

Name	engine module	function description	frequency of use
Deprecated	Development	Indicates that this class is deprecated.	★★★
Experimental	Development	Indicates that this class is an experimental version, currently without documentation, and may be deprecated in the future.	★★★
EarlyAccessPreview	Development	Indicates that this class is a preliminary preview version, more complete than the experimental version, but not yet at the product level.	★★★

# Instance

Name	engine module	function description	frequency of use
Within	Instance	Specifies that when an object is created, it must depend on an object with the OuterClassName as its Outer.	★★★
DefaultToInstanced	Instance	Specifies that all instance properties of this class are set to UPROPERTY(instanced) by default, meaning they create new instances rather than references to objects.	★★★★
EditInlineNew	Instance	Allows objects of this class to be created inline directly in the property details panel, coordinating with the attribute's Instanced setting.	★★★★★
NotEditInlineNew	Instance	Cannot be created using the EditInline button	★

# Scene

Name	engine module	function description	frequency of use
NotPlaceable	Scene	Indicates that this Actor cannot be placed in a level	★★★

<b>Name</b>	<b>engine module</b>	<b>function description</b>	<b>frequency of use</b>
Placeable	Scene	Indicates that this Actor can be placed in a level.	★★★
ConversionRoot	Scene	Allows the Actor to be converted between itself and its subclasses in the scene editor	★

## Config

<b>Name</b>	<b>engine module</b>	<b>function description</b>	<b>frequency of use</b>
Config	Config	Specifies the name of the configuration file to which the object's values are saved in the ini configuration file.	★★★★★
PerObjectConfig	Config	In cases where there is already a config file name, specifies that values should be stored per object instance rather than per class.	★★★★★
ConfigDoNotCheckDefaults	Config	Specifies that the consistency check of the previous level's configuration values is ignored when saving configuration values.	★
DefaultConfig	Config	Specifies that the configuration file level to save to is Project/Config/DefaultXXX.ini.	★★★
GlobalUserConfig	Config	Specifies that the configuration file level to save to is Engine/Config/UserXXX.ini for global user settings.	★★★
ProjectUserConfig	Config	Specifies that the configuration file level to save to is Project/Config/UserXXX.ini for project user settings.	★★★
EditorConfig	Config	Used to save information in the editor state.	★

# Serialization

Name	engine module	function description	frequency of use
Transient	Serialization	Specifies that all objects of this class are skipped during serialization.	★★★
NonTransient	Serialization	Invalidates the Transient specifier inherited from the base class.	★★★
Optional	Serialization	Mark this class's objects as optional, allowing them to be ignored during Cooking.	★
MatchedSerializers	Serialization	Specifies that the class supports text structure serialization	💀

# UINTERFACE(specifier)

## DIIExport

Name	Engine Module	Function Description	Usage Frequency
MinimalAPI	DIIExport	Specifies that the UInterface object should not be exported to other modules	Star

# Blueprint

Name	Engine Module	Function Description	Usage Frequency
Blueprintable	Blueprint	Can be implemented in Blueprints	Five stars
NotBlueprintable	Blueprint	Indicates that it cannot be implemented in Blueprints	Three stars
ConversionRoot	Blueprint	Sets IsConversionRoot metadata flag for this interface.	Skull emoticon

# USTRUCT(specifier)

## UHT

Name	Engine module	Function description	Common usage
NoExport	UHT	Specifies that UHT should not be used	Star

Name	Engine module	Function description	Common usage
		to automatically generate registration code, but rather for lexical analysis to extract metadata only.	
Atomic	UHT	Indicates that the structure always outputs all its properties during serialization, rather than just the changed ones.	Star
IsAlwaysAccessible	UHT	Ensures that UHT can always access the declaration of this structure when generating files; otherwise, a mirrored structure definition must be created in gen.cpp	Skull
HasDefaults	UHT	Indicates that the fields of this structure have default values, allowing functions to provide default values when the structure is used as a parameter or return value.	Skull
HasNoOpConstructor	UHT	Signifies that the structure possesses a ForceInit constructor, which can be invoked to initialize when it is returned as a BP function's value	Skull
IsCoreType	UHT	Indicates that the structure is a core class, and UHT does not require a forward declaration when using it.	Skull

## Blueprint

---

Name	Engine module	Function description	Common usage
BlueprintType	Blueprint	Allows this structure to be declared as variables in Blueprints	★★★★★
BlueprintInternalUseOnly	Blueprint	Prohibits the definition of new BP variables but allows the structure to be exposed as member variables of other classes and passed as variables	★★

Name	Engine module	Function description	Common usage
BlueprintInternalUseOnlyHierarchical	Blueprint	Building on BlueprintInternalUseOnly, this adds a restriction that subclasses are also forbidden from defining new BP variables.	★

## Serialization

Name	Engine module	Function description	Common usage
immutable	Serialization	Immutable is only legal in Object.h and is being phased out, do not use on new structs!	💀

## UENUM(specifier)

### Trait

Name	Engine Module	Function Description	Usage Frequency
Flags	Trait	Use the value of this enumeration as a flag to concatenate and output strings.	★★★★★

## Blueprint

Name	Engine Module	Function Description	Usage Frequency
BlueprintType	Blueprint	Can be used as a blueprint variable	★★★★★

## UFUNCTION(specifier)

### Editor

Name	Engine Module	Function Description	Usage Frequency
Category	Editor	Specify category grouping for the function in the right-click menu of the blueprint, allowing for multi-level nesting	★★★★★

Name	Engine Module	Function Description	Usage Frequency
CallInEditor	Editor	This function can be invoked as a button on the property details panel.	★★★★★

## Blueprint

Name	Engine Module	Function Description	Usage Frequency
BlueprintCallable	Blueprint	Exposed to the blueprint for invocation	★★★★★
BlueprintPure	Blueprint	Designated as a pure function, typically used in Get functions to return values.	★★★★★
BlueprintImplementableEvent	Blueprint	Specify a function call point that can be overloaded within the blueprint.	★★★★★
BlueprintNativeEvent	Blueprint	Implementation can be overridden in Blueprints, with a default implementation also provided in C++.	★★★★★
BlueprintGetter	Blueprint	Specify this function as a custom Get function for the property.	★★
BlueprintSetter	Blueprint	Specify this function as a custom Set function for the property.	★★

## Behavior

Name	Engine Module	Function Description	Usage Frequency
Exec	Behavior	Register a function within a specific class as a console command, enabling parameter acceptance.	★★★
SealedEvent	Behavior	This function cannot be overridden in subclasses. Use the SealedEvent keyword for events only. For non-event functions, declare them as static or final to seal them.	

# Network

Name	Engine Module	Function Description	Usage Frequency
BlueprintAuthorityOnly	Network	This function is executable only on clients with network permissions.	★★★
BlueprintCosmetic	Network	Decorative function; not executable on DS.	★★★
Client	Network	Execute a RPC function on a Client-owned Actor (PlayerController or Pawn). This function runs only on the client. The corresponding implementation function will have an _Implementation suffix.	★★★★★
Server	Network	Execute a RPC function on a Client-owned Actor (PlayerController or Pawn). This function runs only on the server. The corresponding implementation function will have an _Implementation suffix	★★★★★
NetMulticast	Network	Define a multicast RPC function to be executed on both the server and the client. The corresponding implementation function will have an _Implementation suffix.	★★★★★
Reliable	Network	Specify an RPC function as "reliable," which will be resent upon network errors to ensure delivery. Typically used in critical logic functions.	★★★★★
Unreliable	Network	Specify an RPC function as "unreliable," which will be discarded upon network errors. Typically used in functions that express communication effects, where omissions are permissible.	★★★★★
WithValidation	Network	Specify that an RPC function must be verified before execution, and it can only be executed if the verification is successful.	★★★★★
ServiceRequest	Network	This function is an RPC (Remote Procedure Call) service request. RPC service request	💀

Name	Engine Module	Function Description	Usage Frequency
ServiceResponse	Network	This function is an RPC service response. RPC service reply	⌚⌚

## UHT

Name	Engine Module	Function Description	Usage Frequency
BlueprintInternalUseOnly	Blueprint, UHT	Indicate that this function should not be exposed to end users. Internally called by the blueprint and not exposed to users.	★★★
CustomThunk	UHT	Specify that UHT will not generate an auxiliary function for blueprint invocation of this function, and it requires user-defined implementation.	★★★
Variadic	Blueprint, UHT	Mark a function capable of accepting multiple parameters of any type (including input/output)	★★★
FieldNotify	UHT	Create a FieldNotify binding point for this function.	★★★

## UPARAM(specifier)

### Blueprint

Name	Function Description	Engine Module	Usage Frequency
DisplayName	Change the display name of function parameters on blueprint nodes	Blueprint, Parameter	★★★★★
ref	Convert function parameters to reference types	Blueprint, Parameter	★★★★★
Const	Indicate that function parameters are not modifiable	Blueprint, Parameter	★
Required	Ensure that the function parameter node is connected and provides a value	Blueprint, Parameter	★★

# Network

Name	Function Description	Engine Module	Usage Frequency
NotReplicated		Blueprint, Network, Parameter	

# UPROPERTY(specifier)

## Serialization

Name	Engine module	Functional description	Frequency of use
Export	Serialization	When exporting an Asset, this class's objects should export the internal attribute values rather than the object paths.	★
SaveGame	Serialization	When archiving SaveGame, only serialize attributes marked with SaveGame, not others.	★★★★★
SkipSerialization	Serialization	Skip this attribute during binary serialization, but it can still be exported in ExportText.	★★★
TextExportTransient	Serialization	Ignore this attribute when exporting in .COPY format during ExportText.	★
Transient	Serialization	This attribute is not serialized and will be initialized with zeros.	★★★★★
DuplicateTransient	Serialization	Ignore this attribute when copying or exporting the object in COPY format.	★★
NonPIEDuplicateTransient	Serialization	Ignore this attribute when copying the object, unless it is in PIE mode.	★

# Sequencer

Name	Engine module	Functional description	Frequency of use
Interp	Sequencer	Specify that this attribute's value can be exposed for editing in the timeline, used in regular Timeline or UMG animations.	★★★

# Network

Name	Engine module	Functional description	Frequency of use
Replicated	Network	Specify that this property should be replicated over the network.	★★★★★
ReplicatedUsing	Network	Specify a notification callback function to execute after the property is updated over the network.	★★★★★
NotReplicated	Network	Skip replication for this attribute. This applies only to structure members and parameters in service request functions.	★★★
RepRetry	Network	Applies only to structure attributes. If this attribute cannot be fully sent (for example, Object references cannot yet be serialized over the network), the copy attempt will be retried. This is the default for simple references but not optimal for structures, which can incur bandwidth overhead. Therefore, it is disabled by default until this tag is specified.	💀

# UHT

Name	Engine module	Functional description	Frequency of use
FieldNotify	MVVM, UHT	After enabling the MVVM plugin, make this property support FieldNotify.	★★★

## Instance

Name	Engine module	Functional description	Frequency of use
Instanced	Instance	Specify that editing this object's properties should create a new instance as a child, rather than searching for an existing object reference.	★★★

## Editor

Name	Engine module	Functional description	Frequency of use
NonTransactional	Editor	Changes to this property will not be included in the editor's Undo/Redo commands.	★★

## DetailsPanel

Name	Engine module	Functional description	Frequency of use
Category	DetailsPanel, Editor	Specify the category of this attribute, using the	operator to define nested categories.
SimpleDisplay	DetailsPanel, Editor	Visible directly in the details panel without collapsing into the advanced section.	★★★
AdvancedDisplay	DetailsPanel, Editor	Collapsed into the advanced section; needs to be manually expanded. Generally used for less frequently used attributes.	★★★★★
EditAnywhere	DetailsPanel, Editor	Editable in both the default and instance detail panels	★★★★★
EditDefaultsOnly	DetailsPanel, Editor	Can only be edited in the default value panel	★★★★★
EditInstanceOnly	DetailsPanel, Editor	This property can only be edited in the instance's detail panel	★★★★★
VisibleAnywhere	DetailsPanel, Editor	Visible in both the default and instance detail panels but not editable	★★★★★

Name	Engine module	Functional description	Frequency of use
VisibleDefaultsOnly	DetailsPanel, Editor	Visible but not editable in the default details panel	★★★★★
VisibleInstanceOnly	DetailsPanel, Editor	Visible but not editable in the instance details panel	★★★★★
EditFixedSize	DetailsPanel, Editor	Changing the number of elements in this container is not allowed in the details panel.	★★★
NoClear	DetailsPanel, Editor	The Clear button will not appear in the editing options for this attribute, and setting it to null is not allowed.	★★★

## Config

Name	Engine module	Functional description	Frequency of use
Config	Config	Specifying this property as a configuration attribute allows it to be serialized and stored, as well as read and modified, within an INI file (the file path is designated via the 'config' tag within the 'uclass').	★★★
GlobalConfig	Config	Similar to Config, specifying that this attribute can be read and written as a configuration to the ini file, but only the base class values in the configuration file will be read, not the subclass values.	★★★

## Blueprint

Name	Engine module	Functional description	Frequency of use
BlueprintAuthorityOnly	Blueprint, Network	This attribute can only be bound to events with BlueprintAuthorityOnly, allowing the multicast delegate to accept only server-side events	★★★
BlueprintReadWrite	Blueprint	The property can be accessed and manipulated either by reading from or writing to the Blueprint.	★★★★★

Name	Engine module	Functional description	Frequency of use
BlueprintReadOnly	Blueprint	This attribute can be read by Blueprints but cannot be modified.	★★★★★
BlueprintGetter	Blueprint	Define a custom Get function for this attribute to read its value.	★★★
Getter	Blueprint	Add a C++ Get function to this attribute, applicable only at the C++ level.	★★★
Setter	Blueprint	Add a C++ Set function to this attribute, applicable only at the C++ level.	★★★
BlueprintSetter	Blueprint	Use a custom set function to read this attribute.	★★★
BlueprintCallable	Blueprint	This multicast delegate can be invoked in Blueprints	★★★
BlueprintAssignable	Blueprint	This multicast delegate can have events bound to it in Blueprints	★★★

## Behavior

Name	Engine module	Functional description	Frequency of use
Localized	Behavior	The value of this attribute will have a defined localized value, commonly used for strings. Implies ReadOnly. This value has a localized equivalent and is most commonly tagged on strings	💀
Native	Behavior	The property is local: C++ code is responsible for serializing it and exposing it to garbage collection.	💀

## Asset

Name	Engine module	Functional description	Frequency of use
AssetRegistrySearchable	Asset	Mark this attribute as a Tag and Value for AssetRegistry to filter and search assets	★★★

# Meta = (Metadata)

## Actor

Name	engine module	Function description	Frequency
ChildCanTick	Actor	Markers allow their Blueprint subclasses to accept responses to Tick events	★★★
ChildCannotTick	Actor	Used for Actor or ActorComponent subclasses. The flag indicates that its blueprint subclass should not accept and respond to Tick events, even if the parent class can Tick	★★★

## AnimationGraph

Name	engine module	Function description	Frequency
AnimNotifyBoneName	AnimationGraph	Make the FName property under UAnimNotify or UAnimNotifyState serve as BoneName.	★
AnimBlueprintFunction	AnimationGraph	Indicates that it is an internal pure stub function within the animation blueprint, set only during the compilation of the animation blueprint	ՃՃ
CustomizeProperty	AnimationGraph	Used on the member properties of FAnimNode, instructing the editor not to generate a default Details panel control for it. The corresponding edit control will be custom-created within DetailsCustomization later.	★
AnimNotifyExpand	AnimationGraph	Directly expand the properties under UAnimNotify or UAnimNotifyState into the details panel.	ՃՃ
OnEvaluate	AnimationGraph		ՃՃ

Name	engine module	Function description	Frequency
FoldProperty	AnimationGraph	In the animation blueprint, make a certain attribute of the animation node a FoldProperty.	★
BlueprintCompilerGeneratedDefaults	AnimationGraph	Specifies that the value of this property is generated by the compiler, so it does not need to be copied after compilation, which can enhance some compilation performance.	★★
CustomWidget	AnimationGraph		★★
AllowedParamType	AnimationGraph		★★
PinShownByDefault	AnimationGraph	In the animation blueprint, expose a certain attribute of the animation node as a pin from the outset, but it can also be modified.	★★★
AnimGetter	AnimationGraph	Specifies that the function of UAnimInstance and its subclasses should be an AnimGetter function.	★★★
GetterContext	AnimationGraph	Continues to limit where the AnimGetter function can be used. If not specified, it is used by default.	★

## Asset

Name	engine module	Function description	Frequency
RequiredAssetDataTags	Asset	Specify Tags on the UObject* attribute for filtering. Only those with the specified Tags can be selected.	★
DisallowAssetDataTags	Asset	Specify Tags on the UObject* attribute for filtering. Only those without the specified Tags can be selected.	★

Name	engine module	Function description	Frequency
ForceShowEngineContent	Asset	Enforces the selection of built-in engine resources in the resource optional list of the specified UObject* attribute	★
ForceShowPluginContent	Asset	Enforces the selection of built-in resources from other plug-ins in the resource optional list of the specified UObject* attribute	☠
GetAssetFilter	Asset	Specifies a UFUNCTION to exclude filtering of optional resources with UObject* attributes.	★★★

## Blueprint

Name	engine module	Function description	Frequency
IgnoreTypePromotion	Blueprint	Marks this function as not to be included in the type promotion function library	★
Variadic	Blueprint	Specifies that the function accepts multiple parameters	★★★
ForceAsFunction	Blueprint	Forces events defined with BlueprintImplementableEvent or NativeEvent in C++ to be implemented as functions and overridden in subclasses.	★★★
CannotImplementInterfaceInBlueprint	Blueprint	Specifies that the interface cannot be implemented in the blueprint	★★★
CallInEditor	Blueprint	This function can be invoked as a button on the Actor's details panel.	★★★★★
BlueprintProtected	Blueprint	Specifies that the function or attribute can only be accessed or modified within this class and its subclasses, similar to the protected scope restriction in C++. It is not accessible from other Blueprint classes.	★★★
AllowPrivateAccess	Blueprint	Allows a C++ private property to be accessed within a Blueprint.	★★★★★
BlueprintPrivate	Blueprint	Specifies that the function or attribute can only be accessed or modified within this class, similar to the private scope restriction in C++. It is not accessible from other Blueprint classes.	★

Name	engine module	Function description	Frequency
CommutativeAssociativeBinaryOperator	Blueprint	Marks a binary operation function as supporting commutative law and associative law, adding a "+" pin on the blueprint node to allow the dynamic addition of multiple input values.	★★★★
CompactNodeTitle	Blueprint	Changes the display form of the function to a compact mode and specifies a new compact name	★★★
CustomStructureParam	Blueprint	The function parameter marked by CustomStructureParam becomes a wildcard parameter, with the pin type equal to the connected variable type.	★★★★★
DefaultToSelf	Blueprint	Used on functions to specify the default value of a parameter as the Self value	★★★★★
ExpandEnumAsExecs	Blueprint	Specifies multiple enum or bool type function parameters, automatically generating multiple corresponding input or output execution pins based on the entries, and adjusting the control flow accordingly based on different actual parameter values.	★★★★★
ExpandBoolAsExecs	Blueprint	It is an alias of ExpandEnumAsExecs and is completely equivalent to its function.	★★★★★
ArrayParm	Blueprint	Specifies a function as a function using Array<*> with a wildcard generic array element type.	★★★
ArrayTypeDependentParams	Blueprint	When the function specified by ArryParam has two or more Array parameters, the types of the specified array parameters should also be updated accordingly.	⚠️
AdvancedDisplay	Blueprint	Some parameters of the function are collapsed and not displayed. Manually expand the drop-down arrow to edit.	★★★★★
SetParam	Blueprint	Specifies a function as a function using Set with a wildcard generic element type.	★★★
MapParam	Blueprint	Specifies a function as a function using TMap< TKey, TValue > with a wildcard generic element type.	★★★
MapKeyParam	Blueprint	Specifies a function parameter as the Key of the Map, which changes accordingly based on the Key type of the actual Map parameter specified by MapParam.	★★★

Name	engine module	Function description	Frequency
MapValueParam	Blueprint	Specifies a function parameter as the Value of the Map, which changes accordingly based on the Value type of the actual Map parameter specified by MapParam.	★★★
InternalUseParam	Blueprint	Used on function calls to specify the parameter names to be hidden, and also to hide the return value. Multiple can be hidden	★★★★★
Keywords	Blueprint	Specifies a series of keywords for right-clicking to find the function within the blueprint	★★★★★
Latent	Blueprint	Indicates that the function is a deferred asynchronous operation	★★★★★
NeedsLatentFixup	Blueprint	Used on the FLatentActionInfo::Linkage attribute to inform the blueprint VM to generate jump information	★
LatentInfo	Blueprint	Used in conjunction with Latent to indicate which function parameter is the LatentInfo parameter.	★★★
LatentCallbackTarget	Blueprint	Used on the FLatentActionInfo::CallbackTarget attribute to inform the blueprint VM on which object to call the function.	★
NativeMakeFunc	Blueprint	Specifies a function using the MakeStruct icon	★
NativeBreakFunc	Blueprint	Specifies that a function uses the BreakStruct icon.	★
UnsafeDuringActorConstruction	Blueprint	Indicates that the function cannot be called within the Actor's constructor	★
BlueprintAutocast	Blueprint	Informs the blueprint system that this function is intended to support automatic conversion from type A to type B.	★
DeterminesOutputType	Blueprint	Specifies the type of a parameter as a reference type for the function to dynamically adjust the output parameter type	★★★
DynamicOutputParam	Blueprint	Together with DeterminesOutputType, specifies multiple output parameters that support dynamic types.	★★
ReturnDisplayName	Blueprint	Changes the name of the function return value, with the default being ReturnValue	★★★★★
WorldContext	Blueprint	Specifies that a parameter of the specified function automatically receives the WorldContext object to determine the current World	★★★★★

Name	engine module	Function description	Frequency
ShowWorldContextPin	Blueprint	Placed on UCLASS, specifying that all function calls within this class must display the WorldContext pin, regardless of whether it is hidden by default	
CallableWithoutWorldContext	Blueprint	Allows the function to be used without the WorldContextObject	
AutoCreateRefTerm	Blueprint	Automatically creates default values for multiple input reference parameters of the specified function when no connections are made	★★★★★
ProhibitedInterfaces	Blueprint	Lists interfaces that are incompatible with Blueprint classes, preventing their implementation	★
HiddenNode	Blueprint	Hides the specified UBTNode from the right-click menu.	★
HideFunctions	Blueprint	Does not display all functions in the specified category in the property viewer.	★★★
ExposedAsyncProxy	Blueprint	Exposes a proxy object of this class within the Async Task node.	★★★
HasDedicatedAsyncNode	Blueprint		
HideThen	Blueprint	Hides the Then pin of an Async Blueprint node	
HideSpawnParms	Blueprint	Hides certain properties in the UGameplayTask subclass inheritance chain on the blueprint asynchronous node generated by the UGameplayTask subclass.	
NotInputConfigurable	Blueprint	Prevents some UInputModifier and UInputTrigger from being configured within ProjectSettings.	★
BlueprintThreadSafe	Blueprint	Used on classes or functions to indicate that all functions within the class are thread-safe, allowing them to be called in non-game threads such as animation blueprints.	★★★
NotBlueprintThreadSafe	Blueprint	Used on a function to indicate that the function is not thread-safe	★
RestrictedToClasses	Blueprint	Limits functions under the restricted blueprint function library to be created by right-clicking within the class blueprint specified by RestrictedToClasses	★★★
DontUseGenericSpawnObject	Blueprint	Prevents the use of the Generic Create Object node in a Blueprint to instantiate objects of this class.	★

Name	engine module	Function description	Frequency
ObjectSetType	Blueprint	Specifies the object collection type for the statistics page.	★
SparseClassDataTypes	Blueprint		★★★
KismetHideOverrides	Blueprint	Lists Blueprint events that are not allowed to be overridden.	⊕⊕
BlueprintType	Blueprint	Indicates that it can be used as a blueprint variable	★★★★★
IsConversionRoot	Blueprint	Allows Actors to perform conversions between themselves and their subclasses	★★★
BlueprintInternalUseOnlyHierarchical	Blueprint	Indicates that the structure and its subclasses are not exposed to user definition and use and are only used internally within the blueprint system	★
BlueprintSetter	Blueprint	Reads using a custom set function. BlueprintReadWrite is set by default.	★★★
DisplayName	Blueprint	The naming of this node within the Blueprint will be replaced by the value provided here, rather than the code-generated naming.	★★★★★
ExposeOnSpawn	Blueprint	Exposes this property when objects are created, such as with ConstructObject or SpawnActor.	★★★★★
NativeConst	Blueprint	Specifies the const flag from C++	★
CPP_Default_XXX	Blueprint	XXX=parameter name	★★★★★
BlueprintGetter	Blueprint	Reads using a custom get function. If BlueprintSetter or BlueprintReadWrite is not set, BlueprintReadOnly is set by default.	★★★
IsBlueprintBase	Blueprint	Indicates whether this class is an acceptable base class for creating Blueprints, similar to the UCLASS specifier, Blueprintable, or 'NotBlueprintable'.	★★★★★
BlueprintInternalUseOnly	Blueprint	Indicates that this element is used internally by the blueprint system and is not exposed for direct definition or use by the user.	★★★

# Component

Name	engine module	Function description	Frequency
UseComponentPicker	Component	Used on the ComponentReference property to display the Components under the Actor in the selector list for easy selection.	★
AllowAnyActor	Component	Used on the ComponentReference property, in the case of UseComponentPicker, to expand the component picker to other components under other Actors in the scene.	★
BlueprintSpawnableComponent	Component	Allows this component to appear in the Add component panel in the Actor Blueprint.	★★★

# Config

Name	engine module	Function description	Frequency
ConsoleVariable	Config	Synchronizes the value of a Config property with a console variable of the same name.	★★★★★
EditorConfig	Config	Saves the editor configuration	★★★
ConfigHierarchyEditable	Config	Allows a property to be configurable across all levels of Config.	★★★
ConfigRestartRequired	Config	Pops up a dialog box to restart the editor after the properties are changed in the settings.	★★★

## Container

Name	engine module	Function description	Frequency
ReadOnlyKeys	Container	Makes the Key of the TMap attribute non-editable.	★
ArraySizeEnum	Container	Provides an enumeration for a fixed array so that array elements are indexed and displayed according to the enumeration value.	★★★
TitleProperty	Container	Specifies the structure member attribute content in the structure array as the display title of the structure array element.	★
EditFixedOrder	Container	Makes the elements of the array unable to be reordered by dragging.	★
NoElementDuplicate	Container	Removes the Duplicate menu item button from the data item in the TArray attribute.	★

## Debug

Name	engine module	Function description	Frequency
DebugTreeLeaf	Debug	Prevents BlueprintDebugger from expanding the properties of this class to enhance the performance of the debugger within the editor	★

## DetailsPanel

Name	engine module	Function description	Frequency
HideInDetailPanel	DetailsPanel	Hides the dynamic multicast delegate property in the Actor's event panel.	★
DisplayAfter	DetailsPanel	Ensures that this attribute is displayed after the specified attribute.	★★★

Name	engine module	Function description	Frequency
EditCondition	DetailsPanel	Specifies another attribute or expression as a condition for whether this attribute can be edited.	★★★★★
EditConditionHides	DetailsPanel	If there is already an EditCondition, hides this attribute if the EditCondition is not met.	★★★★★
InlineEditConditionToggle	DetailsPanel	When used as an EditCondition, makes this bool attribute inline within the other attribute's row as a radio button, rather than as a separate edit row.	★★★★★
HideEditConditionToggle	DetailsPanel	Used on properties that use EditCondition, indicating that the attribute does not want the attributes used by its EditCondition to be hidden.	★★★★★
DisplayPriority	DetailsPanel	Specifies the display order priority of this attribute in the details panel. The lower the value, the higher the priority.	★★★
AdvancedClassDisplay	DetailsPanel	Specifies that variables of this type are displayed in the advanced display	★★★
bShowOnlyWhenTrue	DetailsPanel	Determines whether this attribute is displayed based on the field value in the editor config configuration file.	★
PrioritizeCategories	DetailsPanel	Prioritizes the display of the specified attribute directory at the forefront	★★★
AutoExpandCategories	DetailsPanel	Automatically expands the property directory within the specified class	★★★

Name	engine module	Function description	Frequency
AutoCollapseCategories	DetailsPanel	Automatically collapses the attribute directory within the specified class	★★★
ClassGroupNames	DetailsPanel	Specifies the name of the ClassGroup	★★★
MaxPropertyDepth	DetailsPanel	Specifies the number of layers at which the object or structure is expanded in the details panel.	★
DeprecatedNode	DetailsPanel	Used for BehaviorTreeNode or EnvQueryNode, indicating that this class has been deprecated, with a red error displayed in the editor and an error ToolTip prompt	★
UsesHierarchy	DetailsPanel	Describes classes that use hierarchical data, used to instantiate the hierarchical editing functionality within the Details panel.	💀
IgnoreCategoryKeywordsInSubclasses	DetailsPanel	Used to cause the first subclass of a class to ignore all inherited ShowCategories and HideCategories specifiers.	★
NoResetToDefault	DetailsPanel	Disables and hides the "reset" functionality of properties on the details panel.	★★★
ReapplyCondition	DetailsPanel	// Properties that have a ReapplyCondition should be disabled behind the specified property when in reapply mode	★
HideBehind	DetailsPanel	This attribute is only displayed when the specified attribute is true or not empty	★

Name	engine module	Function description	Frequency
Category	DetailsPanel	Specifies the category of the attribute within the details panel	★★★★★
HideCategories	DetailsPanel	Hidden categories	★★★
ShowCategories	DetailsPanel	Show category	☠
EditInline	DetailsPanel	Creates an instance of the object property as a child object.	★★★
NoEditInline	DetailsPanel	Object properties pointing to an UObject instance whose class is marked editinline will not show their properties inline in property windows. Useful for getting actor components to appear in the component tree but not inline in the root actor details panel.	☠
AllowEditInlineCustomization	DetailsPanel	Allows object properties that support EditInline to customize the property details panel for editing data within the object.	★
ForceInlineRow	DetailsPanel	Forces the structure key and other values in the TMap attribute to be displayed on the same line	★

## Development

---

Name	engine module	Function description	Frequency
DeprecatedProperty	Development	Marks as deprecated, with a warning triggered in blueprints referencing this property	★
Deprecated	Development	Specifies the engine version number for which this element is to be deprecated.	★

Name	engine module	Function description	Frequency
DevelopmentOnly	Development	1 Making a function DevelopmentOnly means it will only run in Development mode. Useful for features like debug output, but skipped in the final release.	★
DeprecationMessage	Development	3 Define deprecated messages	★
DeprecatedFunction	Development	5 Indicates that a function has been deprecated	★
Comment	Development	7 Used to record the content of comments	★★★
FriendlyName	Development	9 Is it the same as DisplayName?	⌚
DevelopmentStatus	Development	11 Indicates the development status	★
ToolTip	Development	13 Provide a hint text in Meta, overriding the text in the code comments	★★★
ShortTooltip	Development	15 Provide a more concise version of the tooltip text, for example when displaying type selectors	⌚

## Enum

Name	17 Engine module	18 Function description	19 Frequency of use
EnumDisplayNameFn	Enum	20 Provide function callbacks with custom names for enum fields under Runtime	21 ★
Bitflags	Enum	22 Set an enumeration to support bit-marked assignment so that it can be identified as a BitMask in the blueprint	★★★★★
UseEnumValuesAsMaskValuesInEditor	Enum	24 Specifies that the enumeration value is already the shifted value, not the index of the bit mark.	25 ★
Spacer	Enum	26 Hide a value of UENUM	★★★★★
ValidEnumValues	Enum	28 Specifies optional enum value options on enum property values	★★★
InvalidEnumValues	Enum	30 Specifies non-selectable enumeration value options on the enumeration property value to exclude some options	★★★

<b>Name</b>	<b>17 Engine module</b>	<b>18 Function description</b>	<b>19 Frequency of use</b>
GetRestrictedEnumValues	Enum	32 Specifies a function to specify which enumeration options are disabled for enumeration property values	★★★
EnumValueDisplayNameOverrides	Enum	34 Change the display name on an enumeration property value	35 ★
Enum	Enum	36 Specify a String with the name of the value in the enumeration as an option	★★★
<a href="#">DisplayName</a>	Enum	38 Change the display name of enumeration values	★★★★★
<a href="#">Hidden</a>	Enum	40 Hide a value of UENUM	★★★★★
DisplayValue	Enum	Enum /Script/Engine.AnimPhysCollisionType	●●●●●
Grouping	Enum	Enum /Script/Engine.EAlphaBlendOption	●●●●●
TraceQuery	Enum	Enum /Script/Engine.ECollisionChannel	●●●●●
Bitmask	Enum	45 Set an attribute using Bitmask assignment	★★★★★
BitmaskEnum	Enum	47 Enumeration name adopted after using bit flag	★★★★★

## FieldNotify

<b>Name</b>	<b>49 Engine module</b>	<b>50 Function description</b>	<b>51 Frequency of use</b>
FieldNotifyInterfaceParam	FieldNotify	52 Specify a parameter of the function to provide FieldNotify's ViewModel information.	★★★

## GAS

<b>Name</b>	<b>54 Engine module</b>	<b>55 Function description</b>	<b>56 Frequency of use</b>
HideInDetailsView	GAS	57 Hide the attributes in the UAttributeSet subclass in the FGameplayAttribute option list.	★★★
SystemGameplayAttribute	GAS	59 Expose the attributes in the UAbilitySystemComponent subclass to the FGameplayAttribute option box.	★★★

<b>Name</b>	<b>54 Engine module</b>	<b>55 Function description</b>	<b>56 Frequency of use</b>
HideFromModifiers	GAS	61 An attribute under the specified AttributeSet does not appear in the Attribute selection of Modifiers under GameplayEffect.	★★★

## Material

<b>Name</b>	<b>63 Engine module</b>	<b>64 Function description</b>	<b>65 Frequency of use</b>
MaterialParameterCollectionFunction	Material	66 Specify that this function is used to operate UMaterialParameterCollection to support the extraction and verification of ParameterName	★★★
MaterialNewHSLGenerator	Material	68 Identifies this UMaterialExpression as a node using the new HLSL generator, currently hidden in the material blueprint context menu.	★
ShowAsInputPin	Material	70 Make some basic type attributes in UMaterialExpression become pins on the material node.	★★★
MaterialControlFlow	Material	72 Identifies this UMaterialExpression as a control flow node and is currently hidden in the material blueprint right-click menu.	★
OverridingInputProperty	Material	74 Specify other FExpressionInput properties to be overridden by this float in UMaterialExpression.	★★★
RequiredInput	Material	76 Specify in UMaterialExpression whether the FExpressionInput property requires input and the pin appears white or gray.	?
Private	Material	78 Marks this UMaterialExpression as a private node and is currently hidden in the material blueprint right-click menu.	★

# Niagara

<b>Name</b>	<b>80 Engine module</b>	<b>81 Function description</b>	<b>82 Frequency of use</b>
NiagaraClearEachFrame	Niagara	ScriptStruct /Script/Niagara.NiagaraSpawnInfo	
NiagaralInternalType	Niagara	84 Specifies that the structure's type is Niagara's internal type.	

## Numeric

<b>Name</b>	<b>86 Engine module</b>	<b>87 Function description</b>	<b>88 Frequency of use</b>
CtrlMultiplier	Numeric	89 Specifies the ratio at which the value of the numerical input box changes when the mouse wheel scrolls and the mouse drags when Ctrl is pressed.	90 ★
ShiftMultiplier	Numeric	91 Specifies the rate at which the value of the numerical input box changes when the mouse wheel scrolls and the mouse drags when Shift is pressed.	92 ★
SliderExponent	Numeric	93 Specifies the exponential distribution of changes in scroll bar dragging on the numeric input box	★★★★★
Multiple	Numeric	95 The value of the specified number must be an integer multiple of the value provided by Mutliple.	★★★
Units	Numeric	97 Set the unit of the attribute value, and support dynamically changing the displayed unit in real time according to different values.	★★★

<b>Name</b>	<b>86 Engine module</b>	<b>87 Function description</b>	<b>88 Frequency of use</b>
ForceUnits	Numeric	99 The unit of the fixed attribute value remains unchanged and the display unit is not dynamically adjusted based on the value.	★★★
Delta	Numeric	101 Set the amplitude of the numerical input box value change to a multiple of Delta	★★★
LinearDeltaSensitivity	Numeric	103 After setting Delta, further set the numerical input box to change linearly and the sensitivity of the change (the larger the value, the less sensitive it is)	★★★
UIMin	Numeric	105 Specifies the minimum range value for dragging the scroll bar on the numeric input box	★★★★★
UIMax	Numeric	107 Specifies the maximum range value for dragging the scroll bar on the numeric input box	★★★★★
ClampMin	Numeric	109 Specifies the minimum value actually accepted by the numeric input box	★★★★★
ClampMax	Numeric	111 Specifies the maximum value actually accepted by the number input box	★★★★★
SupportDynamicSliderMinValue	Numeric	113 Support the minimum range value of the scroll bar on the numeric input box to be dynamically changed when Alt is pressed	★

<b>Name</b>	<b>86 Engine module</b>	<b>87 Function description</b>	<b>88 Frequency of use</b>
SupportDynamicSlider.MaxValue	Numeric	115 Supports the maximum range value of the scroll bar on the numeric input box to be dynamically changed when Alt is pressed	★
ArrayClamp	Numeric	117 The value of the qualified integer attribute must be within the legal subscript range of the specified array, [0, ArrayClamp.Size () -1 ]	★★★
HideAlphaChannel	Numeric	119 Make the FColor or FLinearColor property hide the alpha channel when editing.	★★★
AllowPreserveRatio	Numeric	121 Add a ratio lock to the FVector property on the details panel.	★★★
NoSpinbox	Numeric	123 Disable the default drag-and-drop and scroll wheel UI editing functions for numerical attributes. Numeric attributes include int series and float series.	124 ★
sRGB	Numeric	125 Make the FColor or FLinearColor property use sRGB mode when editing.	
WheelStep	Numeric	127 Specifies the change value produced by scrolling the mouse wheel up and down on the numeric input box	★★★
InlineColorPicker	Numeric	129 Make the FColor or FLinearColor property directly inline with a color picker when editing.	130 ★
ShowNormalize	Numeric	131 Makes the FVector variable appear with a normalize button in the details panel.	★★★

<b>Name</b>	<b>86 Engine module</b>	<b>87 Function description</b>	<b>88 Frequency of use</b>
ColorGradingMode	Numeric	133 Make an FVector4 property a color display	134 ★

## Object

<b>Name</b>	<b>135 Engine module</b>	<b>136 Function description</b>	<b>137 Frequency of use</b>
DisplayThumbnail	Object	138 Specifies whether to display a thumbnail to the left of this property.	★★★
ThumbnailSize	Object	140 Change thumbnail size.	
LoadBehavior	Object	142 Used to mark the loading behavior of this class on UCLASS so that the corresponding TObjectPtr property supports lazy loading. The optional loading behavior defaults to Eager and can be changed to LazyOnDemand.	★
ShowInnerProperties	Object	144 Display internal properties of object references in the property details panel	★★★★★
ShowOnlyInnerProperties	Object	146 Directly raise the internal attributes of the structural attributes to a higher level and display them directly	147 ★★
FullyExpand	Object		
CollapsibleChildProperties	Object	149 Newly added meta in the TextureGraph module. Used to collapse an internal property of a structure.	
Untracked	Object	151 Soft object reference type properties of TSoftObjectPtr and FSoftObjectPath do not track assets.	★
HideAssetPicker	Object	153 Hide the selection list of AssetPicker on Object type pins	154 ★

<b>Name</b>	<b>135 Engine module</b>	<b>136 Function description</b>	<b>137 Frequency of use</b>
AssetBundles	Object	155 Indicates which AssetBundles the asset referenced by this attribute belongs to.	★★★
IncludeAssetBundles	Object	157 Used for sub-object properties of UPrimaryDataAsset, specifying that recursion should continue into the sub-object to detect AssetBundle data.	158 ★
MustBeLevelActor	Object		
ExposeFunctionCategories	Object	159 Specifies that functions in certain directories in the class to which the Object attribute belongs can be directly exposed on this class.	★★★

## Path

<b>Name</b>	<b>161 Engine module</b>	<b>162 Function description</b>	<b>163 Frequency of use</b>
ContentDir	Path	164 Use UE style to select Content and subdirectories.	★★★
RelativePath	Path	166 Make the result of the system directory selection dialog box be the relative path of the currently running exe.	⊕⊕⊕
RelativeToGameContentDir	Path	168 Make the result of the system directory selection dialog box a relative path relative to the Content.	⊕⊕
RelativeToGameDir	Path	170 If the result of the system directory selection box is a subdirectory of the Project, it is converted to a relative path, otherwise an absolute path is returned.	★★★

<b>Name</b>	<b>161 Engine module</b>	<b>162 Function description</b>	<b>163 Frequency of use</b>
LongPackageName	Path	172 Use UE style to select content and subdirectories, or convert file paths to long package names.	★★★
FilePathFilter	Path	174 Set the extension of the file selector. The rules conform to the format specifications of the system dialog box. Multiple extensions can be filled in.	★★★

## Pin

<b>Name</b>	<b>176 Engine module</b>	<b>177 Function description</b>	<b>178 Frequency of use</b>
HidePin	Pin	179 Used on function calls to specify the parameter names to be hidden, and also to hide the return value. Multiple parameters can be hidden	180 ★
InternalUseParam	Pin	181 Used on function calls to specify the parameter names to be hidden, and also to hide the return value. Multiple parameters can be hidden	182 ★
HideSelfPin	Pin	183 Used in function calls to hide the default SelfPin, which is Target, so that the function can only be called within the OwnerClass.	184 ★
DataTablePin	Pin	185 Specify a function parameter as a DataTable or CurveTable type to provide RowNameList selections for the other parameters of FName.	186 ★
DisableSplitPin	Pin	187 Disable the split function of Struct	188 ★
HiddenByDefault	Pin	189 Pins in Struct's Make Struct and Break Struct nodes are hidden by default	★
AlwaysAsPin	Pin	191 In the animation blueprint, a certain attribute of the animation node is always exposed as a pin	★★★

<b>Name</b>	<b>176 Engine module</b>	<b>177 Function description</b>	<b>178 Frequency of use</b>
NeverAsPin	Pin	193 In the animation blueprint, a certain attribute of the animation node is always not exposed and becomes a pin	★★★
PinHiddenByDefault	Pin	195 Makes the properties in this structure hidden by default when used as pins in blueprints.	196 ★

## RigVMStruct

<b>Name</b>	<b>197 Engine module</b>	<b>198 Function description</b>	<b>199 Frequency of use</b>
Input	RigVMStruct	200 Specify this property under FRigUnit as an input pin.	★★★★★
Constant	RigVMStruct	202 Identifies a property as a constant pin.	203 ★★
Output	RigVMStruct	204 Specify this property under FRigUnit as an output pin.	★★★★★
Visible	RigVMStruct	206 Specify that this property under FRigUnit is a constant pin and variables cannot be connected.	207 ★★
Hidden	RigVMStruct	208 Specify that this attribute under FRigUnit is hidden	209 ★★
DetailsOnly	RigVMStruct	210 Specifies that this property under FRigUnit is only displayed in the details panel.	211 ★★
TemplateName	RigVMStruct	212 Specifies that the FRigUnit becomes a generic template node.	213 ★★
CustomWidget	RigVMStruct	214 Specify that the properties in the FRigUnit should be edited using custom controls.	215 ★
ExpandByDefault	RigVMStruct	216 Expand the attribute pins in FRigUnit by default.	★★★
Aggregate	RigVMStruct	218 Specifies property pins in FRigUnit as operands of the extendable continuous binary operator.	★★★
Varying	RigVMStruct	ScriptStruct /Script/RigVM.RigVMFunction_GetDeltaTime	
MenuDescSuffix	RigVMStruct	221 The name suffix that identifies FRigUnit in the blueprint right-click menu item.	★★★
NodeColor	RigVMStruct	223 Specifies the RGB color value of the FRigUnit Blueprint node.	224 ★
Icon	RigVMStruct	225 Set the icon of the FRigUnit blueprint node.	226 ★

<b>Name</b>	<b>197 Engine module</b>	<b>198 Function description</b>	<b>199 Frequency of use</b>
<a href="#">Deprecated</a>	RigVMStruct	227 Mark the FRigUnit as deprecated and not displayed in the blueprint right-click menu.	228 ★
Abstract	RigVMStruct	229 Mark the FRigUnit as an abstract class and do not need to implement Execute.	230 ★
RigVMTypAllowed	RigVMStruct	231 Specify that a UENUM can be selected in the UEnum* attribute of FRigUnit .	232 ★
Keywords	RigVMStruct	233 Set the keyword of the FRigUnit blueprint node in the right-click menu to facilitate input and search.	★★★

## Scene

<b>Name</b>	<b>235 Engine module</b>	<b>236 Function description</b>	<b>237 Frequency of use</b>
MakeEditWidget	Scene	238 Make FVector and FTransform appear as a movable control in the scene editor.	★★★
ValidateWidgetUsing	Scene	240 Provide a function to verify whether the current attribute value is legal.	★★★
AllowedLocators	Scene	242 Used to locate bindable objects for Sequencer	★

## Script

<b>Name</b>	<b>244 Engine module</b>	<b>245 Function description</b>	<b>246 Frequency of use</b>
ScriptName	Script	247 The name to use when exporting to scripts	★★★
ScriptNoExport	Script	249 This function or property is not exported to the script.	★★★
ScriptConstant	Script	251 Wrap the return value of a static function into a constant value.	★★★
ScriptConstantHost	Script	253 On the basis of ScriptConstant, specify the type of constant generation.	★

<b>Name</b>	<b>244 Engine module</b>	<b>245 Function description</b>	<b>246 Frequency of use</b>
ScriptMethod	Script	255 Export the static function as a member function with the first parameter.	★★★
ScriptMethodMutable	Script	257 Change the first const structure parameter of ScriptMethod to a reference parameter in the call, and the modified value in the function will be saved.	258 ★
ScriptMethodSelfReturn	Script	259 In the case of ScriptMethod, specify that the return value of this function should overwrite the first parameter of the function.	260 ★
ScriptOperator	Script	261 Wrap a static function whose first parameter is a structure into an operator of the structure.	★★★
ScriptDefaultMake	Script	263 Disable HasNativeMake on the structure, do not call the NativeMake function in C++ when constructing in the script, and use the script's built-in default initialization method.	★
ScriptDefaultBreak	Script		★

## Sequencer

---

<b>Name</b>	<b>266 Engine module</b>	<b>267 Function description</b>	<b>268 Frequency of use</b>
TakeRecorderDisplayName	Sequencer	269 Specify the display name of UTakeRecorderSource.	270 ★
SequencerBindingResolverLibrary	Sequencer	271 Use UBlueprintFunctionLibrary with the SequencerBindingResolverLibrary tag as a dynamically bound class.	272 ★
CommandLineD	Sequencer	273 Protocol type marking a subclass of UMovieSceneCaptureProtocolBase.	274 ★

# Serialization

Name	275 Engine module	276 Function description	277 Frequency of use
SkipUCSModifiedProperties	Serialization	278 Allow properties in ActorComponent to be saved after being modified in Actor constructor	★
MatchedSerializers	Serialization	280 Only used in NoExportTypes.h, indicating the use of structure serializer. Whether to support text import and export	⌚

# SparseDataType

Name	282 Engine module	283 Function description	284 Frequency of use
NoGetter	SparseDataType	285 Prevent UHT from generating a C++ get function for this attribute, which only takes effect for attributes in the structural data of sparse classes.	★

# String/Text

Name	287 Engine module	288 Function description	289 Frequency of use
PasswordField	String/Text	290 Make the text property appear as a password box	★★★★★
PropertyValidator	String/Text	292 Specify a UFUNCTION function by name to perform text validation	★★★
MultiLine	String/Text	294 Make the text attribute edit box support line wrapping.	★★★★★
AllowedCharacters	String/Text	296 These characters are only allowed to be entered in the text box.	★★★

Name	287 Engine module	288 Function description	289 Frequency of use
GetOptions	String/Text	298 Specify an external class function to provide options to the FName or FString property. The drop-down option box in the details panel provides a list of values.	★★★★★
GetKeyOptions	String/Text	300 Provide the option value of the option box in the details panel for FName/FString in TMap as Key	⊕⊕
GetValueOptions	String/Text	Provide detailed panel option values for FName/FString in TMap	⊕⊕
MaxLength	String/Text	Limit the maximum length of text within a text edit box	★★★★★

## Struct

Name	Engine module	Function description	Frequency of use
MakeStructureDefaultValue	Struct	Store default values for attributes in custom structures within BP.	★
IgnoreForMemberInitializationTest	Struct	Make this property ignore uninitialized validation for the structure.	★★
HasNativeBreak	Struct	Specify a C++ UFunction as the implementation for the Break node of this structure	★★★★★
HasNativeMake	Struct	Specify a C++ UFunction as the implementation for the Make node of this structure	★★★★★
DataflowFlesh	Struct	ScriptStruct /Script/DataflowNodes.FloatOverrideDataflowNode	⊕⊕

## TypePicker

Name	Engine module	Function description	Frequency of use
AllowedTypes	TypePicker	Specify allowable asset types for FPrimaryAssetId.	★★★
BaseClass	TypePicker	Limit the base class type for FStateTreeEditorNode selection within the StateTree module.	★
AllowedClasses	TypePicker	Use on class or object selectors to specify that selected objects must belong to certain base classes.	★★★

Name	Engine module	Function description	Frequency of use
ExactClass	TypePicker	When setting AllowedClasses and GetAllowedClasses simultaneously, ExactClass specifies that only the intersection of types that are exactly the same in both collections is taken; otherwise, the intersection plus subclasses is taken.	★
DisallowedExceptions	TypePicker	Use on class or object selectors to exclude certain base classes from the selection.	★★★
GetAllowedClasses	TypePicker	Use on class or object selectors to specify through a function that selected objects must belong to certain base classes.	★★
GetDisallowedClasses	TypePicker	Use on class selectors to specify a function that excludes certain base classes from the selected type list.	★★
BaseStruct	TypePicker	Specify that structures selected in the FInstancedStruct attribute's option list must inherit from the structure pointed to by BaseStruct.	★★★
ExcludeBaseStruct	TypePicker	Ignore the base class of the structure pointed to by BaseStruct when using the FInstancedStruct attribute.	★★★
StructTypeConst	TypePicker	The type specified for the FInstancedStruct attribute cannot be selected in the editor.	★
MetaStruct	TypePicker	Set to the UScriptStruct* attribute to specify the parent structure of the selected type.	★★★
ShowDisplayNames	TypePicker	On class and struct attributes, specify an alternative display name in the class selector instead of the class's original name.	★
DisallowedExceptions	TypePicker	Only applicable in the SmartObject module, to exclude a class and its subclasses in the class selector.	★

Name	Engine module	Function description	Frequency of use
RowType	TypePicker	Specify the base class for the optional row types of the FDataTableRowHandle attribute.	★★★
MustImplement	TypePicker	The class selected by the TSubClassOf or FSoftClassPath attribute must implement this interface	★★★
ShowTreeView	TypePicker	For selecting Class or Struct attributes, display them as a tree rather than a list in the class selector.	★★
BlueprintBaseOnly	TypePicker	For class attributes, specify whether to accept only base classes from which Blueprint subclasses can be created	★★
MetaClass	TypePicker	Use on soft reference attributes to specify the base class of the object to be selected	★★
AllowAbstract	TypePicker	For class attributes, specify whether to accept abstract classes.	★★
HideViewOptions	TypePicker	For selecting Class or Struct attributes, hide the option to modify display settings in the class selector.	★
OnlyPlaceable	TypePicker	For class attributes, specify whether to accept only Actors that can be placed into the scene	★★

## UHT

Name	Engine module	Function description	Frequency of use
DocumentationPolicy	UHT	Specify the document validation rules, currently set only to Strict	★
GetByRef	UHT	Specify that UHT should generate C++ code with a return reference for this property	☠

Name	Engine module	Function description	Frequency of use
CustomThunk	UHT	Specify that UHT will not generate a blueprint call helper function for this function, requiring user-defined implementation.	★★★★★
NativeConstTemplateArg	UHT	Indicate that this attribute is a const template parameter.	
CppFromBpEvent	UHT		
IncludePath	UHT	Record the reference path of the UClass	
ModuleRelativePath	UHT	Record the header file path of the type definition, relative to the module's internal path.	

## Widget

Name	Engine module	Function description	Frequency of use
DisableNativeTick	Widget	Disable NativeTick for this UserWidget.	★★★
ViewmodelBlueprintWidgetExtension	Widget	Used to verify that the Object type of InListItems conforms to the EntryWidgetClass's MVVM-bound ViewModelProperty.	
DesignerRebuild	Widget	Specify that the UMG preview interface should be refreshed after a change in a certain attribute value in the Widget.	★
DefaultGraphNode	Widget	Mark the default blueprint nodes created by the engine.	
BindWidget	Widget	Specify that the Widget property in the C++ class must be bound to a corresponding control in UMG.	★★★★★

Name	Engine module	Function description	Frequency of use
BindWidgetOptional	Widget	Specify that the Widget property in the C++ class can be bound to a corresponding control in UMG, or not.	★★★
OptionalWidget	Widget	Specify that the Widget property in the C++ class can be bound to a corresponding control in UMG, or not.	★★★
IsBindableEvent	Widget	Expose a dynamic unicast delegate to the UMG blueprint for binding to corresponding events.	★★★
EntryInterface	Widget	Define the interface that the optional class for EntryWidgetClass must implement, used on DynamicEntryBox and ListView Widgets.	★★★
EntryClass	Widget	Define the base class that the optional class for EntryWidgetClass must inherit from, used on DynamicEntryBox and ListView Widgets.	★★★
BindWidgetAnim	Widget	Specify that the UWidgetAnimation property in the C++ class must be bound to an animation under UMG	★★★★★
BindWidgetAnimOptional	Widget	Specify that the UWidgetAnimation property in the C++ class can be bound to an animation under UMG, or not.	★★★

# ClassFlags :

Name	Feature	Trait	Value	Description	UCLASS	Related to UPROPERTY
CLASS_Abstract	Blueprint		0x00000001	Specifies that this class is an abstract base class and cannot be instantiated	Abstract	
CLASS_Const	Blueprint	Inherit	0x00010000	All properties and functions of this class are const and should also be exposed as const	Const	
CLASS_CompiledFromBlueprint	Blueprint		0x00040000u	Indicates that the class is created from the blueprint's compilation process		
CLASS_NewerVersionExists	Blueprint		0x80000000u			
CLASS_NoExport	UHT		0x00000100u	Not exposed in the C++ header files, and no registration code is generated	NoExport	
CLASS_CustomConstructor	UHT		0x00008000u	Does not create a default constructor, intended for use only in the C++ environment	CustomConstructor	
CLASS_Deprecated	Editor	Inherit	0x02000000u	Displays a deprecation warning	Deprecated	
CLASS_HideDropDown	Editor		0x04000000u	The class is not shown in the context menu selection box	HideDropDown	
CLASS_EditInlineNew	Editor		0x00001000u	Objects can be constructed using the EditInlineNew button	EditInlineNew, NotEditInlineNew	
CLASS_Hidden	Editor		0x01000000u	Not displayed in the editor's class browser or in the edit inline new feature		
CLASS_CollapseCategories	Editor		0x00002000u	Properties are displayed without categorization	CollapseCategories, DontCollapseCategories	
CLASS_NotPlaceable	Behavior	Inherit	0x00000200u	Cannot be placed in the scene	Deprecated, NotPlaceable, Placeable	
CLASS_ReplicationDetailsSetUp	Behavior		0x00000800u	Does the class still need to call SetUpRuntimeReplicationData		
CLASS_MinimalAPI	DllExport		0x00080000u	Specifies minimal exports for the class, only exporting functions that retrieve class pointers	MinimalAPI	
CLASS_RequiredAPI	DllExport	DefaultC++, Internal	0x00100000u	Requires that the class has DLL export capabilities, exporting all functions and properties	UCLASS_Empty	
	DllExport					
CLASS_DefaultToInstanced	LoadConstruct	Inherit	0x00200000u	Specifies that all references to this class will automatically create an instance object by default	DefaultToInstanced	
CLASS_HasInstancedReference	LoadConstruct	Inherit	0x00800000u	Classes possess component properties		
CLASS_Parsed	LoadConstruct		0x00000010u	Parsing completed successfully		
CLASS_TokenStreamAssembled	LoadConstruct	DefaultC++	0x00400000u	The TokenStream of the specified parent class has been successfully merged into the current class	UCLASS_Empty	
CLASS_LayoutChanging	LoadConstruct			Indicates that the memory layout of this class has been altered, thus CDO cannot be created at this time		
CLASS_Constructed	LoadConstruct	DefaultC++	0x20000000u	The class has been fully constructed	UCLASS_Empty	
CLASS_NeedsDeferredDependencyLoading	LoadConstruct	Inherit		Specifies that this class requires delayed dependency loading	NeedsDeferredDependencyLoading	
CLASS_Transient	LoadConstruct	Inherit	0x00000008u	Transparent and skipped during serialization	Transient, NonTransient	
CLASS_MatchedSerializers	LoadConstruct	DefaultC++, Internal	0x00000020u		UCLASS_Empty, MatchedSerializers	
CLASS_Native	Traits	DefaultC++	0x00000080u	Designated as a native class, created within C++	UCLASS_Empty	
CLASS_Intrinsic	Traits	DefaultC++	0x10000000u	Classes are defined in C++ without UHT-generated code	Intrinsic, UCLASS_Empty	
CLASS_Interface	Traits		0x00004000u	This class serves as an interface	Interface	
CLASS_Optional	Traits	Inherit	0x00000010u	This object type may not be available in certain contexts. (i.e. game runtime or in certain configuration). Optional class data is saved separately to other object types. (i.e. might use sidecar files)	Optional	
CLASS_Config	Config	Inherit	0x00000004u	Loads the object's configuration settings during construction		
CLASS_DefaultConfig	Config	Inherit	0x00000002u	Save the object configuration to DefaultXXX.ini instead of Local , which must be used together with CLASS_Config	DefaultConfig	
CLASS_ProjectUserConfig	Config	Inherit	0x00000040u	Specifies that the config file of settings is saved in Project/User*.ini similar to CLASS_GlobalUserConfig	ProjectUserConfig	
CLASS_PerObjectConfig	Config	Inherit	0x00000400u	Configuration is performed on a per-object basis rather than at the class level	PerObjectConfig	
CLASS_GlobalUserConfig	Config	Inherit	0x08000000u	class Settings is saved to / .... /Blah.ini	GlobalUserConfig	

Name	Feature	Trait	Value	Description	UCLASS	Related to UPROPERTY
CLASS_ConfigDoNotCheckDefaults	Config	Inherit	0x40000000u	Specifying object configuration will not check base/defaults ini	ConfigDoNotCheckDefaults	
HasCustomFieldNotify					CustomFieldNotify	

## StructFlags :

Name	Value	Description	USTRUCT
STRUCT_NoFlags	0x00000000		
STRUCT_Native	0x00000001		
STRUCT_IdenticalNative	0x00000002	If set, this struct will be compared using native code	
STRUCT_HasInstancedReference	0x00000004		
STRUCT_NoExport	0x00000008		
STRUCT_Atomic	0x00000010	Indicates that this struct should always be serialized as a single unit	Atomic
STRUCT.Immutable	0x00000020	Indicates that this struct uses binary serialization; it is unsafe to add/remove members from this struct without incrementing the package version	immutable
STRUCT_AddStructReferencedObjects	0x00000040	If set, native code needs to be run to find referenced objects	
STRUCT_RequiredAPI	0x00000200	Indicates that this struct should be exportable/importable at the DLL layer. Base structs must also be exportable for this to work.	
STRUCT_NetSerializeNative	0x00000400	If set, this struct will be serialized using the CPP net serializer	
STRUCT_SerializeNative	0x00000800	If set, this struct will be serialized using the CPP serializer	
STRUCT_CopyNative	0x00001000	If set, this struct will be copied using the CPP operator=	
STRUCT_IsPlainOldData	0x00002000	If set, this struct will be copied using memcpy	
STRUCT_NoDestructor	0x00004000	If set, this struct has no destructor and none will be called. STRUCT_IsPlainOldData implies STRUCT_NoDestructor	
STRUCT_ZeroConstructor	0x00008000	If set, this struct will not be constructed because it is assumed that memory is zero before construction.	
STRUCT_ExportTextItemNative	0x00010000	If set, native code will be used to export text	
STRUCT_ImportTextItemNative	0x00020000	If set, native code will be used to import text	

Name	Value	Description	USTRUCT
STRUCT_PostSerializeNative	0x00040000	If set, this struct will have PostSerialize called on it after CPP serializer or tagged property serialization is complete	
STRUCT_SerializeFromMismatchedTag	0x00080000	If set, this struct will have SerializeFromMismatchedTag called on it if a mismatched tag is encountered.	
STRUCT_NetDeltaSerializeNative	0x00100000	If set, this struct will be serialized using the CPP net delta serializer	
STRUCT_PostScriptConstruct	0x00200000	If set, this struct will be have PostScriptConstruct called on it after a temporary object is constructed in a running blueprint	
STRUCT_NetSharedSerialization	0x00400000	If set, this struct can share net serialization state across connections	
STRUCT_Trashed	0x00800000	If set, this struct has been cleaned and sanitized (trashed) and should not be used	
STRUCT_NewerVersionExists	0x01000000	If set, this structure has been replaced via reinstancing	
STRUCT_CanEditChange	0x02000000	If set, this struct will have CanEditChange on it in the editor to determine if a child property can be edited	

## EnumFlags :

Name	Feature	Value	Description	UENUM	UENUM 1
Flags	Trait	0x00000001	Whether the UEnum represents a set of flags		Flags
NewerVersionExists	Trait	0x00000002	If set, this UEnum has been replaced by a newer version		

## FunctionFlags :

Name	Feature	Value	Description	UFUNCTION/UDELEGATE	UFUNCTION/UDELEGATE 1	USTRUCT
FUNC_Final	Trait	0x00000001	Function is final (prebindable, non-overridable function).	SealedEvent		
FUNC_RequiredAPI	Dll	0x00000002	Indicates this function is DLL exported/imported.			

Name	Feature	Value	Description	UFUNCTION/UDELEGATE	UFUNCTION/UDELEGATE 1	USTRUCT
FUNC_BlueprintAuthorityOnly	Network	0x00000004	Function will only run if the object has network authority	BlueprintAuthorityOnly		
FUNC_BlueprintCosmetic	Network	0x00000008	Function is cosmetic in nature and should not be invoked on dedicated servers	BlueprintCosmetic		
FUNC_Net	Network	0x00000040	Function is network-replicated.	Client, NetMulticast, Server, ServiceRequest, ServiceResponse		
FUNC_NetReliable	Network	0x00000080	Function should be sent reliably on the network.	Reliable, ServiceRequest, ServiceResponse		
FUNC_NetRequest	Network	0x00000100	Function is sent to a net service	ServiceRequest		
FUNC_Exec	Trait	0x00000200	Executable from command line.	Exec		
FUNC_Native	Trait	0x00000400	Native function.	BlueprintImplementableEvent		
FUNC_Event	Trait	0x00000800	Event function.	BlueprintImplementableEvent, BlueprintNativeEvent, ServiceRequest, ServiceResponse		
FUNC_NetResponse	Network	0x00001000	Function response from a net service	ServiceResponse		
FUNC_Static		0x00002000	Static function.			
FUNC_NetMulticast	Network	0x00004000	Function is networked multicast Server -> All Clients	NetMulticast		
FUNC_UbergraphFunction	Blueprint	0x00008000	Function is used as the merge 'ubergraph' for a blueprint, only assigned when using the persistent 'ubergraph' frame			
FUNC_MulticastDelegate	Trait	0x00010000	Function is a multi-cast delegate signature (also requires FUNC_Delegate to be set)			
FUNC_Public	Trait	0x00020000	Function is accessible in all classes (if overridden, parameters must remain unchanged).			
FUNC_Private	Trait	0x00040000	Function is accessible only in the class it is defined in (cannot be overridden, but function name may be reused in subclasses. IOW: if overridden, parameters don't need to match, and Super.Func() cannot be accessed since it's private.)			
FUNC_Protected	Trait	0x00080000	Function is accessible only in the class it is defined in and subclasses (if overridden, parameters must remain unchanged).			
FUNC_Delegate	Trait	0x00100000	Function is delegate signature (either single-cast or multi-cast, depending on whether FUNC_MulticastDelegate is set.)			
FUNC_NetServer	Network	0x00200000	Function is executed on servers (set by replication code if passes check)	Server		
FUNC_HasOutParms	Trait	0x00400000	function has out (pass by reference) parameters			
FUNC_HasDefaults	Trait	0x00800000	function has structs that contain defaults			HasDefaults
FUNC_NetClient	Network	0x01000000	function is executed on clients	Client		
FUNC_DLLImport	DLL	0x02000000	function is imported from a DLL			
FUNC_BlueprintCallable	Blueprint	0x04000000	function can be called from blueprint code	BlueprintGetter, BlueprintPure, BlueprintSetter, BlueprintCallable		
FUNC_BlueprintEvent	Blueprint	0x08000000	function can be overridden/implemented from a blueprint	BlueprintImplementableEvent, BlueprintNativeEvent		
FUNC_BlueprintPure	Blueprint	0x10000000	function can be called from blueprint code, and is also pure (produces no side effects). If you set this, you should set FUNC_BlueprintCallable as well.	BlueprintGetter, BlueprintPure		
FUNC_EditorOnly	Trait	0x20000000	function can only be called from an editor script.			
FUNC_Const	Trait	0x40000000	function can be called from blueprint code, and only reads state (never writes state)			

Name	Feature	Value	Description	UFUNCTION/UDELEGATE	UFUNCTION/UDELEGATE 1	USTRUCT
FUNC_NetValidate	Network	0x80000000	function must supply a _Validate implementation	WithValidation		

# PropertyFlags :

Name	Feature	Value	Description	UPARAM	UPROPERTY
CPF_Edit	Editor	0x0000000000000001	Property is user-settable in the editor.		EditAnywhere, EditDefaultsOnly, EditInstanceOnly, VisibleAnywhere, VisibleDefaultsOnly, VisibleInstanceOnly, Interp
CPF_ConstParm	Trait	0x0000000000000002	This is a constant function parameter	Const (Specifier/UPARAM/Const.md)	
CPF_BlueprintVisible	Blueprint	0x0000000000000004	This property can be read by blueprint code		BlueprintReadWrite, BlueprintReadOnly, BlueprintSetter, BlueprintGetter, Interp
CPF_ExportObject	Serialization	0x0000000000000008	Object can be exported with actor.		Instanced, Export
CPF_BlueprintReadOnly	Blueprint	0x0000000000000010	This property cannot be modified by blueprint code		BlueprintReadOnly, BlueprintGetter
CPF_Net	Network	0x0000000000000020	Property is relevant to network replication.		Replicated, ReplicatedUsing
CPF_EditFixedSize	Editor	0x0000000000000040	Indicates that elements of an array can be modified, but its size cannot be changed.		EditFixedSize
CPF_Parm	Function	0x0000000000000080	Function/When call parameter.		
CPF_OutParm	Function	0x0000000000000100	Value is copied out after function call.		
CPF_ZeroConstructor	Trait	0x0000000000000200	memset is fine for construction		
CPF_ReturnParm	Function	0x0000000000000400	Return value.		
CPF_DisableEditOnTemplate	Editor	0x0000000000000800	Disable editing of this property on an archetype/sub-blueprint		EditInstanceOnly, VisibleInstanceOnly
CPF_NonNullable	Trait	0x0000000000001000	Object property can never be null		
CPF_Transient	Serialization	0x0000000000002000	Property is transient: shouldn't be saved or loaded, except for Blueprint CDOs.		Transient
CPF_Config	Config	0x0000000000004000	Property should be loaded/saved as permanent profile.		Config
CPF_RequiredParm	Editor	0x0000000000008000	Parameter must be linked explicitly in blueprint. Leaving the parameter out results in a compile error.	Required (Specifier/UPARAM/Required.md)	
CPF_DisableEditOnInstance	Editor	0x0000000000010000	Disable editing on an instance of this class		EditDefaultsOnly, VisibleDefaultsOnly
CPF_EditConst	Editor	0x0000000000020000	Property is uneditable in the editor.		VisibleAnywhere
CPF_GlobalConfig	Config	0x0000000000040000	Load config from base class, not subclass.		GlobalConfig
CPF_InstancedReference	Trait	0x0000000000080000	Property is a component references.		Instanced
CPF_DuplicateTransient	Serialization	0x0000000000200000	Property should always be reset to the default value during any type of duplication (copy/paste, binary duplication, etc.)		DuplicateTransient
CPF_SaveGame	Serialization	0x0000000001000000	Property should be serialized for save games, this is only checked for game-specific archives with ArlsSaveGame		
CPF_NoClear	Editor	0x0000000002000000	Hide clear button.		NoClear
CPF_ReferenceParm	Function	0x0000000008000000	Value is passed by reference; CPF_OutParam and CPF_Parm should also be set.	ref (Specifier/UPARAM/ref.md)	
CPF_BlueprintAssignable	Blueprint	0x0000000010000000	MC Delegates only. Property should be exposed for assigning in blueprint code		BlueprintAssignable

Name	Feature	Value	Description	UPARAM	UPROPERTY
CPF_Deprecated	Trait	0x0000000020000000	Property is deprecated. Read it from an archive, but don't save it.		
CPF_IsPlainOldData	Trait	0x0000000040000000	If this is set, then the property can be memcopied instead of CopyCompleteValue / CopySingleValue		
CPF_RepSkip	Network	0x0000000800000000	Not replicated. For non replicated properties in replicated structs	NotReplicated (Specifier/UPARAM/NotReplicated.md)	NotReplicated
CPF_RepNotify	Network	0x0000001000000000	Notify actors when a property is replicated		ReplicatedUsing
CPF_Interp	Editor	0x0000002000000000	interpolatable property for use with cinematics		Interp
CPF_NonTransactional	Editor	0x0000004000000000	Property isn't transacted		NonTransactional
CPF_EditorOnly	Editor	0x0000008000000000	Property should only be loaded in the editor		
CPF_NoDestructor	Trait	0x0000010000000000	No destructor		
CPF_AutoWeak	Trait	0x0000040000000000	Only used for weak pointers, means the export type is autoweak		
CPF_ContainsInstancedReference	Trait	0x0000080000000000	Property contains component references.		
CPF_AssetRegistrySearchable	Editor	0x0000100000000000	asset instances will add properties with this flag to the asset registry automatically		AssetRegistrySearchable
CPF_SimpleDisplay	Editor	0x0000200000000000	The property is visible by default in the editor details view		SimpleDisplay
CPF_AdvancedDisplay	Editor	0x0000400000000000	The property is advanced and not visible by default in the editor details view		AdvancedDisplay
CPF_Protected	Editor	0x0000800000000000	property is protected from the perspective of script		
CPF_BlueprintCallable	Blueprint	0x0001000000000000	MC Delegates only. Property should be exposed for calling in blueprint code		BlueprintCallable
CPF_BlueprintAuthorityOnly	Network	0x0002000000000000	MC Delegates only. This delegate accepts (only in blueprint) only events with BlueprintAuthorityOnly.		BlueprintAuthorityOnly
CPF_TextExportTransient	Serialization	0x0004000000000000	Property shouldn't be exported to text format (e.g. copy/paste)		TextExportTransient
CPF_NonPIEDuplicateTransient	Serialization	0x0008000000000000	Property should only be copied in PIE		NonPIEDuplicateTransient
CPF_ExposeOnSpawn	Trait	0x0010000000000000	Property is exposed on spawn		
CPF_PersistentInstance	Serialization	0x0020000000000000	A object referenced by the property is duplicated like a component. (Each actor should have an own instance.)		Instanced
CPF_UObjectWrapper	Trait	0x0040000000000000	Property was parsed as a wrapper class like TSubclassOf, FScriptInterface etc., rather than a USomething*		
CPF_HasGetValueTypeHash	Trait	0x0080000000000000	This property can generate a meaningful hash value.		
CPF_NativeAccessSpecifierPublic	Trait	0x0100000000000000	Public native access specifier		
CPF_NativeAccessSpecifierProtected	Trait	0x0200000000000000	Protected native access specifier		
CPF_NativeAccessSpecifierPrivate	Trait	0x0400000000000000	Private native access specifier		
CPF_SkipSerialization	Serialization	0x0800000000000000	Property shouldn't be serialized, can still be exported to text		SkipSerialization

# NoExport

- **Function description:** Instruct UHT not to automatically generate registration code, but to perform lexical analysis and extract metadata exclusively.
- **Engine module:** UHT
- **Metadata type:** bool
- **Mechanism of action:** Add EClassFlags: CLASS\_NoExport to ClassFlags
- **Frequency of use:** 0

Specify that UHT should not be used to automatically generate registration code, but only for lexical analysis to extract metadata.

There are a lot of this type in the engine NoExportTypes.h which are specially provided for UHT to extract information. Usually it is wrapped with #if ! CPP //noexport class to avoid compilation. At the same time, this class will be defined in another place. Because StaticRegisterNatives##TClass is not generated, GetPrivateStaticClass cannot be called successfully, so NewObject cannot be used. Generally noexport and Intrinsic are used together. Because DECLARE\_CLASS\_INTRINSIC internally declares static void StaticRegisterNatives##TClass () {} to allow successful calls.

Structures in the engine often use noexport to prevent UHT registration from being generated. Because the structure doesn't actually need to call GetPrivateStaticClass to create metadata. As long as there is an object Z\_Construct\_UScriptStruct\_XXX to generate and construct the corresponding UScriptStruct object.

## Test Code:

```
UCLASS(noexport)
class INSIDER_API UMyClass_NoExport :public UObject
{
    GENERATED_BODY()
public:
};
```

## Test Results:

编译的时候生成错误：

```
error LNK2019: unresolved external symbol "private: static void __cdecl UMyClass_NoExport::StaticRegisterNativeSumyClass_NoExport(void)" (?StaticRegisterNativeSumyClass_NoExport@UMyClass_NoExport@@CAXXZ) referenced in function "private: static class UClass * __cdecl UMyClass_NoExport::GetPrivateStaticClass(void)" (?GetPrivateStaticClass@UMyClass_NoExport@@CAPEAVUClass@@XZ)
```

# Intrinsic

- **Function Description:** Specify that UHT should not generate any code for this class, requiring manual implementation.
- **Engine Module:** UHT
- **Metadata Type:** bool

- **Action Mechanism:** Add CLASS\_Intrinsic to ClassFlags

- **Usage Frequency:** 0

Specify that UHT should not generate any code for this class, necessitating manual coding.

Only set directly in C++; typically, new classes do not use this setting. Classes marked with this are native to UE4, indicating that their metadata code is already handwritten in the source code.

noexport will still parse and generate metadata, but it lacks registration. As a result, all metadata flags for the intrinsic class need to be manually set. However, intrinsic does not generate any code at all; both generated.h and .gen.cpp remain empty. Currently, the only class in noexporttyps.h using intrinsics is UCLASS(noexport, Intrinsic) class UModel{}, but it is still not compiled by the C++ compiler.

```
//UCLASS(Intrinsic)
//class INSIDER_API UMyClass_Intrinsic :public UObject //syntax error: missing
';' before '<class-head>'
//{
//  GENERATED_BODY()
//
//};

//.h
class INSIDER_API UMyClass_Intrinsic :public UObject
{
    DECLARE_CLASS_INTRINSIC(UMyClass_Intrinsic, UObject,
CLASS_MatchedSerializers, TEXT("/Script/Insider"))
};

//.cpp
IMPLEMENT_INTRINSIC_CLASS(UMyClass_Intrinsic, INSIDER_API, UObject, INSIDER_API,
"/Script/Insider", {})

class COREUOBJECT_API UInterface : public UObject
{
    DECLARE_CLASS_INTRINSIC(UInterface, UObject, CLASS_Interface |
CLASS_Abstract, TEXT("/Script/coreuobject"))
};
```

## Interface

- **Function Description:** Indicates that this class is an interface.
- **Engine Module:** UHT
- **Metadata Type:** bool
- **Mechanism of Action:** Add CLASS\_Interface to ClassFlags
- **Usage Frequency:** 0

Indicates that this class is an interface.

This setting is only applied in NoExportTypes.h. Our self-defined UInterfaces do not require manual configuration.

This is automatically handled by UHT when generating the .generated.h file for UInterfaces.

## Source Code Example:

```
UCLASS(abstract, noexport, intrinsic, interface, Config = Engine)
class UInterface : public UObject
{}
```

## Principle:

```
bool FKismetEditorUtilities::IsClassABlueprintInterface(const UClass* Class)
{
    if (Class->HasAnyClassFlags(CLASS_Interface) && !Class-
>HasAnyClassFlags(CLASS_NewerVersionExists))
    {
        return true;
    }
    return false;
}
```

## UCLASS()

- **Function Description:** The default behavior for leaving it blank is that it cannot be inherited within blueprints, variables cannot be defined within blueprints, yet it retains reflective capabilities.
- **Engine Module:** UHT
- **Metadata Type:** bool
- **Functionality Mechanism:** CLASS\_MatchedSerializers, CLASS\_Native, CLASS\_RequiredAPI, CLASS\_TokenStreamAssembled, CLASS\_Intrinsic, CLASS\_Constructed are added to ClassFlags
- **Related items:** Do not write UCLASS ()
- **Commonly Used:** ★★★★☆

It cannot be inherited in blueprints, nor can variables be defined within blueprints.

However, they can still be instantiated using the blueprint ConstructObject. This is particularly suitable for those who desire reflective capabilities without the intention of using them within blueprints.

## Sample Code:

```
/*
[MyClass_Default    Class->Struct->Field->Object
/Script/Insider.MyClass_Default] [IncludePath = Class/MyClass_Default.h,
ModuleRelativePath = Class/MyClass_Default.h]
ObjectFlags:    RF_Public | RF_Standalone | RF_Transient
Outer:  Package /Script/Insider
ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI |
CLASS_TokenStreamAssembled | CLASS_Intrinsic | CLASS_Constructed
Size:  48
{
public: void ExecuteUbergraph(int32 EntryPoint);
};
```

```
 */
UCLASS()
class INSIDER_API UMyClass_Default :public UObject
{
    GENERATED_BODY()
public:
};
```

By default, it possesses the following flags: CLASS\_MatchedSerializers | CLASS\_Native | CLASS\_RequiredAPI | CLASS\_TokenStreamAssembled | CLASS\_Intrinsic | CLASS\_Constructed

## Without\_UCLASS

- **Function description:** Serves merely as a regular C++ object without reflection capabilities.
- **Engine module:** UHT
- **Metadata type:** bool
- **Related Items:** UCLASS\_Empty
- **Commonality:** ★

Exists solely as a regular C++ object, devoid of reflection features.

Objects typically inheriting from UObject will at least have one UCLASS() to enable reflection. However, take note that calling UMyClass\_NoUCLASS::StaticClass() will return the base class UObject's Class, as the subclass does not override it. Thus, it can also be said that this class does not instantiate its own UClass metadata object.

```
class INSIDER_API UMyClass_NoUCLASS :public UObject
{
};
```

By default, the Class of UObject is marked as: CLASS\_Abstract | CLASS\_MatchedSerializers | CLASS\_Native | CLASS\_TokenStreamAssembled | CLASS\_Intrinsic | CLASS\_Constructed. Hence, it cannot be instantiated with NewObject. After manually removing CLASS\_Abstract, it can be instantiated normally with 'new', but the object's name remains "Object," indicating that it is using the Object's Class.

## CustomThunkTemplates

- **Function Description:** Defines the structure that encapsulates the implementations of CustomThunk
- **Engine Module:** UHT
- **Metadata Type:** boolean

Cannot find a reference within the source code

# CustomConstructor

- **Function Description:** Prevent the automatic generation of constructor declarations.
- **Engine Module:** UHT
- **Metadata Type:** bool
- **Action Mechanism:** Add CLASS\_CustomConstructor to ClassFlags

UHT will not generate the default constructor for NO\_API UMyClass\_ModuleAPI(const FObjectInitializer& ObjectInitializer = FObjectInitializer::Get());. However, this is typically used in conjunction with GENERATED\_UCLASS\_BODY, as GENERATED\_BODY will automatically generate a default constructor. It is generally used when customizing this function is required. (But using GENERATED\_BODY is also acceptable.)

Currently deprecated:

```
CLASS_CustomConstructor UE_DEPRECATED(5.1, "CLASS_CustomConstructor should no longer be used. It is no longer being set by engine code.") = 0x00008000u,
```

# CustomFieldNotify

- **Function description:** Prevent UHT from generating the FieldNotify-related code for this class.
- **Engine module:** UHT
- **Metadata type:** bool
- **Action mechanism:** Add HasCustomFieldNotify to ClassFlags
- **Usage frequency:** 0

Prevent UHT from generating FieldNotify-related code for this class.

Used only in the source code for UWidget, for instance, the bIsEnabled field in this class is marked with FieldNotify. Normally, UHT would generate code for it. However, if the class wishes to manually write the UHT code, CustomFieldNotify can be added to prevent UHT from generating the code. In UWidget's .cpp file, because it needs to use the UE\_FIELD\_NOTIFICATION\_IMPLEMENT\_CLASS\_DESCRIPTOR method, it must reject UHT generation.

If your class also requires manual implementation of UE\_FIELD\_NOTIFICATION\_IMPLEMENT\_CLASS\_DESCRIPTOR, you can use CustomFieldNotify.

## Source code example:

```
//E:\P4V\Engine\Source\Runtime\UMG\Public\FieldNotification\FieldNotificationDeclaration.h
UCLASS(Abstract, BlueprintType, Blueprintable, CustomFieldNotify)
class UMG_API UWidget : public UVisual, public INotifyFieldValueChanged
{
    GENERATED_UCLASS_BODY()
public:
    UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_BASE_BEGIN(UMG_API)
        UE_FIELD_NOTIFICATION_DECLARE_FIELD(ToolTipText)
        UE_FIELD_NOTIFICATION_DECLARE_FIELD(Visibility)
        UE_FIELD_NOTIFICATION_DECLARE_FIELD(bIsEnabled)
```

```

UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_BEGIN(ToolTipText)
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(Visibility)
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(bIsEnabled)
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_END()

UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_BASE_END();

UPROPERTY(EditAnywhere, BlueprintReadWrite, FieldNotify,
Getter="GetIsEnabled", Setter="SetIsEnabled", BlueprintGetter="GetIsEnabled",
BlueprintSetter="SetIsEnabled", Category="Behavior")
uint8 bIsEnabled:1;

//cpp
UE_FIELD_NOTIFICATION_IMPLEMENT_CLASS_DESCRIPTOR_ThreeFields(UWidget,
ToolTipText, Visibility, bIsEnabled);

```

## Principle:

The condition for checking includes the presence of HasCustomFieldNotify.

```

protected static bool NeedFieldNotifyCodeGen(UhtClass classObj)
{
    return

!classObj.ClassExportFlags.HasAnyFlags(UhtClassExportFlags.HasCustomFieldNotify)
&&

classObj.ClassExportFlags.HasAnyFlags(UhtClassExportFlags.HasFieldNotify);
}

```

## Blueprintable

- **Function description:** Can be inherited within blueprints, and its latent effects can also serve as variable types
- **Engine module:** Blueprint
- **Metadata type:** bool
- **Mechanism of action:** IsBlueprintBase and BlueprintType are added to Meta
- **Associated items:** NotBlueprintable
- **Commonality:** ★★★★☆

Can be inherited in blueprints, and its latent effects can also be used as variable types.

When the Blueprintable tag is set, the metadata for BlueprintType = true is implicitly set. Conversely, when the tag is removed, the BlueprintType = true metadata is also removed.

## Sample Code:

```
/*
(BlueprintType = true, IncludePath = Class/MyClass_Blueprintable.h,
IsBlueprintBase = true, ModuleRelativePath = Class/MyClass_Blueprintable.h)
*/
UCLASS(Blueprintable)
class INSIDER_API UMyClass_Blueprintable :public UObject
{
    GENERATED_BODY()
};

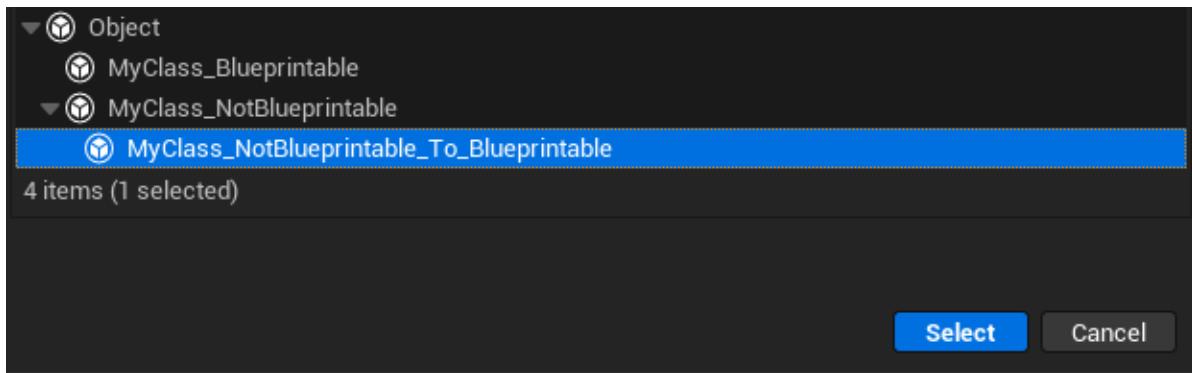
/*
(IncludePath = Class/MyClass_Blueprintable.h, IsBlueprintBase = false,
ModuleRelativePath = Class/MyClass_Blueprintable.h)
*/
UCLASS(NotBlueprintable)
class INSIDER_API UMyClass_NotBlueprintable :public UObject
{
    GENERATED_BODY()
};

/*
(BlueprintType = true, IncludePath = Class/MyClass_Blueprintable.h,
IsBlueprintBase = true, ModuleRelativePath = Class/MyClass_Blueprintable.h)
*/
UCLASS(Blueprintable)
class INSIDER_API UMyClass_NotBlueprintable_To_Blueprintable :public
UMyClass_NotBlueprintable
{
    GENERATED_BODY()
};

/*
(IncludePath = Class/MyClass_Blueprintable.h, IsBlueprintBase = false,
ModuleRelativePath = Class/MyClass_Blueprintable.h)
*/
UCLASS(NotBlueprintable)
class INSIDER_API UMyClass_Blueprintable_To_NotBlueprintable :public
UMyClass_Blueprintable
{
    GENERATED_BODY()
};
```

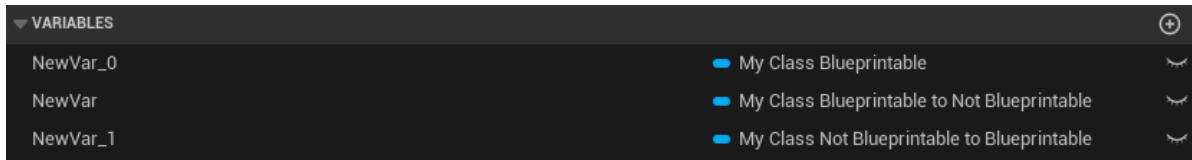
## Example Effect:

Only items marked as Blueprintable can be chosen as base classes.



However, the rule for whether it can be used as a variable still depends on the Blueprint tag of the parent class. Therefore, the following three can be used as variables.

UMyClass\_Blueprintable\_To\_NotBlueprintable can be used as a variable because its parent class, UMyClass\_Blueprintable, can be used as a variable, thus inheriting the property.



## Principle:

The MD\_IsBlueprintBase attribute is used to determine whether subclasses can be created

```
bool FKismetEditorUtilities::CanCreateBlueprintOfClass(const UClass* Class)
{
    bool bCanCreateBlueprint = false;

    if (Class)
    {
        bool bAllowDerivedBlueprints = false;
        GConfig->GetBool(TEXT("Kismet"), TEXT("AllowDerivedBlueprints"), /*out*/
bAllowDerivedBlueprints, GEngineIni);

        bCanCreateBlueprint = !Class->HasAnyClassFlags(CLASS_Deprecated)
            && !Class->HasAnyClassFlags(CLASS_NewerVersionExists)
            && (!Class->ClassGeneratedBy || (bAllowDerivedBlueprints &&
!IsClassABlueprintSkeleton(Class)));

        const bool bIsBPGC = (Cast<UBlueprintGeneratedClass>(Class) != nullptr);

        const bool bIsValidClass = Class-
>GetBoolMetaDataHierarchical(FBlueprintMetadata::MD_IsBlueprintBase)
            || (Class == UObject::StaticClass())
            || (Class == USceneComponent::StaticClass() || Class ==
UActorComponent::StaticClass())
            || bIsBPGC; // BPs are always considered inheritable

        bCanCreateBlueprint &= bIsValidClass;
    }

    return bCanCreateBlueprint;
}
```

# NotBlueprintable

---

- **Function description:** Cannot be inherited within blueprints, and its inherent effects cannot be treated as variables
- **Engine module:** Blueprint
- **Metadata type:** bool
- **Mechanism of action:** IsBlueprintBase and BlueprintType are removed from Meta
- **Associated items:** Blueprintable
- **Commonly used:** ★★★★

# BlueprintType

---

- **Function description:** Can serve as a variable type
- **Engine module:** Blueprint
- **Metadata type:** bool
- **Action mechanism:** Add BlueprintType to Meta
- **Associated items:** NotBlueprintType
- **Commonly used:** ★★★★★

Can serve as a variable type.

The key is to set the two metadata, BlueprintType and NotBlueprintType.

## Sample Code:

---

```
/*
(BlueprintType = true, IncludePath = Class/MyClass_BlueprintType.h,
ModuleRelativePath = Class/MyClass_BlueprintType.h)
*/
UCLASS(BlueprintType)
class INSIDER_API UMyClass_BlueprintType :public UObject
{
    GENERATED_BODY()
};

/*
(IncludePath = Class/MyClass_BlueprintType.h, ModuleRelativePath =
Class/MyClass_BlueprintType.h)
*/
UCLASS()
class INSIDER_API UMyClass_BlueprintType_Child :public UMyClass_BlueprintType
{
    GENERATED_BODY()
};

/*
(IncludePath = Class/MyClass_BlueprintType.h, ModuleRelativePath =
Class/MyClass_BlueprintType.h, NotBlueprintType = true)
*/
```

```

UCLASS(NotBlueprintType)
class INSIDER_API UMyClass_NotBlueprintType :public UObject
{
    GENERATED_BODY()
};

/*
(BlueprintType = true, IncludePath = Class/MyClass_BlueprintType.h,
ModuleRelativePath = Class/MyClass_BlueprintType.h)
*/
UCLASS(BlueprintType)
class INSIDER_API UMyClass_NotBlueprintType_To_BlueprintType:public
UMyClass_NotBlueprintType
{
    GENERATED_BODY()
};

/*
(IncludePath = Class/MyClass_BlueprintType.h, ModuleRelativePath =
Class/MyClass_BlueprintType.h, NotBlueprintType = true)
*/
UCLASS(NotBlueprintType)
class INSIDER_API UMyClass_BlueprintType_To_NotBlueprintType:public
UMyClass_BlueprintType
{
    GENERATED_BODY()
};

```

## Example Results:

Only items with BlueprintType set to true can be used as variables



## Principle:

The three overloaded functions in UEdGraphSchema\_K2::IsAllowableBlueprintVariableType each determine whether UEnum, UClass, and UScriptStruct can be used as variables.

用UEdGraphSchema\_K2::IsAllowableBlueprintVariableType来判断

```

const UClass* ParentClass = InClass;
while(ParentClass)
{
    // Climb up the class hierarchy and look for "BlueprintType" and
    "NotBlueprintType" to see if this class is allowed.
    if(ParentClass-
>GetBoolMetaData(FBlueprintMetadata::MD_AllowableBlueprintVariableType)
    || ParentClass-
>HasMetaData(FBlueprintMetadata::MD_BlueprintSpawnableComponent))

```

```

    {
        return true;
    }
    else if(ParentClass->GetBoolMetaData(FBlueprintMetadata::MD_NotAllowableBlueprintVariableType))
    {
        return false;
    }
    ParentClass = ParentClass->GetSuperclass();
}

```

## NotBlueprintType

- **Function description:** Not to be used as a variable type
- **Engine module:** Blueprint
- **Metadata type:** boolean
- **Action mechanism:** Meta removal of BlueprintType
- **Associated items:** BlueprintType
- **Commonality:** ★★★★

## Abstract

- **Function Description:** This class is designated as an abstract base class. It can be inherited but not instantiated.
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Functionality Mechanism:** Add CLASS\_Abstract to ClassFlags
- **Common Usage:** ★★★★★

Designates this class as an abstract base class. It can be inherited but not instantiated.

Usually applied in base classes such as XXXBase.

## Sample Code:

```

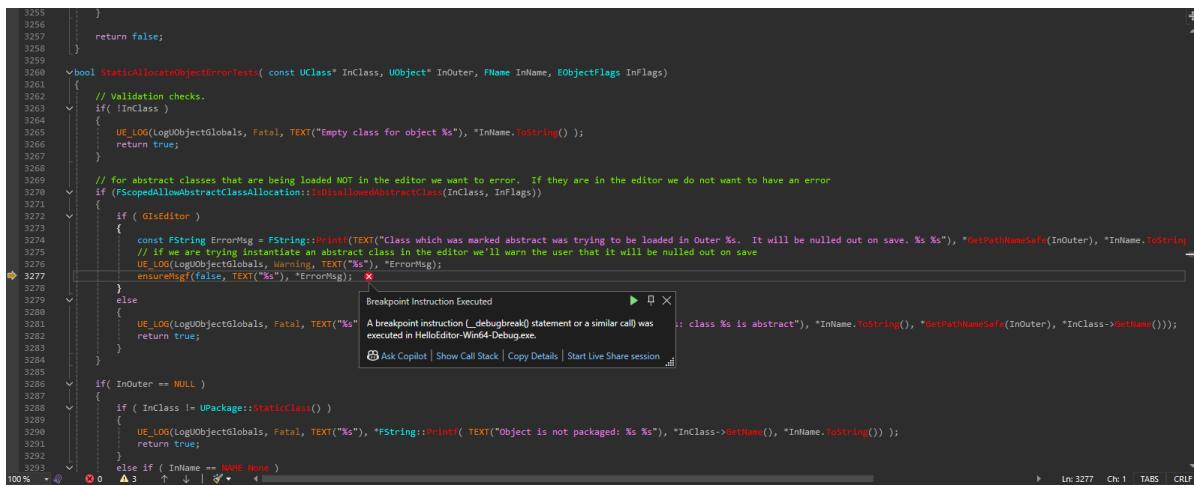
/*
    ClassFlags: CLASS_Abstract | CLASS_MatchedSerializers | CLASS_Native | 
    CLASS_RequiredAPI | CLASS_TokenStreamAssembled | CLASS_Intrinsic | 
    CLASS_Constructed
*/
UCLASS(Blueprintable, abstract)
class INSIDER_API UMyClass_Abstract :public UObject
{
    GENERATED_BODY()
};

//Test Statement:
UMyClass_Abstract* obj=NewObject<UMyClass_Abstract>();

```

# Example Effect:

In the blueprint's ConstructObject, this class will not appear. Simultaneously, using NewObject in C++ will result in an error.



A screenshot of a debugger interface showing a breakpoint instruction executed at line 3277. The code is in a file named `StaticAllocateObjectErrorTests.cpp`. The line contains a `ensureMsgf` call. A tooltip from the debugger indicates: "Breakpoint Instruction Executed" and "A breakpoint instruction (\_debugbreak) statement or a similar call was executed in HelloEditor-Win64-Debug.exe". The tooltip also shows the message being logged: "class %s is abstract", "InName.ToString()", "GetPathNameSafe(Outer)", "InClass->GetName()". The debugger status bar at the bottom right shows "Ln: 3277 Ch: 1 TABS CRLF".

```
3255     }
3256     return false;
3257 }
3258 
3259 bool StaticAllocateObjectErrorTests( const UClass* InClass, UObject* InOuter, FName InName, EObjectFlags InFlags)
3260 {
3261     // Validation checks.
3262     if( !InClass )
3263     {
3264         UE_LOG(LogUObjectGlobals, Fatal, TEXT("Empty class for object %s"), *InName.ToString() );
3265         return true;
3266     }
3267     // for abstract classes that are being loaded NOT in the editor we want to error. If they are in the editor we do not want to have an error
3268     if ( FScopedAllowAbstractClassAllocation::IsDisallowedAbstractClass(InClass, InFlags))
3269     {
3270         if ( GIseEditor )
3271         {
3272             const FString ErrorMsg = FString::Printf(TEXT("Class which was marked abstract was trying to be loaded in Outer %s. It will be nulled out on save. %s %s"), *GetPathNameSafe(Outer), *InName.ToString(), *InClass->GetName());
3273             UE_LOG(LogUObjectGlobals, Warning, TEXT("%s"), *ErrorMsg);
3274             ensureMsgf(false, TEXT("%s"), *ErrorMsg); X
3275         }
3276         else
3277         {
3278             UE_LOG(LogUObjectGlobals, Fatal, TEXT("%s" : class %s is abstract"), *InName.ToString(), *GetPathNameSafe(Outer), *InClass->GetName()));
3279             return true;
3280         }
3281     }
3282     if( InOuter == NULL )
3283     {
3284         if ( InClass != UPackage::StaticClass() )
3285         {
3286             UE_LOG(LogUObjectGlobals, Fatal, TEXT("%s"), *FString::Printf(TEXT("Object is not packaged: %s %s"), *InClass->GetName(), *InName.ToString() ) );
3287             return true;
3288         }
3289         else if ( InName == NAME_None )
3290     }
3291 }
```

## Principle:

During the execution of NewObject, an abstract check is performed.

```
bool StaticAllocateObjectErrorTests( const UClass* InClass, UObject* InOuter,
FName InName, EObjectFlags InFlags)
{
    // validation checks.
    if( !InClass )
    {
        UE_LOG(LogUObjectGlobals, Fatal, TEXT("Empty class for object %s"),
*InName.ToString() );
        return true;
    }

    // for abstract classes that are being loaded NOT in the editor we want to
    // error. If they are in the editor we do not want to have an error
    if ( FScopedAllowAbstractClassAllocation::IsDisallowedAbstractClass(InClass,
InFlags))
    {
        if ( GIseEditor )
        {
            const FString ErrorMsg = FString::Printf(TEXT("Class which was marked
abstract was trying to be loaded in outer %. It will be nulled out on save. %s
%s"), *GetPathNameSafe(Outer), *InName.ToString(), *InClass->GetName());
            // if we are trying instantiate an abstract class in the editor we'll
            // warn the user that it will be nulled out on save
            UE_LOG(LogUObjectGlobals, Warning, TEXT("%s"), *ErrorMsg);
            ensureMsgf(false, TEXT("%s"), *ErrorMsg);
        }
        else
        {
            UE_LOG(LogUObjectGlobals, Fatal, TEXT("%s"), *FString::Printf(
TEXT("Can't create object %s in outer %s: class %s is abstract"),
*InName.ToString(), *GetPathNameSafe(Outer), *InClass->GetName()));
            return true;
        }
    }
}
```

```

        }

    bool FScopedAllowAbstractClassAllocation::IsDisallowedAbstractClass(const
    UClass* InClass, EObjectFlags InFlags)
    {
        if (((InFlags & RF_ClassDefaultObject) == 0) && InClass-
>HasAnyClassFlags(CLASS_Abstract))
        {
            if (AllowAbstractCount == 0)
            {
                return true;
            }
        }

        return false;
    }

```

## Const

---

- **Function Description:** Indicates that the internal attributes of this class are immutable in Blueprints, making them read-only and non-writable.
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Action Mechanism:** Add CLASS\_Abstract to ClassFlags
- **Common Usage:** ★★★

Indicates that the internal attributes of this class cannot be modified in Blueprints, and are only readable and non-writable.

The same applies to Blueprint subclasses. Essentially, this automatically applies a const attribute to both the class and its subclasses. Note that this is only enforced within Blueprints; C++ can still modify these attributes freely, adhering to C++ rules. Therefore, this const is exclusive to Blueprints and is enforced within them. Functions can still be called as usual, but the Set methods for attributes are unavailable, and changes are not permitted.

## Sample Code:

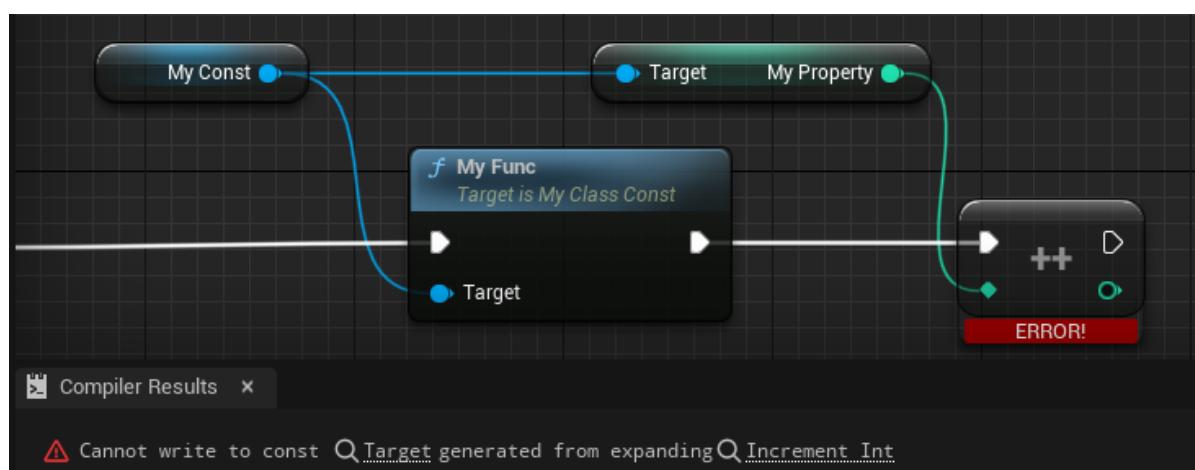
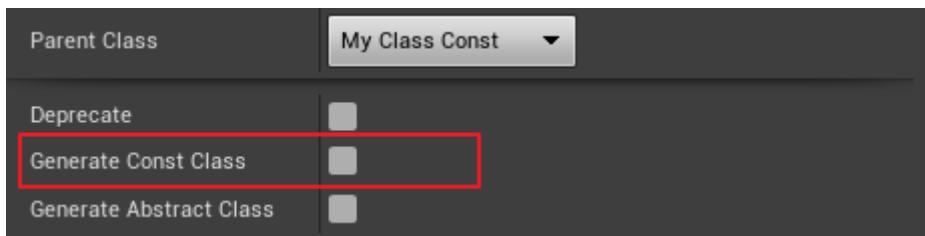
```
/*
    ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_Const |
    CLASS_RequiredAPI | CLASS_TokenStreamAssembled | CLASS_Intrinsic |
    CLASS_Constructed
*/
UCLASS(Blueprintable, Const)
class INSIDER_API UMyClass_Const :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
    UFUNCTION(BlueprintCallable)
    void MyFunc() { ++MyProperty; }
};
```

## Example Effect:

Attempting to modify an attribute in a Blueprint subclass will result in an error.



It is equivalent to enabling this option in the Blueprint Class Settings



## Principle:

Const class instances have attributes marked with const, which prevents modification of their own properties.

```
void FKCHandler_VariableSet::InnerAssignment(FKismetFunctionContext& Context,
UEdGraphNode* Node, UEdGraphPin* VariablePin, UEdGraphPin* ValuePin)
{
    if (!(*VariableTerm)->IsTermwritable())
    {
        CompilerContext.MessageLog.Error(*LOCTEXT("WriteConst_Error", "Cannot
write to const @@").ToString(), VariablePin);
    }
}

bool FBPTerminal::IsTermwritable() const
{
    return !bIsLiteral && !bIsConst;
}
```

## ShowFunctions

- **Function description:** Reopen certain functions within the subclass function override list.
- **Engine module:** Blueprint
- **Metadata type:** strings =( abc , "d|e", "x|y|z")
- **Mechanism of action:** Remove HideFunctions within the Meta class
- **Related items:** HideFunctions
- **Frequency:** ★★

Reopen certain functions within the subclass function override list.

Refer to HideFunctions for test code and visual results.

## Principle:

In the UHT code, it is evident that the purpose of ShowFunctions is to undo the effects of previously set HideFunctions.

```
private void MergeCategories()
{
    MergeShowCategories();

    // Merge ShowFunctions and HideFunctions
    AppendStringListMetaData(SuperClass, UhtNames.HideFunctions, HideFunctions);
    foreach (string value in ShowFunctions)
    {
        HideFunctions.RemoveSwap(value);
    }
    ShowFunctions.Clear();
}
```

# HideFunctions

- **Function description:** Hide certain functions in the subclass function override list.
- **Engine module:** Blueprint
- **Metadata type:** strings = (abc, "d|e", "x|y|z")
- **Mechanism:** Add HideFunctions in the Meta class
- **Associated items:** ShowFunctions
- **Commonly used:** ★★

Hide certain functions in the subclass function override list.

- In the Blueprint, you can still view the functions marked as BlueprintCallable by right-clicking on the class; they remain callable. This marking is only applied to the function override list of the class.
- HideFunctions only accepts function names. To hide functions within a directory, you'll need to use HideCategories for additional specification.

It is utilized only at a single location within the source code. A prime example is the SetFieldOfView and SetAspectRatio functions defined in UCameraComponent. These are irrelevant to UCineCameraComponent and would be better hidden.

```
class ENGINE_API UCameraComponent : public USceneComponent
{
    UFUNCTION(BlueprintCallable, Category = Camera)
        virtual void SetFieldofview(float InFieldofview) { Fieldofview =
    InFieldofview; }
    UFUNCTION(BlueprintCallable, Category = Camera)
        void SetAspectRatio(float InAspectRatio) { AspectRatio = InAspectRatio; }

    UCLASS(HideCategories = (CameraSettings), HideFunctions = (SetFieldofview,
    SetAspectRatio), Blueprintable, ClassGroup = Camera, meta =
    (BlueprintSpawnableComponent), Config = Engine)
    class CINEMATICCAMERA_API UCineCameraComponent : public UCameraComponent
```

## Sample code:

```
UCLASS(Blueprintable, HideFunctions = (MyFunc1, MyEvent2), hideCategories=
EventCategory2)
class INSIDER_API AMyClass_HideFunctions :public AActor
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
        void MyFunc1() {}

    UFUNCTION(BlueprintCallable)
        void MyFunc2() {}

    UFUNCTION(BlueprintCallable, Category = "FuncCategory1")
        void MyFuncInCategory1() {}
```

```

UFUNCTION(BlueprintCallable, Category = "FuncCategory2")
    void MyFuncInCategory2() {}

public:
UFUNCTION(BlueprintImplementableEvent)
    void MyEvent1();

UFUNCTION(BlueprintImplementableEvent)
    void MyEvent2();

UFUNCTION(BlueprintImplementableEvent, Category = "EventCategory1")
    void MyEventInCategory1();

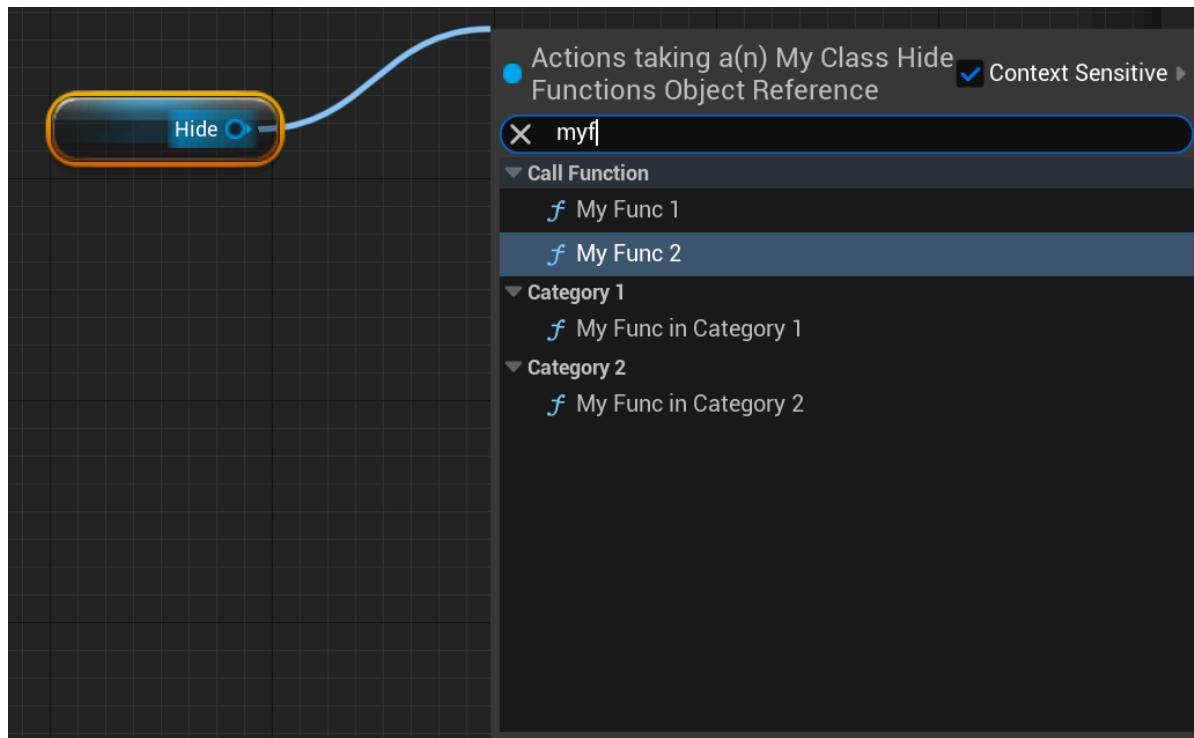
UFUNCTION(BlueprintImplementableEvent, Category = "EventCategory2")
    void MyEventInCategory2();
};

UCLASS(Blueprintable, ShowFunctions = (MyEvent2), showCategories= EventCategory2)
class INSIDER_API AMyClass_ShowFunctions :public AMyClass_HideFunctions
{
    GENERATED_BODY()
public:
};

```

## Example effect:

It is observed that Callable functions are still callable.



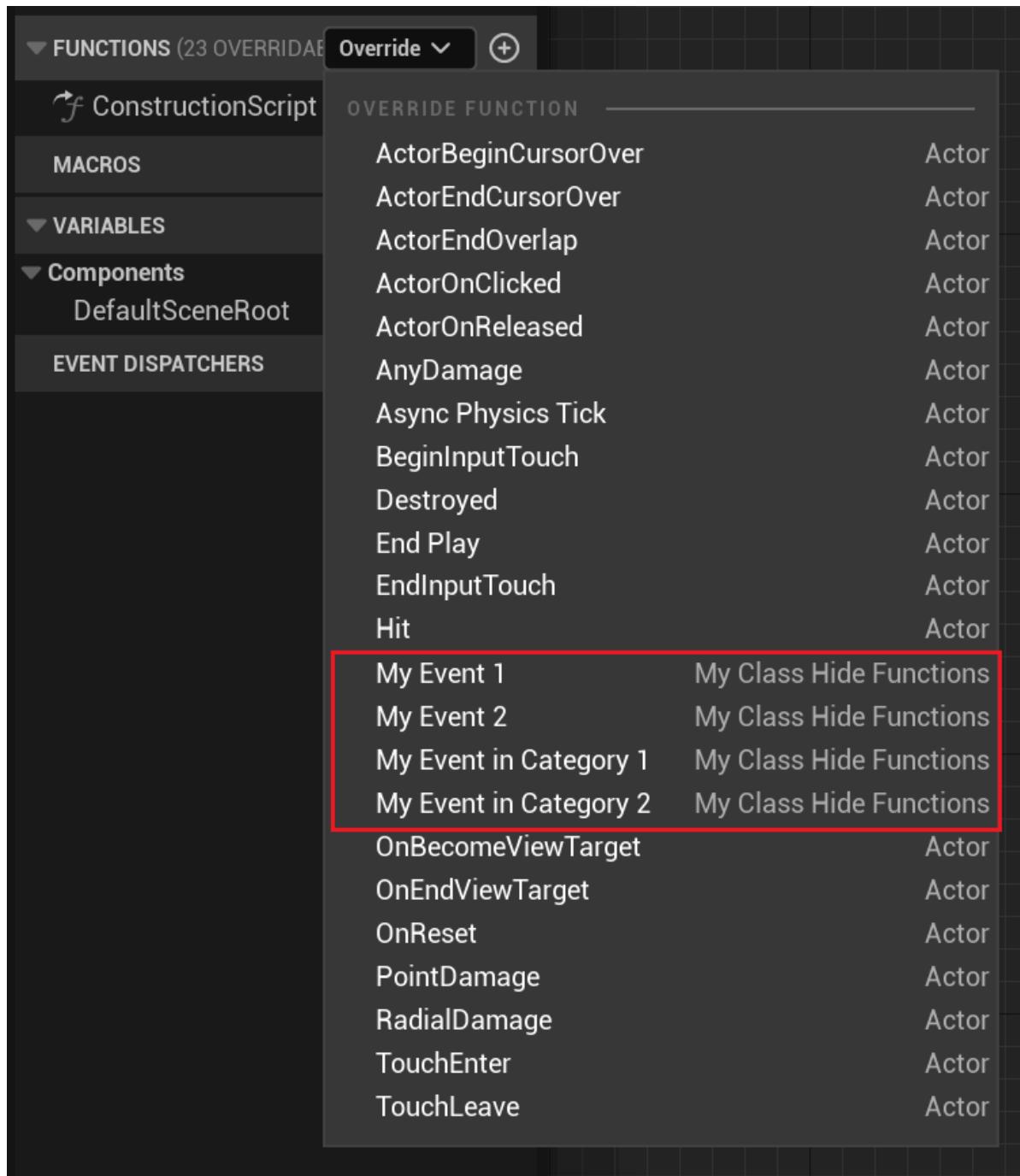
In subclasses using HideFunctions, two functions will be missing from the function overload list

The screenshot shows the 'FUNCTIONS' tab in the ConstructionScript Overrides panel. The left sidebar lists categories: MACROS, VARIABLES, Components (with 'DefaultSceneRoot' selected), and EVENT DISPATCHERS. The main area displays a table of functions and their overrides:

	Override	
ActorBeginCursorOver		Actor
ActorEndCursorOver		Actor
ActorEndOverlap		Actor
ActorOnClicked		Actor
ActorOnReleased		Actor
AnyDamage		Actor
Async Physics Tick		Actor
BeginInputTouch		Actor
Destroyed		Actor
End Play		Actor
EndInputTouch		Actor
Hit		Actor
My Event 1	My Class Hide Functions	
My Event in Category 1	My Class Hide Functions	
OnBecomeViewTarget		Actor
OnEndViewTarget		Actor
OnReset		Actor
PointDamage		Actor
RadialDamage		Actor
TouchEnter		Actor
TouchLeave		Actor

A red box highlights the two rows for 'My Event 1' and 'My Event in Category 1'. Below the table, a note states: 'Event2 and EventCategory2 can be re-enabled in the subclass of ShowFunction'

Event2 and EventCategory2 can be re-enabled in the subclass of ShowFunction



## Principle:

As demonstrated by the principle, HideFunctions should only include function names. To conceal functions within a directory, one must define HideCategories as an additional step.

```
bool IsFunctionHiddenFromClass( const UFunction* InFunction, const UClass* Class )
{
    bool bResult = false;
    if( InFunction )
    {
        bResult = Class->IsFunctionHidden( *InFunction->GetName() );

        static const FName FunctionCategory(TEXT("Category")); // FBlueprintMetadata::MD_FunctionCategory
        if( !bResult && InFunction->HasMetaData( FunctionCategory ) )
    }
}
```

```

        FString const& FuncCategory = InFunction-
>GetMetaData(FunctionCategory);
    bResult = FEditorCategoryUtils::IsCategoryHiddenFromClass(Class,
FuncCategory);
}
}

return bResult;
}

```

## SparseClassDataType

---

- **Function description:** To store some repetitive and immutable data of the Actor in a common structure to reduce the amount of content used
- **Engine module:** Blueprint
- **Metadata type:** string="abc"
- **Mechanism of action:** Add SparseClassDataTypes in Meta
- **Associated items:** NoGetter
- **Commonly used:** ★★★

This is a point of refactoring and performance optimization. When using SparseClassDataType, it is divided into two scenarios: one is for existing Actors to leverage this feature for optimization, and the other is for newly created Actors to use this feature from the outset.

## Example usage:

---

Divided into two parts:

1. Existing Actor has redundant properties

In short, these are properties that do not change in Blueprint. In C++, if these properties are modified, they should also be retrieved using the Get function to transfer them to the SparseDataStruct.

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_SparseClassDataTypes :public AActor
{
GENERATED_BODY()

public:
    UPROPERTY(EditDefaultsOnly)
        int32 MyInt_EditDefaultOnly = 123;

    UPROPERTY(BlueprintReadOnly)
        int32 MyInt_BlueprintReadOnly = 1024;

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
        FString MyString_EditDefault_ReadOnly = TEXT("MyName");

    UPROPERTY(EditAnywhere)
        float MyFloat_EditAnywhere = 555.f;

    UPROPERTY(BlueprintReadWrite)
        float MyFloat_BlueprintReadWrite = 666.f;

```

```
};
```

Change to the following code. Wrap the attributes with WITH\_EDITORONLY\_DATA to indicate that operations are only performed under editor , and runtime has been eliminated. Adding \_DEPRECATED the suffix mark is also to further remind the original BP that the access needs to be removed. Overload MoveDataToSparseClassDataStruct to copy the value currently configured in BP Class Defaults to the new FMySparseClassData structure value.

```
USTRUCT(BlueprintType)
struct FMySparseClassData
{
    GENERATED_BODY()

    UPROPERTY(EditDefaultOnly)
    int32 MyInt_EditDefaultOnly = 123;

    UPROPERTY(EditDefaultOnly, BlueprintReadOnly)
    int32 MyInt_BlueprintReadOnly = 1024;

    // "GetByRef" means that Blueprint graphs access a const ref instead of a
    // copy.
    UPROPERTY(EditDefaultOnly, BlueprintReadOnly, meta=(GetByRef))
    FString MyString_EditDefault_ReadOnly = TEXT("MyName");
};

UCLASS(Blueprintable, BlueprintType, SparseClassDataTypes= MySparseClassData)
class INSIDER_API AMyActor_SparseClassDataTypes :public AActor
{
    GENERATED_BODY()

public:
#if WITH_EDITOR
    // ~ This function transfers existing data into FMySparseClassData.
    virtual void MoveDataToSparseClassDataStruct() const override;
#endif // WITH_EDITOR
public:
#if WITH_EDITORONLY_DATA
    UPROPERTY()
    int32 MyInt_EditDefaultOnly_DEPRECATED = 123;

    UPROPERTY()
    int32 MyInt_BlueprintReadOnly_DEPRECATED = 1024;

    UPROPERTY()
    FString MyString_EditDefault_ReadOnly_DEPRECATED = TEXT("MyName");
#endif // WITH_EDITORONLY_DATA
public:
    UPROPERTY(EditAnywhere)
    float MyFloat_EditAnywhere = 555.f;

    UPROPERTY(BlueprintReadWrite)
    float MyFloat_BlueprintReadWrite = 666.f;
};

//cpp
```

```

#ifndef WITH_EDITOR
void AMyActor_SparseClassDataTypes::MoveDataToSparseClassDataStruct() const
{
    // make sure we don't overwrite the sparse data if it has been saved already
    UBlueprintGeneratedClass* BPClass = Cast<UBlueprintGeneratedClass>
    (GetClass());
    if (BPClass == nullptr || BPClass->bIsSparseClassDataSerializable == true)
    {
        return;
    }

    Super::MoveDataToSparseClassDataStruct();

#ifndef WITH_EDITORONLY_DATA
    // Unreal Header Tool (UHT) will create GetMySparseClassData automatically.
    FMySparseClassData* SparseClassData = GetMySparseClassData();

    // Modify these lines to include all Sparse Class Data properties.
    SparseClassData->MyInt_EditDefaultOnly = MyInt_EditDefaultOnly_DEPRECATED;
    SparseClassData->MyInt_BlueprintReadOnly =
    MyInt_BlueprintReadOnly_DEPRECATED;
    SparseClassData->MyString_EditDefault_ReadOnly =
    MyString_EditDefault_ReadOnly_DEPRECATED;
#endif // WITH_EDITORONLY_DATA

}
#endif // WITH_EDITOR

```

After the BP's PostLoad is executed, MoveDataToSparseClassDataStruct will be called automatically, so bIsSparseClassDataSerializable must be checked internally.

```

void UBlueprintGeneratedClass::PostLoadDefaultObject(UObject* Object)
{
    FScopeLock SerializeAndPostLoadLock(&SerializeAndPostLoadCritical);

    Super::PostLoadDefaultObject(Object);

    if (Object == ClassDefaultObject)
    {
        // Rebuild the custom property list used in post-construct initialization
        // logic. Note that PostLoad() may have altered some serialized properties.
        UpdateCustomPropertyListForPostConstruction();

        // Restore any property values from config file
        if (HasAnyClassFlags(CLASS_Config))
        {
            ClassDefaultObject->LoadConfig();
        }
    }

#ifndef WITH_EDITOR
    Object->MoveDataToSparseClassDataStruct();

    if (Object->GetSparseClassDataStruct())
    {

```

```

        // now that any data has been moved into the sparse data structure we can
        // safely serialize it
        bIsSparseClassDataSerializable = true;
    }

    ConformSparseClassData(object);
#endif
}

```

Within UClass

```

protected:
    /** This is where we store the data that is only changed per class instead of
    per instance */
    void* SparseClassData;

    /** The struct used to store sparse class data. */
    UScriptStruct* SparseClassDataStruct;

```

在构造UClass的时候，会SetSparseClassDataStruct来把结构传进去，因此就把结构关联起来。

```

UCLASS()
void Z_Construct_UClass_AMyActor_SparseClassDataTypes()
{
    if (!Z_Registration_Info_UClass_AMyActor_SparseClassDataTypes.OuterSingleton)
    {

UECodeGen_Private::Construct_UClass(Z_Registration_Info_UClass_AMyActor_SparseClassDataTypes.OuterSingleton,
Z_Construct_UClass_AMyActor_SparseClassDataTypes_Statics::ClassParams);
    Z_Registration_Info_UClass_AMyActor_SparseClassDataTypes.OuterSingleton->SetSparseClassDataStruct(AMyActor_SparseClassDataTypes::StaticGetMySparseClassDataScriptStruct());
    }
    return
Z_Registration_Info_UClass_AMyActor_SparseClassDataTypes.OuterSingleton;
}

```

Note that at this time, you cannot use blueprint get for ReadOnly variables in BP, as \_DEPRECATED is in use. One approach is to define additional Getter methods:

```

UFUNCTION(BlueprintPure)
int32 GetMyMyInt_BlueprintReadOnly() const
{
    return GetMySparseClassData()>>MyInt_BlueprintReadOnly;
}

```

2. Another approach is to simply remove the redundant properties in AMyActor\_SparseClassDataTypes after MoveDataToSparseClassDataStruct (make sure to open the editor, and save after opening the subclass BP Blueprint) and use all values from FMySparseClassData. Thus, it becomes:

```

USTRUCT(BlueprintType)
struct FMySparseClassData
{

```

```

GENERATED_BODY()

UPROPERTY(EditDefaultsOnly)
int32 MyInt_EditDefaultOnly = 123;

UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
int32 MyInt_BlueprintReadOnly = 1024;

// "GetByRef" means that Blueprint graphs access a const ref instead of a
copy.
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, meta=(GetByRef))
FString MyString_EditDefault_ReadOnly = TEXT("MyName");
};

UCLASS(Blueprintable, BlueprintType, SparseClassDataTypes= MySparseClassData)
class INSIDER_API AMyActor_SparseClassDataTypes :public AActor
{
GENERATED_BODY()

public:
UPROPERTY(EditAnywhere)
float MyFloat_EditAnywhere = 555.f;

UPROPERTY(BlueprintReadWrite)
float MyFloat_BlueprintReadWrite = 666.f;
};

```

This achieves the final effect, which is also applicable to new Actors that adopt redundant properties. Note that at this point, BlueprintReadOnly properties can still be accessed in BP, as the UHT and BP system have already added a layer of convenient access control for us.

## Example effect:

UHT will generate C++ access functions for us:

```

#define
FID_Hello_Source_Insider_Class_Trait_MyClass_SparseClassDataTypes_h_30_SPARSE_DAT
A \
FMySparseClassData* GetMySparseClassData(); \
FMySparseClassData* GetMySparseClassData() const; \
const FMySparseClassData* GetMySparseClassData(EGetSparseClassDataMethod
GetMethod) const; \
static UScriptStruct* StaticGetMySparseClassDataScriptStruct(); \
int32 GetMyInt_EditDefaultOnly() \
{ \
    return GetMySparseClassData()->MyInt_EditDefaultOnly; \
} \
int32 GetMyInt_EditDefaultOnly() const \
{ \
    return GetMySparseClassData()->MyInt_EditDefaultOnly; \
} \
int32 GetMyInt_BlueprintReadOnly() \
{ \
    return GetMySparseClassData()->MyInt_BlueprintReadOnly; \
} \

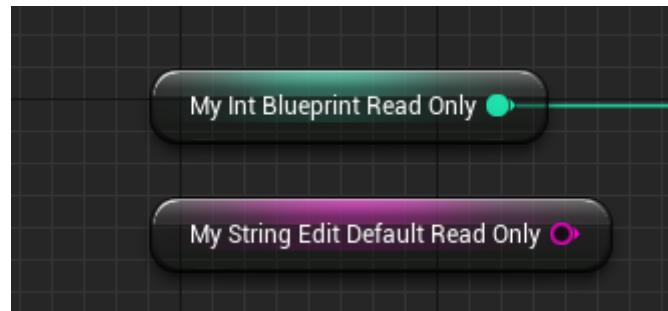
```

```

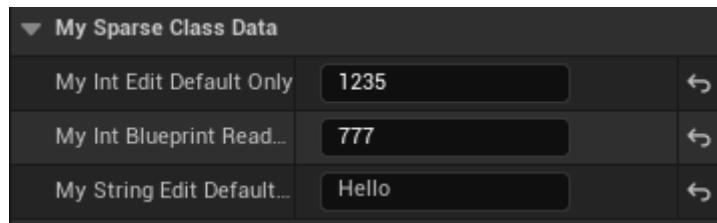
int32 GetMyInt_BlueprintReadOnly() const \
{ \
    return GetMySparseClassData()->MyInt_BlueprintReadOnly; \
} \
const FString& GetMyString_EditDefault_ReadOnly() \
{ \
    return GetMySparseClassData()->MyString_EditDefault_ReadOnly; \
} \
const FString& GetMyString_EditDefault_ReadOnly() const \
{ \
    return GetMySparseClassData()->MyString_EditDefault_ReadOnly; \
}

```

Still accessible in BP:



Values can also be changed in Class Defaults:



## NeedsDeferredDependencyLoading

- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Function Mechanism:** Add CLASS\_NeedsDeferredDependencyLoading to ClassFlags

### Source Code Example:

```

UCLASS(NeedsDeferredDependencyLoading, MinimalAPI)
class UBlueprintGeneratedClass : public UClass, public
IBlueprintPropertyGuidProvider
{
}

```

# Principle:

```
if (ClassFlags.HasAnyFlags(EClassFlags.NeedsDeferredDependencyLoading) &&
    !IsChildof(Session.UClass))
{
    // CLASS_NeedsDeferredDependencyLoading can only be set on classes derived
    // from UClass
    this.LogError($"'NeedsDeferredDependencyLoading' is set on '{SourceName}' but
    the flag can only be used with classes derived from UClass.");
}
```

## MinimalAPI

- **Function Description:** The class's functions are not exported to the DLL; only type information is exported as variables.
- **Engine Module:** DllExport
- **Metadata Type:** bool
- **Action Mechanism:** Add CLASS\_MinimalAPI to ClassFlags
- **Common Usage:** ★★★

Do not export the class's functions to the DLL; only export type information as variables.

- Other modules that reference this one can use pointers for conversion but cannot call the aforementioned functions. However, they are still accessible in Blueprints.
- The benefit is that it can reduce compilation information and speed up linking, as there are fewer dllexport functions.
- Note that MinimalAPI cannot be used in conjunction with MODULENAME\_API, as MinimalAPI is for non-exporting, while MODULENAME\_API is for exporting. The effect of MinimalAPI is not equivalent to not writing MODULENAME\_API, because MinimalAPI also exports GetPrivateStaticClass to allow NewObject. Therefore, if a class does not want to be known by another module at all, no export is needed. If you want another module to be aware of the type but not call any functions, MinimalAPI can be used to prevent this.
- It is recommended not to export game modules. External modules of plugins should be exported, while internal base classes can consider using MinimalAPI, and private classes can be completely unexported. MinimalAPI is frequently used in the engine, resulting in classes that can be used as variables but not inherited or methods called.
- It is typically used in conjunction with BlueprintType, allowing it to be used as a variable in Blueprints.
- Functions and properties can be called normally in Blueprints. This is because blueprint calls only require reflection information, which is registered into the system by the module itself.

## Sample Code:

```
UCLASS()
class UMyClass_NotMinimalAPI :public UObject
{
    GENERATED_BODY()
public:
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite)
int32 MyProperty;
UFUNCTION(BlueprintCallable)
void MyFunc();
};

UCLASS(MinimalAPI)
class UMyClass_MinimalAPI :public UObject
{
GENERATED_BODY()
public:
UPROPERTY(EditAnywhere, BlueprintReadWrite)
int32 MyProperty;
UFUNCTION(BlueprintCallable)
void MyFunc();
};

UCLASS(MinimalAPI, BlueprintType)
class UMyClass_MinimalAPI_BlueprintType :public UObject
{
GENERATED_BODY()
public:
UPROPERTY(EditAnywhere, BlueprintReadWrite)
int32 MyProperty;
UFUNCTION(BlueprintCallable)
void MyFunc() {}
};

UCLASS(MinimalAPI)
class UMyClass_MinimalAPI_BlueprintFunctionLibrary :public
UBlueprintFunctionLibrary
{
GENERATED_BODY()
public:

UFUNCTION(BlueprintCallable)
static void MyFuncInMinimalAPI();

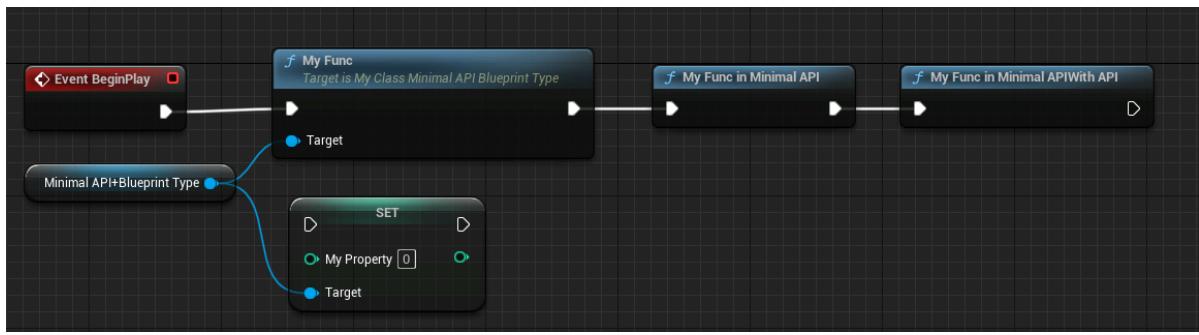
UFUNCTION(BlueprintCallable)
static INSIDER_API void MyFuncInMinimalAPIWithAPI();
};

```

## Example Effect:

---

Functions and properties can be called normally in Blueprints. Methods in the blueprint function library can also be called, indicating that UHT still generates reflection call information for MinimalAPI. Blueprint calls only require reflection information, as the module itself registers function and attribute pointers into the system, thus no DLL export is necessary. However, this function is not found when viewing the list of functions exported by the DLL in the export tool.



View the list of DLL exported functions:

```

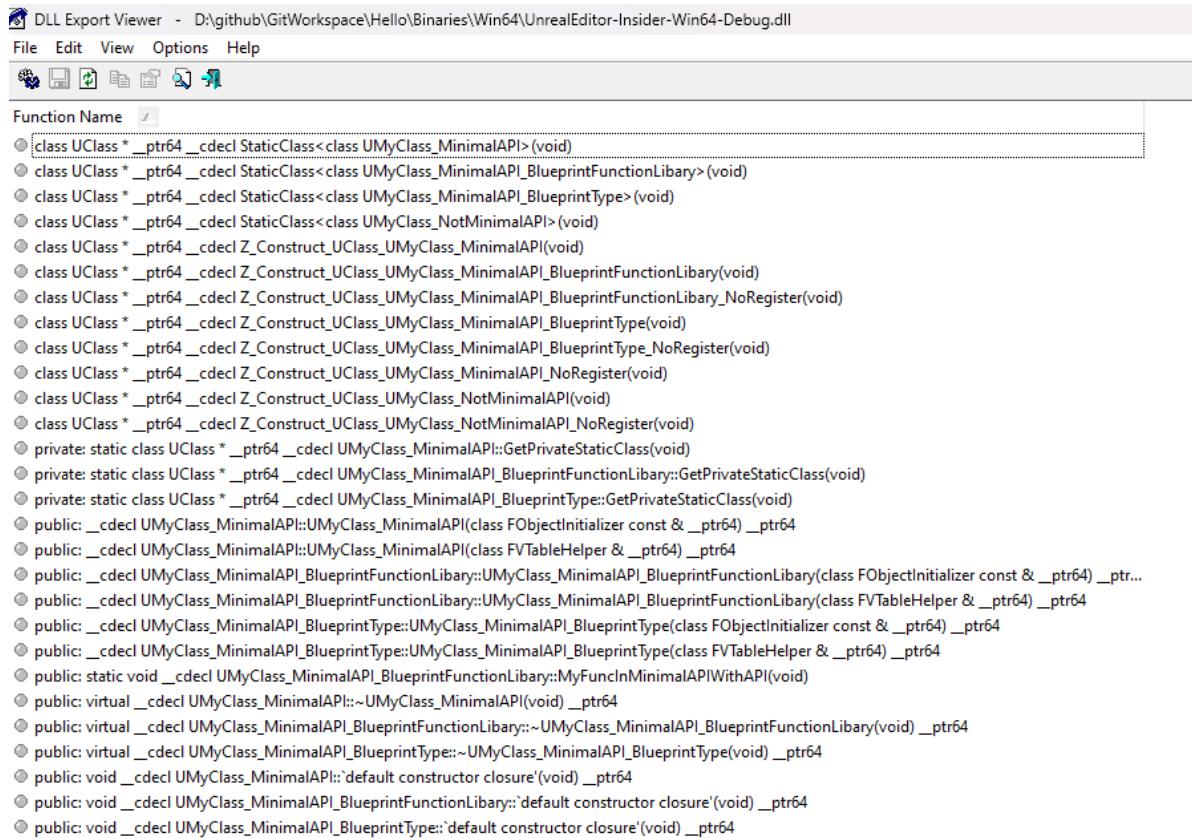
class UClass * __ptr64 __cdecl staticClass<class UMyClass_MinimalAPI>(void)
class UClass * __ptr64 __cdecl staticClass<class
UMyClass_MinimalAPI_BlueprintFunctionLibrary>(void)
class UClass * __ptr64 __cdecl staticClass<class
UMyClass_MinimalAPI_BlueprintType>(void)
class UClass * __ptr64 __cdecl StaticClass<class UMyClass_NotMinimalAPI>(void)
class UClass * __ptr64 __cdecl Z_Construct_UClass_UMyClass_MinimalAPI(void)
class UClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_MinimalAPI_BlueprintFunctionLibrary(void)
class UClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_MinimalAPI_BlueprintFunctionLibrary_NoRegister(void)
class UClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_MinimalAPI_BlueprintType(void)
class UClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_MinimalAPI_BlueprintType_NoRegister(void)
class UClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_MinimalAPI_NoRegister(void)
class UClass * __ptr64 __cdecl Z_Construct_UClass_UMyClass_NotMinimalAPI(void)
class UClass * __ptr64 __cdecl
Z_Construct_UClass_UMyClass_NotMinimalAPI_NoRegister(void)
private: static class UClass * __ptr64 __cdecl
UMyClass_MinimalAPI::GetPrivateStaticClass(void)
private: static class UClass * __ptr64 __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::GetPrivateStaticClass(void)
private: static class UClass * __ptr64 __cdecl
UMyClass_MinimalAPI_BlueprintType::GetPrivateStaticClass(void)
public: __cdecl UMyClass_MinimalAPI::UMyClass_MinimalAPI(class FObjectInitializer
const & __ptr64) __ptr64
public: __cdecl UMyClass_MinimalAPI::UMyClass_MinimalAPI(class FVTableHelper &
__ptr64) __ptr64
public: __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::UMyClass_MinimalAPI_BlueprintFunc
nLibrary(class FObjectInitializer const & __ptr64) __ptr64
public: __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::UMyClass_MinimalAPI_BlueprintFunc
nLibrary(class FVTableHelper & __ptr64) __ptr64
public: __cdecl
UMyClass_MinimalAPI_BlueprintType::UMyClass_MinimalAPI_BlueprintType(class
FObjectInitializer const & __ptr64) __ptr64
public: __cdecl
UMyClass_MinimalAPI_BlueprintType::UMyClass_MinimalAPI_BlueprintType(class
FVTableHelper & __ptr64) __ptr64
public: static void __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::MyFuncInMinimalAPIwithAPI(void)

```

```

public: virtual __cdecl UMyClass_MinimalAPI::~UMyClass_MinimalAPI(void) __ptr64
public: virtual __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::~UMyClass_MinimalAPI_BlueprintFunctionLibrary(void) __ptr64
public: virtual __cdecl
UMyClass_MinimalAPI_BlueprintType::~UMyClass_MinimalAPI_BlueprintType(void)
__ptr64
public: void __cdecl UMyClass_MinimalAPI::`default constructor closure'(void)
__ptr64
public: void __cdecl UMyClass_MinimalAPI_BlueprintFunctionLibrary::`default
constructor closure'(void) __ptr64
public: void __cdecl UMyClass_MinimalAPI_BlueprintType::`default constructor
closure'(void) __ptr64

```



When calling across modules, a link error will be triggered because there is no DLL export.

```

UMyClass_MinimalAPI* a = NewObject<UMyClass_MinimalAPI>();

//First error
//error LNK2019: unresolved external symbol "public: void __cdecl
UMyClass_MinimalAPI::MyFunc(void)" (?MyFunc@UMyClass_MinimalAPI@@QEAXXZ)
referenced in function "public: void __cdecl
UMyClass_UseMinimalAPI::TestFunc(void)" (?TestFunc@UMyClass_UseMinimalAPI@@QEAXXZ)
//a->MyFunc();

a->MyProperty++;

//Second error
//error LNK2019: unresolved external symbol "private: static class uclass *
__cdecl UMyClass_NotMinimalAPI::GetPrivateStaticClass(void)" (?GetPrivateStaticClass@UMyClass_NotMinimalAPI@@CAPEAVUClass@@XZ)

```

```

//referenced in function "class UMyClass_NotMinimalAPI * __cdecl NewObject<class
UMyClass_NotMinimalAPI>(class UObject *)" (??
$NewObject@VUMyClass_NotMinimalAPI@@@YAPEAVUMyClass_NotMinimalAPI@@PEAVUObject@@
@Z)
auto* a = NewObject<UMyClass_NotMinimalAPI>();

//Third error
//error LNK2019: unresolved external symbol "public: static void __cdecl
UMyClass_MinimalAPI_BlueprintFunctionLibrary::MyFuncInMinimalAPI(void)" (??
MyFuncInMinimalAPI@UMyClass_MinimalAPI_BlueprintFunctionLibrary@@SAXXZ)
//referenced in function "public: void __cdecl
UMyClass_UseMinimalAPI::TestFunc(void)" (??
TestFunc@UMyClass_UseMinimalAPI@@QEAXXXZ)
UMyClass_MinimalAPI_BlueprintFunctionLibrary::MyFuncInMinimalAPI();

UMyClass_MinimalAPI_BlueprintFunctionLibrary::MyFuncInMinimalAPIwithAPI();

```

# ClassGroup

---

- **Function description:** Specifies the grouping of components within the AddComponent panel in an Actor, as well as the grouping in the right-click menu of the Blueprint.
- **Engine module:** Category, Editor
- **Metadata type:** string = "a" | b | c"
- **Mechanism of action:** Adds ClassGroupNames in the Meta data
- **Commonly used:** ★★★

Specifies the grouping of components within the AddComponent panel of an Actor, and in the right-click menu of the Blueprint.

## 1 Sample Code:

---

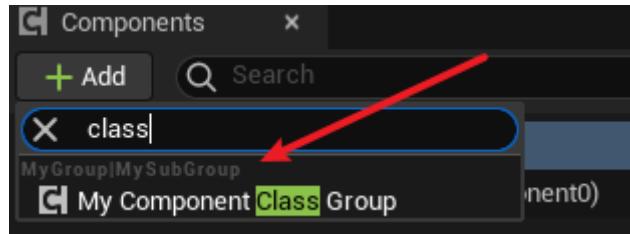
```

//2 ClassGroup must be a valid BlueprintSpawnableComponent
/*
(BlueprintSpawnableComponent = , BlueprintType = true, ClassGroupNames =
MyGroup|MySubGroup, IncludePath = Class/MyComponent_ClassGroup.h, IsBlueprintBase
= true, ModuleRelativePath = Class/MyComponent_ClassGroup.h)
*/
UCLASS(Blueprintable, ClassGroup="MyGroup|MySubGroup", meta =
(BlueprintSpawnableComponent))
class INSIDER_API UMyComponent_ClassGroup:public UActorComponent
{
    GENERATED_BODY()
public:
};

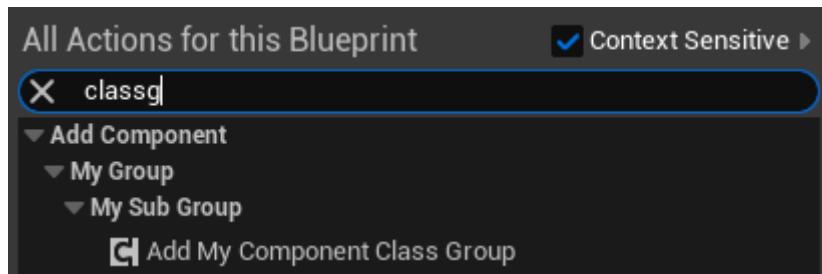
```

## 3 Example Effect:

When adding components:



- 4 Right-click "Add Component" within the blueprint. This test is applicable only to UActorComponents that include BlueprintSpawnableComponent, as it is the only component type that can be dynamically added to a blueprint.



## 5 Principle:

- 6 In the Metadata, ClassGroupNames are accessed via UClass::GetClassGroupNames, a method that is also employed within the BlueprintComponentNodeSpawner. Additionally, it is referenced in ComponentTypeRegistry.cpp, where it aids in component classification. Thus, the ClassGroup is exclusively utilized within the Component context.

```
static FText GetDefaultMenuCategory(const TSubclassOf<UActorComponent>
ComponentClass)
{
    TArray< FString> ClassGroupNames;
    ComponentClass->GetClassGroupNames(ClassGroupNames);

    if (FKismetEditorUtilities::IsClassABlueprintSpawnableComponent(class))
    {
        TArray< FString> ClassGroupNames;
        Class->GetClassGroupNames(ClassGroupNames);
    }
}
```

## ShowCategories

- **Function description:** Displays properties of certain categories in the ClassDefaults property panel of the class.
- **Engine module:** Category
- **Metadata type:** strings =( abc , "d|e" , "x|y|z" )
- **Mechanism of action:** Add HideCategories in Meta
- **Related items:** HideCategories
- **Commonly used:** ★★★

In the ClassDefaults property panel of the class, display properties of certain categories. This invalidates the HideCategories specifier inherited from the base class for the listed categories.

ShowCategories will be analyzed by UHT but will not be saved in the UClass metadata. Its function is to override the HideCategories attributes previously set by the base class. ShowCategories can be inherited by subclasses.

## Sample Code:

```
/*
(BlueprintType = true, HideCategories = MyGroup1, IncludePath =
Class/Display/MyClass_ShowCategories.h, IsBlueprintBase = true,
ModuleRelativePath = Class/Display/MyClass_ShowCategories.h)
*/
UCLASS(Blueprintable, hideCategories = MyGroup1)
class INSIDER_API UMyClass_ShowCategories :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup1)
        int Property_Group1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2 | MyGroup3")
        int Property_Group23;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
};

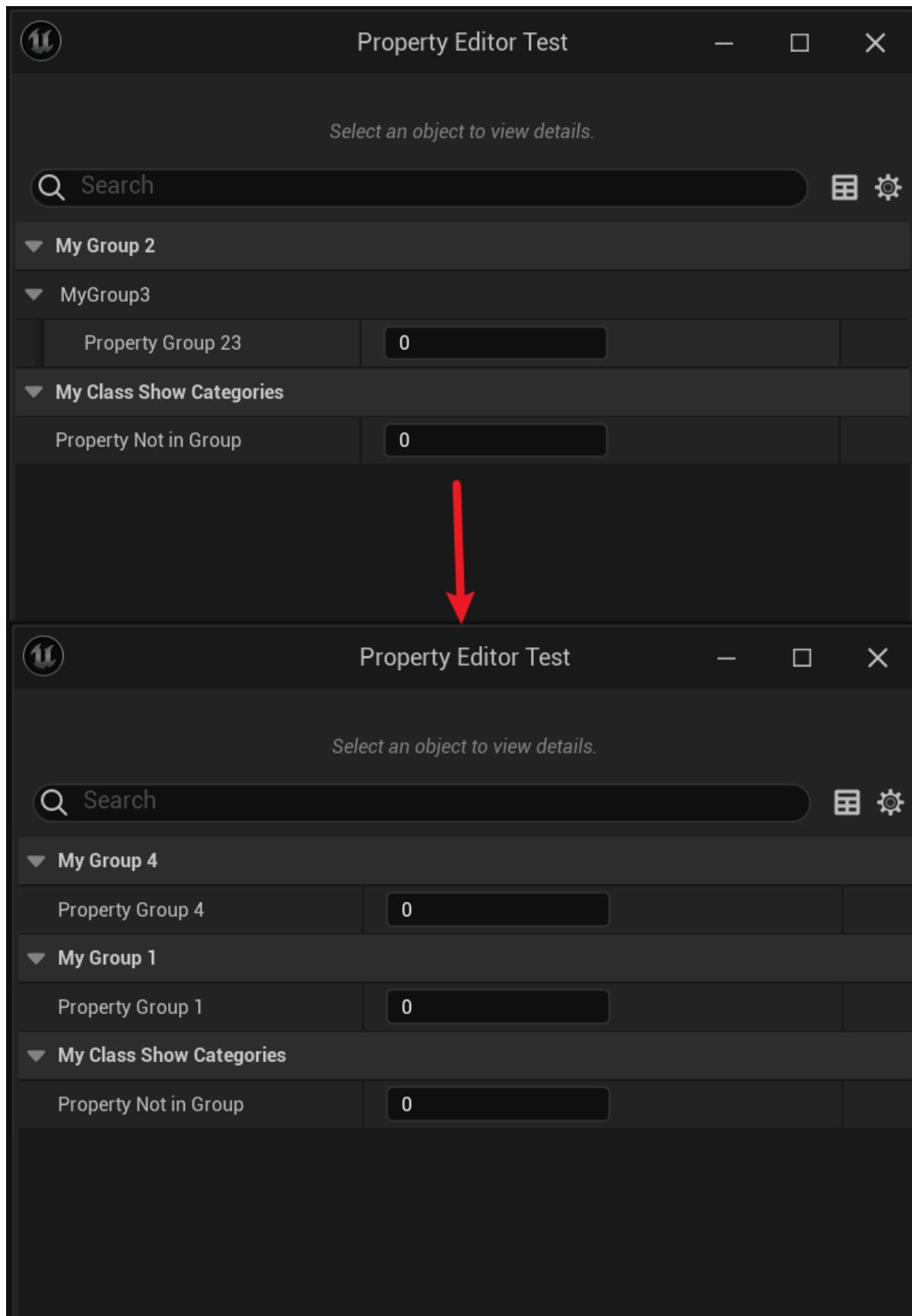
/*
(BlueprintType = true, HideCategories = MyGroup2 | MyGroup3, IncludePath =
Class/Display/MyClass_ShowCategories.h, IsBlueprintBase = true,
ModuleRelativePath = Class/Display/MyClass_ShowCategories.h)
*/
UCLASS(Blueprintable, showCategories = MyGroup1, hideCategories = "MyGroup2 |
MyGroup3")
class INSIDER_API UMyClass_ShowCategoriesChild :public UMyClass_ShowCategories
{
    GENERATED_BODY()
public:

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2")
        int Property_Group2;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup3")
        int Property_Group3;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup4)
        int Property_Group4;
};
```

## Example Effect:



## Principle:

Actually, what UHT saves is only in `HideCategories`, which can be verified by examining the class metadata.

```

void FEditorCategoryUtils::GetClassShowCategories(const UStruct* Class,
TArray<FString>& CategoriesOut)
{
    CategoriesOut.Empty();

    using namespace FEditorCategoryUtilsImpl;
    if (Class->HasMetaData(ClassShowCategoriesMetaKey))
    {
        const FString& ShowCategories = Class-
>GetMetaData(ClassShowCategoriesMetaKey);
        ShowCategories.ParseIntoArray(CategoriesOut, TEXT(" "), /*InCullEmpty
= */true);

        for (FString& Category : CategoriesOut)
        {
            Category =
GetCategoryDisplayString(FText::FromString(Category)).ToString();
        }
    }
}

void FEditorCategoryUtils::GetClassHideCategories(const UStruct* Class,
TArray<FString>& CategoriesOut, bool bHomogenize)
{
    CategoriesOut.Empty();

    using namespace FEditorCategoryUtilsImpl;
    if (Class->HasMetaData(ClassHideCategoriesMetaKey))
    {
        const FString& HideCategories = Class-
>GetMetaData(ClassHideCategoriesMetaKey);

        HideCategories.ParseIntoArray(CategoriesOut, TEXT(" "), /*InCullEmpty
= */true);

        if (bHomogenize)
        {
            for (FString& Category : CategoriesOut)
            {
                Category = GetCategoryDisplayString(Category);
            }
        }
    }
}

```

## HideCategories

---

- **Function description:** Hide certain properties under the Category in the ClassDefaults property panel of a class.
- **Engine module:** Category
- **Metadata type:** strings = (abc, "d" | |y|z")
- **Related items:** ShowCategories
- **Commonly used:** ★★★★

Hide certain Category properties in the ClassDefaults property panel of the class.

Note that you should first define the attribute in the class and then assign it to a Category. The information of HideCategories will be analyzed by UHT and stored in the metadata of UClass. The HideCategories information can be inherited by subclasses.

## Sample code:

```
UCLASS(Blueprintable, hideCategories = MyGroup1)
class INSIDER_API UMyClass_HideCategories :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup1)
        int Property_Group1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2 | MyGroup3")
        int Property_Group23;

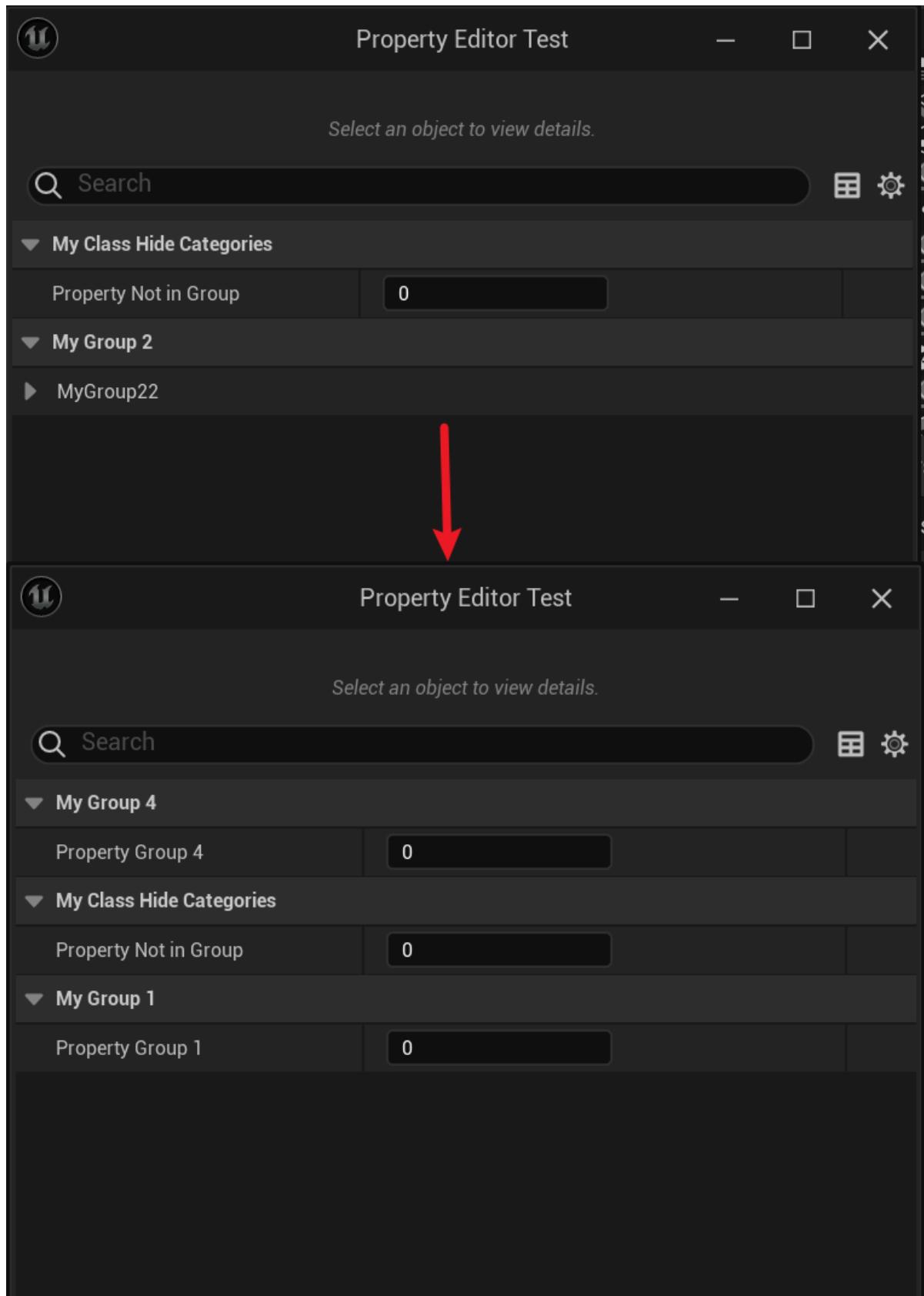
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
};

/*
(BlueprintType = true, HideCategories = MyGroup2 | MyGroup3, IncludePath =
Class/Display/MyClass_ShowCategories.h, IsBlueprintBase = true,
ModuleRelativePath = Class/Display/MyClass_ShowCategories.h)
*/

UCLASS(Blueprintable, showCategories = MyGroup1, hideCategories = "MyGroup2 |
MyGroup3")
class INSIDER_API UMyClass_HideCategoriesChild :public UMyClass_ShowCategories
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2")
        int Property_Group2;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup3")
        int Property_Group3;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup4)
        int Property_Group4;
};
```

## Example effect:

Note here that MyGroup2 and MyGroup3 are also not displayed individually. Therefore, the criterion is simply that the directory matches a specific category name.



## Principle:

Check the `ClassHideCategoriesMetaKey` metadata in `GetClassHideCategories`.

```
void FEditorCategoryUtils::GetClassShowCategories(const UStruct* class,
TArray< FString>& CategoriesOut)
{
    CategoriesOut.Empty();
```

```

using namespace FEditorCategoryUtilsImpl;
if (Class->HasMetaData(ClassShowCategoriesMetaKey))
{
    const FString& ShowCategories = Class-
>GetMetaData(ClassShowCategoriesMetaKey);
    ShowCategories.ParseIntoArray(CategoriesOut, TEXT(" "), /*InCullEmpty
= */true);

    for (FString& Category : CategoriesOut)
    {
        Category =
GetCategoryDisplayString(FText::FromString(Category)).ToString();
    }
}
}

void FEditorCategoryUtils::GetClassHideCategories(const UStruct* Class,
TArray<FString>& CategoriesOut, bool bHomogenize)
{
    CategoriesOut.Empty();

    using namespace FEditorCategoryUtilsImpl;
    if (Class->HasMetaData(ClassHideCategoriesMetaKey))
    {
        const FString& HideCategories = Class-
>GetMetaData(ClassHideCategoriesMetaKey);

        HideCategories.ParseIntoArray(CategoriesOut, TEXT(" "), /*InCullEmpty
= */true);

        if (bHomogenize)
        {
            for (FString& Category : CategoriesOut)
            {
                Category = GetCategoryDisplayString(Category);
            }
        }
    }
}

```

## CollapseCategories

---

- **Function Description:** Hide all properties with the "Category" attribute in the class's property panel, but this effect is only applied to properties with multiple levels of nested "Category."
- **Engine Module:** Category
- **Metadata Type:** bool
- **Action Mechanism:** Add CLASS\_CollapseCategories to ClassFlags
- **Associated Items:** DontCollapseCategories
- **Common Usage:** ★★

All properties with "Category" in the class's property panel are hidden, but this applies only to properties with multiple nested "Category."

## Sample Code:

```
/*
ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_CollapseCategories | 
CLASS_RequiredAPI | CLASS_TokenStreamAssembled | CLASS_Intrinsic | 
CLASS_Constructed
*/
UCLASS(Blueprintable, collapseCategories)
class INSIDER_API UMyClass_CollapseCategories :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup1")
        int Property_Group1;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2|MyGroup22")
        int Property_Group22;

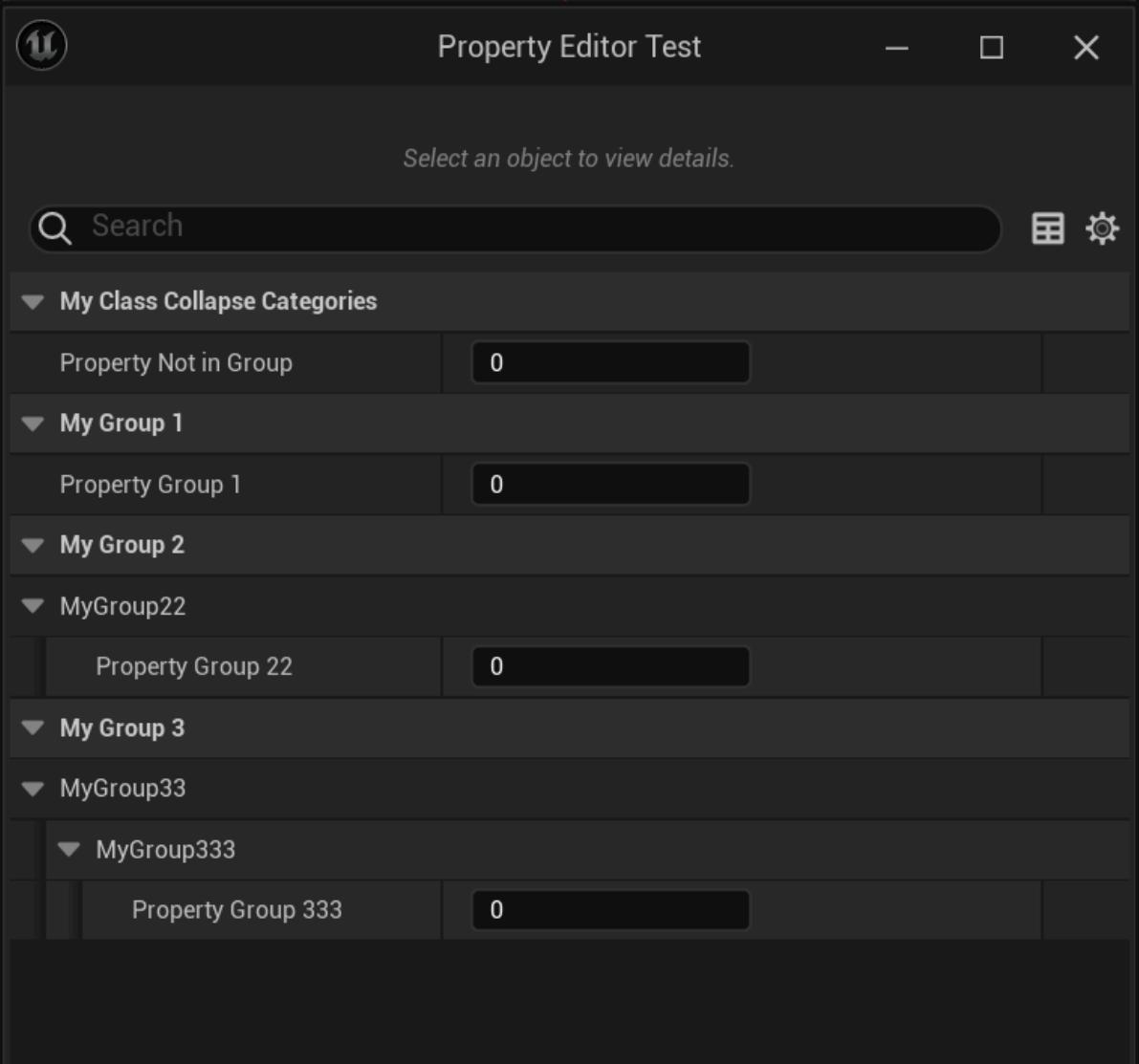
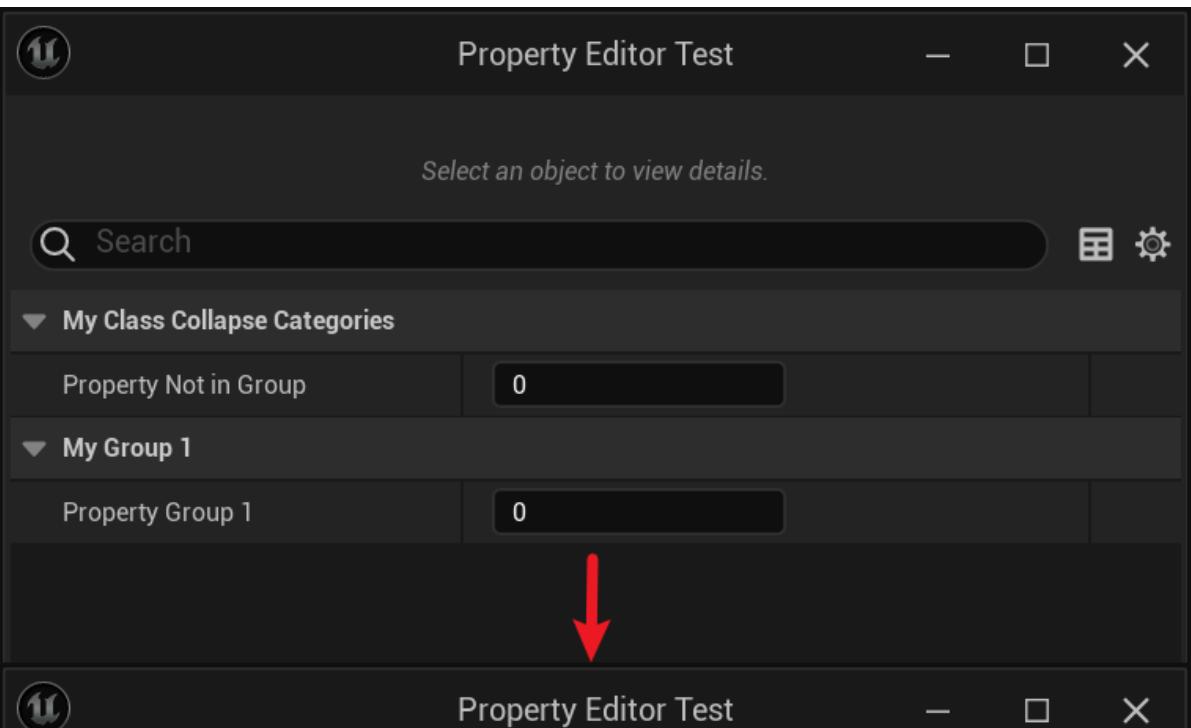
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MyGroup3|MyGroup33|MyGroup333")
        int Property_Group333;
};

/*
ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI | 
CLASS_TokenStreamAssembled | CLASS_Intrinsic | CLASS_Constructed
*/
UCLASS(Blueprintable, dontCollapseCategories)
class INSIDER_API UMyClass_DontCollapseCategories :public UMyClass_CollapseCategories
{
    GENERATED_BODY()

public:
};
```

## Example Effect:

The first is the effect of UMyClass\_CollapseCategories, and the second is the effect of UMyClass\_DontCollapseCategories, where it is evident that some properties are hidden.



# Fundamental Principle:

```
if (Specifier == TEXT("collapseCategories"))
{
    // Class' properties should not be shown categorized in the editor.
    ClassFlags |= CLASS_CollapseCategories;
}
else if (Specifier == TEXT("dontCollapseCategories"))
{
    // Class' properties should be shown categorized in the editor.
    ClassFlags &= ~CLASS_CollapseCategories;
}
```

## DontCollapseCategories

- **Function description:** Overrides the CollapseCategories specifier inherited from the base class.
- **Engine module:** Category
- **Metadata type:** bool
- **Mechanism of action:** Removes the CLASS\_CollapseCategories flag from ClassFlags
- **Related Items:** CollapseCategories
- **Commonly used:** ★★

This effectively removes the CLASS\_CollapseCategories flag from the class, allowing all attributes to be displayed.

## AutoExpandCategories

- **Function description:** Specify the Category that objects of this class should automatically expand in the details panel.
- **Engine module:** Category
- **Metadata type:** strings = (abc, "d|e", "x|y|z")
- **Mechanism of action:** In Meta, remove AutoCollapseCategories and add AutoExpandCategories
- **Associated items:** AutoCollapseCategories
- **Commonly used:** ★

Specify the Category that objects of this class should automatically expand in the details panel.

- Categories can be filled in multiple times, corresponding to the Categories defined for the properties within this class.
- It is worth noting that the editor automatically saves the expanded and closed state of the property directory. It will also be affected by the configuration of DetailPropertyExpansion After opening the window, SDetailsViewBase::UpdateFilteredDetails () will save the currently expanded attribute items, which should be automatically expanded the next time it is opened. The saved code is GConfig->SetSingleLineArray ( TEXT ( "DetailPropertyExpansion" ), \*Struct->GetName (), ExpandedPropertyItems , GEditorPerProjectIni ) ; thus saving under \ Hello \ Saved \ Config \ WindowsEditor \ EditorPerProjectUserSettings.ini . Therefore, in order

to better test the status of this metadata. You should manually clear the saved values in ini before testing.

```
[DetailCategories]
MyClass_AutoExpandCategories.MyClass_AutoExpandCategories=False
MyClass_AutoExpandCategories.MyGroup1=False
MyClass_AutoExpandCategories.MyGroup2=False
MyClass_AutoExpandCategories.MyGroup3=True
MyClass_AutoExpandCategories.MyGroup4=True

[DetailPropertyExpansion]
GeometryCache="\\"Object.GeometryCache.Materials\\" \\"Object.GeometryCache.Tracks\\"
"
Object="\\"Object.MyGroup2.MyGroup22\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44.MyGroup4|MyGroup44|MyGroup444\\"
"
GeometryCacheCodecV1="\\"Object.GeometryCache.TopologyRanges\\"
"
GeometryCacheCodecBase="\\"Object.GeometryCache.TopologyRanges\\"
"
MassSettings="\\"Object.Mass\\"
"
DeveloperSettings=
SmartObjectSettings="\\"Object.SmartObject\\"
"
MyClass_ShowCategories=
MyClass_ShowCategoriesChild=
MyClass_DontCollapseCategories="\\"Object.MyGroup2.MyGroup22\\"
\\"Object.MyGroup3.MyGroup3|MyGroup33\\"
\\"Object.MyGroup3.MyGroup3|MyGroup33.MyGroup3|MyGroup33|MyGroup333\\"
"
MyClass_CollapseCategories="\\"Object.MyGroup2.MyGroup2|MyGroup22\\"
\\"Object.MyGroup3.MyGroup3|MyGroup33\\"
\\"Object.MyGroup3.MyGroup3|MyGroup33.MyGroup3|MyGroup33|MyGroup333\\"
"
MyClass_AutoExpandCategories="\\"Object.MyGroup2.MyGroup2|MyGroup22\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44.MyGroup4|MyGroup44|MyGroup444\\"
"
MyClass_AutoExpandCategoriesCompare=
MyClass_AutoCollapseCategories="\\"Object.MyGroup2.MyGroup2|MyGroup22\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44\\"
\\"Object.MyGroup4.MyGroup4|MyGroup44.MyGroup4|MyGroup44|MyGroup444\\"
"
```

According to the code search rules, the values of AutoExpandCategories and AutoCollapseCategories should be separated by spaces. The top-level directory is initially set to open by default, so AutoExpandCategories is generally used for subdirectories. There is also a restriction that requires opening level by level. Directly opening the deepest subdirectory is not sufficient. Therefore, in the sample code, the intermediate second-level directories "MyGroup4" and "MyGroup44" must also be specified | Ensure that "MyGroup44" is also included.

## Sample Code:

```
UCLASS(Blueprintable, AutoExpandCategories = ("MyGroup2|MyGroup22",
"MyGroup4|MyGroup44", "MyGroup4|MyGroup44|MyGroup444"))
class INSIDER_API UMyClass_AutoExpandCategories :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup1")
    int Property_Group1;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2")
    int Property_Group2;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2|MyGroup22")
    int Property_Group22;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup3|MyGroup33")
    int Property_Group33;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MyGroup4|MyGroup44|MyGroup44")
    int Property_Group44;
};

源码里最复杂的样例:

```

```

UCLASS(Config = Engine, PerObjectConfig, BlueprintType, AutoCollapseCategories =
("Data Layer|Advanced"), AutoExpandCategories = ("Data Layer|Editor", "Data
Layer|Advanced|Runtime"))
class ENGINE_API UDataLayerInstance : public UObject

```

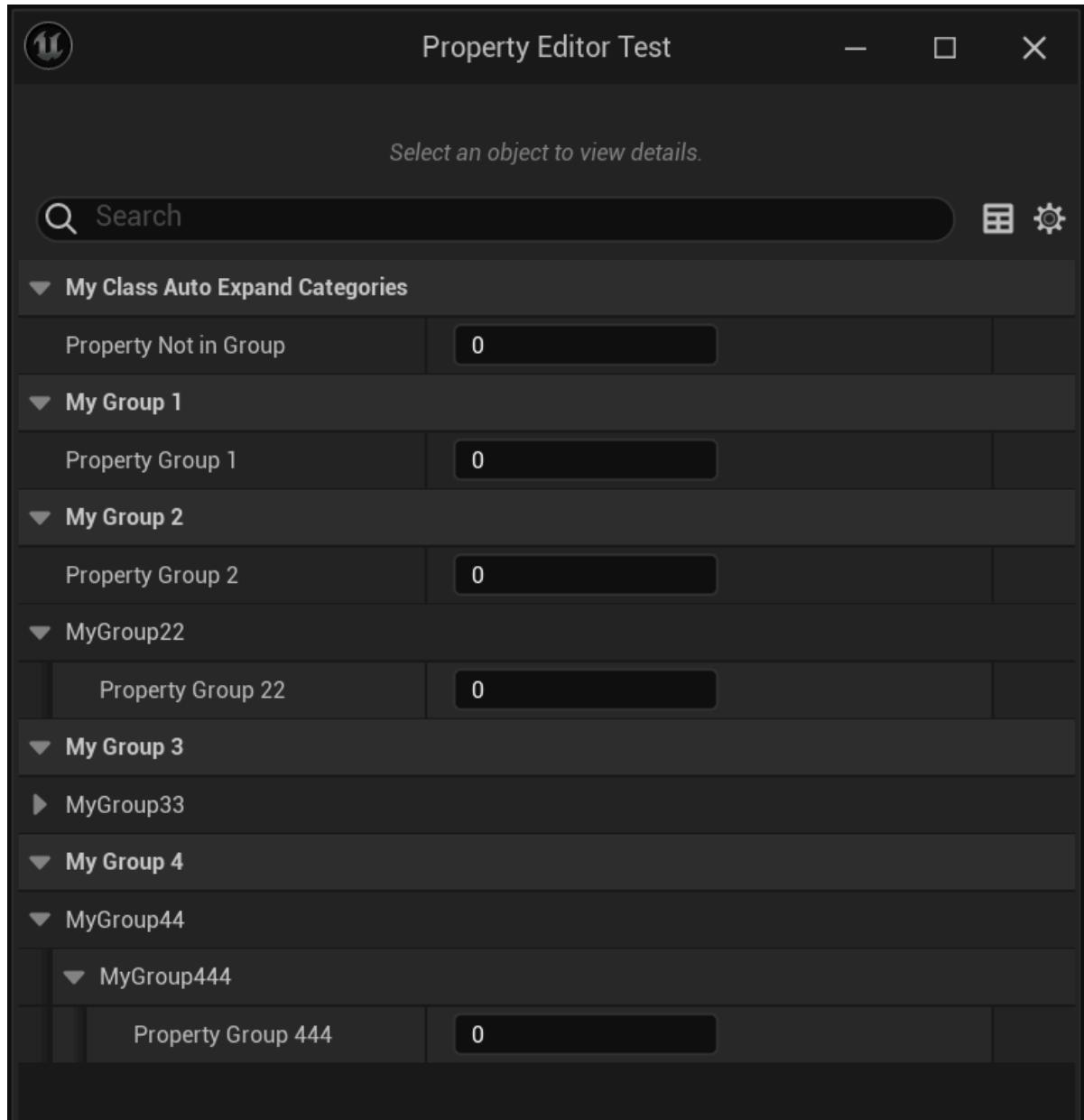
可以打开子目录: UCLASS(Blueprintable, AutoExpandCategories = ("MyGroup2|MyGroup22",
"MyGroup4|MyGroup44", "MyGroup4|MyGroup44|MyGroup44"))

不可以打开子目录: UCLASS(Blueprintable, AutoExpandCategories =
("MyGroup2|MyGroup22", "MyGroup4|MyGroup44|MyGroup44"))

## Example Effect:

After deleting the MyClass\_AutoCollapseCategories value under DetailCategories and DetailPropertyExpansion in Saved\EditorPerProjectUserSettings, open the window using testprops class=MyClass\_AutoExpandCategories:

By comparison, it can be seen that Expand indeed automatically expands subdirectories for immediate editing. The requirement is that the directories listed in AutoExpandCategories must match the Categories on the properties



## Principle:

UClass extracts the metadata of AutoExpandCategories and AutoCollapseCategories to determine whether a Category should be displayed.

```
if (BaseClass->IsAutoExpandCategory(*categoryName.ToString()) &&
!BaseClass->IsAutoCollapseCategory(*categoryName.ToString()))
{
    NewCategoryNode->SetNodeFlags(EPropertyNodeFlags::Expanded, true);
}

bool UClass::IsAutoExpandCategory(const TCHAR* InCategory) const
{
    static const FName NAME_AutoExpandCategories(TEXT("AutoExpandCategories"));
    if (const FString* AutoExpandCategories =
FindMetaData(NAME_AutoExpandCategories))
    {
        return !FCString::StrfindDelim(**AutoExpandCategories, InCategory,
TEXT(" "));
    }
    return false;
}
```

```

}

bool UCLASS::IsAutoCollapseCategory(const TCHAR* InCategory) const
{
    static const FName
NAME_AutoCollapseCategories(TEXT("AutoCollapseCategories"));
    if (const FString* AutoCollapseCategories =
FindMetaData(NAME_AutoCollapseCategories))
    {
        return !FCString::StrfindDelim(**AutoCollapseCategories, InCategory,
TEXT(" "));
    }
    return false;
}

```

## AutoCollapseCategories

---

- **Function description:** The AutoCollapseCategories specifier overrides the effect of the AutoExpandCategories specifier on the listed categories of the parent class.
- **Engine module:** Category
- **Metadata type:** strings = (abc, "d|e", "x|y|z")
- **Mechanism of action:** Add AutoCollapseCategories to Meta and remove AutoExpandCategories
- **Related items:** DontAutoCollapseCategories, AutoExpandCategories
- **Frequency:** ★

## Sample Code:

---

```

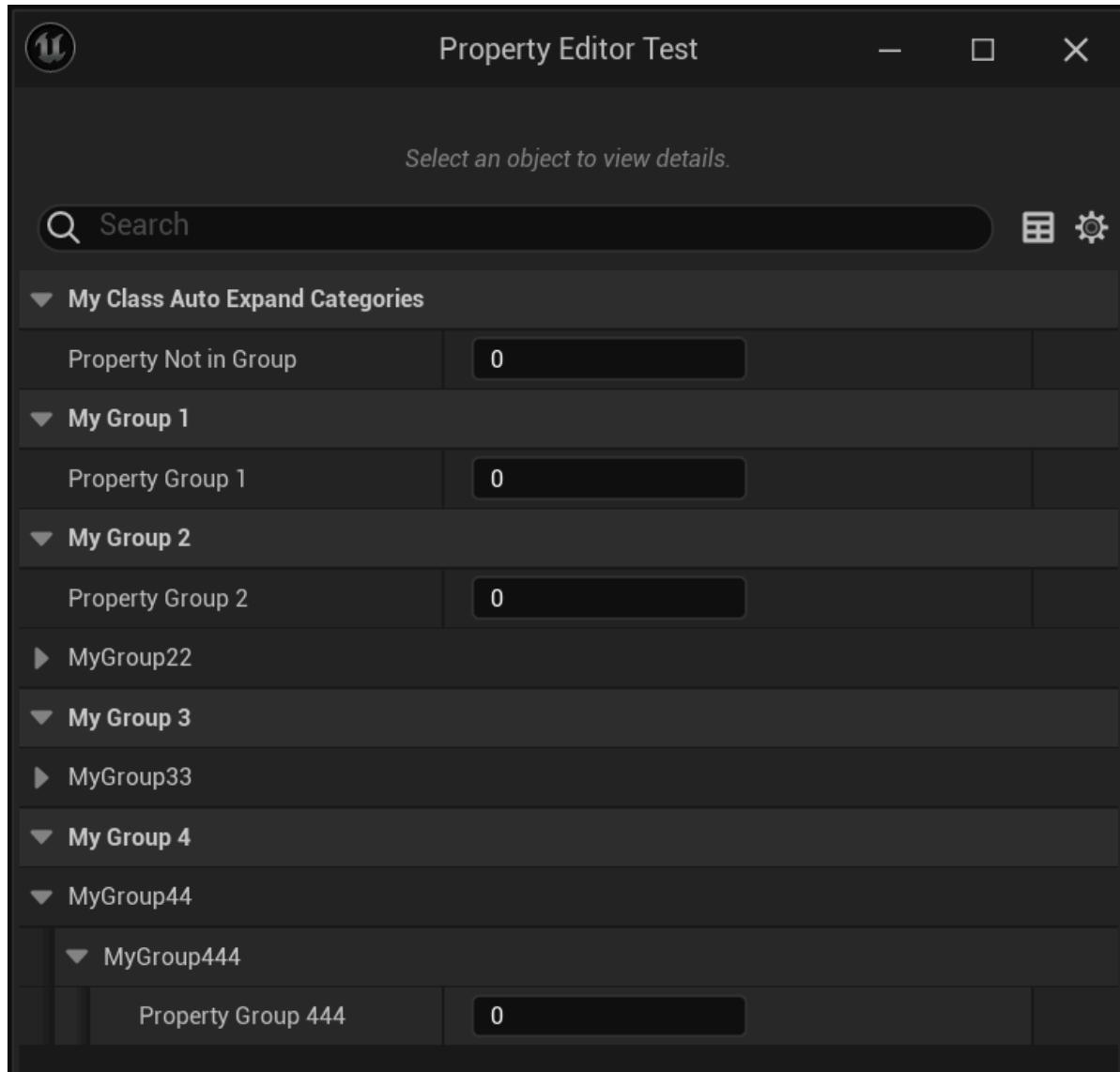
UCLASS(Blueprintable, AutoCollapseCategories = ("MyGroup2|MyGroup22"))
class INSIDER_API UMyClass_AutoCollapseCategories :public
UMyClass_AutoExpandCategories
{
    GENERATED_BODY()
public:
};

```

## Example Results:

---

The expansion of Group22 is disabled, but the expansion of 444 is still inherited



## DontAutoCollapseCategories

- **Function description:** Makes the AutoCollapseCategories specifier for the listed categories inherited from the parent class ineffective.
- **Engine module:** Category
- **Metadata type:** strings = "a, b, c"
- **Mechanism of action:** Remove AutoCollapseCategories from the Meta data
- **Related Items:** AutoCollapseCategories
- **Commonly used:** ★

The code simply removes AutoCollapseCategories. Unlike AutoExpandCategories, it does not automatically add an expansion. A search of the source code did not reveal its usage. Furthermore, the current source code implementation has bugs that prevent its removal.

```
case EClassMetadataSpecifier::AutoExpandCategories:  
  
    FHeaderParser::RequirespecifierValue(*this, PropSpecifier);  
  
    for (FString& value : PropSpecifier.values)  
    {  
        AutoCollapseCategories.RemoveSwap(value);  
    }
```

```

        AutoExpandCategories.AddUnique(MoveTemp(Value));
    }
    break;

case EClassMetadataSpecifier::AutoCollapseCategories:

    FHeaderParser::RequireSpecifierValue(*this, PropSpecifier);

    for (FString& Value : PropSpecifier.Values)
    {
        AutoExpandCategories.RemoveSwap(Value);
        AutoCollapseCategories.AddUnique(MoveTemp(Value));
    }
    break;
case EClassMetadataSpecifier::DontAutoCollapseCategories:

    FHeaderParser::RequireSpecifierValue(*this, PropSpecifier);

    for (const FString& Value : PropSpecifier.Values)
    {
        AutoCollapseCategories.RemoveSwap(Value); //The current value of
        AutoCollapseCategories is empty. Removing it is pointless
    }
    break;

```

改动:

```

FUnrealClassDefinitionInfo::MergeClassCategories()放最后:
// Merge DontAutoCollapseCategories and AutoCollapseCategories
for (const FString& Value : DontAutoCollapseCategories)
{
    AutoCollapseCategories.RemoveSwap(Value);
}
DontAutoCollapseCategories.Empty();

```

改为:

```

case EClassMetadataSpecifier::DontAutoCollapseCategories:

    FHeaderParser::RequireSpecifierValue(*this, PropSpecifier);

    for (FString& Value : PropSpecifier.Values)
    {
        DontAutoCollapseCategories.AddUnique(MoveTemp(Value));
        //AutoCollapseCategories.RemoveSwap(Value);
    }
    break;

```

## PrioritizeCategories

- **Function description:** Display the specified attribute directory at the forefront of the details panel.
- **Engine module:** Category
- **Metadata type:** strings = (abc, "d|e", "x|y|z")
- **Action mechanism:** Add PrioritizeCategories in Meta

- Commonly used: ★★★

Display the specified attribute directory at the forefront of the details panel.

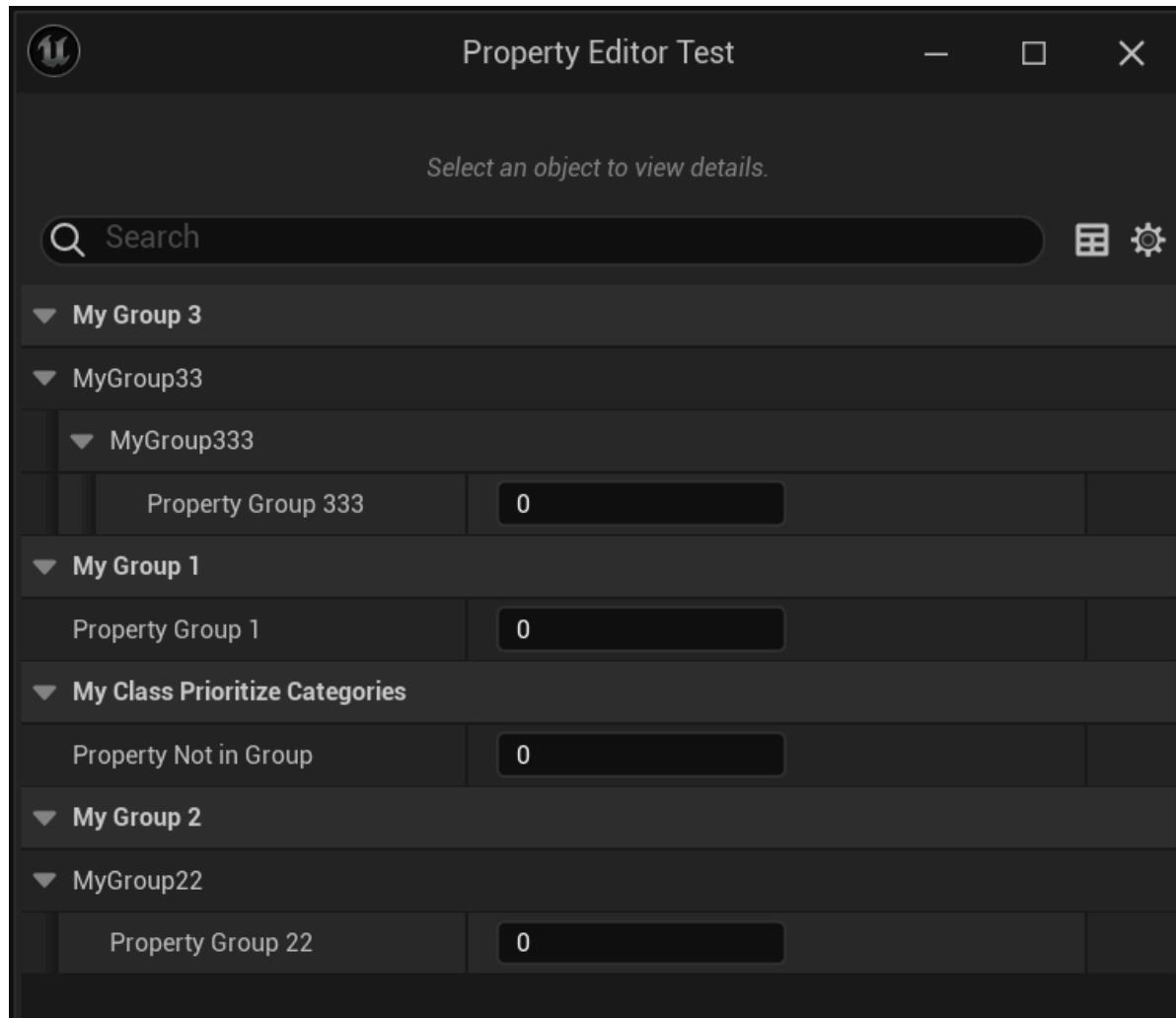
## Sample Code:

```
UCLASS(Blueprintable, PrioritizeCategories=
("MyGroup3|MyGroup33|MyGroup333", "MyGroup1"))
class INSIDER_API UMyClass_PrioritizeCategories :public Uobject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup1")
        int Property_Group1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2|MyGroup22")
        int Property_Group22;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MyGroup3|MyGroup33|MyGroup333")
        int Property_Group33;
};
```

## Sample Code:

Property\_Group33 is now at the top.



## Example Results:

exist UClass::GetPrioritizeCategories ( TArray & OutPrioritizedCategories ) to get the priority directory. The principle is to put them in SortedCategories in the specified order, so the attribute directory will be created first.

```
TArray< FString> ClassPrioritizeCategories;
Class->GetPrioritizeCategories(ClassPrioritizeCategories);
for (const FString& ClassPrioritizeCategory : ClassPrioritizeCategories)
{
    FName PrioritizeCategoryName = FName(ClassPrioritizeCategory);
    SortedCategories.AddUnique(PrioritizeCategoryName);
    PrioritizeCategories.AddUnique(PrioritizeCategoryName);
}
```

## ComponentWrapperClass

- **Function Description:** Designate this class as a simple wrapper class, disregarding Category-related configurations in its subclasses.
- **Engine Module:** Category
- **Metadata Type:** bool
- **Functionality Mechanism:** Add IgnoreCategoryKeywordsInSubclasses in the Meta section
- **Commonality:** ★★

Designate this class as a simple wrapper class, disregarding the Category-related settings of its subclasses.

As suggested by the name, it serves as a wrapper for a component, essentially an Actor that contains just one Component. This simple wrapping relationship is exemplified by ALight wrapping ULightComponent and ASkeletalMeshActor wrapping USkeletalMeshComponent.

Control over the hideCategories and showCategories defined in subclasses is bypassed, and the directory definitions from the base class are directly adopted, which corresponds to the directory settings on this component's wrapper class. Currently, only BlueprintEditorUtils.cpp in the source code utilizes this, which is part of the blueprint opening process. Therefore, this setting only takes effect when a blueprint is opened by double-clicking. For standard UObject classes, windows created using testprops will not be affected since they are not opened by double-clicking a blueprint.

Searching the source code for ComponentWrapperClass reveals that it is only used by some Actors.

## Sample Code:

```
UCLASS(Blueprintable, BlueprintType, Componentwrapperclass, hideCategories =
MyGroup3) //Property_Group3 will still be displayed
class AMyActor_ComponentwrapperClass : public AActor
{
    GENERATED_UCLASS_BODY()
public:
    UPROPERTY(BlueprintReadOnly, visibleAnywhere)
```

```

class UPointLightComponent* PointLightComponent;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup3)
int Property_Group3;
};

UCLASS(Blueprintable, BlueprintType, hideCategories = MyGroup3)
class AMyActor_NoComponentWrapperClass : public AActor //Property_Group3 will be
hidden
{
GENERATED_UCLASS_BODY()
public:
UPROPERTY(BlueprintReadOnly, VisibleAnywhere)
class UPointLightComponent* PointLightComponent;

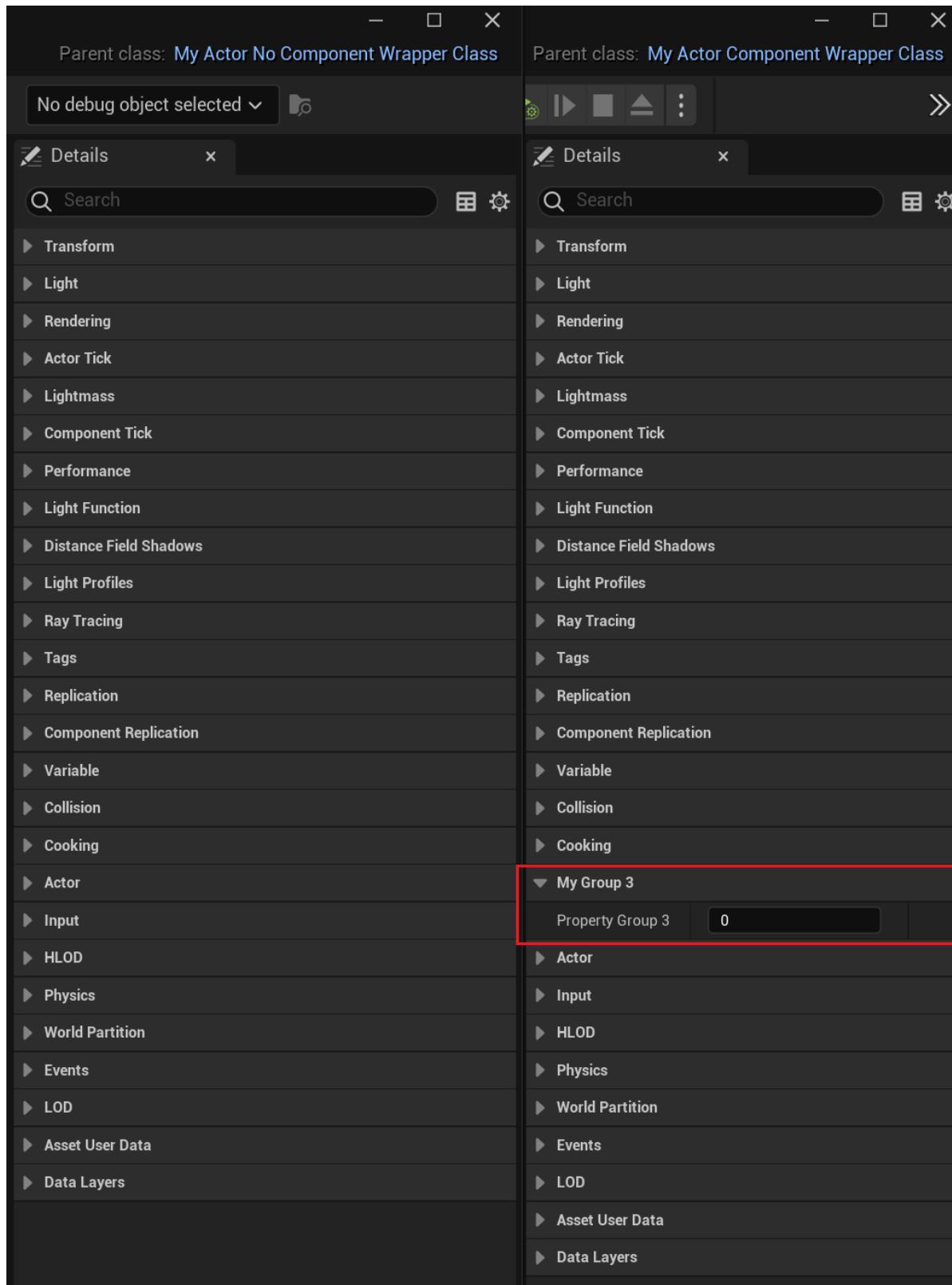
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup3)
int Property_Group3;
};

```

## The Actual Functionality of the Subclass:

---

Even if MyGroup3 in the subclass is hidden, it will still be displayed.



## Principles:

ComponentWrapperClass effectively adds the metadata `IgnoreCategoryKeywordsInSubclasses=true`. Consequently, once the `IgnoreCategoryKeywordsInSubclasses` metadata is present, subsequent settings such as `ShowCategories` are not evaluated.

Currently, only `BlueprintEditorUtils.cpp` in the source code is using this, which is part of the blueprint opening process. Thus, this setting only takes effect when a blueprint is opened by double-clicking. For standard UObject classes, windows created directly using testprops will not be affected since they are not opened by double-clicking a blueprint.

```

case EClassMetadataSpecifier::ComponentWrapperClass:
    MetaData.Add(NAME_IgnoreCategoryKeywordsInSubclasses, TEXT("true"));
    //IgnoreCategoryKeywordsInSubclasses"
    break;
/////////////////////////////////////////////////////////////////
E:\P4V\Engine\Source\Editor\UnrealEd\Private\Kismet2\BlueprintEditorUtils.cpp
void FBlueprintEditorUtils::RecreateClassMetaData(UBlueprint* Blueprint, uclass* Class, bool bRemoveExistingMetaData)

if (!ParentClass-
>HasMetaData(FBlueprintMetadata::MD_IgnoreCategoryKeywordsInSubclasses)) //should
this setting be omitted:
{
    // we want the categories just as they appear in the parent class
    // (set bHomogenize to false) - especially since homogenization
    // could inject spaces

    //with the absence of this setting, the subclass will inherit the
directory settings from the parent class.

    FEditorCategoryUtils::GetClassHideCategories(ParentClass,
AllHideCategories, /*bHomogenize =*/false);
    if (ParentClass->HasMetaData(TEXT("ShowCategories")))
    {
        Class->SetMetaData(TEXT("ShowCategories"), *ParentClass-
>GetMetaData("ShowCategories"));
    }
    if (ParentClass->HasMetaData(TEXT("AutoExpandCategories")))
    {
        Class->SetMetaData(TEXT("AutoExpandCategories"), *ParentClass-
>GetMetaData("AutoExpandCategories"));
    }
    if (ParentClass->HasMetaData(TEXT("AutoCollapseCategories")))
    {
        Class->SetMetaData(TEXT("AutoCollapseCategories"), *ParentClass-
>GetMetaData("AutoCollapseCategories"));
    }
    if (ParentClass->HasMetaData(TEXT("PrioritizeCategories")))
    {
        Class->SetMetaData(TEXT("PrioritizeCategories"), *ParentClass-
>GetMetaData("PrioritizeCategories"));
    }
}

```

## AdvancedClassDisplay

- **Function description:** By default, display all properties of this class in the advanced directory
- **Engine module:** Category
- **Metadata type:** bool
- **Action mechanism:** Add AdvancedClassDisplay to the Meta class
- **Commonly used:** ★★★★

Make all properties of this class appear under the "Advanced" section of the class's Detail panel.

However, this can be overridden by using SimpleDisplay on an individual property. After searching through the source code, it was found that only 3 Actors use AdvancedClassDisplay, and none of these 3 Actors have redefined properties.

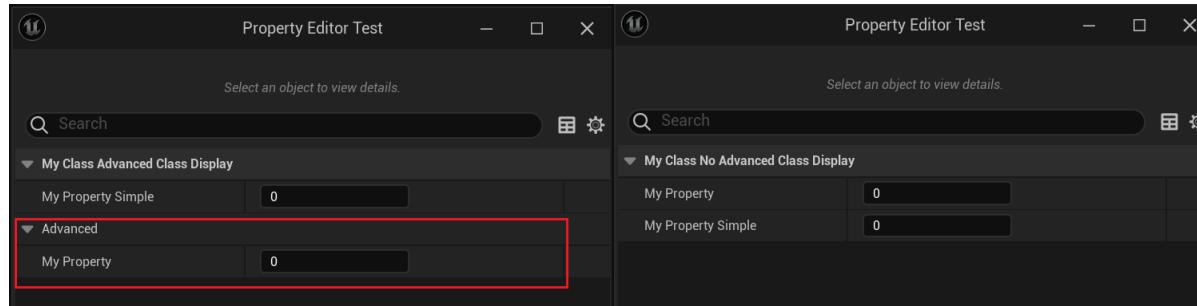
## Sample Code:

```
UCLASS(Blueprintable,AdvancedClassDisplay)
class INSIDER_API UMyClass_AdvancedClassDisplay :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, SimpleDisplay)
    int32 MyProperty_Simple;
};

UCLASS(Blueprintable)
class INSIDER_API UMyClass_NoAdvancedClassDisplay :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, SimpleDisplay)
    int32 MyProperty_Simple;
};
```

## Example Effect:

MyProperty\_Simple remains a straightforward display, even within the AdvancedClassDisplay class.



## Principle:

```
// Property is advanced if it is marked advanced or the entire class is advanced
// and the property not marked as simple
static const FName Name_AdvancedClassDisplay("AdvancedClassDisplay");
bool bAdvanced = Property.IsValid() ? (Property->HasAnyPropertyFlags(CPF_AdvancedDisplay) || (!Property->HasAnyPropertyFlags(CPF_SimpleDisplay) && Property->GetOwnerClass() && Property->GetOwnerClass()->GetBoolMetaData(Name_AdvancedClassDisplay) ) ) : false;
```

# HideDropDown

- **Function Description:** Hide this class within the class selector
- **Engine Module:** TypePicker
- **Metadata Type:** bool
- **Action Mechanism:** Add CLASS\_HideDropDown to ClassFlags
- **Commonality:** ★★

Within the class selector, hide this class, typically triggered by TSubClassOf or initiated by the Class variable. In such cases, this specifier can prevent its appearance. In source code, this is usually applied to deprecated classes, Test classes, Abstract classes, and base classes.

## Example Code:

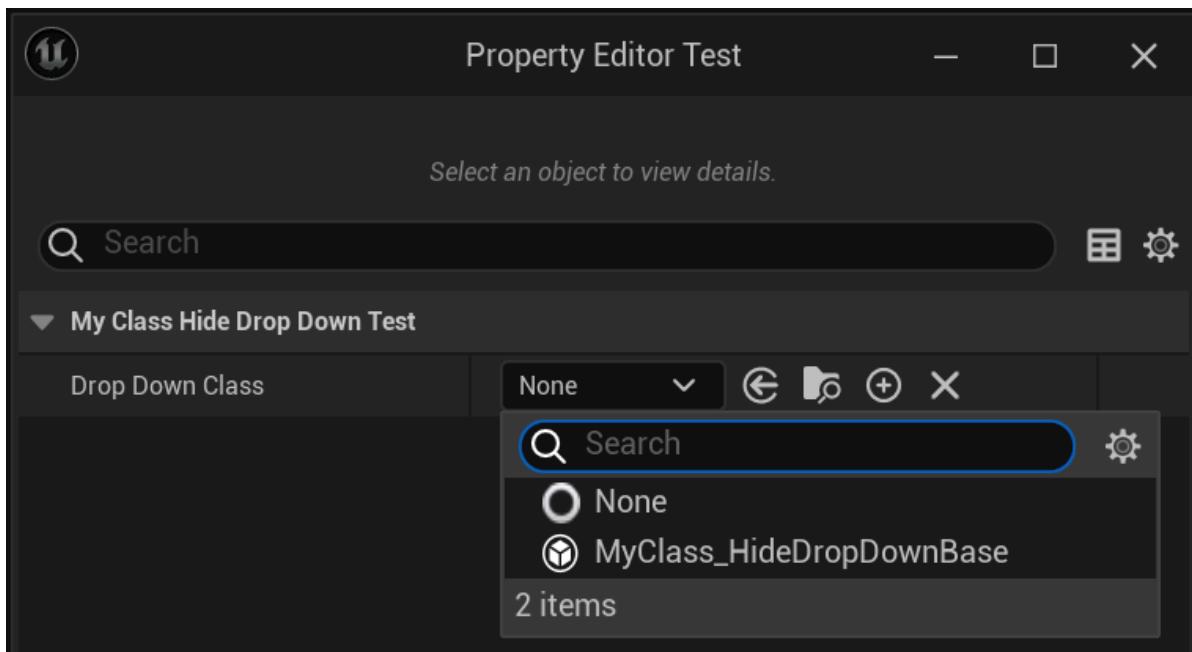
```
UCLASS(Blueprintable)
class INSIDER_API UMyClass_HideDropDownBase :public UObject
{
    GENERATED_BODY()
public:
};

UCLASS(Blueprintable, hidedropdown)
class INSIDER_API UMyClass_HideDropDown :public UMyClass_HideDropDownBase
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty;
};

UCLASS(Blueprintable, hidedropdown)
class INSIDER_API UMyClass_NoHideDropDown :public UMyClass_HideDropDownBase
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty;
};

UCLASS(Blueprintable)
class INSIDER_API UMyClass_HideDropDown_Test :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TSubclassOf<UMyClass_HideDropDownBase> DropDownClass;
};
```

## Example Results:



## Conceptual Framework:

The HideDropDown flag sets the CLASS\_HideDropDown marker, thereby excluding the class from the type UI customization list.

```
template <typename TClass>
bool FPropertyEditorClassFilter::IsClassAllowedHelper(TClass InClass)
{
    bool bMatchesFlags = !InClass->HasAnyClassFlags(CLASS_Hidden |
CLASS_HideDropDown | CLASS_Deprecated) &&
    (bAllowAbstract || !InClass->HasAnyClassFlags(CLASS_Abstract));

    if (bMatchesFlags && InClass->IsChildOf(ClassPropertyMetaClass)
        && (!InterfaceThatMustBeImplemented || InClass-
>ImplementsInterface(InterfaceThatMustBeImplemented)))
    {
        auto PredicateFn = [InClass](const UClass* Class)
        {
            return InClass->IsChildOf(Class);
        };

        if (DisallowedClassFilters.FindByPredicate(PredicateFn) == nullptr &&
            (AllowedClassFilters.Num() == 0 ||
AllowedClassFilters.FindByPredicate(PredicateFn) != nullptr))
        {
            return true;
        }
    }

    return false;
}
```

# Deprecated

- **Function description:** Indicates that this class is deprecated.
- **Engine module:** Development
- **Metadata type:** bool
- **Mechanism of action:** Add CLASS\_Deprecated and CLASS\_NotPlaceable to ClassFlags, and add DeprecationMessage and DeprecatedProperty to Meta
- **Commonly used:** ★★★

Indicates that this class is deprecated.

- Classes will be filtered out of the inheritance list.

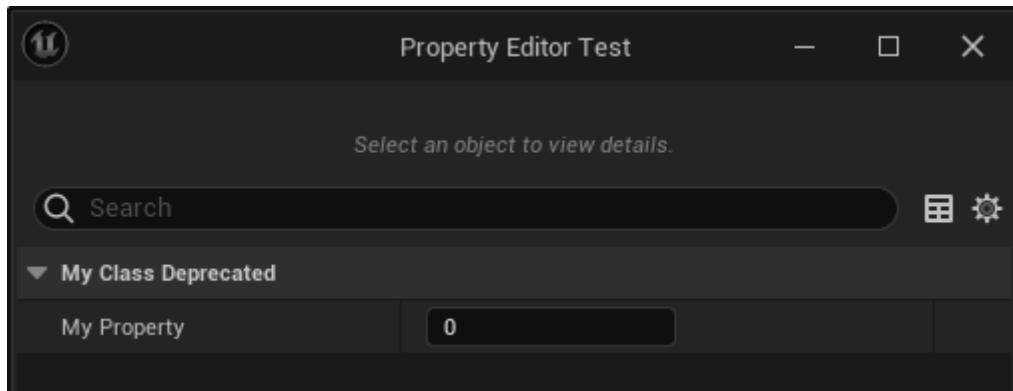
## Sample Code 1:

```
UCLASS(Blueprintable)
class INSIDER_API UMyClass_Deprecated :public UObject
{
    GENERATED_BODY()
};

//Change to:
UCLASS(Blueprintable, Deprecated)
class INSIDER_API UDEPRECATED_MyClass_Deprecated :public UObject
{
    GENERATED_BODY()
};
```

## Example Effect 1:

NewObject can still be used.



## Sample Code 2:

However, note that this is a UE marking. In the source code, you may see many uses of the UE\_DEPRECATED macro, which is a VS compiler-level marking that generates warnings during the compilation process based on usage references.

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_Deprecated_Test :public UObject
{
    GENERATED_BODY()
```

```

public:

    UE_DEPRECATED(5.2, "MyClass_Deprecated has been deprecated, please remove
it.")

    UDEPRECATED_MyClass_Deprecated* MyProperty_Deprecated;

    UE_DEPRECATED(5.2, "MyIntProperty has been deprecated, please remove
it.")

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta=(DeprecatedProperty,
DeprecationMessage = "MyIntProperty has been deprecated."))
    int MyIntProperty;

    UE_DEPRECATED(5.2, "MyClass_Deprecated has been deprecated, please remove
it.")

    void MyFunc(UDEPRECATED_MyClass_Deprecated* obj){}

    UFUNCTION(BlueprintCallable, meta = (DeprecatedProperty,
DeprecationMessage="MyVoidFunc has been deprecated."))
    void MyVoidFunc(){}
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_Deprecated_Usage :public UObject
{
    GENERATED_BODY()
public:

    void MyFunc()
    {
        UMyClass_Deprecated_Test* obj=NewObject<UMyClass_Deprecated_Test>();
        UDEPRECATED_MyClass_Deprecated* obj2 =
        NewObject<UDEPRECATED_MyClass_Deprecated>();
        obj->MyProperty_Deprecated= obj2;
        obj->MyProperty_Deprecated->MyFunc();

        obj->MyIntProperty++;
        obj->MyFunc(obj2);
        obj->MyVoidFunc();
    }
};

```

编译警告:

```

warning C4996: 'UMyClass_Deprecated_Test::MyProperty_Deprecated':
MyClass_Deprecated has been deprecated, please remove it. Please update your code
to the new API before upgrading to the next release, otherwise your project will
no longer compile.
warning C4996: 'UMyClass_Deprecated_Test::MyProperty_Deprecated':
MyClass_Deprecated has been deprecated, please remove it. Please update your code
to the new API before upgrading to the next release, otherwise your project will
no longer compile.
warning C4996: 'UMyClass_Deprecated_Test::MyIntProperty': MyIntProperty has been
deprecated, please remove it. Please update your code to the new API before
upgrading to the next release, otherwise your project will no longer compile.
warning C4996: 'UMyClass_Deprecated_Test::MyFunc': MyClass_Deprecated has been
deprecated, please remove it. Please update your code to the new API before
upgrading to the next release, otherwise your project will no longer compile.

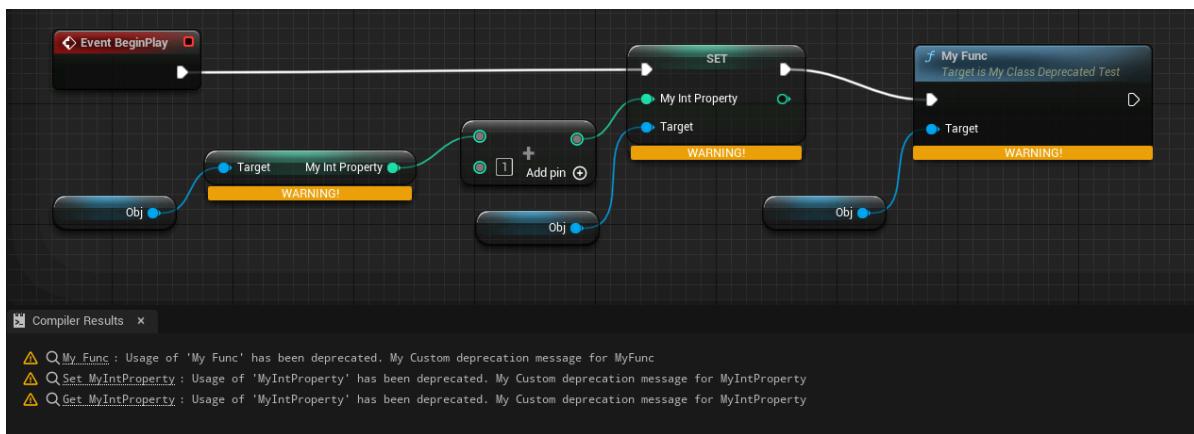
```

注意如果没有UE\_DEPRECATED标记，则不会生成编译警告。

```
UPROPERTY(EditAnywhere, BlueprintReadWrite) int MyInt2Property_DEPRECATED;  
会触发：  
warning : Member variable declaration: Deprecated property  
'MyInt2Property_DEPRECATED' should not be marked as blueprint visible without  
having a BlueprintGetter  
warning : Member variable declaration: Deprecated property  
'MyInt2Property_DEPRECATED' should not be marked as blueprint writable without  
having a BlueprintSetter  
warning : Member variable declaration: Deprecated property  
'MyInt2Property_DEPRECATED' should not be marked as visible or editable  
因此只能改成：  
UPROPERTY() int MyInt2Property_DEPRECATED;
```

## Example Effect 2:

When attributes and functions are marked with Deprecated, warnings will be generated during BP compilation. Note that the function must first be a normal function, and after connections are completed in BP, marking DeprecatedFunction in C++ will generate a warning. Otherwise, a deprecated function cannot be called in BP.



## Principle:

In the source code, there are numerous judgments for CLASS\_DEPRECATED, such as in SpawnActor:

```
AActor* Uworld::SpawnActor( Uclass* Class, FTransform const* UserTransformPtr,  
const FActorSpawnParameters& SpawnParameters )  
{  
    if( Class->HasAnyClassFlags(CLASS_Deprecated) )  
    {  
        UE_LOG(LogSpawn, Warning, TEXT("SpawnActor failed because class %s is  
deprecated"), *Class->GetName());  
        return NULL;  
    }  
}
```

# Experimental

- **Function Description:** Indicates that this class is a trial version, currently without documentation, and may be deprecated in the future.
- **Engine Module:** Development
- **Metadata Type:** bool
- **Functionality Mechanism:** Add `DevelopmentStatus` to `Meta`, marking the class as Experimental
- **Usage Frequency:** ★★★

Indicates that this class is a trial version, currently without documentation, and may be deprecated in the future.

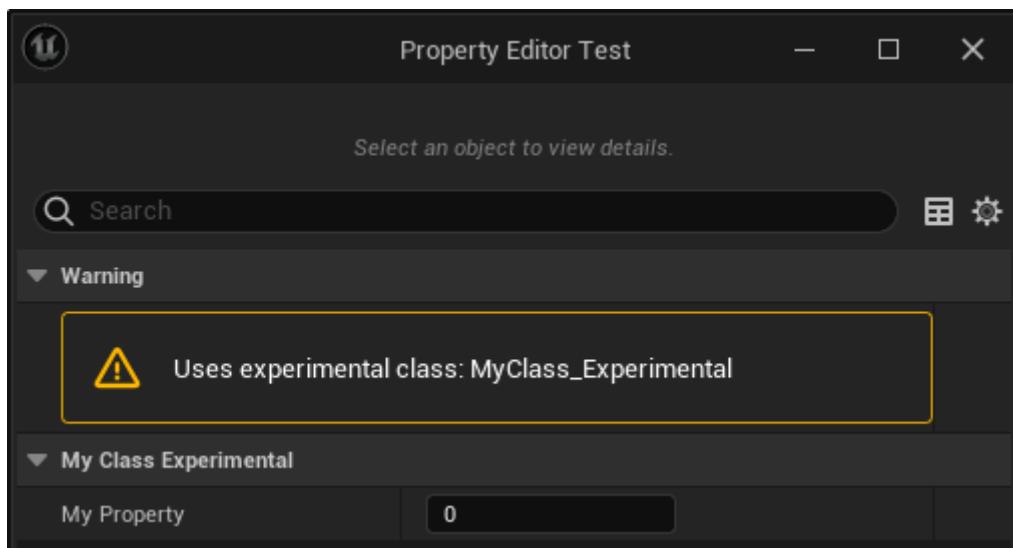
The example in the source code is the `Paper2D` class. This tag adds `{ "DevelopmentStatus": "Experimental" }` to the class's metadata.

## Example Code:

```
/*
(BlueprintType = true, DevelopmentStatus = Experimental, IncludePath =
Class/Display/MyClass_Deprecated.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Display/MyClass_Deprecated.h)
*/
UCLASS(Blueprintable, Experimental)
class INSIDER_API UMyClass_Experimental :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty;
    UFUNCTION(BlueprintCallable)
        void MyFunc();
};
```

## Example Effect:



# EarlyAccessPreview

- **Function Description:** Indicates that this class is an early preview version, which is more refined than a beta version but is not yet at the product level.
- **Engine Module:** Development
- **Metadata Type:** bool
- **作用机制:** 在Meta中添加DevelopmentStatus，将类标记为Early Access
- **Common Usage:** ★★★

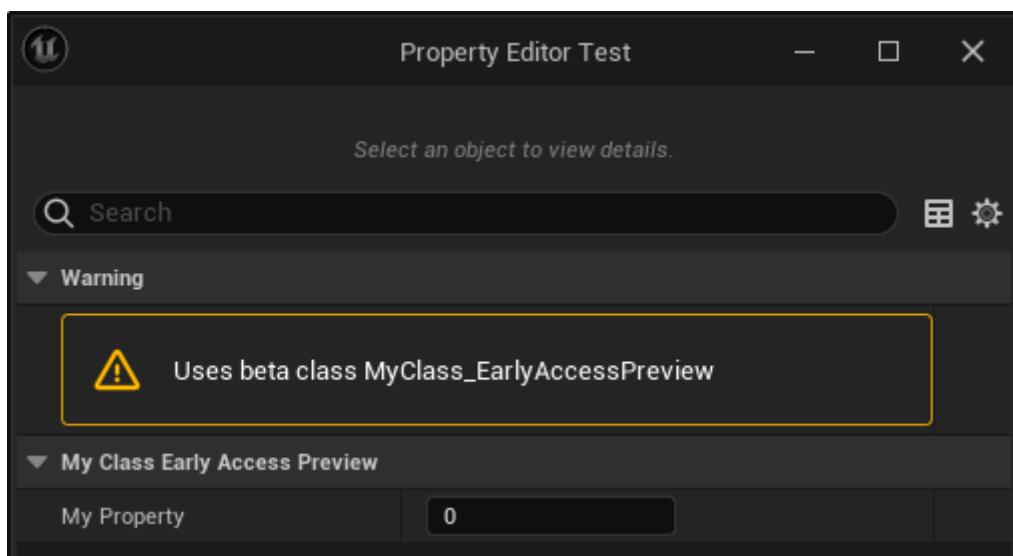
Indicates that this class is an early preview version, which is more complete than a trial version but still not at the product level.

This tag will append { "DevelopmentStatus", "EarlyAccess" } to the class's metadata.

## Sample Code:

```
//(BlueprintType = true, DevelopmentStatus = EarlyAccess, IncludePath =
Class/Display/MyClass_Deprecated.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Display/MyClass_Deprecated.h)
UCLASS(Blueprintable, EarlyAccessPreview)
class INSIDER_API UMyClass_EarlyAccessPreview :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
    UFUNCTION(BlueprintCallable)
    void MyFunc() {}
};
```

## Example Results:



# Within

- **Function description:** Specifies that an object must be created with a dependency on an instance of OuterClassName as its Outer.
- **Engine module:** Instance
- **Metadata type:** string="abc"
- **Action mechanism:** Stored in the XXX field of UClass\* UClass::ClassWithin=XXX
- **Commonly used:** ★★★

Specifies that an object must depend on an instance of OuterClassName as its Outer when it is created.

Objects of this class cannot exist outside of an instance of OuterClassName. This implies that to instantiate an object of this class, an instance of OuterClassName must be provided as its Outer object.

This class is typically used as a sub-object in such scenarios.

## Sample Code:

```
UCLASS(within= MyClass_within_Outer)
class INSIDER_API UMyClass_within :public UObject
{
    GENERATED_BODY()
};

UCLASS()
class INSIDER_API UMyClass_within_Outer :public UObject
{
    GENERATED_BODY()
public:
};
```

## Example Results:

```
//Error! Fatal error: Object MyClass_within None created in Package instead of
MyClass_within_Outer
UMyClass_within* obj=NewObject<UMyClass_within>();

//Correct:
UMyClass_within_Outer* objouter = NewObject<UMyClass_within_Outer>();
UMyClass_within* obj=NewObject<UMyClass_within>(objouter);
```

## Principle:

The generated UClass field: UClass\* ClassWithin will hold this information, and during creation, StaticAllocateObject will check (bCreatingCDO || !InOuter || InOuter->IsA(InClass->ClassWithin)). Hence, the Within object must be created first.

```

bool staticAllocateObjectErrorTests( const UClass* InClass, UObject* InOuter,
FName InName, EObjectFlags InFlags)
{
    if ( (InFlags & (RF_ClassDefaultObject|RF_ArchetypeObject)) == 0 )
    {
        if ( InOuter != NULL && !InOuter->IsA(InClass->ClassWithin) )
        {
            UE_LOG(Log UObject Globals, Fatal, TEXT("%s"), *FString::Printf(
TEXT("Object %s %s created in %s instead of %s"), *InClass->GetName(),
*InName.ToString(), *InOuter->GetClass()->GetName(), *InClass->ClassWithin-
>GetName() );
            return true;
        }
    }
}

```

You can find many usages of Within by searching through the source code

```

UCLASS(Within=Engine, config=Engine, transient)
class ENGINE_API ULocalPlayer

UCLASS(Abstract, DefaultToInstanced, Within=UserWidget)
class UMG_API UUserWidgetExtension : public UObject
{

```

## DefaultToInstanced

---

- **Function Description:** Specifies that all instance attributes of this class are set to UPROPERTY(instanced) by default, meaning that new instances are created rather than references to existing objects.
- **Engine Module:** Instance
- **Metadata Type:** bool
- **Functionality Mechanism:** Add CLASS\_DefaultToInstanced to ClassFlags
- **Common Usage:** ★★★★

Specifies that all instance properties of this class are set to UPROPERTY(instanced) by default, meaning that new instances are created rather than references to existing objects.

UPROPERTY(instanced) signifies that the property will have the CPF\_InstancedReference attribute, meaning an object instance is created for this property.

An instance refers to creating an object for the UObject pointer, rather than defaulting to referencing an existing object within the engine.

It is often used in conjunction with EditInlineNew to allow object instances to be created within the details panel.

UActorComponent inherently has the DefaultToInstanced attribute.

## Sample Code:

```
UCLASS(Blueprintable)
class INSIDER_API UMyClass_NotDefaultToInstanced :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
};

// ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI |
CLASS_DefaultToInstanced | CLASS_TokenStreamAssembled | CLASS_Intrinsic |
CLASS_Constructed
UCLASS(Blueprintable, DefaultToInstanced)
class INSIDER_API UMyClass_DefaultToInstanced :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
};

// ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_EditInlineNew |
CLASS_RequiredAPI | CLASS_DefaultToInstanced | CLASS_TokenStreamAssembled |
CLASS_Intrinsic | CLASS_Constructed
UCLASS(Blueprintable, DefaultToInstanced, EditInlineNew)
class INSIDER_API UMyClass_DefaultToInstanced_EditInlineNew :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
};

UCLASS(Blueprintable, EditInlineNew)
class INSIDER_API UMyClass_NotDefaultToInstanced_EditInlineNew :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_DefaultToInstanced_Test :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "NormalProperty")
    UMyClass_NotDefaultToInstanced* MyObject_NotDefaultToInstanced;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "NormalProperty")
    UMyClass_DefaultToInstanced* MyObject_DefaultToInstanced;
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category =
"NormalProperty | Instanced")
UMyClass_NotDefaultToInstanced* MyObject_NotDefaultToInstanced_Instanced;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category =
"NormalProperty | Instanced")
UMyClass_DefaultToInstanced* MyObject_DefaultToInstanced_Instanced;

public:
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "EditInlineNew")
UMyClass_NotDefaultToInstanced_EditInlineNew*
MyObject_NotDefaultToInstanced_EditInlineNew;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "EditInlineNew")
UMyClass_DefaultToInstanced_EditInlineNew*
MyObject_DefaultToInstanced_EditInlineNew;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category =
>EditInlineNew | Instanced")
UMyClass_NotDefaultToInstanced_EditInlineNew*
MyObject_NotDefaultToInstanced_EditInlineNew_Instanced;

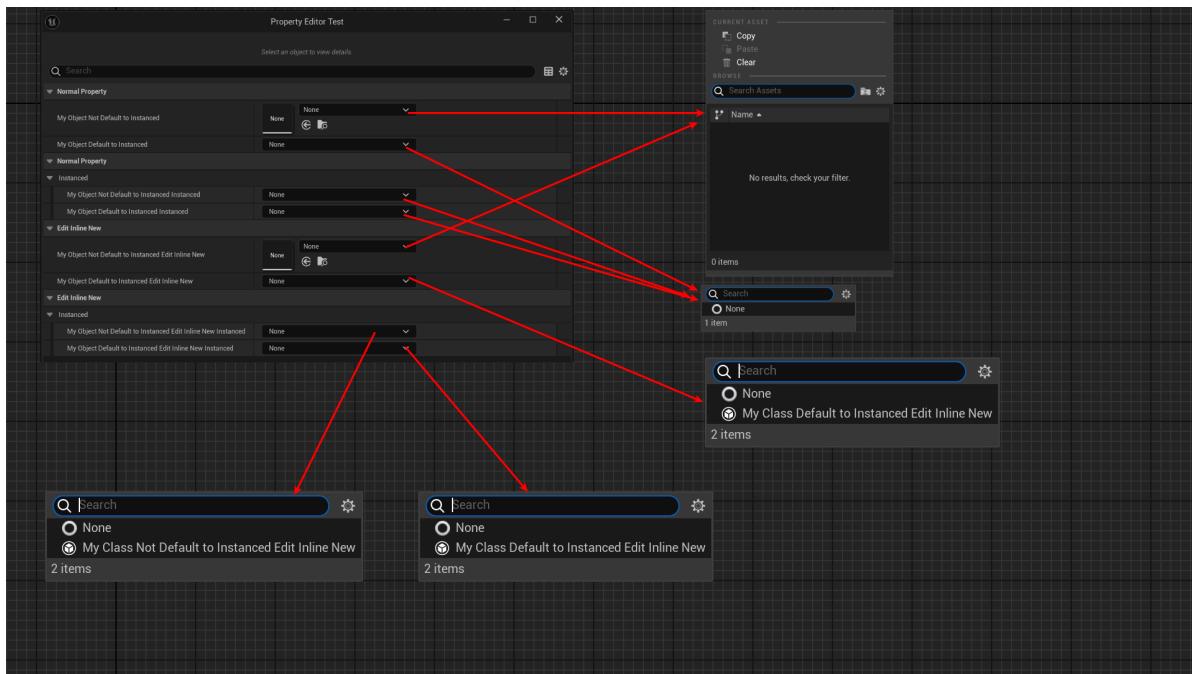
UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category =
>EditInlineNew | Instanced")
UMyClass_DefaultToInstanced_EditInlineNew*
MyObject_DefaultToInstanced_EditInlineNew_Instanced;
};

```

## Example Effect:

---

- MyObject\_NotDefaultToInstanced and MyObject\_NotDefaultToInstanced\_EditInlineNew lack the instanced mark on their attributes, so they present a list of object references when opened.
- MyObject\_DefaultToInstanced has the DefaultToInstanced attribute on the class, making the property instanced. Of course, the instanced mark can also be added manually to attributes, as seen with MyObject\_NotDefaultToInstanced\_Instanced and MyObject\_DefaultToInstanced\_Instanced. A window for creating instances appears, but instances cannot yet be created directly within the details panel.
- MyObject\_DefaultToInstanced\_EditInlineNew, MyObject\_NotDefaultToInstanced\_EditInlineNew\_Instanced, and MyObject\_DefaultToInstanced\_EditInlineNew\_Instanced can all create object instances directly in the details panel. This is because the class itself must have EditInlineNew, and the attribute must also have the Instanced mark (either by setting DefaultToInstanced on the class, making all attributes of the class automatically instanced, or by setting Instanced individually on the attribute)



## Principle:

```
uobject* FObjectInstancingGraph::InstancePropertyValue(uobject*
SubObjectTemplate, uobject* CurrentValue, uobject* Owner,
EInstancePropertyValueFlags Flags)
{
    if (CurrentValue->GetClass()>HasAnyClassFlags(CLASS_DefaultToInstanced))
    {
        bCausesInstancing = true; // these are always instanced no matter what
    }
}
```

## EditInlineNew

- Function Description:** Specifies that objects of this class can be directly created inline within the property details panel, and this should be coordinated with the attribute's Instanced property.
- Engine Module:** Instance
- Metadata Type:** bool
- Functionality Mechanism:** Add CLASS\_EditInlineNew to ClassFlags
- Associated Items:** NotEditInlineNew (NotEditInlineNew.md)
- Commonly Used:** ★★★★☆

Objects of this class can be directly created inline within the property details panel.

If you wish to create objects directly in the details panel, the attribute must first be marked as Instanced or ShowInnderProperties.

EditInlineNew is primarily used on subclasses of UObject. Typically, classes not marked with EditInlineNew are used for references to Actors or assets. Note that EditInlineNew indicates the capability to create object instances directly from the property details panel, not a restriction to creation only within the panel. Of course, you can also manually create a new object and assign it to the object reference attribute.

This is independent of the Instanced capability on UPROPERTY. If EditInlineNew is not added to UCLASS, but Instanced is added to an attribute, after manually assigning a new object to the attribute, the attribute will also expand to show internal properties for editing. This is because the Instanced attribute automatically adds the EditInline meta to the property.

This specifier is inherited by all subclasses; subclasses can override it using the NotEditInlineNew specifier.

## Sample Code:

```
UCLASS(Blueprintable, EditInlineNew)
class INSIDER_API UMyClass_EditInlineNew :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty;
};

UCLASS(Blueprintable, NotEditInlineNew)
class INSIDER_API UMyClass_NotEditInlineNew :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_Edit_Test :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category = InstancedProperty)
        UMyClass>EditInlineNew* MyEditInlineNew;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category = InstancedProperty)
        UMyClass>_NotEditInlineNew* MyNotEditInlineNew;

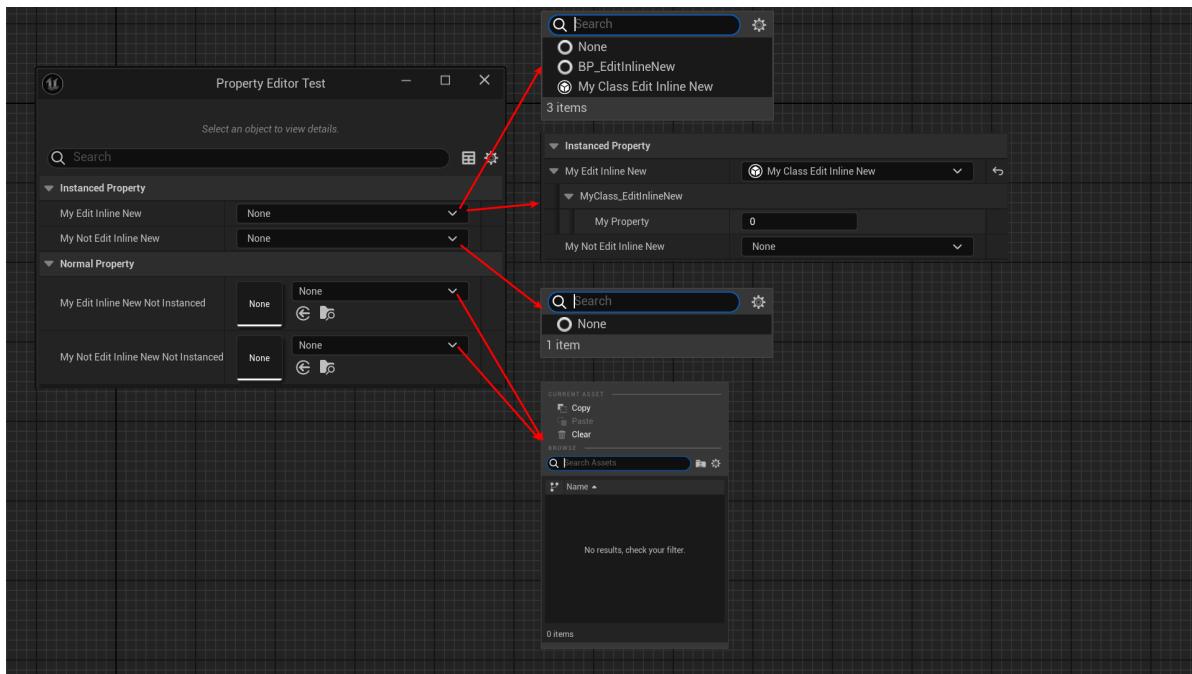
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NormalProperty)
        UMyClass>EditInlineNew* MyEditInlineNew_NotInstanced;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NormalProperty)
        UMyClass>_NotEditInlineNew* MyNotEditInlineNew_NotInstanced;
};
```

## Example Effect:

EditInlineNew supports direct creation of object instances from C++ or BP subclasses, allowing editing of the instances on them.

Properties marked with NotEditInlineNew cannot find a supported class to create an object.

If an attribute does not have Instanced, you can only attempt to reference it (the object cannot be found).



## Principle:

The class's ability to be created inline for editing is determined by whether it has the CLASS\_EditInlineNew flag.

```
template <typename TClass, typename TIsChildOfFunction>
bool FPropertyEditorInlineClassFilter::IsClassAllowedHelper(TClass* InClass,
TIsChildOfFunction IsClassChildOf, TSharedRef< FClassViewerFilterFuncs >
InFilterFuncs)
{
    const bool bMatchesFlags = InClass->HasAnyClassFlags(CLASS>EditInlineNew) &&
        !InClass->HasAnyClassFlags(CLASS>Hidden | CLASS>HideDropDown |
CLASS>Deprecated) &&
        (bAllowAbstract || !InClass->HasAnyClassFlags(CLASS>Abstract));
}
```

## NotEditInlineNew

- **Function description:** Cannot create via the EditInline button
- **Engine module:** Instance
- **Metadata type:** boolean
- **Action mechanism:** Remove CLASS>EditInlineNew from ClassFlags
- **Associated items:** EditInlineNew (EditInlineNew.md)
- **Commonality:** ★

## NotPlaceable

- **Function description:** Indicates that this Actor cannot be placed within a level
- **Engine module:** Behavior
- **Metadata type:** bool
- **Action mechanism:** Adds CLASS>NotPlaceable to ClassFlags

- **Associated items:** Placeable (Placeable.md)

- **Commonality:** ★★★

Indicates that this Actor cannot be placed in a level and cannot be dragged into the scene. It invalidates the Placeable specifier inherited from the base class. It will be marked with CLASS\_NotPlaceable in ClassFlags, a mark that is inheritable, meaning all its subclasses are also not placeable by default. For instance, AWorldSettings is an Actor that is not placeable.

However, note that this class can still be dynamically spawned into the level using SpawnActor.

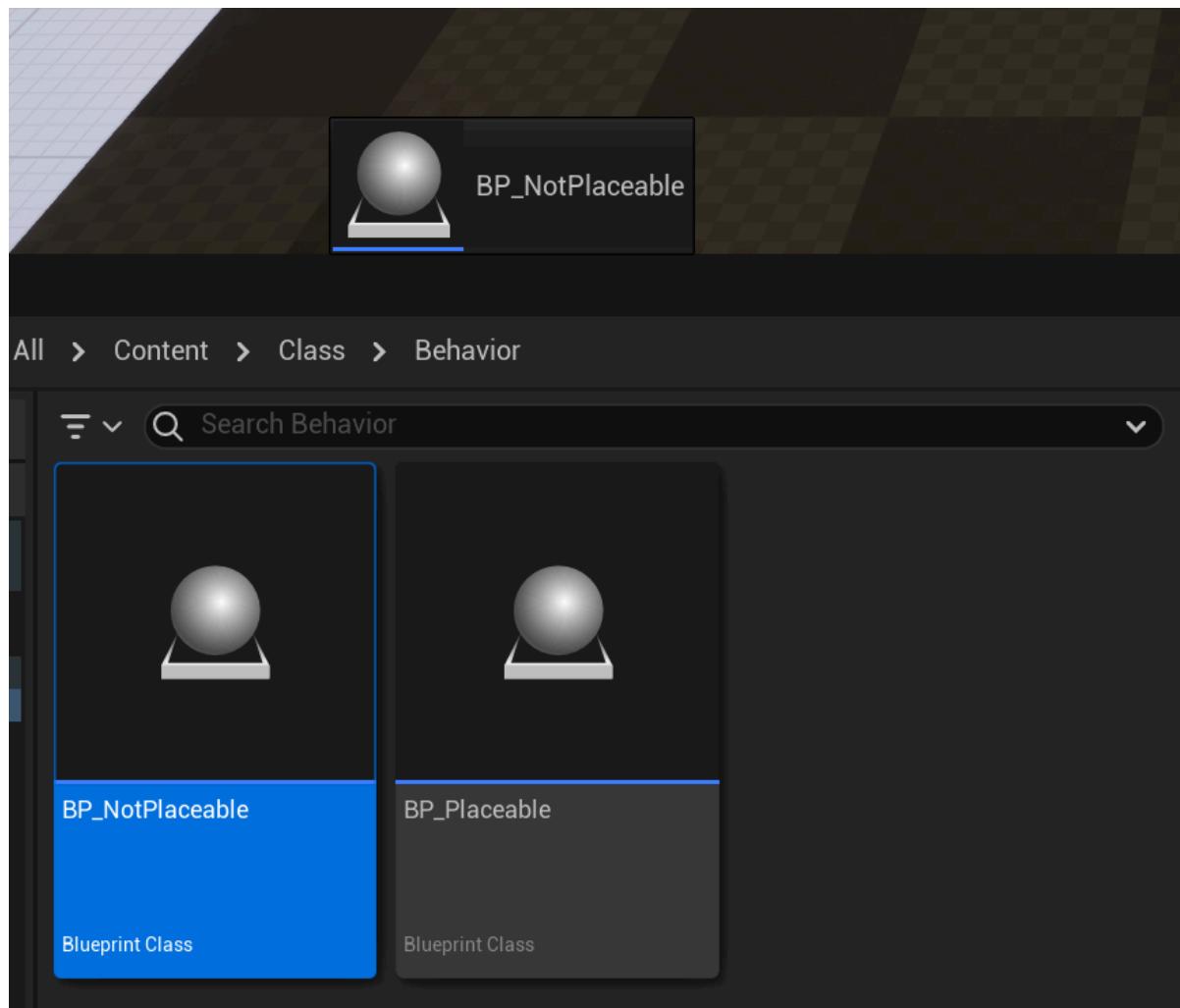
Classes marked as NotPlaceable do not appear in the class selection for PlaceMode.

## Sample Code:

```
UCLASS(Blueprintable, BlueprintType, NotPlaceable)
class INSIDER_API AMyActor_NotPlaceable :public AActor
{
    GENERATED_BODY()
};
```

## Example Effect:

When dragged into the scene, it will be impossible to create the Actor.



## Fundamental Principle:

If it is a direct C++ class, such as AMyActor\_NotPlaceable, it can be directly dragged from the ContentBrowser into the scene. The source code reveals that only subclasses that inherit from Blueprints are subject to this restriction.

```
TArray<AActor*> FLevelEditorviewportClient::TryPlacingActorFromObject( ULevel*  
InLevel, UObject* ObjToUse, bool bSelectActors, EObjectFlags ObjectFlags,  
UActorFactory* FactoryToUse, const FName Name, const FViewportCursorLocation*  
Cursor )  
{  
  
    bool bPlace = true;  
    if (ObjectClass->IsChildOf(UBlueprint::StaticClass()))  
    {  
        UBlueprint* BlueprintObj = StaticCast<UBlueprint*>(ObjToUse);  
        bPlace = BlueprintObj->GeneratedClass != NULL;  
        if(bPlace)  
        {  
            check(BlueprintObj->ParentClass == BlueprintObj->GeneratedClass->GetSuperClass());  
            if (BlueprintObj->GeneratedClass->HasAnyClassFlags(CLASS_NotPlaceable  
| CLASS_Abstract))  
            {  
                bPlace = false;  
            }  
        }  
    }  
  
    if (bPlace)  
    {  
        PlacedActor = FActorFactoryAssetProxy::AddActorForAsset( ObjToUse,  
bSelectActors, ObjectFlags, FactoryToUse, Name );  
        if ( PlacedActor != NULL )  
        {  
            PlacedActors.Add(PlacedActor);  
            PlacedActor->PostEditMove(true);  
        }  
    }  
}
```

## Placeable

- **Function Description:** Indicates that this Actor can be placed within a level.
- **Engine Module:** Scene
- **Metadata Type:** bool
- **Action Mechanism:** Remove CLASS\_NotPlaceable from ClassFlags
- **Associated Items:** NotPlaceable
- **Common Usage:** ★★★

Indicates that the Actor is placeable in the level.

It is placeable by default, so Placeable is not currently used in the source code.

Subclasses can override this flag using the NotPlaceable specifier, just as AInfo and the like set NotPlaceable themselves.

Indicates that this class can be created in the Editor and can be placed into a level, UI scene, or Blueprint (depending on the class type). This flag is inherited by all subclasses

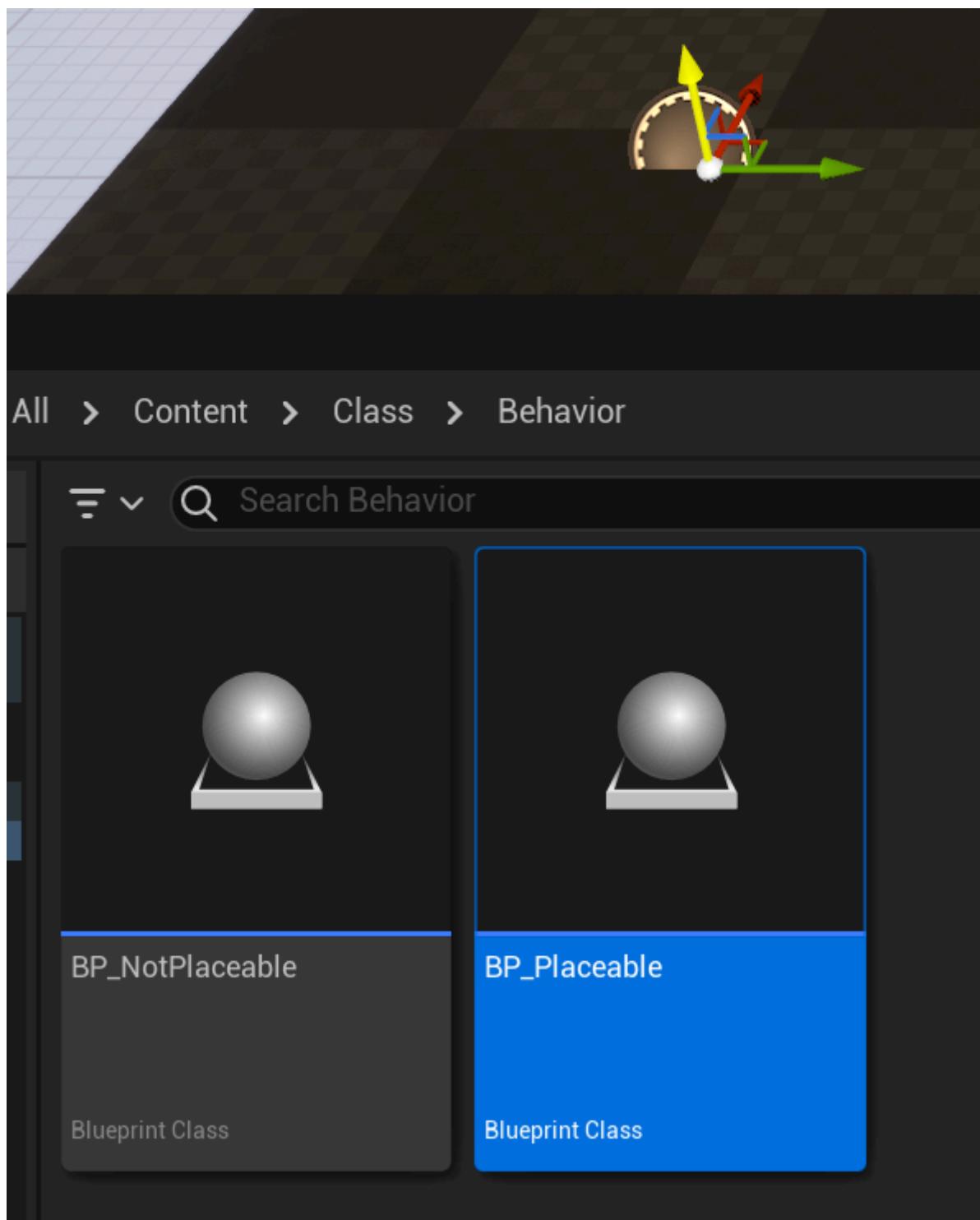
placeable cannot clear the notplaceable mark of the parent class.

## Example Code:

```
UCLASS(Blueprintable, BlueprintType, placeable)
class INSIDER_API AMyActor_Placeable :public AMyActor_NotPlaceable
{
    GENERATED_BODY()
};

error : The 'placeable' specifier cannot override a 'nonplaceable' base class.
Classes are assumed to be placeable by default. Consider whether using the
'abstract' specifier on the base class would work.
```

## Example Effect:



## ConversionRoot

- **Function Description:** Allows Actors within the scene editor to transform between themselves and their subclasses
- **Engine Module:** Scene
- **Metadata Type:** bool
- **Action Mechanism:** Adds IsConversionRoot to the Meta data
- **Usage Frequency:** ★

Generally used on Actors to restrict the level of transformation during an Actor's conversion, such as ASkeletalMeshActor, AStaticMeshActor, etc.

Often appears together with ComponentWrapperClass.

According to the code, the IsConversionRoot in the meta data will restrict the propagation to this level only, without searching further up the hierarchy.

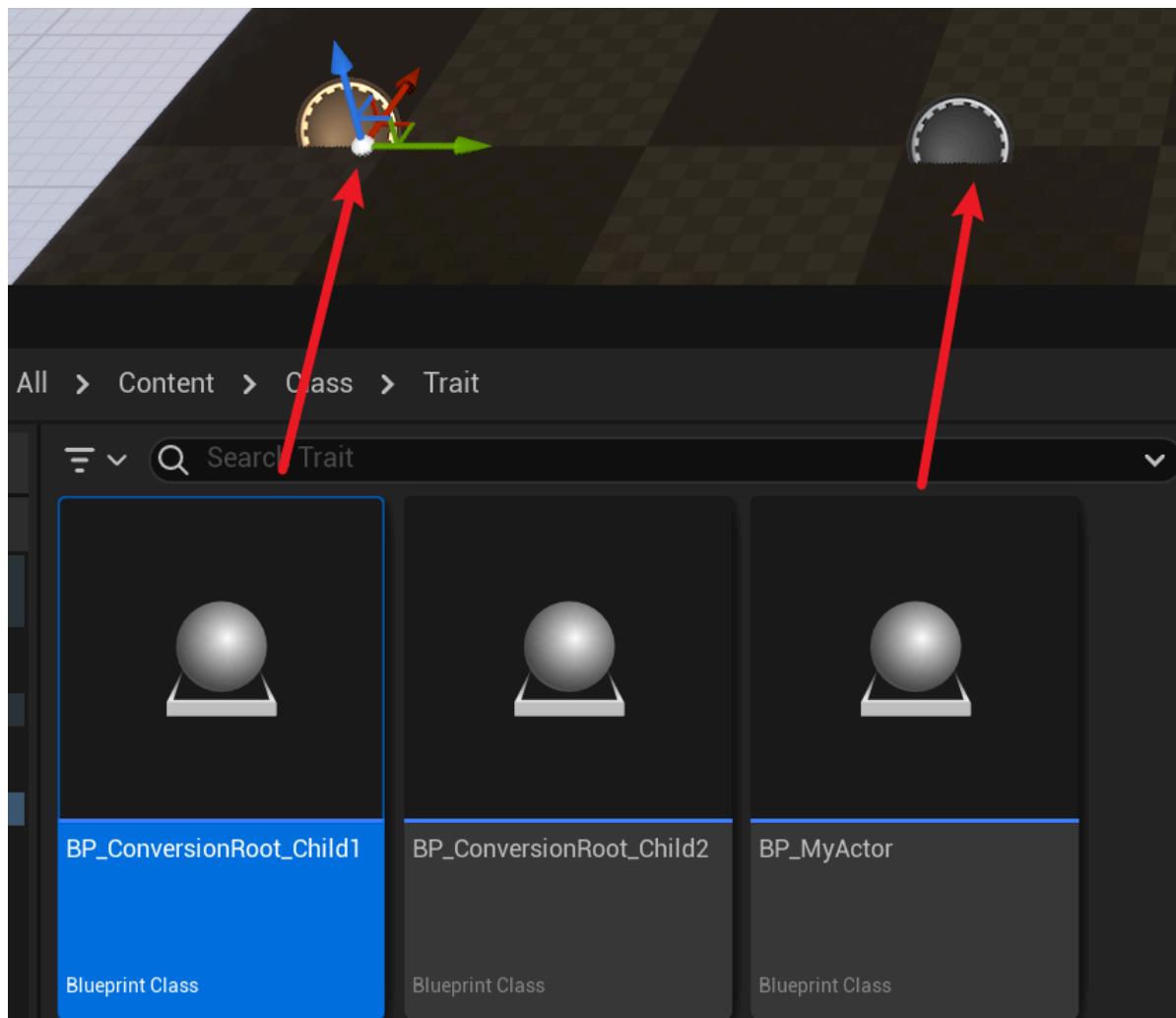
Only Actors with a ConversionRoot set are allowed to perform the Convert Actor action; otherwise, it is disabled.

## Sample Code:

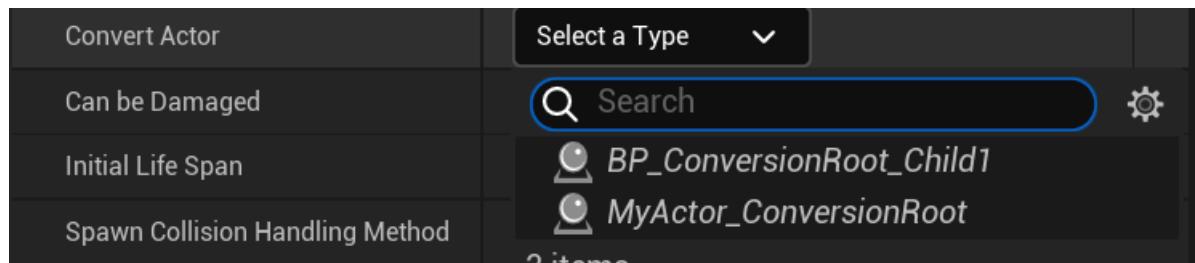
```
//(BlueprintType = true, IncludePath = Class/Trait/MyClass_ConversionRoot.h,
IsBlueprintBase = true, IsConversionRoot = true, ModuleRelativePath =
Class/Trait/MyClass_ConversionRoot.h)
UCLASS(Blueprintable, BlueprintType, ConversionRoot)
class INSIDER_API AMyActor_ConversionRoot :public AActor
{
    GENERATED_BODY()
};
```

## Example Effect:

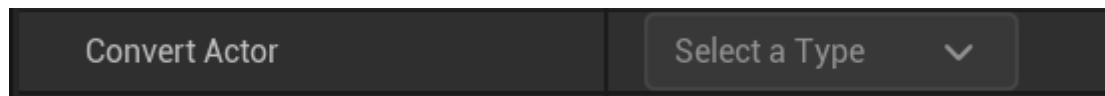
Create subclasses BP\_ConversionRoot\_Child1 and BP\_ConversionRoot\_Child2 in the Blueprint. Then, drag and drop BP\_ConversionRoot\_Child1 into the scene to create an Actor, and also create a standard Blueprint Actor for comparison.



Selecting Child1 in the level will enable the ConvertActor function, allowing transformations between itself and all subclasses of the ConversionRoot.



If it is a standard Actor, no transformation can occur because ConversionRoot is not defined.



## Fundamental Principle:

In the level, when an Actor is selected, the DetailsPanel will display the ConverActor property bar, where another Actor can be selected for transformation.

TSharedRef FActorDetails::MakeConvertMenu( const FSelectedActorInfo& SelectedActorInfo )

This function is used to create the menu for the Select Type Combo Button. Internally, it calls CreateClassPickerConvertActorFilter:

```
uclass* FActorDetails::GetConversionRoot( uclass* InCurrentClass ) const
{
    Uclass* ParentClass = InCurrentClass;

    while(ParentClass)
    {
        if( ParentClass->GetBoolMetaData(FName(TEXT("IsConversionRoot"))) )
        {
            break;
        }
        ParentClass = ParentClass->GetSuperclass();
    }

    return ParentClass;
}

void FActorDetails::CreateClassPickerConvertActorFilter(const
TweakObjectPtr<AActor> ConvertActor, class FClassViewerInitializationOptions*&
ClassPickerOptions)
Filter->AllowedChildofRelationship.Add(RootConversionClass); //Limits the
selection to subclasses below this base class
```

## Config

- **Function description:** Specify the name of the configuration file and save the object's values to the INI configuration file.
- **Engine module:** Config
- **Metadata type:** string="abc"

- **Action mechanism:** The name of the Config file is stored in the parameter FName UClass::ClassConfigName
- **Related items:** PerObjectConfig, ConfigDoNotCheckDefaults, DefaultConfig, GlobalUserConfig, ProjectUserConfig
- **Commonly used:** ★★★★☆

Specify the name of the configuration file and save the object's values to the INI configuration file.

- An entire class is represented by only one section in the INI file, so it is generally a CDO object that is saved, but a regular object can also be used.
- The metadata value for the Config file name is stored in FName UClass::ClassConfigName.
- By default, it is saved in the Local file under Saved/XXX.ini.
- This specifier propagates to all subclasses and cannot be disabled; however, subclasses can change the configuration file by re-declaring the config specifier and providing a different ConfigName.
- Common values for ConfigName include "Engine", "Editor", "Input", and "Game".
- You can manually invoke SaveConfig and LoadConfig to read and write configuration values. The CDO's values will be updated by the engine reading from the configuration.
- Properties intended for saving in the configuration file must be annotated with UPROPERTY(config).

## Sample Code:

```
UCLASS(Config = Game)
class INSIDER_API UMyClass_Config :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertywithConfig = 123;
};

//Test Code
UMyClass_Config* testObject = NewObject<UMyClass_Config>
(GetTransientPackage(), TEXT("testObject"));
testObject->SaveConfig();

//Generate
\Hello\Saved\Config\WindowsEditor\Game.ini
[/Script/Insider.MyClass_Config]
MyPropertywithConfig=123
```

## Principle:

When the engine starts, UObjectLoadAllCompiledInDefaultProperties loads the CDO for all classes, and after a series of function calls, it automatically invokes LoadConfig to initialize the CDO's values.

```

static void UobjectLoadAllCompiledInDefaultProperties(TArray<UClass*>&
OutAllNewClasses)
{
    for (UClass* Class : NewClasses)
    {
        UE_LOG(LogUObjectBootstrap, Verbose, TEXT("GetDefaultObject Begin %s"),
        *Class->GetOutermost()->GetName(), *Class->GetName());
        Class->GetDefaultObject();
        UE_LOG(LogUObjectBootstrap, Verbose, TEXT("GetDefaultObject End %s %s"),
        *Class->GetOutermost()->GetName(), *Class->GetName());
    }
}

```

# PerObjectConfig

---

- **Function description:** Specifies that when a config file name already exists, values should be stored for each object instance rather than for the entire class.
- **Engine module:** Config
- **Metadata type:** bool
- **Action mechanism:** Add CLASS\_PerObjectConfig to ClassFlags
- **Associated items:** Config
- **Commonly used:** ★★★★☆

When a config file name already exists, it specifies that values should be stored per object instance rather than per class.

- The configuration information for this class will be stored per object. In the .ini file, each object will have a dedicated section named after the object, formatted as [ObjectName ClassName].
- This specifier is inherited by subclasses. It indicates that the configuration should be saved individually for each object.

## 1 Example Code:

---

Note that ObjectName must be consistent

```

UCLASS(Config = Game, PerObjectConfig)
class INSIDER_API UMyClass_PerObjectConfig :public Uobject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertyWithConfig = 123;
};

void UMyClass_Config_Test::TestPerObjectConfigSave()
{
    UMyClass_PerObjectConfig* testObject1 = NewObject<UMyClass_PerObjectConfig>
    (GetTransientPackage(), TEXT("testObject1"));
}

```

```

testObject1->MyPropertywithConfig = 456;
testObject1->SaveConfig();

UMyClass_PerObjectConfig* testObject2 = NewObject<UMyClass_PerObjectConfig>
(GetTransientPackage(), TEXT("testObject2"));
testObject2->MyPropertywithConfig = 789;
testObject2->SaveConfig();

}

void UMyClass_Config_Test::TestPerObjectConfigLoad()
{
    UMyClass_PerObjectConfig* testObject1 = NewObject<UMyClass_PerObjectConfig>
(GetTransientPackage(), TEXT("testObject1"));
    //testObject1->LoadConfig();    // LoadConfig does not need to be explicitly
called

    UMyClass_PerobjectConfig* testObject2 = NewObject<UMyClass_PerObjectConfig>
(GetTransientPackage(), TEXT("testObject2"));
    //testObject2->LoadConfig();
}

//\Saved\Config\WindowsEditor\Game.ini
[testObject1 MyClass_PerobjectConfig]
MyPropertyWithConfig=456

[testObject2 MyClass_PerobjectConfig]
MyPropertyWithConfig=789

```

## 2 Principle:

At the end of object construction, the object attempts to read its configuration.

```

void FObjectInitializer::PostConstructInit()
{
    //During the NewObject construction, it will be called subsequently
    if (bIsCDO || Class->HasAnyClassFlags(CLASS_PerObjectConfig))
    {
        Obj->LoadConfig(NULL, NULL, bIsCDO ? UE::LCPF_ReadParentSections :
UE::LCPF_None);
    }
}

```

## ConfigDoNotCheckDefaults

- **Function description:** Specifies to overlook the consistency check of the parent configuration values when saving configuration values.
- **Engine module:** Config
- **Metadata type:** bool
- **Action mechanism:** Add CLASS\_ConfigDoNotCheckDefaults to ClassFlags
- **Associated items:** Config

- **Commonly used:** ★

Specifies that the consistency check of the parent configuration values is ignored when saving configuration values.

- During configuration saving, it determines whether to check the attributes for consistency based on the Base or Default configuration first. If consistent, there is no need to serialize and write them. However, with this flag set, the values will be saved regardless of whether they match the parent configuration values.

UCLASS(config=XXX,configdonotcheckdefaults): Indicates that the configuration file corresponding to this class will not check if the DefaultXXX configuration file at the XXX level contains this information (the hierarchy will be explained later), and will be saved directly to the Saved directory.

## Sample Code:

```
UCLASS(Config = Game)
class INSIDER_API UMyClass_Config :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertyWithConfig = 123;
};

UCLASS(Config = Game, configdonotcheckdefaults)
class INSIDER_API UMyClass_ConfigDoNotCheckDefaults :public UMyClass_Config
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertyWithConfigSub = 123;
};

UCLASS(Config = Game)
class INSIDER_API UMyClass_ConfigDefaultChild :public UMyClass_Config
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertyWithConfigSub = 123;
};
```

## Example Effect:

```
void UMyClass_Config_Test::TestConfigCheckDefaultSave()
{
    auto* testObject = NewObject<UMyClass_ConfigDoNotCheckDefaults>(GetTransientPackage(), TEXT("testObjectCheckDefault"));
    auto* testObject2 = NewObject<UMyClass_ConfigDefaultChild>(GetTransientPackage(), TEXT("testObjectDefaultChild"));
```

```

    testObject->SaveConfig();
    testObject2->SaveConfig();
}

生成:
[/script/Insider.MyClass_Config]
MyPropertyWithConfig=777

[/script/Insider.MyClass_ConfigDoNotCheckDefaults]
MyPropertyWithConfigSub=123
MyPropertyWithConfig=777

[/script/Insider.MyClass_ConfigDefaultChild]
MyPropertyWithConfigSub=123

```

As seen here, the value of MyPropertyWithConfig in MyClass\_ConfigDoNotCheckDefaults is identical to the 777 value in UMyClass\_Config by default, but it will still be written in. In the MyClass\_ConfigDefaultChild class, the value of MyPropertyWithConfig will be omitted because it has not been altered.

When searching for configdonotcheckdefaults in the source code, it is often found to be used in conjunction with defaultconfig. When should configdonotcheckdefaults be used? It seems to be for maintaining completeness, ensuring everything is written in regardless. In defaultConfig, you can ignore the values in Base and write a copy to the Default configuration, making the editing more comprehensive.

## Principle:

```

const bool bShouldCheckIfIdenticalBeforeAdding = !GetClass()->HasAnyClassFlags(CLASS_ConfigDoNotCheckDefaults) && !bPerObject && bIsPropertyInherited;
//Simple example judgment
if (!bPropDeprecated && (!bShouldCheckIfIdenticalBeforeAdding || !Property->Identical_InContainer(this, SuperClassDefaultObject, Index)))
{
    FString value;
    Property->ExportText_InContainer( Index, value, this, this, this, PortFlags );
    Config->SetString( *Section, *Key, *Value, PropFileName );
}
else
{
    // If we are not writing it to config above, we should make sure that this
    // property isn't stagnant in the cache.
    Config->RemoveKey( *Section, *Key, PropFileName );
}

```

# DefaultConfig

- **Function description:** Specifies that the configuration file level to save to is Project/Config/DefaultXXX.ini.
- **Engine module:** Config
- **Metadata type:** bool
- **Action mechanism:** Add CLASS\_DefaultConfig to ClassFlags
- **Associated items:** Config
- **Commonly used:** ★★★

The specified level for saving the configuration file is Project/Config/DefaultXXX.ini.

- Instead of the default Saved/XXX.ini
- Usually used in the editor to automatically save Settings to Project/Config/DefaultXXX.ini

## Sample Code:

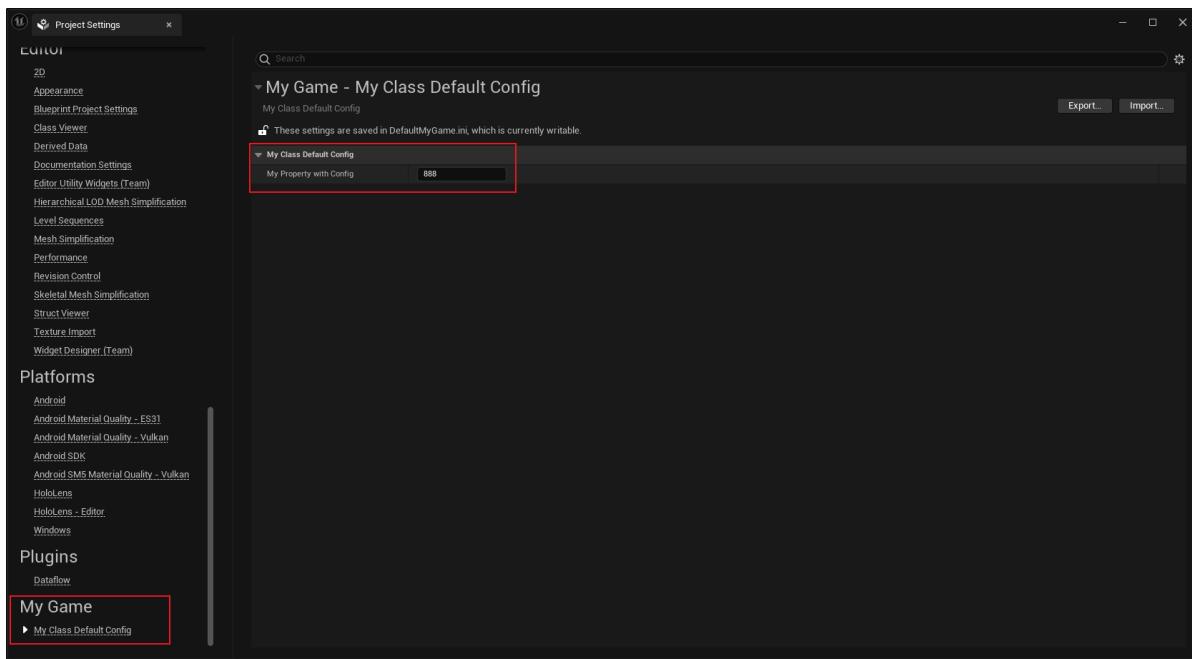
```
UCLASS(Config = MyGame, DefaultConfig)
class INSIDER_API UMyClass_DefaultConfig :public UDeveloperSettings
{
    GENERATED_BODY()

public:
    /** Gets the settings container name for the settings, either Project or
Editor */
    virtual FName GetContainerName() const override { return TEXT("Project"); }
    /** Gets the category for the settings, some high level grouping like,
Editor, Engine, Game...etc. */
    virtual FName GetCategoryName() const override { return TEXT("MyGame"); }
    /** The unique name for your section of settings, uses the class's FName. */
    virtual FName GetSectionName() const override { return TEXT("MyGame"); }

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
        int32 MyPropertywithConfig = 123;
};

//Saved Results:
//Config/DefaultMyGame.ini
[/Script/Insider.MyClass_DefaultConfig]
MyPropertywithConfig=888
```

# Example Results:



## Principle:

In the code, one should use `Settings->TryUpdateDefaultConfigFile();`, but it has been observed that `TryUpdateDefaultConfigFile` can be called regardless of whether `DefaultConfig` exists, and it will save to the Default configuration. Therefore, which `SaveConfig` method (`(TryUpdateDefaultConfigFile, UpdateGlobalUserConfigFile, UpdateProjectUserConfigFile)`) to call can be manually specified.

However, when editing in the editor, the logic can be handled through well-written code. For instance, in `SSettingsEditor.cpp`, calling `Section->Save()`; in `NotifyPostChange` will internally invoke the following code:

```
bool FSettingsSection::Save()
{
    if (ModifiedDelegate.IsBound() && !ModifiedDelegate.Execute())
    {
        return false;
    }

    if (SaveDelegate.IsBound())
    {
        return SaveDelegate.Execute();
    }

    //Update to the Correct File
    if (settingsobject.IsValid())
    {
        if (SettingsObject->GetClass()->HasAnyClassFlags(CLASS_DefaultConfig))
        {
            SettingsObject->TryUpdateDefaultConfigFile();
        }
        else if (SettingsObject->GetClass()->HasAnyClassFlags(CLASS_GlobalUserConfig))
        {
    }
```

```

        SettingsObject->UpdateGlobalUserConfigFile();
    }
    else if (SettingsObject->GetClass()->HasAnyClassFlags(CLASS_ProjectUserConfig))
    {
        SettingsObject->UpdateProjectUserConfigFile();
    }
    else
    {
        SettingsObject->SaveConfig();
    }

    return true;
}

return false;
}

```

## GlobalUserConfig

- **Function description:** Specifies that the configuration file level to save to is the global user settings at Engine/Config/UserXXX.ini.
- **Engine module:** Config
- **Metadata type:** bool
- **Action mechanism:** Add CLASS\_GlobalUserConfig within ClassFlags
- **Associated items:** Config
- **Commonly used:** ★★★

Specifies that the configuration file level to be saved to is the global user settings at Engine/Config/UserXXX.ini.

## Sample Code:

Attributes can be referred to as Config or GlobalConfig, either is acceptable.

```

UCLASS(Config = MyGame, GlobalUserConfig)
class INSIDER_API UMyClass_GlobalUserConfig:public UDeveloperSettings
{
    GENERATED_BODY()
public:
    /** Gets the settings container name for the settings, either Project or
Editor */
    virtual FName GetContainerName() const override { return TEXT("Project"); }
    /** Gets the category for the settings, some high level grouping like,
Editor, Engine, Game...etc. */
    virtual FName GetCategoryName() const override { return TEXT("MyGame"); }
    /** The unique name for your section of settings, uses the class's FName. */
    virtual FName GetSectionName() const override { return TEXT("MyGlobalGame"); }
}
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
        int32 MyPropertyWithConfig = 123;

```

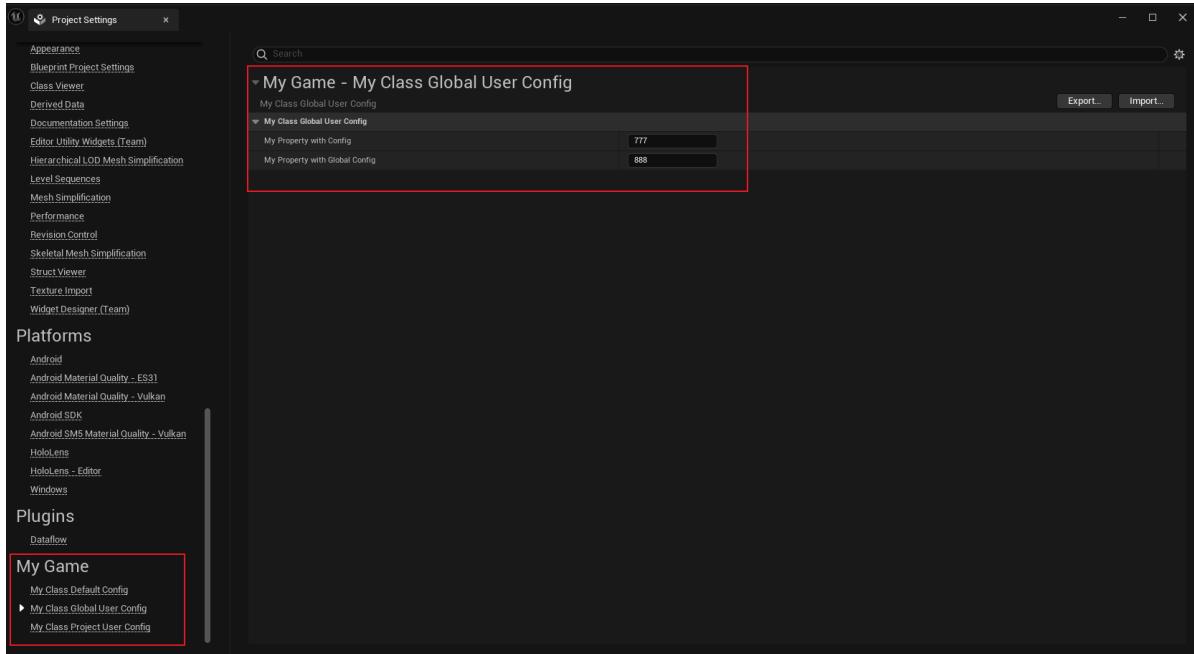
```

UPROPERTY(EditAnywhere, BlueprintReadWrite, GlobalConfig)
    int32 MyPropertywithGlobalConfig = 456;
};

```

保存到C:\Users\jack.fu\AppData\Local\Unreal Engine\Engine\Config\UserMyGame.ini  
[./script/Insider.uMyClass\_GlobalUserConfig]  
MyPropertywithGlobalConfig=999

## Example Effect:



## Source Code Illustration:

```

UCLASS(config=Engine, globaluserconfig)
class ANDROIDPLATFORMEDITOR_API UAndroidSDKSettings : public uobject
{
public:
    GENERATED_UCLASS_BODY()

    // Location on disk of the Android SDK (falls back to ANDROID_HOME
    environment variable if this is left blank)
    UPROPERTY(GlobalConfig, EditAnywhere, Category = SDKConfig, Meta =
    (DisplayName = "Location of Android SDK (the directory usually contains 'android-
    sdk-')"))
    FDirectoryPath SDKPath;

    // Location on disk of the Android NDK (falls back to NDKROOT environment
    variable if this is left blank)
    UPROPERTY(GlobalConfig, EditAnywhere, Category = SDKConfig, Meta =
    (DisplayName = "Location of Android NDK (the directory usually contains 'android-
    ndk-')"))
    FDirectoryPath NDKPath;

    // Location on disk of Java (falls back to JAVA_HOME environment variable if
    this is left blank)

```

```

UPROPERTY(GlobalConfig, EditAnywhere, Category = SDKConfig, Meta =
(DisplayName = "Location of JAVA (the directory usually contains 'jdk')"))
FDirectoryPath JavaPath;

// Which SDK to package and compile Java with (a specific version or (without
// quotes) 'latest' for latest version on disk, or 'matchndk' to match the NDK API
// Level)
UPROPERTY(GlobalConfig, EditAnywhere, Category = SDKConfig, Meta =
(DisplayName = "SDK API Level (specific version, 'latest', or 'matchndk' - see
tooltip")))
FString SDKAPILevel;

// Which NDK to compile with (a specific version or (without quotes) 'latest'
// for latest version on disk). Note that choosing android-21 or later won't run on
// pre-5.0 devices.
UPROPERTY(GlobalConfig, EditAnywhere, Category = SDKConfig, Meta =
(DisplayName = "NDK API Level (specific version or 'latest' - see tooltip")))
FString NDKAPILevel;
};

```

## ProjectUserConfig

- **Function Description:** Specifies that the configuration file level to save to is the project user settings at Project/Config/UserXXX.ini.
- **Engine Module:** Config
- **Metadata Type:** bool
- **Action Mechanism:** Add CLASS\_ProjectUserConfig within ClassFlags
- **Associated Items:** Config
- **Common Usage:** ★★★

Specifies that the configuration file level to save to is the project user settings at Project/Config/UserXXX.ini.

## Sample Code:

The directory to save to is \Hello\Config\UserMyGame.ini

```

UCLASS(Config = MyGame, ProjectUserConfig)
class INSIDER_API UMyClass_ProjectUserConfig :public UDeveloperSettings
{
    GENERATED_BODY()
public:
    /** Gets the settings container name for the settings, either Project or
Editor */
    virtual FName GetContainerName() const override { return TEXT("Project"); }
    /** Gets the category for the settings, some high level grouping like,
Editor, Engine, Game...etc. */
    virtual FName GetCategoryName() const override { return TEXT("MyGame"); }
    /** The unique name for your section of settings, uses the class's FName. */
    virtual FName GetSectionName() const override { return TEXT("MyProjectGame"); }
}
public:

```

```

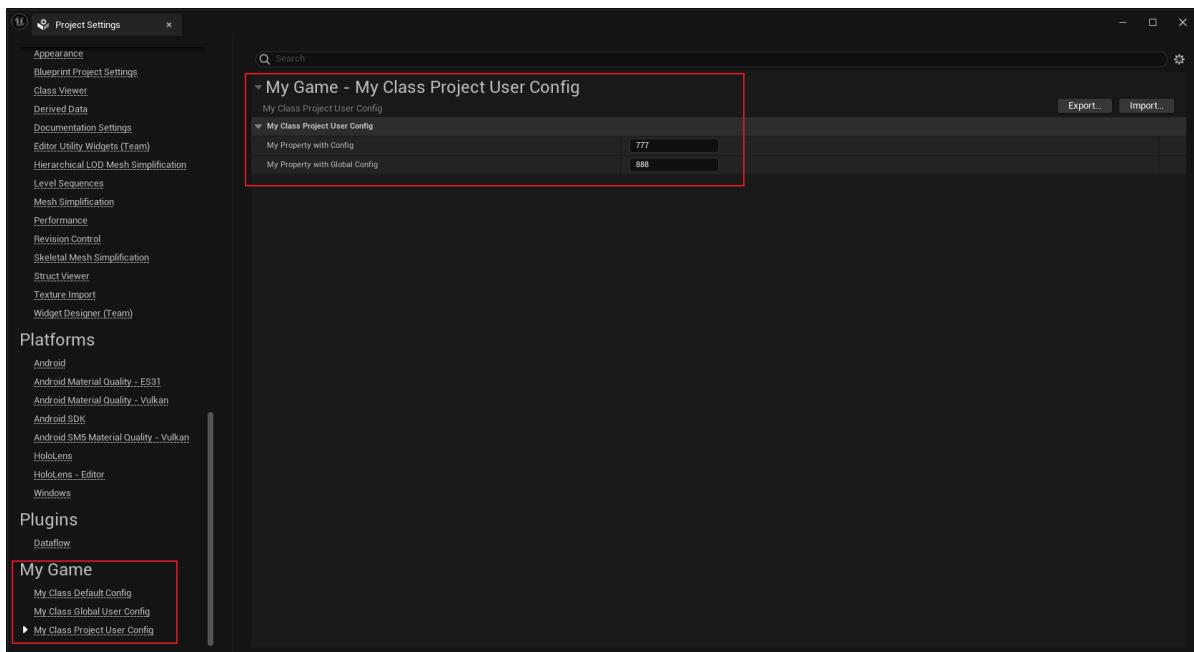
UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
int32 MyPropertyWithConfig = 123;

UPROPERTY(EditAnywhere, BlueprintReadWrite, GlobalConfig)
int32 MyPropertyWithGlobalConfig = 456;
};

//Result: \Hello\Config\userMyGame.ini
[/Script/Insider.MyClass_ProjectUserConfig]
MyPropertyWithConfig=777
MyPropertyWithGlobalConfig=888

```

## Example Effect:



## Search in the source code:

```

UCLASS(config = Engine, projectuserconfig, meta = (DisplayName = "Rendering
Overrides (Local)"))
class ENGINE_API URenderOverridesSettings : public UDeveloperSettings
{
}

```

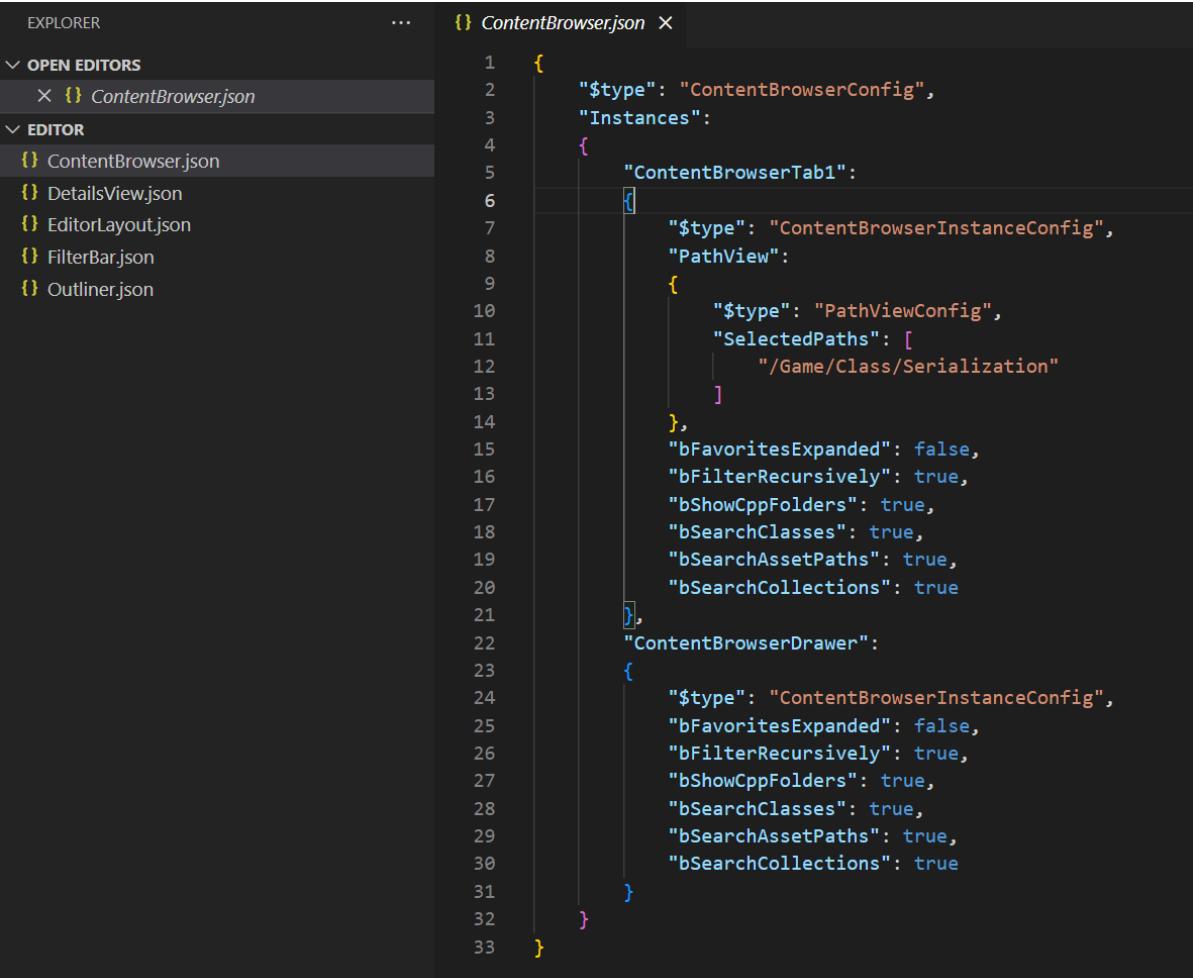
## EditorConfig

- **Function description:** Used for saving information in the editor mode.
- **Engine modules:** Config, Editor
- **Metadata type:** string="abc"
- **Action mechanism:** Add EditorConfig within Meta
- **Commonly used:** ★

Used for saving information in the editor mode.

Generally used in the EditorTarget module for configuring corresponding editor settings, such as column width, bookmarks, etc., saved in JSON format.

Saved in: C:\Users\user name\AppData\Local\UnrealEngine\Editor. Presently includes:



The screenshot shows a code editor with the file 'ContentBrowser.json' open. The left sidebar shows other files like 'ContentBrowserInstanceConfig.h' and 'ContentBrowserTab.h'. The code itself is a JSON configuration object with nested properties for different tabs and their configurations.

```
{
    "$type": "ContentBrowserConfig",
    "Instances": [
        {
            "ContentBrowserTab1": [
                {
                    "$type": "ContentBrowserInstanceConfig",
                    "PathView": [
                        {
                            "$type": "PathViewConfig",
                            "SelectedPaths": [
                                "/Game/Class/Serialization"
                            ]
                        }
                    ],
                    "bFavoritesExpanded": false,
                    "bFilterRecursively": true,
                    "bShowCppClassFolders": true,
                    "bSearchClasses": true,
                    "bSearchAssetPaths": true,
                    "bSearchCollections": true
                },
                "ContentBrowserDrawer": [
                    {
                        "$type": "ContentBrowserInstanceConfig",
                        "bFavoritesExpanded": false,
                        "bFilterRecursively": true,
                        "bShowCppClassFolders": true,
                        "bSearchClasses": true,
                        "bSearchAssetPaths": true,
                        "bSearchCollections": true
                    }
                ]
            }
        }
    ]
}
```

After searching the source code, it must be used by inheriting from the base class:

```
/** Inherit from this class to simplify saving and loading properties from editor
configs. */
UCLASS()
class EDITORCONFIG_API UEditorConfigBase : public UObject
{
    GENERATED_BODY()

public:

    /** Load any properties of this class into properties marked with metadata
tag "EditorConfig" from the class's EditorConfig */
    bool LoadEditorConfig();

    /** Save any properties of this class in properties marked with metadata tag
"EditorConfig" into the class's EditorConfig. */
    bool SaveEditorConfig() const;
};
```

## Sample code:

```
UCLASS(EditorConfig = "MyEditorGame")
class INSIDER_API UMyClass_EditorConfig : public UEditorConfigBase
{
public:
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditorConfig))
    int32 MyPropertywithConfig = 123;
};

void UMyClass_EditorConfig_Test::TestConfigSave()
{
    //must run after editor initialization
    auto* testObject = NewObject<UMyClass_EditorConfig>(GetTransientPackage(),
TEXT("testObject_EditorConfig"));
    testObject->MyPropertywithConfig = 777;
    testObject->SaveEditorConfig();

}

void UMyClass_EditorConfig_Test::TestConfigLoad()
{
    auto* testObject = NewObject<UMyClass_EditorConfig>(GetTransientPackage(),
TEXT("testObject_EditorConfig"));
    testObject->LoadEditorConfig();
}

//Save result after executing Save:
C:\users\jack.fu\AppData\Local\UnrealEngine\Editor\MyEditorGame.json

{
    "$type": "MyClass_EditorConfig",
    "MyPropertywithConfig": 777
}
```

## Transient

- **Function description:** Indicates that all instances of this class will be skipped during serialization.
- **Engine module:** Serialization
- **Metadata type:** bool
- **Mechanism of action:** Add CLASS\_Transient to ClassFlags
- **Associated items:** NonTransient
- **Commonality:** ★★★

Indicates that all instances of this class are to be skipped during serialization.

- Objects of this class are never saved to disk. This specification is inherited by subclasses but can be overridden by the NonTransient specifier. It can also be overridden in the subclasses.
- It results in the corresponding Object having the RF\_Transient flag set.

- Note: UPROPERTYTransient specifies that a particular property should not be serialized, whereas UCLASSTransient applies to all objects of the class.

## Sample code:

```

UCLASS(Blueprintable, Transient)
class INSIDER_API UMyClass_Transient :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty = 123;
};

UCLASS(Blueprintable, NonTransient)
class INSIDER_API UMyClass_NonTransient :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyProperty = 123;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_Transient_Test :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        UMyClass_Transient* MyTransientObject;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        UMyClass_NonTransient* MyNonTransientObject;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Normal=123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
        int32 MyInt_Transient =123;
};

```

The key to serializing object pointers is to use the SerializeItem method Slot << ObjectValue; in FObjectProperty

Testing with an Actor's Class Default is not feasible because:

Transient properties are serialized for (Blueprint) Class Default Objects but should not be serialized for any 'normal' instances of classes.

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_Transient_Test :public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        UMyClass_Transient* MyTransientObject;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)

```

```

UMyClass_NonTransient* MyNonTransientObject;

UPROPERTY(EditAnywhere, BlueprintReadWrite)
int32 MyInt_Normal=123;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
int32 MyInt_Transient =123;
};

```

Nor can it be tested with FObjectAndNameAsStringProxyArchive, as this archive internally looks up the name before serializing an object.

```

FArchive& FObjectAndNameAsStringProxyArchive::operator<<(UObject*& Obj)
{
    if (IsLoading())
    {
        // Load the path name to the object
        FString LoadedString;
        InnerArchive << LoadedString;
        // Look up the object by fully qualified pathname
        Obj = FindObject<UObject>(nullptr, *LoadedString, false);
        // If we couldn't find it, and we want to load it, do that
        if(!Obj && bLoadIfFindFails)
        {
            Obj = LoadObject<UObject>(nullptr, *LoadedString);
        }
    }
    else
    {
        // save out the fully qualified object name
        FString SavedString(Obj->GetPathName());
        InnerArchive << SavedString;
    }
    return *this;
}

```

Therefore, testing is done using a Package:

```

FString packageName = TEXT("/Game/MyTestPackage");
FString assetPath = FPackageName::LongPackageNameToFilename(packageName,
FPackageName::GetAssetPackageExtension());
UPackage* package = CreatePackage(*packageName);
FSavePackageArgs saveArgs{};
saveArgs.Error = GError;

//ObjectFlags: RF_NoFlags
UMyClass_Transient_Test* testObject = NewObject<UMyClass_Transient_Test>(package,
TEXT("testObject"));
//ObjectFlags: RF_Transient
testObject->MyTransientObject = NewObject<UMyClass_Transient>(testObject,
TEXT("MyTransientObject"));
//ObjectFlags: RF_NoFlags
testObject->MyNonTransientObject = NewObject<UMyClass_NonTransient>(testObject,
TEXT("MyNonTransientObject"));

```

```

testObject->MyTransientObject->MyProperty = 456;
testObject->MyNonTransientObject->MyProperty = 456;
testObject->MyInt_Normal = 456;
testObject->MyInt_Transient = 456;

bool result = UPackage::SavePackage(package, testObject, *assetPath, saveArgs);

```

After saving, the Package should be reloaded:

```

FString packageName = TEXT("/Game/MyTestPackage");
FString assetPath = FPackageName::LongPackageNameToFilename(packageName,
FPackageName::GetAssetPackageExtension());

UPackage* package = LoadPackage(nullptr, *assetPath, LOAD_None);
package->FullyLoad();

UMyClass_Transient_Test* newTestObject=LoadObject<UMyClass_Transient_Test>
(package, TEXT("testObject"), *assetPath);

```

## Example effect:

It can be observed that MyTransientObject has not been serialized to disk and therefore will not be loaded.

newTestObject	0x0000075046120f00 (Name="testObject")
UObject	(Name="testObject")
MyTransientObject	0x0000000000000000 <NULL>
MyNonTransientObject	0x000007503f7ff7d0 (Name="MyNonTransientObject")
UObject	(Name="MyNonTransientObject")
MyProperty	456
MyInt_Normal	456
MyInt_Transient	123

## Principle:

During SavePackage, RF\_Transient causes the object not to be included in the HarvestExport process and not to be placed in the SaveContext

```

void FPackageHarvester::TryHarvestExport(UObject* Inobject)
{
    // Those should have been already validated
    check(Inobject && Inobject->IsInPackage(SaveContext.GetPackage()));

    // Get the realm in which we should harvest this export
    EIllegalRefReason Reason = EIllegalRefReason::None;
    ESaveRealm HarvestContext = GetObjectHarvestingRealm(InObject, Reason);
    if (!SaveContext.GetHarvestedRealm(HarvestContext).IsExport(Inobject))
    {
        SaveContext.MarkUnsaveable(InObject);
        bool bExcluded = false;
        if (!Inobject->HasAnyFlags(RF_Transient))
        {
            bExcluded = ConditionallyExcludeObjectForTarget(SaveContext,
InObject, HarvestContext);
        }
    }
}

```

```

    }

    if (!Inobject->HasAnyFlags(RF_Transient) && !bExcluded)
    {
        // It passed filtering so mark as export
        HarvestExport(Inobject, HarvestContext);
    }

    // If we have a illegal ref reason, record it
    if (Reason != EIllegalRefReason::None)
    {

SaveContext.RecordIllegalReference(currentExportDependencies.CurrentExport,
Inobject, Reason);
    }
}

}

```

## NonTransient

---

- **Function Description:** Invalidate the Transient specifier inherited from the base class.
- **Engine Module:** Serialization
- **Metadata Type:** bool
- **Action Mechanism:** Remove CLASS\_Transient from ClassFlags
- **Associated Items:** Transient
- **Commonality:** ★★★

## Optional

---

- **Function Description:** Indicates that objects of this class are optional and can be excluded from saving during the Cooking process.
- **Engine Module:** Serialization
- **Functionality Mechanism:** Add CLASS\_Optional to ClassFlags
- **Commonly Used:** ★

Objects of this class are optional and can be chosen to be ignored during the Cooking process.

- Usually, data marked as EditorOnly, such as MetaData, does not exist at runtime and is saved in separate specific files.
- Optional objects are typically enclosed within the WITH\_EDITORONLY\_DATA macro and are used exclusively in the editor.
- During the Cooking process, the engine will, based on the EDITOROPTIONAL configuration, add SAVE\_Optional to decide whether to serialize the object value along with others, such as metadata.

## Example Code:

```
//ClassFlags: CLASS_Optional | CLASS_MatchedSerializers | CLASS_Native |  
CLASS_RequiredAPI | CLASS_TokenStreamAssembled | CLASS_Intrinsic |  
CLASS_Constructed  
UCLASS(Optional)  
class INSIDER_API UMyClass_Optional :public UObject  
{  
    GENERATED_BODY()  
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite)  
        int32 MyProperty = 123;  
};  
  
UCLASS()  
class INSIDER_API UMyClass_NotOptional :public UObject  
{  
    GENERATED_BODY()  
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite)  
        int32 MyProperty = 123;  
};  
  
UCLASS()  
class INSIDER_API UMyClass_Optional_Test :public UObject  
{  
    GENERATED_BODY()  
public:  
  
#if WITH_EDITORONLY_DATA  
    UPROPERTY()  
        UMyClass_Optional* MyOptionalObject;  
  
#endif // WITH_EDITORONLY_DATA  
  
public:  
    UPROPERTY()  
        UMyClass_NotOptional* MyNotOptionalObject;  
public:  
    static void CreatePackageAndSave();  
    static void LoadPackageAndTest();  
};  
  
void UMyClass_Optional_Test::CreatePackageAndSave()  
{  
    FString packageName = TEXT("/Game/MyOptionTestPackage");  
    FString assetPath = FPackageName::LongPackageNameToFilename(packageName,  
FPackageName::GetAssetPackageExtension());  
  
    IFileManager::Get().Delete(*assetPath, false, true);  
  
    UPackage* package = CreatePackage(*packageName);  
    FSavePackageArgs saveArgs{};  
    //saveArgs.TopLevelFlags = EObjectFlags::RF_Public |  
    EObjectFlags::RF_Standalone;
```

```

    saveArgs.Error = GError;
    saveArgs.SaveFlags=SAVE_NoError;

        //SAVE_Optional = 0x00008000,    ///Indicate that we to save optional exports. This flag is only valid while cooking. Optional exports are filtered if not specified during cooking.

        UMyClass_Optional_Test* testObject = NewObject<UMyClass_Optional_Test>(package, TEXT("testObject"));

#if WITH_EDITORONLY_DATA
    testObject->MyOptionalObject = NewObject<UMyClass_Optional>(testObject,
TEXT("MyOptionalObject"));
    testObject->MyOptionalObject->MyProperty = 456;
#endif

    testObject->MyNotOptionalObject = NewObject<UMyClass_NotOptional>(testObject,
TEXT("MyNotOptionalObject"));

    testObject->MyNotOptionalObject->MyProperty = 456;

    FString str = UIInsiderSubsystem::Get().PrintObject(package,
EInsiderPrintFlags::All);
    FString str2 = UIInsiderSubsystem::Get().PrintObject(testObject,
EInsiderPrintFlags::All);
    FString str3 =
UIInsiderSubsystem::Get().PrintObject(UMyClass_Optional::StaticClass(),
EInsiderPrintFlags::All);
    FString str4 =
UIInsiderSubsystem::Get().PrintObject(UMyClass_NotOptional::StaticClass(),
EInsiderPrintFlags::All);

    bool result = UPackage::SavePackage(package, testObject, *assetPath,
saveArgs);

}

void UMyClass_Optional_Test::LoadPackageAndTest()
{
    FString packageName = TEXT("/Game/MyOptionTestPackage");
    FString assetPath = FPackageName::LongPackageNameToFilename(packageName,
FPackageName::GetAssetPackageExtension());

    UPackage* package = LoadPackage(nullptr, *assetPath, LOAD_None);
    package->FullyLoad();

    UMyClass_Optional_Test* newTestObject = LoadObject<UMyClass_Optional_Test>(package, TEXT("testObject"), *assetPath);
    //UMyClass_Transient_Test* newTestObject = nullptr;

    /*const TArrray<FObjectExport>& exportMap = package->GetLinker()->ExportMap;
    for (const auto& objExport : exportMap)
    {
        if (objExport.ObjectName == TEXT("testObject"))
        {
            newTestObject = Cast<UMyClass_Transient_Test>(objExport.Object);
        }
    }
}

```

```

        break;
    }
} */
FString str = UInsiderSubsystem::Get().PrintObject(package,
EInsiderPrintFlags::All);

}

```

## Example Effect:

The standard SavePackage method is ineffective; serialization and saving will still occur. The special saving method occurs during the Cook stage, and this example does not specifically test it.

newTestObject	0x0000095c3e7ec640 (Name="testObject") (Name="testObject")	UMyClass_Optional_Test *
↳ UObject	0x0000095c0e9750a0 (Name="MyOptionalObject") (Name="MyOptionalObject")	UObject
↳ MyOptionalObject	456	TObjectPtr<UMyClass_Optional>
↳ UObject	{ObjectPtr=0x0000095c3e9750a0 (Name="MyOptionalObject") DebugPtr=0x0000095c3e9750a0 (Name="MyOptio...")	UObject
↳ MyProperty	0x0000095c3e7ec5d0 (Name="MyNotOptionalObject") (Name="MyNotOptionalObject")	int
↳ [Raw View]	456	UMyClass_NotOptional *
↳ MyNotOptionalObject	0x0000095c3e7ec5d0 (Name="MyNotOptionalObject") (Name="MyNotOptionalObject")	UObject
↳ UObject	456	int
↳ MyProperty		

Searching for "Optional" in the source code reveals that it is generally used by the EditorOnlyData and CookedMetaData classes.

```

UCLASS(optional, within=Enum)
class ENGINE_API UEnumCookedMetaData : public UObject
UCLASS(optional, within=ScriptStruct)
class ENGINE_API UStructCookedMetaData : public UObject
UCLASS(optional, Within=Class)
class ENGINE_API UClassCookedMetaData : public UObject

UMaterialInterfaceEditorOnlyData* UMaterialInterface::CreateEditorOnlyData()
{
    const UClass* Editoronlyclass = GetEditorOnlyDataClass();
    check(Editoronlyclass);
    check(Editoronlyclass->HasAllClassFlags(CLASS_Optional));

    const FString EditorOnlyName =
MaterialInterface::GetEditorOnlyDataName(*GetName());
    const EObjectFlags EditorOnlyFlags =
GetMaskedFlags(RF_PropagateToSubObjects);
    return NewObject<UMaterialInterfaceEditorOnlyData>(this, Editoronlyclass,
*EditorOnlyName, EditorOnlyFlags);
}

```

There are also some verifications within the engine:

```

UnrealTypeDefinitionInfo.cpp:
// Validate if we are using editor only data in a class or struct definition
if (HasAnyClassFlags(CLASS_Optional))
{
    for (TSharedRef<FunrealPropertyDefinitionInfo> PropertyDef : GetProperties())
    {
        if (PropertyDef->GetPropertyBase().IsEditorOnlyProperty())
        {

```

```
        PropertyDef->LogError(TEXT("Cannot specify editor only property  
inside an optional class."));  
    }  
    else if (PropertyDef->GetPropertyBase().ContainsEditorOnlyProperties())  
    {  
        PropertyDef->LogError(TEXT("Do not specify struct property containing  
editor only properties inside an optional class."));  
    }  
}
```

Discovered through the source code:

```
//SAVE_Optional = 0x00008000, //< Indicate that we to save optional exports. This flag is only valid while cooking. Optional exports are filtered if not specified during cooking.
```

`SAVE_Optional` is applied to `UUserDefinedEnum`, `UUserDefinedStruct`, and `UBlueprintGeneratedClass` `MetaData` objects.

```

void UUserDefinedStruct::PreSaveRoot(FObjectPreSaveRootContext ObjectSaveContext)
{
    Super::PreSaveRoot(ObjectSaveContext);

    if (ObjectSaveContext.IsCooking() && (ObjectSaveContext.GetSaveFlags() &
SAVE_Optional))
    {
        //This object, with 'this' as its Outer, marked with RF_Standalone | RF_Public, will result in the sub-object being serialized
        UStructCookedMetaData* CookedMetaData = NewCookedMetaData();
        CookedMetaData->CacheMetaData(this);

        if (!CookedMetaData->HasMetaData())
        {
            PurgeCookedMetaData(); //Cleanup of this CookedMetaData object is required
        }
    }
    else
    {
        PurgeCookedMetaData();
    }
}

```

Moreover, during the Cooking process, if specified,

```
bCookEditorOptional = Switches.Contains(TEXT("EDITOROPTIONAL")); // Produce the optional editor package data alongside the cooked data.
```

the CookEditorOptional attribute will be added

```
CookFlags |= bCookEditorOptional ? ECookInitializationFlags::CookEditorOptional :  
ECookInitializationFlags::None;
```

Subsequently, `SAVE_Optional` will be conveyed to the Package's `SaveFlags`

```
SaveFlags |= COTFS.IsCookFlagSet(ECookInitializationFlags::CookEditorOptional) ? SAVE_Optional :  
SAVE_None;
```

Thus, when including a Package, the IsSaveOptional() check will determine whether to create an Optional Realm

```
TArray<ESaveRealm> FSaveContext::GetHarvestedRealmsToSave()
{
    TArray<ESaveRealm> HarvestedContextsToSave;
    if (IsCooking())
    {
        HarvestedContextsToSave.Add(ESaveRealm::Game);
        if (IsSaveOptional())
        {
            HarvestedContextsToSave.Add(ESaveRealm::Optional);
        }
    }
    else
    {
        HarvestedContextsToSave.Add(ESaveRealm::Editor);
    }
    return HarvestedContextsToSave;
}
```

Also, if an Object is found with CLASS\_Optional, it will not be treated as an Export (child object) but as an Import (referenced object), and the Optional object may be stored in an external independent file.

```
ESavePackageResult HarvestPackage(FSaveContext& SaveContext)
{
    // If we have a valid optional context and we are saving it,
    // transform any harvested non optional export into imports
    // Mark other optional import package as well
    if (!SaveContext.IsSaveAutoOptional() &&
        SaveContext.IsSaveOptional() &&
        SaveContext.IsCooking() &&
        SaveContext.GetHarvestedRealm(ESaveRealm::Optional).GetExports().Num() &&
        SaveContext.GetHarvestedRealm(ESaveRealm::Game).GetExports().Num())
    {
        bool bHasNonOptionalSelfReference = false;
        FHarvestedRealm& OptionalContext =
        SaveContext.GetHarvestedRealm(ESaveRealm::Optional);
        for (auto It = OptionalContext.GetExports().CreateIterator(); It; ++It)
        {
            if (!It->Obj->GetClass()->HasAnyClassFlags(CLASS_Optional))
            {
                // Make sure the export is found in the game context as well
                if (FTaggedExport* GameExport =
                    SaveContext.GetHarvestedRealm(ESaveRealm::Game).GetExports().Find(It->Obj))
                {
                    // Flag the export in the game context to generate it's
                    public hash
                    GameExport->bGeneratePublicHash = true;
                    // Transform the export as an import
                    OptionalContext.AddImport(It->Obj);
                    // Flag the package itself to be an import
                    bHasNonOptionalSelfReference = true;
                }
            }
        }
    }
}
```

```

        }

        // if not found in the game context and the reference directly
        // came from an optional object, record an illegal reference
        else if (It->bFromOptionalReference)
        {
            SaveContext.RecordIllegalReference(nullptr, It->Obj,
EIllegalRefReason::ReferenceFromOptionalToMissingGameExport);
        }
        It.RemoveCurrent();
    }

}

// Also add the current package itself as an import if we are referencing
any non optional export
if (bHasNonOptionalSelfReference)
{
    OptionalContext.AddImport(SaveContext.GetPackage());
}
}
}
}

```

## MatchedSerializers

- **Function Description:** Specifies that the class supports serialization of text structure
- **Engine Module:** Serialization
- **Metadata Type:** bool
- **Functionality Mechanism:** Add CLASS\_MatchedSerializers to ClassFlags and MatchedSerializers to Meta
- **Usage Frequency:** 0

This specifier is only allowed to be used in NoExportTypes.h and is considered an internal specifier for the engine's use.

Effectively, most classes possess this tag, except for those that do not export themselves, typically including those defined in NoExportTypes.h (unless MatchedSerializers is manually added, such as UObject), or metadata defined with DECLARE\_CLASS\_INTRINSIC directly in the source code.

Therefore, the majority of classes actually have this tag. In UHT, unless a class is marked NoExport, this tag is automatically added.

```

// Force the MatchedSerializers on for anything being exported
if (!ClassExportFlags.HasAnyFlags(UhtClassExportFlags.NoExport))
{
    ClassFlags |= EClassFlags::MatchedSerializers;
}

```

## Structured Serializer:

If a class supports text format, the StructuredArchive's purpose is to serially expand the class's fields in a tree-like structure for serialization and display, facilitating human comprehension. If text format is not supported, all field values are compressed into a binary buffer (Data field), which is also the method used during runtime.

Test Code:

```
UCLASS(Blueprintable, BlueprintType, editinlinenew)
class INSIDER_API UMyClass_MatchedSerializersSub :public UObject
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Default = 123;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_MatchedSerializersTestAsset:public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Default = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced)
    UMyClass_MatchedSerializersSub* SubObject;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UStruct* MyStructType;

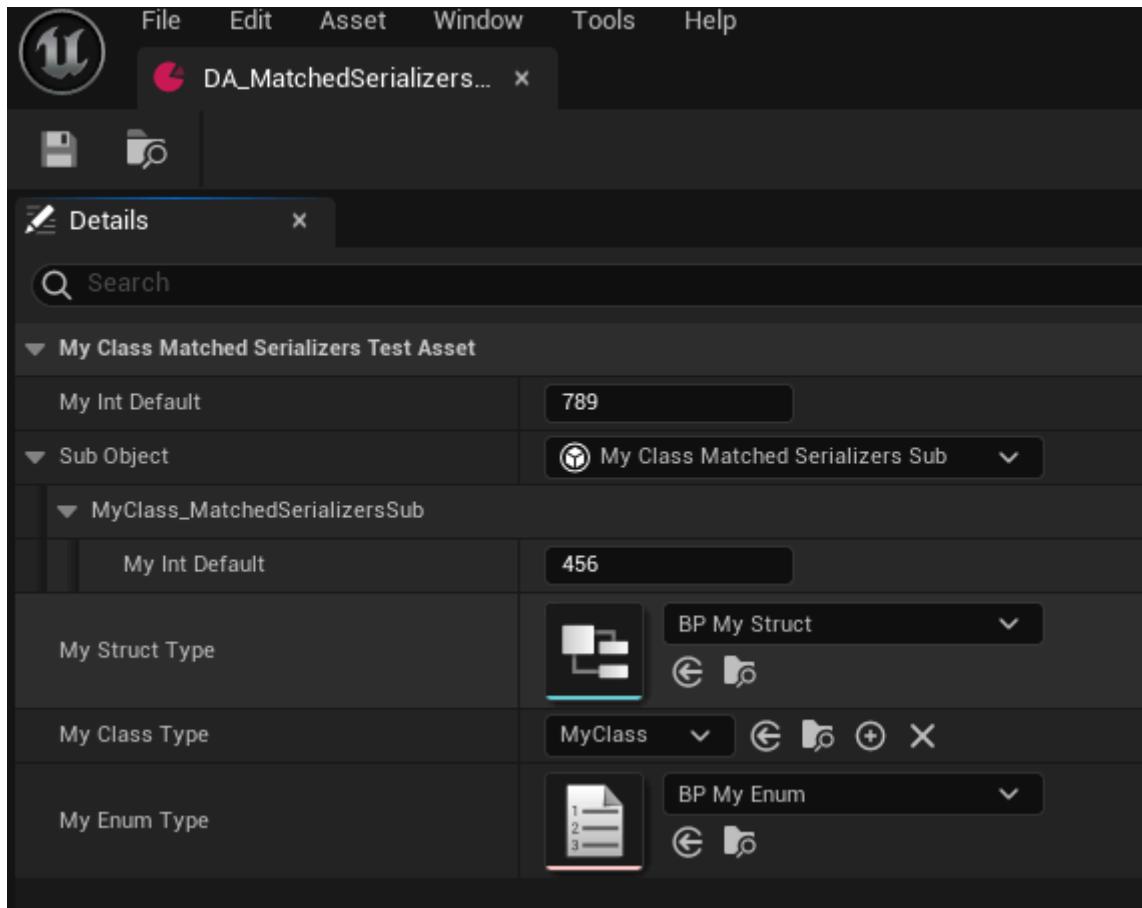
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UClass* MyClassType;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UEnum* MyEnumType;
};

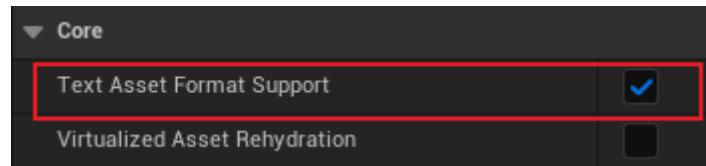
void UMyClass_MatchedSerializers_Test::ApplyClassFlag()
{
    UMyClass_MatchedSerializersTestAsset::StaticClass()->ClassFlags |=
    CLASS_MatchedSerializers;
    UMyClass_MatchedSerializersSub::StaticClass()->ClassFlags |=
    CLASS_MatchedSerializers;
}

void UMyClass_MatchedSerializers_Test::RemoveClassFlag()
{
    UMyClass_MatchedSerializersTestAsset::StaticClass()->ClassFlags &=
    ~CLASS_MatchedSerializers;
    UMyClass_MatchedSerializersSub::StaticClass()->ClassFlags &=
    ~CLASS_MatchedSerializers;
}
```

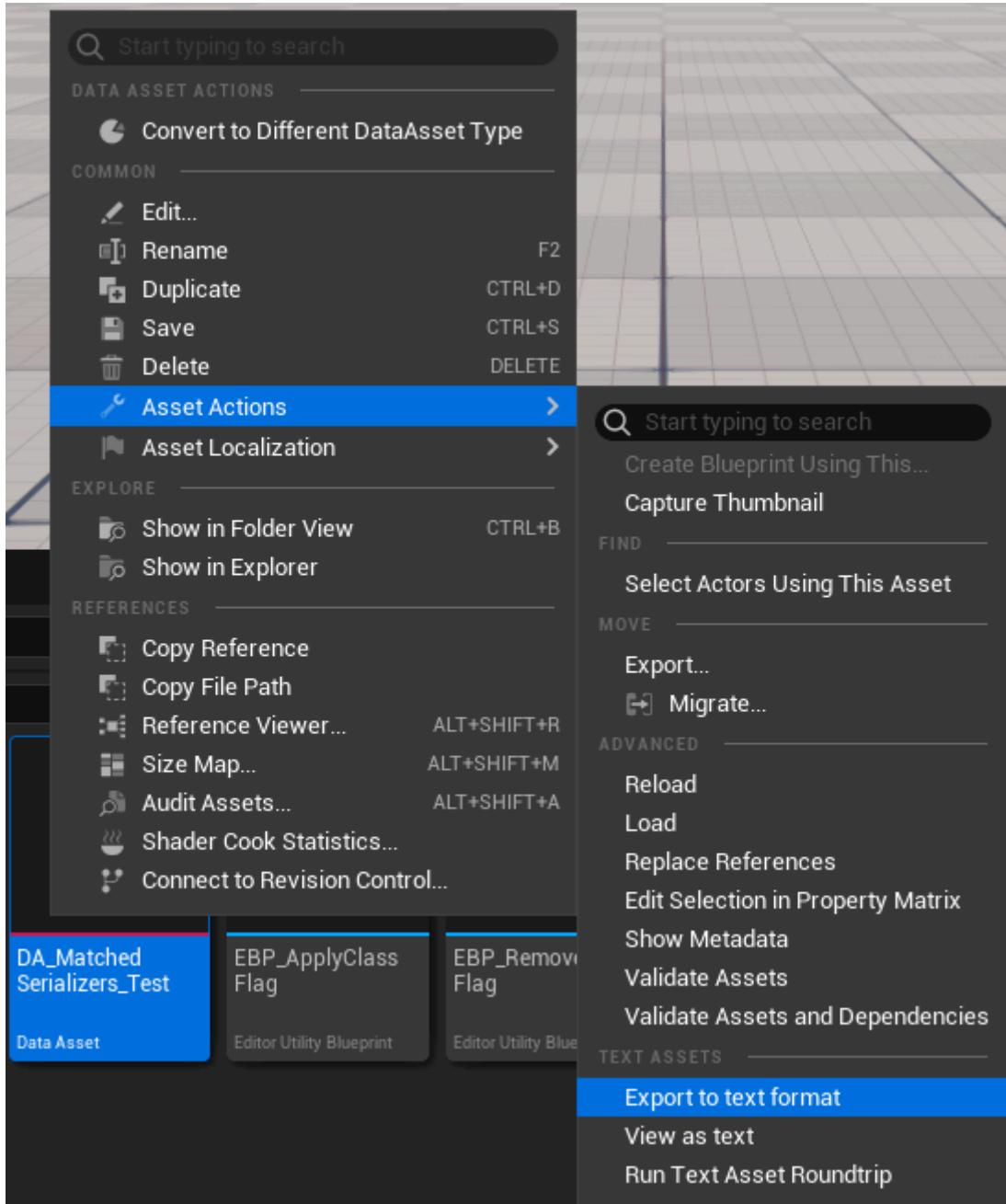
Create a test data Asset in the editor



Then enable `TextAssetFormatSupport` (`UEditorExperimentalSettings::bTextAssetFormatSupport`) in the Editor options



Three menus will then appear on the asset, supporting the export of the asset as text.



`ExportToTextFormat` will generate a `.txt` file in the same directory as the blueprint asset, formatted as JSON. By dynamically adding or removing the `CLASS_MatchedSerializers` tag, the differences produced by this tag can be compared:

```
"RootMetaDataMember": [
    {
        "Namespace": "PackageLocalizationNamespace",
        "B292B0E8CCFA805473A903D7AEE63B37"
    }
],
"DA_MatchedSerializers_Test:MyClass_MatchedSerializersSub_0": {
    "Properties": [
        "SerializationControlExtensions": 0,
        "Value": {
            "MyInt_Default": [
                {
                    "Type": "IntProperty",
                    "PropertyExtensions": 0,
                    "Value": 458
                }
            ]
        }
    ]
},
"DA_MatchedSerializers_Pest": {
    "Properties": [
        "SerializationControlExtensions": 0,
        "Value": {
            "MyInt_Default": [
                {
                    "Type": "IntProperty",
                    "PropertyExtensions": 0,
                    "Value": 789
                }
            ],
            "SubObject": [
                {
                    "Type": "ObjectProperty",
                    "PropertyExtensions": 0,
                    "Value": 12
                }
            ],
            "MyStructType": [
                {
                    "Type": "ObjectProperty",
                    "PropertyExtensions": 0,
                    "Value": -11
                }
            ],
            "MyClassType": [
                {
                    "Type": "ObjectProperty",
                    "PropertyExtensions": 0,
                    "Value": -4
                }
            ],
            "MyEnumType": [
                {
                    "Type": "ObjectProperty",
                    "PropertyExtensions": 0,
                    "Value": -107
                }
            ]
        }
    ]
},
"Summary": {
    "Summary": ...
}
},
"DA_MatchedSerializers_Test:MyClass_MatchedSerializersSub_0": {
    "Properties": [
        "Base64": "AAAABBBBBBBBBBAAQAAAAAAYAAAAAIAAAAAAA",
        "Name": "Base64",
        "MyInt_Default",
        "IntProperty",
        "None"
    ]
},
"DA_MatchedSerializers_Pest": {
    "Properties": [
        "Digest": "f49b6e5b3eff8a5749743b49b447842bb36fd1",
        "Base64": [
            "AAAABBBBBBBBBBAAQAAAAAAYAAAAAIAAAAAAA",
            "AAAABBBBBBBBBBAAQAAAAAAYAAAAAIAAAAAAA"
        ],
        "Objects": [
            {
                "ID": "1",
                "Name": "1",
                "Type": "Object"
            },
            {
                "ID": "2",
                "Name": "2",
                "Type": "Object"
            },
            {
                "ID": "3",
                "Name": "3",
                "Type": "Object"
            }
        ],
        "Names": [
            "MyInt_Default",
            "IntProperty",
            "SubObject",
            "ObjectProperty",
            "MyStructType",
            "MyClassType",
            "MyEnumType",
            "None"
        ]
    ]
},
"Summary": {
    "Summary": ...
}
```

It can be observed that there are significant differences in the serialized content. The right-side result without the CLASS\_MatchedSerializers tag compresses all field values into a binary buffer (Data field).

## Internal Mechanism Principle:

The CLASS\_MatchedSerializers tag is used in UClass::IsSafeToSerializeToStructuredArchives to indicate the use of a structured serializer. Support for text import and export is only considered in the editor environment.

Only SavePackage2.cpp and LinkerLoad.cpp are affected, occurring when saving UPackage as a subclass object. Therefore, simple in-memory Archive serialization cannot be used for testing.

```
bool UClass::IsSafeToSerializeToStructuredArchives(UClass* InClass)
{
    while (InClass)
    {
        if (!InClass->HasAnyClassFlags(CLASS_MatchedSerializers))
        {
            return false;
        }
        InClass = InClass->GetSuperClass();
    }
    return true;
}

//LinkerLoad.cpp
bool bClassSupportsTextFormat =
UClass::IsSafeToSerializeToStructuredArchives(Object->GetClass());
if (IsTextFormat()) //If Ar serialization is in text format
{
    FStructuredArchiveslot Exportslot = GetExportslot(Export.ThisIndex);

    if (bClassSupportsTextFormat) //If the class itself supports text
format
    {
        Object->GetClass()->SerializeDefaultObject(Object, Exportslot);
    }
    else
    {
        FStructuredArchiveChildReader ChildReader(Exportslot);
        FArchiveUObjectFromStructuredArchive
Adapter(ChildReader.GetRoot());
        Object->GetClass()->SerializeDefaultObject(Object,
Adapter.GetArchive());
    }
}

//SavePackage2.cpp
#if WITH_EDITOR
    bool bSupportsText =
UClass::IsSafeToSerializeToStructuredArchives(Export.Object->GetClass());
#else
    bool bSupportsText = false;
#endif
```

```

if (bSupportsText)
{
    Export.Object->GetClass()->SerializeDefaultObject(Export.Object,
ExportsSlot);
}
else
{
    FArchiveUObjectFromStructuredArchive Adapter(ExportsSlot);
    Export.Object->GetClass()->SerializeDefaultObject(Export.Object,
Adapter.GetArchive());
    Adapter.Close();
}

```

Text format is only effective in the editor environment.

As seen in the source code, if the class itself supports text format serialization, it can be directly serialized when Ar is in text format, using the default SerializeTaggedProperties. Otherwise, FArchiveUObjectFromStructuredArchive must be used to adapt, converting the object pointer to a combination of object path + int32 Index.

When printing all classes in the engine that contain or do not contain the CLASS\_MatchedSerializers tag, it is found that classes in the UStruct inheritance chain begin to include the tag (but UClass does not), while classes above UField do not include it, such as various Property types. The class list can be found in the Doc txt file.

## MinimalAPI

---

- **Function Description:** Specifies that the UInterface object should not be exported to other modules
- **Metadata Type:** bool
- **Engine Module:** DllExport
- **Common Usage:** ★

Refer to the explanation of MinimalAPI within UCLASS for guidance.

Primarily, UInterface objects serve merely as auxiliary interface objects, and therefore do not possess functions that need to be exposed. As a result, the majority of UInterface objects in the source code are designated as MinimalAPI to expedite compilation and prevent their use by other modules.

```

UINTERFACE(MinimalAPI, BlueprintType)
class USoundLibraryProviderInterface : public UInterface
{
    GENERATED_BODY()
};

```

# Blueprintable

- **Function Description:** Can be realized in blueprints
- **Metadata Type:** boolean
- **Engine Module:** Blueprint
- **Action Mechanism:** Include IsBlueprintBase and BlueprintType in Meta
- **Associated Items:** NotBlueprintable
- **Common Usage:** ★★★★☆

Whether it can be realized in blueprints.

## Sample Code:

```
UINTERFACE(Blueprintable,MinimalAPI)
class UMyInterface_Blueprintable:public UInterface
{
    GENERATED_UINTERFACE_BODY()
};

class INSIDER_API IMyInterface_Blueprintable
{
    GENERATED_IINTERFACE_BODY()
public:
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
        void Func_ImplementableEvent() const;

    UFUNCTION(BlueprintCallable, BlueprintNativeEvent)
        void Func_NativeEvent() const;
};

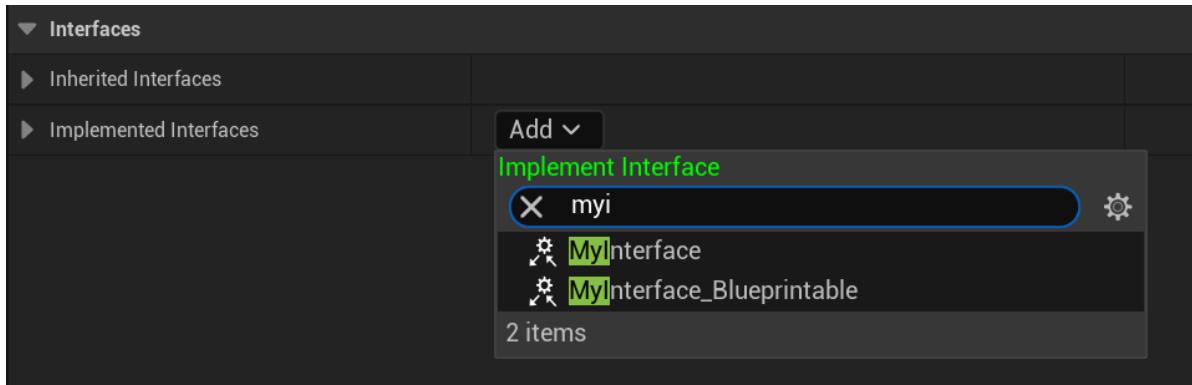
UINTERFACE(NotBlueprintable,MinimalAPI)
class UMyInterface_NotBlueprintable:public UInterface
{
    GENERATED_UINTERFACE_BODY()
};

class INSIDER_API IMyInterface_NotBlueprintable
{
    GENERATED_IINTERFACE_BODY()
public:
    //Blueprint functions should not be defined either, as they can no longer be
    //realized in blueprints
    //UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
    //void Func_ImplementableEvent() const;

    // UFUNCTION(BlueprintCallable, BlueprintNativeEvent)
    // void Func_NativeEvent() const;
};
```

## Example Effect:

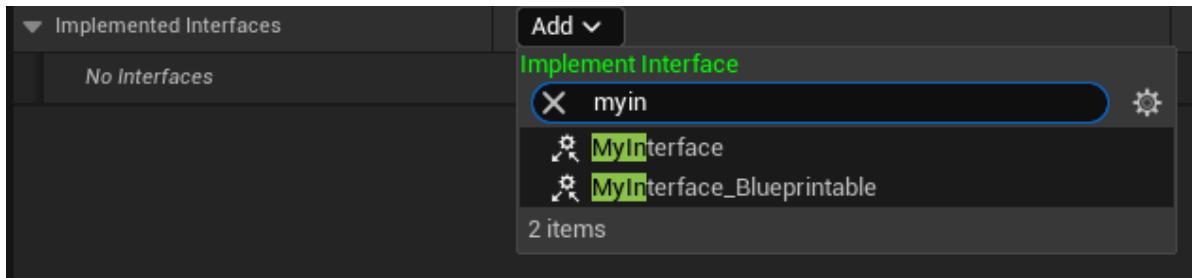
Tested in blueprints and found that UMyInterface\_NotBlueprintable cannot be located.



## NotBlueprintable

- **Function Description:** Specifies that it cannot be implemented in blueprints
- **Metadata Type:** bool
- **Engine Module:** Blueprint
- **Action Mechanism:** Removes IsBlueprintBase and BlueprintType from Meta, equivalent to CannotImplementInterfaceInBlueprint
- **Associated Items:** Blueprintable
- **Common Usage:** ★★★

In the Class Settings under Interface, the interface that cannot be implemented is not found.



When is it necessary to employ this marker? Although it cannot be realized within blueprints, it is still feasible to implement it in C++, and reflection can also be utilized to ascertain whether an object has implemented the interface.

## ConversionRoot

- **Function Description:** Sets the IsConversionRoot metadata flag for this interface.
- **Metadata Type:** boolean
- **Engine Module:** Blueprint
- **Action Mechanism:** Incorporates IsConversionRoot into the Meta

An example of this usage is not found in the source code

# NoExport

- **Function Description:** Instruct UHT not to automatically generate registration code, but to perform lexical analysis and extract metadata only.
- **Metadata Type:** bool
- **Engine Module:** UHT
- **Commonly Used:** ★

Instructions for UHT not to automatically generate registration code, but to perform lexical analysis and extract metadata only.

NoExportTypes.h frequently uses this example. Structures defined are often enclosed with the !CPP macro to prevent compilation in C++, thus typically used only within the engine.

We can actually use it if desired, provided the memory layout in C++ remains consistent, allowing for additional self-defined structures. Use case: Defining a UHT header for UHT analysis and then defining the actual C++ elsewhere. A typical application is where multiple classes in C++ inherit from a template base class, such as FVector2MaterialInput, enabling the definition of a UHT type alias for each specialized subclass. Another purpose is to place all headers to be analyzed by UHT in a single file to accelerate analysis and generation, eliminating the need to analyze multiple files, as long as the UHT information and memory layout are correct.

```
#if !CPP      // begin noexport class
USTRUCT(noexport, BlueprintType) //If "noexport" is not specified, an error will
occur: Expected a GENERATED_BODY() at the start of the struct
struct FFfloatRK4SpringInterpolator
{

    UPROPERTY(EditAnywhere, Category = "FloatRK4SpringInterpolator")
    float StiffnessConstant;

    /** 0 = Undamped, <1 = Underdamped, 1 = Critically damped, >1 = Over damped */
    UPROPERTY(EditAnywhere, Category = "FloatRK4SpringInterpolator")
    float DampeningRatio;

    bool bIsInitialized;
    bool bIsInMotion;
    float TimeRemaining;
    FRK4SpringConstants SpringConstants;

    float LastPosition;
    RK4Integrator::FRK4State<float> State;
};

#endif // end noexport class

//Practical Applications:
template <typename T>
struct FRK4SpringInterpolator
{
protected:
    float StiffnessConstant;
    float DampeningRatio;
```

```

    bool bIsInitialized;
    bool bIsInMotion;
    float TimeRemaining;
    FRK4SpringConstants SpringConstants;

    T LastPosition;
    RK4Integrator::FRK4State<T> State;
}

struct FFloatRK4SpringInterpolator : FRK4SpringInterpolator<float>
struct FVectorRK4SpringInterpolator : FRK4SpringInterpolator< FVector>

```

Excluded Code Includes:

```

USTRUCT(BlueprintType,noexport)
struct INSIDER_API FMyStruct_NoExport
{
    //suppression: Explanation of code generated by GENERATED_BODY():
    //static class UScriptStruct* StaticStruct();

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score;
};

//Suppression:
//template<> INSIDER_API UScriptStruct* StaticStruct<struct FMyStruct_NoExport>();

```

No generation in .h files, thus not used in other modules

```
template<> INSIDER_API UScriptStruct* StaticStruct<struct FMyStruct_NoExport>();
```

However, the Z\_Construct\_UScriptStruct\_FMyStruct\_NoExport call will still be generated in Module.init.gen.cpp, thus it will still be exposed in blueprints.

```

#include "UObject/GeneratedCppIncludes.h"
PRAGMA_DISABLE_DEPRECATED_WARNINGS
void EmptyLinkFunctionForGeneratedCodeInsider_init() {}
INSIDER_API UScriptStruct* Z_Construct_UScriptStruct_FMyStruct_NoExport();
static FPackageRegistrationInfo Z_Registration_Info_UPackage__Script__Insider;
FORCEINLINE UPackage* Z_Construct_UPackage__Script__Insider()
{
    if (!Z_Registration_Info_UPackage__Script__Insider.outersingleton)
    {
        static UObject* (*const SingletonFuncArray[])() = {
            (	UObject* (*)()
        })Z_Construct_UScriptStruct_FMyStruct_NoExport,//Injection of the call here
    };
    static const UECodeGen_Private::FPackageParams PackageParams = {
        "/script/Insider",
        SingletonFuncArray,
        UE_ARRAY_COUNT(SingletonFuncArray),
        PKG_CompiledIn | 0x00000000,
    };
}
```

```

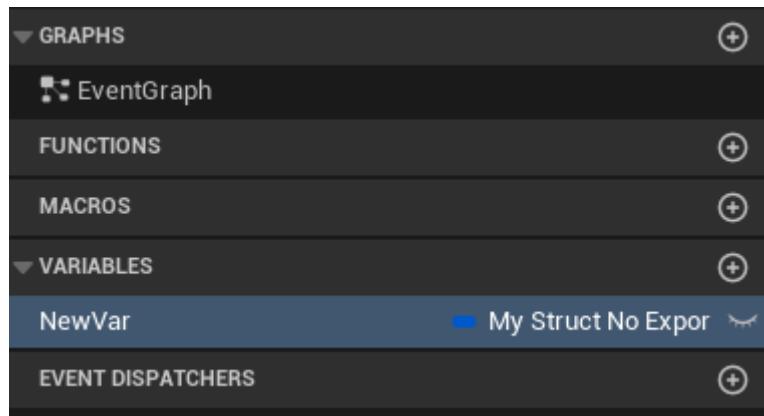
        0x02A7B98C,
        0xFA17C3C4,
        METADATA_PARAMS(0, nullptr)
    };

UECodeGen_Private::ConstructUPackage(Z_Registration_Info_UPackage__Script_Insider
.OuterSingleton, PackageParams);
}
return Z_Registration_Info_UPackage__Script_Insider.OuterSingleton;
}

static FRegisterCompiledInInfo
Z_CompiledInDeferPackage_UPackage__Script_Insider(Z_Construct_UPackage__Script_In
sider, TEXT("/Script/Insider"), Z_Registration_Info_UPackage__Script_Insider,
CONSTRUCT_RELOAD_VERSION_INFO(FPackageReloadVersionInfo, 0x02A7B98C,
0xFA17C3C4));
PRAGMA_ENABLE_DEPRECATED_WARNINGS

```

Effect in Blueprints: Still can be used as a variable.



The screenshot shows the 'Details' panel for the variable 'NewVar'. The 'Variable' section contains the following properties:

Variable Name	NewVar
Variable Type	My Struct No Export
Description	(empty)
Instance Editable	<input checked="" type="checkbox"/>
Blueprint Read Only	<input checked="" type="checkbox"/>
Expose on Spawn	<input checked="" type="checkbox"/>
Private	<input checked="" type="checkbox"/>
Category	Default
Replication	None
Replication Condition	None

Below the variable section, there are sections for 'Advanced' and 'Default Value'. Under 'Default Value', there is a row for 'New Var' with a 'Score' field set to '0.0'.

The difference with adding "noexport" is the inability to use StaticStruct and the absence of TCppStructOps, preventing certain optimizations. Otherwise, it can still be used normally, similar to FVector.

Missing code can also be manually added to achieve the same functionality.

```
USTRUCT(BlueprintType,noexport)
struct INSIDER_API FMyStruct_NoExport
{
    //GENERATED_BODY() //missing type specifier - int assumed, ..generated.h only
    //defines a StaticStruct() function

    static class UScriptStruct* StaticStruct(); //Can be self-defined

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score;

};

template<> INSIDER_API UScriptStruct* StaticStruct<struct FMyStruct_NoExport>()
//Can be self-defined

//.cpp
//Link function declarations, which have already been implemented in other .cpp
files, allowing for normal invocation
INSIDER_API UScriptStruct* Z_Construct_UScriptStruct_FMyStruct_NoExport();
UPackage* Z_Construct_UPackage__Script_Insider();

static FStructRegistrationInfo
Z_Registration_Info_UScriptStruct_MyStruct_NoExport;

class UScriptStruct* FMyStruct_NoExport::StaticStruct()
{
    if (!Z_Registration_Info_UScriptStruct_MyStruct_NoExport.OuterSingleton)
    {
        Z_Registration_Info_UScriptStruct_MyStruct_NoExport.OuterSingleton =
GetStaticStruct(Z_Construct_UScriptStruct_FMyStruct_NoExport,
Z_Construct_UPackage__Script_Insider(), TEXT("MyStruct_NoExport"));
    }
    return Z_Registration_Info_UScriptStruct_MyStruct_NoExport.OuterSingleton;
}

template<> INSIDER_API UScriptStruct* StaticStruct<FMyStruct_NoExport>()
{
    return FMyStruct_NoExport::StaticStruct();
}
```

## Atomic

- **Function description:** Specifies that the entire structure will always be serialized with all its properties, rather than just the changed ones.
- **Metadata type:** bool
- **Engine module:** UHT

- **Mechanism of action:** Include STRUCT\_Atomic in StructFlags
- **Commonly used:** ★

Specifies that the entire structure will always be serialized with all its properties, not just the altered ones.

Atomic serialization refers to the process where, if any field attribute of the structure differs from the default value, even if the rest of the fields are the same, the entire structure is serialized at once rather than being split. Note that this is only effective under the standard SerializeVersionedTaggedProperties, as it is compared against the default values. It does not work under Bin serialization. The actual mechanism is that when atomic serialization is employed, the default values of internal attributes are not checked, thus the entire attribute is serialized in all cases.

UE's noexporttype.h contains a large number of atomic base structures, such as FVector, because Immutable also sets STRUCT\_Atomic simultaneously, but no instances of setting Atomic alone were found.

## Sample Code:

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_InnerItem
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 A = 1;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 B = 2;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 C = 3;

    bool operator==(const FMyStruct_InnerItem& other) const
    {
        return A == other.A;
    }
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_NoAtomic
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FMyStruct_InnerItem Item;
};

USTRUCT(Atomic, BlueprintType)
struct INSIDER_API FMyStruct_Atomic
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
```

```

        FMyStruct_InnerItem Item;
    };

template<>
struct TStructOpsTypeTraits<FMyStruct_InnerItem> : public
TStructOpsTypeTraitsBase2<FMyStruct_InnerItem>
{
    enum
    {
        WithIdenticalViaEquality = true,
    };
};

void USerializationLibrary::SaveStructToMemory(UScriptStruct* structClass, void*
structObject, const void* structDefaults, TArray<uint8>& outSaveData,
EInsiderSerializationFlags flags/*=EInsiderSerializationFlags::None*/)
{
    FMemoryWriter MemoryWriter(outSaveData, false);
    MemoryWriter.SetWantBinaryPropertySerialization(EnumHasAnyFlags(flags,
EInsiderSerializationFlags::UseBinary));
    if (!EnumHasAnyFlags(flags, EInsiderSerializationFlags::CheckDefaults))
    {
        structDefaults=nullptr;
    }
    structClass->SerializeItem(MemoryWriter, structObject, structDefaults);
}

```

测试代码：

```
FMyStruct_NoAtomic NoAtomicStruct;
NoAtomicStruct.Item.A=3;

FMyStruct_Atomic AtomicStruct;
AtomicStruct.Item.A=3;

TArray<uint8> NoAtomicMemoryChanged;
USerializationLibrary::SaveStructToMemory(NoAtomicStruct, NoAtomicMemoryChanged, EInsiderSerializationFlags::CheckDefaults);

TArray<uint8> AtomicMemoryChanged;
USerializationLibrary::SaveStructToMemory(AtomicStruct, AtomicMemoryChanged, EInsiderSerializationFlags::CheckDefaults);
```

## Example Effect:

It is evident that `AtomicMemoryChanged` occupies more memory than `NonAtomicMemoryChanged`, because although the properties of both structures have changed, `AtomicStruct` always serializes all properties.

## Principle:

The mechanism at play is that if an external structure is marked as Atomic and its internal properties are found to have changed, these internal properties must be part of another structure. This is because if it's just an Int attribute, the default values of internal attributes are not compared. However, if it's an internal structure attribute, and one of the ID fields differs, the entire structure is deemed unequal during comparison (even though the rest of the internal structure's attributes are identical, which is why the example code only modifies A and provides an == comparison function). The default approach is to continue comparing default values within the internal structure's internal attributes, but with atomic serialization, the comparison is cut off at the default value being null, meaning the child attributes have no default values to compare against, and thus the entire internal attribute is serialized. Therefore, Atomic is used on external structures, and it's essentially meaningless to use it on structures like FVector that are not typically further disassembled.

```
void Ustruct::SerializeversionedTaggedProperties(FStructuredArchive::FSlot slot,
uint8* Data, Ustruct* DefaultsStruct, uint8* Defaults, const Uobject* BreakRecursionIfFullyLoad) const
{
//...
/** If true, it means that we want to serialize all properties of this struct if
any properties differ from defaults */
    bool buseAtomicSerialization = false;
    if (DefaultsscriptStruct)
    {
        buseAtomicSerialization = DefaultsscriptStruct-
>ShouldSerializeAtomically(UnderlyingArchive);
    }

    if (buseAtomicSerialization)
    {
        DefaultValue = NULL;
    }
}
```

## IsAlwaysAccessible

- **Function Description:** Ensures that the declaration of the specified structure is always accessible when UHT generates files, otherwise a mirrored structure definition must be created in gen.cpp
- **Metadata Type:** bool
- **Engine Module:** UHT
- **Restriction Type:** Exclusive to NoExportTypes.h for UHT usage
- **Usage Frequency:** 0

Indicates whether the declaration of this structure is always accessible in the gen.cpp file generated by UHT for NoExportTypes.h.

In other words, it determines if these structures are declared within GeneratedCppIncludes.h. If not found, a mirrored structure definition must be manually created when generating similar constructs like Z\_Construct\_UScriptStruct\_FMatrix44dStatics. If found, such as FGuid, no additional definition is needed.

Thus, this is a manual internal marker, aiding the UHT program in identifying which structures require mirrored structure definitions.

When examining structures in NoExportTypes.h, some structures (85/135) are marked with IsAlwaysAccessible, while others are not. This is because when UHT generates gen.cpp for NoExportTypes.h,

```
\unrealEngine\Engine\Source\Runtime\CoreUObject\Public\UObject\GeneratedCppIncludes.h
#include "UObject/Object.h"
#include "UObject/UObjectGlobals.h"
#include "UObject/CoreNative.h"
#include "UObject/Class.h"
#include "UObject/MetaData.h"
#include "UObject/UnrealType.h"
#include "UObject/EnumProperty.h"
#include "UObject/TextProperty.h"
#include "UObject/FieldPathProperty.h"

#if UE_ENABLE_INCLUDE_ORDER_DEPRECATED_IN_5_2
#include "CoreMinimal.h"
#endif

\Hello\Intermediate\Build\Win64\HelloEditor\Inc\CoreUObject\UHT\NoExportTypes.gen
.cpp:
// Copyright Epic Games, Inc. All Rights Reserved.
/*=====
   Generated code exported from UnrealHeaderTool.
   DO NOT modify this manually! Edit the corresponding .h files instead!
=====*/
//The following two lines:
#include "UObject/GeneratedCppIncludes.h" //A
#include "UObject/NoExportTypes.h" //B
PRAGMA_DISABLE_DEPRECATED_WARNINGS
void EmptyLinkFunctionForGeneratedCodeNoExportTypes() {}
```

If the struct definition can be found directly in the first two included headers, no additional definition is needed at points A and B in gen.cpp where the structure is required.

```

const UECodeGen_Private::FStructParams
Z_Construct_UScriptStruct_FMatrix44f_Statics::ReturnStructParams = {
    (UObject* (*)())Z_Construct_UPackage__Script_CoreUObject,
    nullptr,
    nullptr,
    "Matrix44f",
    Z_Construct_UScriptStruct_FMatrix44f_Statics::PropPointers,

UE_ARRAY_COUNT(Z_Construct_UScriptStruct_FMatrix44f_Statics::PropPointers),
    sizeof(FMatrix44f), //Location A
    alignof(FMatrix44f), //Location B
    RF_Public|RF_Transient|RF_MarkAsNative,
    EStructFlags(0x00000038),

METADATA_PARAMS(UE_ARRAY_COUNT(Z_Construct_UScriptStruct_FMatrix44f_Statics::Struct_MetaDataParams),
Z_Construct_UScriptStruct_FMatrix44f_Statics::Struct_MetaDataParams)
};

```

If the definition is not found, for example, FMatrix44f, which is defined in Engine\Source\Runtime\Core\Public\Math\Matrix.h, an identical definition must be generated for it (omitting includes to speed up compilation):

```

struct Z_Construct_UScriptStruct_FMatrix44f_Statics
{
    struct FMatrix44f //A definition with a consistent memory layout
    {
        FPlane4f XPlane;
        FPlane4f YPlane;
        FPlane4f ZPlane;
        FPlane4f WPlane;
    };

    static_assert(sizeof(FMatrix44f) < MAX_uint16);
    static_assert(alignof(FMatrix44f) < MAX_uint8);

```

Of course, if the definitions of subfields or parent classes are also not found, only the parent definition needs to be written first. Therefore, FindNoExportStructsRecursive in cs is used to identify related structures. The absence of the IsAlwaysAccessible flag indicates that a dummy structure definition will be generated

```

private static void FindNoExportStructsRecursive(List<UhtScriptStruct>
outScriptStructs, UhtStruct structObj)
{
    for (UhtStruct? current = structObj; current != null; current =
current.SuperStruct)
    {
        // Is isn't true for noexport structs
        if (current is UhtScriptStruct scriptStruct)
        {
            if
(scriptStruct.ScriptStructFlags.HasAnyFlags(EStructFlags.Native))
            {

```

```

        break;
    }

    // these are a special cases that already exists and if wrong
if exported naively
    if (!scriptStruct.IsAlwaysAccessible)
    {
        outScriptStructs.Remove(scriptStruct);
        outScriptStructs.Add(scriptStruct);
    }
}

foreach (uhtType type in current.Children)
{
    if (type is UhtProperty property)
    {
        foreach (UhtType referenceType in
property.EnumerateReferencedTypes())
        {
            if (referenceType is UhtScriptStruct
propertyScriptStruct)
            {
                FindNoExportStructsRecursive(outScriptStructs,
propertyScriptStruct);
            }
        }
    }
}
}

```

## HasDefaults

- **Function Description:** Indicates that the fields within this structure possess default values. Consequently, if this structure is passed as a function parameter or returned as a value, the function can assign default values to it.
- **Metadata Type:** bool
- **Engine Module:** UHT
- **Restriction Type:** Used exclusively by UHT in NoExportTypes.h
- **Mechanism:** Adds FUNC\_HasDefaults to the FunctionFlags
- **Commonality:** 0

The structure's fields have default values specified.

This does not refer to whether there are default values declared in NoExportTypes.h but rather to the actual declaration location, where its internal attributes are initialized with default values. This allows the function to provide default values when the structure is used as a function parameter or return value.

Most structures in NoExportTypes.h have this structure (88/135), and none are like FPackedXXX.

## Principle:

If a function within a class uses a structure as a parameter, and if that structure has the HasDefaults attribute, it results in EFunctionFlags having the HasDefaults flag set

```
// The following code is only performed on functions in a class.
if (Outer is UhtClass)
{
    foreach (UhtType type in children)
    {
        if (type is UhtProperty property)
        {
            if (property.PropertyFlags.HasExactFlags(EPropertyFlags.ReturnParm | EPropertyFlags.OutParm, EPropertyFlags.OutParm))
            {
                FunctionFlags |= EFunctionFlags.HasOutParms;
            }
            if (property is UhtStructProperty structProperty)
            {
                if (structProperty.ScriptStruct.HasDefaults)
                {
                    FunctionFlags |= EFunctionFlags.HasDefaults;
                }
            }
        }
    }
}
```

## HasNoOpConstructor

- **Function Description:** Specifies that the structure possesses a constructor with ForceInit, allowing it to be invoked for initialization when it serves as the return value of a BP function
- **Metadata Type:** bool
- **Engine Module:** UHT
- **Restriction Type:** Used exclusively by UHT in NoExportTypes.h
- **Frequency of Use:** 0

Specifies that the structure has a constructor with ForceInit, ensuring that when it is used as a return value or parameter for a BP Function, the engine is aware of this constructor's existence for initialization purposes.

The functionality is applied within the AppendEventParameter method of UhtHeaderCodeGenerator, where glue code needs to be generated for an Event exposed to BP. For instance, FLinearColor is marked as HasNoOpConstructor. Consider the following function:

```
UFUNCTION(BlueprintNativeEvent, Category = "Modifier")
FLinearColor GetVisualizationColor(FInputActionValue SampleValue,
FInputActionValue FinalValue) const;
```

Generated code:

```

struct InputModifier_eventGetVisualizationColor_Parms
{
    FInputActionValue Samplevalue;
    FInputActionValue Finalvalue;
    FLinearColor ReturnValue;

    /** Constructor, initializes return property only **/
    InputModifier_eventGetVisualizationColor_Parms()
        : ReturnValue(ForceInit)//Enforce initialization
    {
    }
};

static FName NAME_UInputModifier_GetVisualizationColor =
FName(TEXT("GetVisualizationColor"));
    FLinearColor UInputModifier::GetVisualizationColor(FInputActionValue
Samplevalue, FInputActionValue Finalvalue) const
{
    InputModifier_eventGetVisualizationColor_ParmsParms;
   Parms.Samplevalue=Samplevalue;
   Parms.Finalvalue=Finalvalue;
    const_cast<UInputModifier*>(this)-
>ProcessEvent(FindFunctionChecked(NAME_UInputModifier_GetVisualizationColor),&Par
ms);
    return Parm.ReturnValue;
}

```

Thus, the structure must have a constructor with ForceInit

```

FORCEINLINE explicit FLinearColor(EForceInit)
    : R(0) , G(0) , B(0) , A(0)
{}
```

## Principle:

```

if (scriptStruct.HasNoOpConstructor)
{
    //If true, the an argument will need to be added to the constructor
    PropertyCaps |= UhtPropertyCaps.RequiresNullConstructorArg;
}
```

## IsCoreType

- **Functional Description:** Indicates that this structure is a core class, and no forward declaration is necessary when UHT uses it.
- **Metadata Type:** bool
- **Engine Module:** UHT
- **Restriction Type:** Used exclusively by UHT in NoExportTypes.h
- **Commonality:** 0

Indicates that this structure is a core class, and UHT does not require a forward declaration when utilizing it.

## Principle:

Examining the UHT source code reveals the use of the 'struct' keyword in instances where parameters or attributes are being referenced.

```
public override string? UhtStructProperty::GetForwardDeclarations()
{
    if (ScriptStruct.IsCoreType)
    {
        return null;
    }

    if (Templatewrapper != null)
    {
        StringBuilder builder = new();
        Templatewrapper.AppendForwardDeclarations(builder);
        return builder.ToString();
    }

    return $"struct {ScriptStruct.SourceName};";
}
```

## BlueprintType

- **Functional Description:** Permits the declaration of variables within this structure in Blueprints
- **Metadata Type:** boolean
- **Engine Module:** Blueprint
- **Functionality Mechanism:** Incorporate BlueprintType into the Meta
- **Common Usage:** ★★★★☆

Comparable to UCLASS, this structure enables the declaration of variables in Blueprints

## Sample Code:

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_BlueprintType
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score;
};

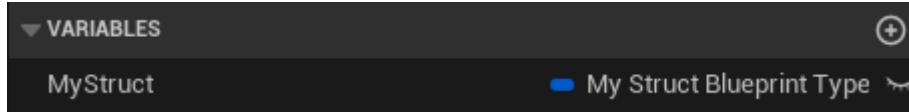
USTRUCT()
struct INSIDER_API FMyStruct_NoBlueprintType
{
    GENERATED_BODY()
```

```

UPROPERTY(EditAnywhere)
float Score;
};

```

## Test Blueprint:



# BlueprintInternalUseOnly

- **Function description:** No new BP variables can be defined, but they can be exposed as member variables of other classes or passed as variables
- **Metadata type:** bool
- **Engine module:** Blueprint
- **Mechanism of action:** Include BlueprintInternalUseOnly and BlueprintType in the Meta
- **Commonly used:** ★★

Indicates that this STRUCT is a BlueprintType, yet it cannot declare new variables in the Blueprint editor; however, it can be exposed as a member variable of other classes within Blueprints.

What is the difference if BlueprintType is not specified?

Without BlueprintType, it cannot be used as a member variable of other classes at all.

BlueprintInternalUseOnly inhibits the ability to define new variables but allows them to be passed as variables. For instance, defining variables in C++ and then passing them in Blueprints.

For example, FTableRowBase itself cannot define new variables, but its subclasses (when marked with BlueprintType) can define new variables and are used normally.

## Sample Code:

```

//(BlueprintInternalUseOnly = true, BlueprintType = true, ModuleRelativePath =
Struct/MyStruct_BlueprintInternaluseOnly.h)
USTRUCT(BlueprintInternaluseOnly)
struct INSIDER_API FMyStruct_BlueprintInternaluseOnly
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite>EditAnywhere)
    float Score=0.f;
};

USTRUCT()
struct INSIDER_API FMyStruct_NoBlueprintInternaluseOnly
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere)
    float Score=0.f;
};

```

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyClass_BlueprintInternalUseOnlyTest :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FMyStruct_BlueprintInternalUseOnly MyInternalStruct;

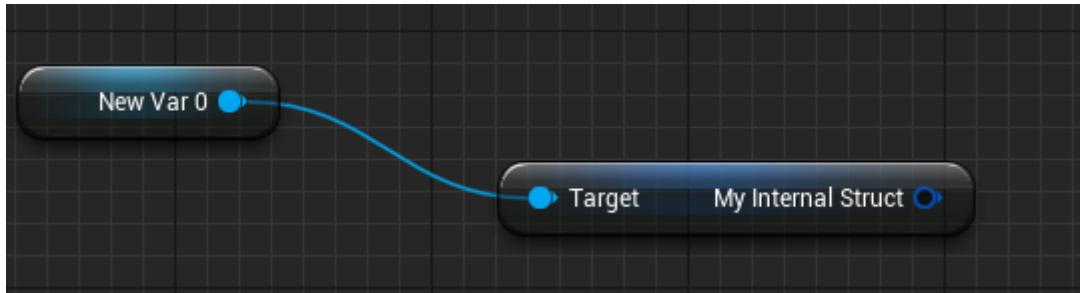
    /*UPROPERTY(BlueprintReadWrite, EditAnywhere) //no supported by BP
    FMyStruct_NoBlueprintInternalUseOnly MyStruct;*/

};


```

## Example Effect:

NewVar is of type UMyClass\_BlueprintInternalUseOnlyTest and still has access to the internal MyInternalStruct variable.



The source code is available at:

```

USTRUCT(BlueprintInternalUseOnly)
struct FLatentActionInfo
{};

USTRUCT(BlueprintInternalUseOnly)
struct FTableRowBase
{};


```

## Principle:

```

bool UEdGraphSchema_K2::IsAllowableBlueprintVariableType(const UScriptStruct* InStruct, const bool bForInternalUse)
{
    if (const UUserDefinedStruct* UDStruct = Cast<const UUserDefinedStruct>(InStruct))
    {
        if (EUserDefinedStructureStatus::UDSS_UpToDate != UDStruct->Status.GetValue())
        {
            return false;
        }

        // User-defined structs are always allowed as BP variable types.
        return true;
    }
}


```

```

    // struct needs to be marked as BP type
    if (InStruct && InStruct-
>GetBoolMetaDataHierarchical(FBlueprintMetadata::MD_AllowableBlueprintVariableTyp
e))
    {
        // for internal use, all BP types are allowed
        if (bForInternalUse)
        {
            return true;
        }

        // for user-facing use case, only allow structs that don't have the
        internal-use-only tag
        // struct itself should not be tagged
        if (!InStruct-
>GetBoolMetaData(FBlueprintMetadata::MD_BlueprintInternalUseOnly))
        {
            // struct's base structs should not be tagged
            if (!InStruct-
>GetBoolMetaDataHierarchical(FBlueprintMetadata::MD_BlueprintInternalUseOnlyHiera
rchical))
            {
                return true;
            }
        }
    }

    return false;
}

// If Node->IsIntermediateNode() returns true, it is used as an intermediate node,
// and this sets bForInternalUse to true
if (!UK2Node_MakeStruct::CanBeMade(Node->StructType, Node->IsIntermediateNode()))

```

## BlueprintInternalUseOnlyHierarchical

- **Function Description:** On top of BlueprintInternalUseOnly, it imposes the restriction that subclasses are also prohibited from defining new BP variables.
- **Metadata Type:** bool
- **Engine Module:** Blueprint
- **Action Mechanism:** Include BlueprintInternalUseOnlyHierarchical in the Meta section
- **Commonality:** ★

Building upon BlueprintInternalUseOnly, it introduces the limitation that subclasses are not allowed to define new BP variables.

Currently, this feature is only found to have one application, and there are no subclasses. If we define new subclasses in C++, none of these subclasses will be able to define variables. It is important to note that this differs from FTableRowBase, where subclasses can still define new variables because the BlueprintInternalUseOnly attribute of FTableRowBase only applies to itself.

## Sample Code:

```
USTRUCT(BlueprintInternaluseOnlyHierarchical)
struct GAMEPLAYABILITIESEDITOR_API FGameplayAbilityAuditRow : public
FTableRowBase
{};

USTRUCT(BlueprintInternaluseOnly)
struct FTableRowBase
{}
```

## Principle:

Utilized exclusively in this context, GetBoolMetaDataHierarchical checks whether any of the structure's parent classes have a specific tag. If any parent class possesses this tag, the definition of new variables is prohibited.

```
bool UEdGraphSchema_K2::IsAllowableBlueprintVariableType(const UScriptStruct*
InStruct, const bool bForInternalUse)
{
    if (const UUserDefinedStruct* UDStruct = Cast<const UUserDefinedStruct>
(InStruct))
    {
        if (EUserDefinedStructureStatus::UDSS_UpToDate != UDStruct-
>Status.GetValue())
        {
            return false;
        }

        // User-defined structs are always allowed as BP variable types.
        return true;
    }

    // struct needs to be marked as BP type
    if (InStruct && InStruct-
>GetBoolMetaDataHierarchical(FBlueprintMetadata::MD_AllowableBlueprintVariableTyp
e))
    {
        // for internal use, all BP types are allowed
        if (bForInternalUse)
        {
            return true;
        }

        // for user-facing use case, only allow structs that don't have the
internal-use-only tag
        // struct itself should not be tagged
        if (!InStruct-
>GetBoolMetaData(FBlueprintMetadata::MD_BlueprintInternaluseOnly))
        {
            // struct's base structs should not be tagged
            if (!InStruct-
>GetBoolMetaDataHierarchical(FBlueprintMetadata::MD_BlueprintInternaluseOnlyHiera
rchical))
```

```

        {
            return true;
        }
    }

    return false;
}

```

## immutable

- **Function Description:** Immutable is exclusively valid in Object.h and is being phased out; do not use it in new structs!
- **Metadata Type:** bool
- **Engine Module:** Serialization
- **Mechanism of Operation:** Incorporate STRUCT\_Immutable into StructFlags

Currently, a multitude of structs are identified within noexporttypes.h

The fields of this specified structure have been fully defined and will not be altered in the future. Consequently, they can be serialized with the aid of UseBinarySerialization, and there is no need to support the addition or removal of fields.

```

//USTRUCT(BlueprintType,Immutable) //error : Immutable is being phased out in
favor of SerializeNative, and is only legal on the mirror structs declared in
UObject
//struct INSIDER_API FMyStruct_Immutable
//{
//    GENERATED_BODY()
//
//    UPROPERTY(BlueprintReadWrite,EditAnywhere)
//    float Score;
//
//};

Struct[67] WithFlags:STRUCT_Immutable
Struct: ScriptStruct /Script/CoreUObject.Guid
Struct: ScriptStruct /Script/CoreUObject.DateTime
Struct: ScriptStruct /Script/CoreUObject.Box
Struct: ScriptStruct /Script/CoreUObject.Vector
Struct: ScriptStruct /Script/CoreUObject.Box2D
Struct: ScriptStruct /Script/CoreUObject.Vector2D
Struct: ScriptStruct /Script/CoreUObject.Box2f
Struct: ScriptStruct /Script/CoreUObject.Vector2f
Struct: ScriptStruct /Script/CoreUObject.Box3d
Struct: ScriptStruct /Script/CoreUObject.Vector3d
Struct: ScriptStruct /Script/CoreUObject.Box3f
Struct: ScriptStruct /Script/CoreUObject.Vector3f
Struct: ScriptStruct /Script/CoreUObject.Color
Struct: ScriptStruct /Script/CoreUObject.Int32Point
Struct: ScriptStruct /Script/CoreUObject.Int32Vector
Struct: ScriptStruct /Script/CoreUObject.Int32Vector2

```

```
Struct: ScriptStruct /Script/CoreUObject.Int32Vector4
Struct: ScriptStruct /Script/CoreUObject.Int64Point
Struct: ScriptStruct /Script/CoreUObject.Int64Vector
Struct: ScriptStruct /Script/CoreUObject.Int64Vector2
Struct: ScriptStruct /Script/CoreUObject.Int64Vector4
Struct: ScriptStruct /Script/CoreUObject.LinearColor
Struct: ScriptStruct /Script/CoreUObject.Quat
Struct: ScriptStruct /Script/CoreUObject.TwoVectors
Struct: ScriptStruct /Script/CoreUObject.IntPoint
Struct: ScriptStruct /Script/CoreUObject.IntVector
Struct: ScriptStruct /Script/CoreUObject.IntVector2
Struct: ScriptStruct /Script/CoreUObject.IntVector4
Struct: ScriptStruct /Script/CoreUObject.Matrix
Struct: ScriptStruct /Script/CoreUObject.Plane
Struct: ScriptStruct /Script/CoreUObject.Matrix44d
Struct: ScriptStruct /Script/CoreUObject.Plane4d
Struct: ScriptStruct /Script/CoreUObject.Matrix44f
Struct: ScriptStruct /Script/CoreUObject.Plane4f
Struct: ScriptStruct /Script/CoreUObject.OrientedBox
Struct: ScriptStruct /Script/CoreUObject.PackedNormal
Struct: ScriptStruct /Script/CoreUObject.PackedRGB10A2N
Struct: ScriptStruct /Script/CoreUObject.PackedRGBA16N
Struct: ScriptStruct /Script/CoreUObject.Quat4d
Struct: ScriptStruct /Script/CoreUObject.Quat4f
Struct: ScriptStruct /Script/CoreUObject.Ray
Struct: ScriptStruct /Script/CoreUObject.Ray3d
Struct: ScriptStruct /Script/CoreUObject.Ray3f
Struct: ScriptStruct /Script/CoreUObject.Rotator
Struct: ScriptStruct /Script/CoreUObject.Rotator3d
Struct: ScriptStruct /Script/CoreUObject.Rotator3f
Struct: ScriptStruct /Script/CoreUObject.Sphere
Struct: ScriptStruct /Script/CoreUObject.Sphere3d
Struct: ScriptStruct /Script/CoreUObject.Sphere3f
Struct: ScriptStruct /Script/CoreUObject.Timespan
Struct: ScriptStruct /Script/CoreUObject.Transform3d
Struct: ScriptStruct /Script/CoreUObject.Transform3f
Struct: ScriptStruct /Script/CoreUObject.Uint32Point
Struct: ScriptStruct /Script/CoreUObject.Uint32Vector
Struct: ScriptStruct /Script/CoreUObject.Uint32Vector2
Struct: ScriptStruct /Script/CoreUObject.Uint32Vector4
Struct: ScriptStruct /Script/CoreUObject.Uint64Point
Struct: ScriptStruct /Script/CoreUObject.Uint64Vector
Struct: ScriptStruct /Script/CoreUObject.Uint64Vector2
Struct: ScriptStruct /Script/CoreUObject.Uint64Vector4
Struct: ScriptStruct /Script/CoreUObject.UintPoint
Struct: ScriptStruct /Script/CoreUObject.UintVector
Struct: ScriptStruct /Script/CoreUObject.UintVector2
Struct: ScriptStruct /Script/CoreUObject.UintVector4
Struct: ScriptStruct /Script/CoreUObject.Vector4
Struct: ScriptStruct /Script/CoreUObject.Vector4d
Struct: ScriptStruct /Script/CoreUObject.Vector4f
```

# Flags

- **Function description:** Concatenate strings for output by using the value of this enumeration as a flag.
- **Metadata type:** bool
- **Engine module:** Trait
- **Mechanism of action:** Add Flags to EnumFlags
- **Commonly used:** ★★★★☆

Concatenate strings for output by using the value of this enumeration as a flag.

Specifies the context where this is applied: when a value is output as a string. The methods of conversion to a string include: one is to search directly for an exact match, then find the specific enumeration value; the second is to treat it as a flag, outputting values in the format "A...C" that correspond to the flags | B | Flags indicate the use of the second method.

However, note that the enumeration value itself does not affect any changes. This is different from directly defining enumeration values as markers.

Be careful to distinguish it from meta(bitflags), which indicates that the enumeration can be used as a flag and can be filtered as a bitmask.

## Sample Code:

```
UENUM(BlueprintType)
enum class EMyEnum_Normal: uint8
{
    First,
    Second,
    Third,
};

/*
[EMyEnum_Flags  Enum->Field->Object /Script/Insider.EMyEnum_Flags]
(BlueprintType = true, First.Name = EMyEnum_Flags::First, ModuleRelativePath =
Enum/MyEnum_Flags.h, Second.Name = EMyEnum_Flags::Second, Third.Name =
EMyEnum_Flags::Third)
    ObjectFlags:    RF_Public | RF_Transient
    Outer:    Package /Script/Insider
    EnumFlags:  EEnumFlags::Flags
    EnumDisplayNameFn: 0
    CppType:    EMyEnum_Flags
    CppForm:    EnumClass
{
    First = 0,
    Second = 1,
    Third = 2,
    EMyEnum_MAX = 3
};
*/
UENUM(BlueprintType, Flags)
enum class EMyEnum_Flags: uint8
{
```

```

First,
Second,
Third,
};

void UMyActor_EnumBitFlags_Test::TestFlags()
{
    int value = 3;

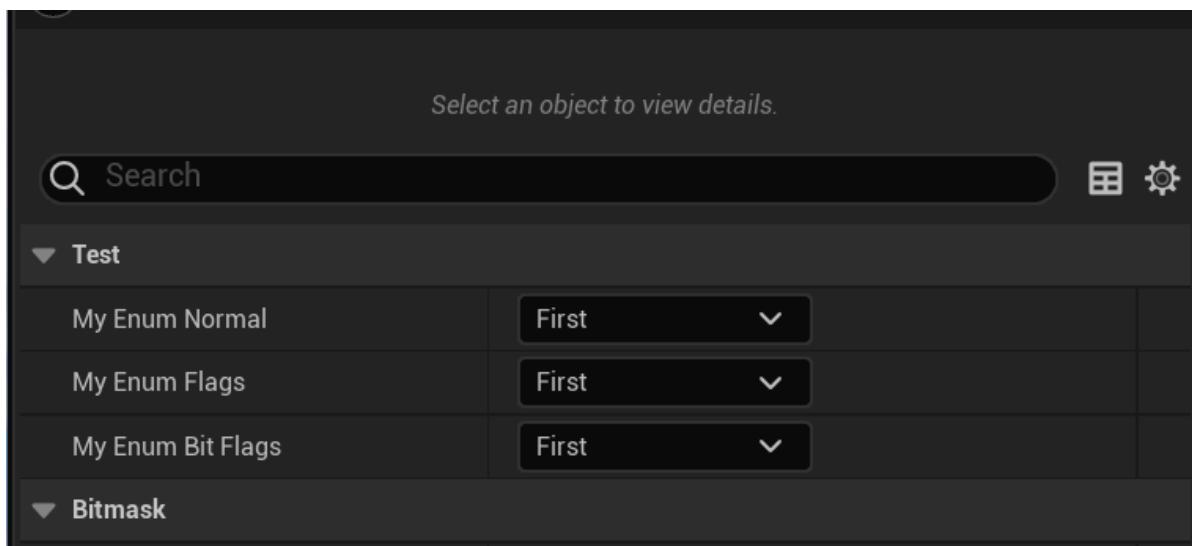
    FString outStr_Normal = StaticEnum<EMyEnum_Normal>()-
>GetValueOrBitfieldAsString(value);
    FString outStr_Flags = StaticEnum<EMyEnum_Flags>()-
>GetValueOrBitfieldAsString(value);
    FString outStr_BitFlags = StaticEnum<EMyEnum_BitFlags>()-
>GetValueOrBitfieldAsString(value);

}

```

## Example Effect:

In the blueprint representation, only a single item can still be selected.



And the string printed in the test code:

It can be seen that the printing of outStr\_Flags is string concatenation.

UEnum::GetValueOrBitfieldAsString returned	L"EMyEnum_MAX"	Q View ▾ FString &
outStr_BitFlags	L"EMyEnum_MAX"	Q View ▾ FString
outStr_Flags	L"Second   Third"	Q View ▾ FString
outStr_Normal	L"EMyEnum_MAX"	Q View ▾ FString
value	3	int

## Principle:

It only takes effect in the GetValueOrBitfieldAsString function, so it must be tested with this method to take effect.

```

FString UEnum::GetValueOrBitfieldAsString(int64 InValue) const
{
    if (!HasAnyEnumFlags(EEEnumFlags::Flags) || InValue == 0)
    {
        return GetNameStringByValue(InValue);
    }
}

```

```

    }
    else
    {
        FString Bitfieldstring;
        bool WroteFirstFlag = false;
        while (InValue != 0)
        {
            int64 NextValue = 111 << FMath::CountTrailingZeros64(InValue);
            InValue = InValue & ~NextValue;
            if (WroteFirstFlag)
            {
                // we don't just want to use the NameValuePair.Key because we
                // want to strip enum class prefixes
                BitfieldString.Appendf(TEXT(" | %s"),
                *GetNameStringByValue(NextValue));
            }
            else
            {
                // we don't just want to use the NameValuePair.Key because we
                // want to strip enum class prefixes
                BitfieldString.Appendf(TEXT("%s"),
                *GetNameStringByValue(NextValue));
                WroteFirstFlag = true;
            }
        }
        return BitfieldString;
    }
}

```

## BlueprintType

---

- **Function Description:** Can serve as a variable within blueprints
- **Metadata Type:** Boolean
- **Engine Module:** Blueprint
- **Functionality Mechanism:** Add BlueprintType to the Meta section
- **Commonality:** ★★★★☆

The usage of BlueprintType is consistent with its application in other contexts.

## Category

---

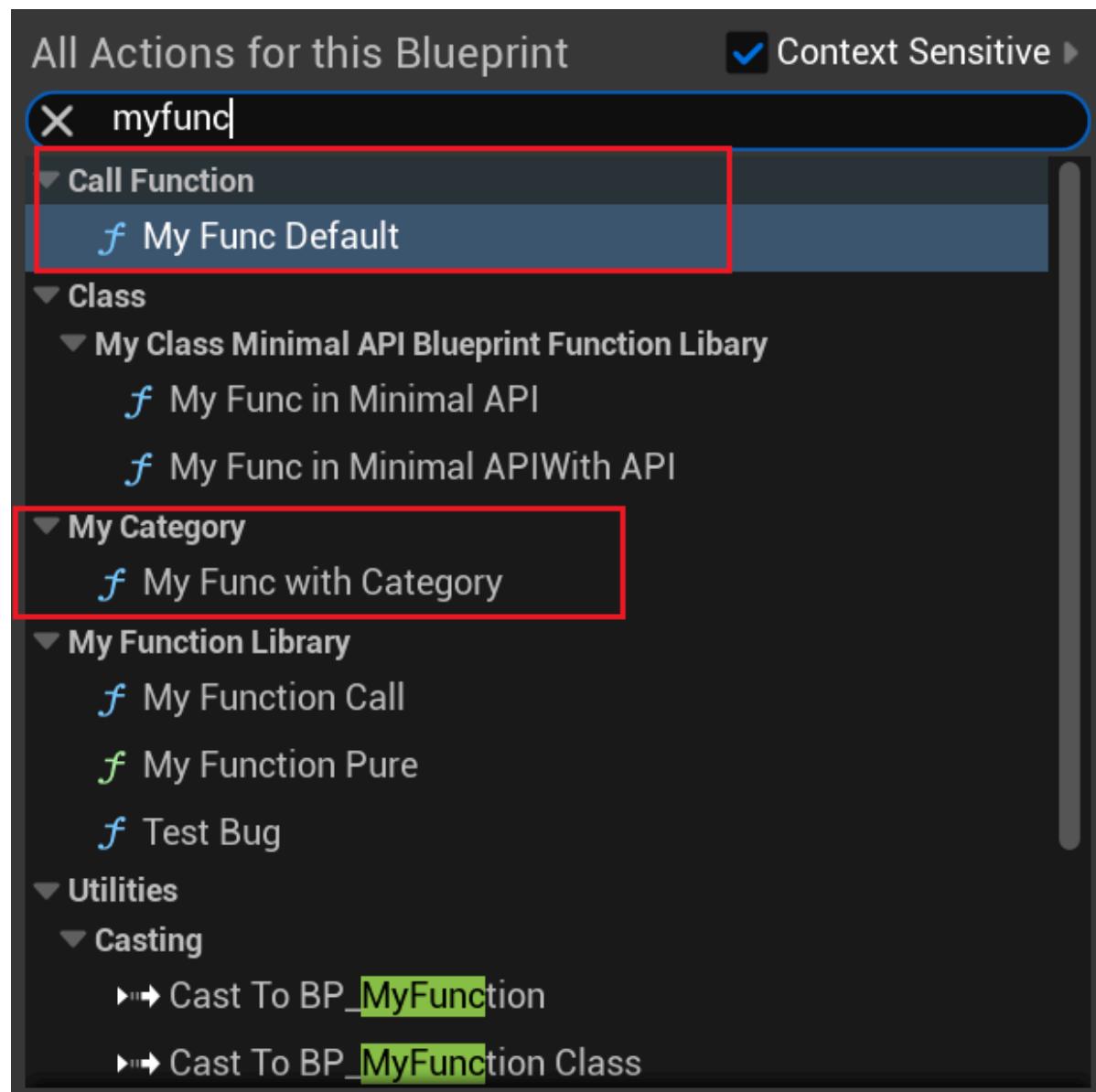
- **Function Description:** In the right-click menu of the blueprint, assign a category group to the function, allowing for multi-level nesting
- **Metadata Type:** strings = "a|b|c"
- **Engine Module:** Editor
- **Action Mechanism:** Include Category in the Meta
- **Common Usage:** ★★★★☆

Assign a category group to the function in the blueprint's right-click menu.

## Testing Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Default :public AActor
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, Category = MyCategory)
    void MyFunc_WithCategory(){}
    UFUNCTION(BlueprintCallable)
    void MyFunc_Default(){}
};
```

## Display in the Blueprint:



# CallInEditor

---

- **Function Description:** Can be invoked as a button on the property details panel.
- **Metadata Type:** bool
- **Engine Module:** Editor
- **Functionality Mechanism:** Add CallInEditor in Meta
- **Common Usage:** ★★★★☆

Can be invoked as a button on the property details panel.

The function can be implemented in either an AActor or a UObject subclass, provided there is an associated property details panel.

Note that this is typically within the Editor execution environment. A classic example is the Recapture button in ASkyLight. Functions may sometimes call editor-specific functions. However, care should be taken to avoid mixing them with runtime functions, as this can easily lead to errors.

## Test Code:

---

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Default :public AActor
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(CallInEditor)
    void MyFunc_CallInEditor(){}
};
```

## Blueprint Demonstration:

The screenshot shows the Unreal Engine's World Settings panel. At the top, there are tabs for "Details" and "World Settings". Below that, the title bar displays "BP\_MyFunction" and "World Settings". A toolbar on the right includes a green plus icon for "Add", a lock icon, and a dropdown menu. The main area shows a list of components: "BP\_MyFunction (Self)" (selected), "DefaultSceneRoot", and "Edit in Blueprint".

Below the list is a search bar with a magnifying glass icon and a "Search" placeholder. Underneath are several tabs: "General", "Actor", "Misc", "Streaming", and "All" (which is selected). A horizontal scroll bar is visible below these tabs.

The configuration section starts with a "Replication" group, which contains the setting "Net Load on Client" with a checked checkbox. The next group is "Rendering", containing "Actor Hidden In Game" (unchecked) and "Editor Billboard Scale" set to "1.0".

A red box highlights the "Default" group, which contains a single entry: "My Func Call in Editor".

Following the "Default" group is the "Collision" group, which includes settings like "Generate Overlap Events During Le...", "Update Overlaps Method During L...", and "Default Update Overlaps Method D...".

A "Advanced" section follows, indicated by a right-pointing arrow.

The final group shown is "HLOD", containing the setting "Include Actor in HLOD" with a checked checkbox.

The last group is "Input", containing "Auto Receive Input" set to "Disabled" and "Input Priority" set to "0".

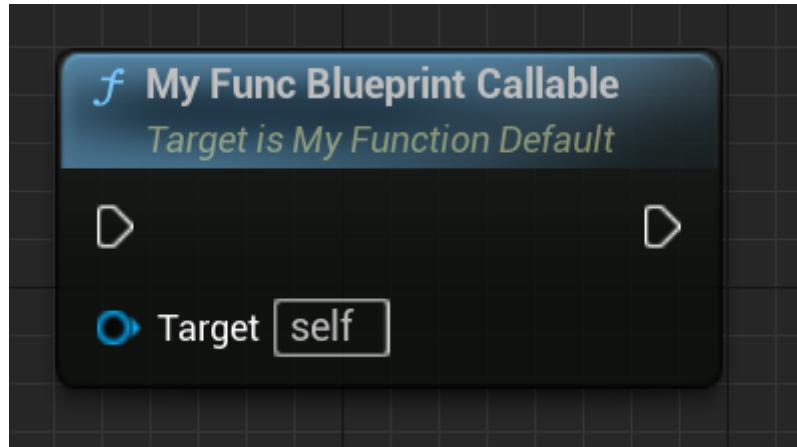
# BlueprintCallable

- **Function Description:** Exposed to the blueprint and callable
- **Metadata Type:** boolean
- **Engine Module:** Blueprint
- **Action Mechanism:** Add FUNC\_BlueprintCallable to the Function Flags
- **Commonly Used:** ★★★★

## Test Code:

```
UFUNCTION(BlueprintCallable)
void MyFunc_BlueprintCallable() {}
```

## Effect Display:



# BlueprintPure

- **Function Description:** Designated as a pure function, typically used by Get functions to return values.
- **Metadata Type:** bool
- **Engine Module:** Blueprint
- **Functionality Mechanism:** Adds FUNC\_BlueprintCallable and FUNC\_BlueprintPure to FunctionFlags
- **Common Usage:** ★★★★

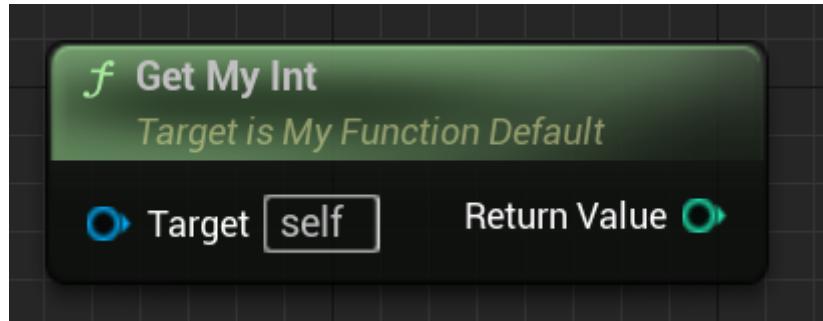
Designated as a pure function, typically used by Get functions to return values.

- Pure functions are those without execution pins, not to be confused with const functions.
- Pure functions can have multiple return values, which can be added to the function using reference parameters.
- Cannot be used for void functions; otherwise, an error will be reported: "error: BlueprintPure specifier is not allowed for functions with no return value and no output parameters."

## Test Code:

```
UFUNCTION(BlueprintPure)
    int32 GetMyInt() const { return MyInt; }
private:
    int32 MyInt;
```

## Effect Display:



# BlueprintImplementableEvent

- **Function Description:** Specifies a function call point that can be overridden in the blueprint.
- **Metadata Type:** bool
- **Engine Module:** Blueprint
- **Action Mechanism:** Adds FUNC\_Event, FUNC\_Native, FUNC\_BlueprintEvent to FunctionFlags
- **Commonly Used:** ★★★★☆

Specifies a function call point that can be overridden in the blueprint, providing a convenient method for C++ to invoke blueprint functions.

If no implementation is provided in the blueprint, invoking the function is equivalent to calling an empty function.

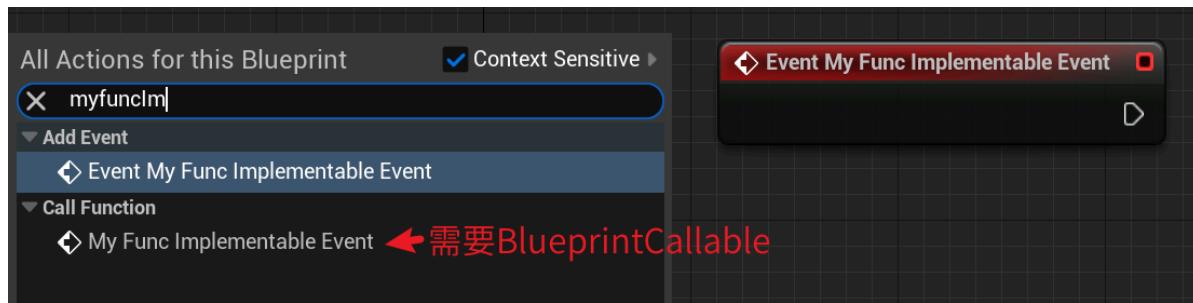
BlueprintImplementableEvent must be used in conjunction with BlueprintCallable. Without BlueprintCallable, it can only be called from C++, and the Call Function node will not be found in the blueprint.

## Test Code:

```
//FunctionFlags: FUNC_Event | FUNC_Public | FUNC_BlueprintCallable |
FUNC_BlueprintEvent
UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
void MyFunc_ImplementableEvent();
```

## Effect Display:

Right-click to add a custom implementation



## Principle:

When called in C++, the function searches for the name using `FindFunctionChecked`. If an implementation is found in the blueprint, it will be invoked. If called directly from the blueprint, the function also uses `FindFunctionChecked` to search; if a definition exists in the blueprint, it will be found directly.

```
void AMyFunction_Default::MyFunc_ImplementableEvent()
{
    ProcessEvent(FindFunctionChecked(NAME_AMyFunction_Default_MyFunc_ImplementableEvent), NULL);
}
```

## BlueprintNativeEvent

- **Function description:** The implementation can be overridden entirely in Blueprints, but a default implementation is also provided in C++.
- **Metadata type:** bool
- **Engine module:** Blueprint
- **Mechanism of action:** FUNC\_Event and FUNC\_BlueprintEvent are added to the FunctionFlags
- **Frequency of use:** ★★★★☆

The implementation can be overridden in Blueprints, but a default implementation is also provided in C++.

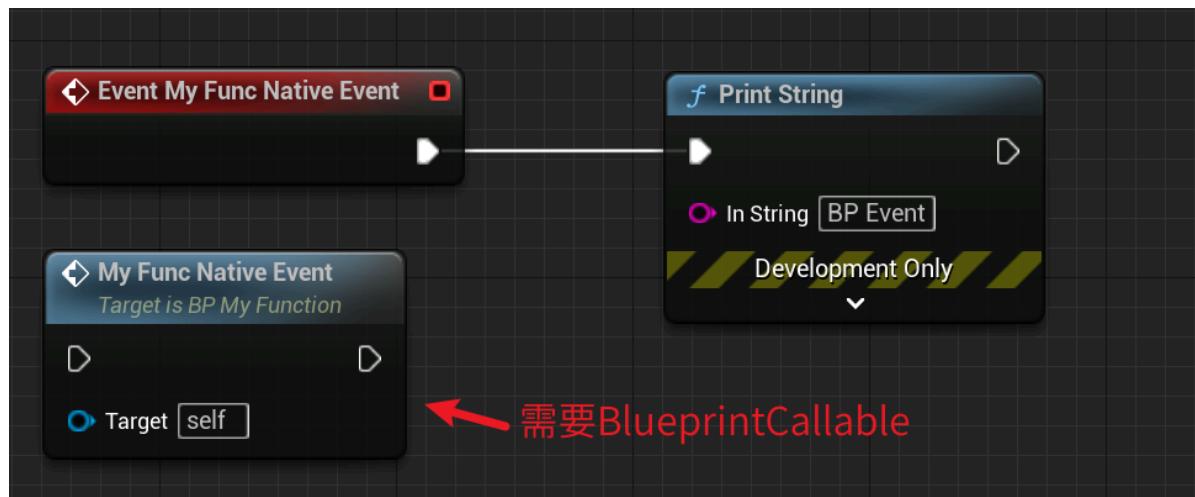
An additional function needs to be declared in C++ with the same name as the main function, but with "*Implementation*" appended to the end. *If no Blueprint override is found, the automatically generated code will invoke the "[FunctionName]Implementation" method.* This is typically used for functions like OnXXX, where C++ provides the implementation, allowing for the default C++ version to be called if no Blueprint override exists.

BlueprintNativeEvent must be accompanied by BlueprintCallable; otherwise, it can only be called within C++.

## Test Code:

```
//FunctionFlags: FUNC_Native | FUNC_Event | FUNC_Public |  
FUNC_BlueprintCallable | FUNC_BlueprintEvent  
UFUNCTION(BlueprintCallable, BlueprintNativeEvent)  
void MyFunc_NativeEvent();  
  
void AMyFunction_Default::MyFunc_NativeEvent_Implementation()  
{  
    GEngine->AddOnScreenDebugMessage(-1, 3.f, FColor::Red,  
    "MyFunc_NativeEvent_Implementation");  
}
```

## Effect Display:



## Principle:

When calling MyFunc\_NativeEvent, the internal FindFunctionChecked will search for the function by name. If it is defined in a Blueprint, the Blueprint's implementation will be found; otherwise, the implementation for execMyFunc\_NativeEvent will be located, thus calling MyFunc\_NativeEvent\_Implementation.

```
DEFINE_FUNCTION(AMyFunction_Default::execMyFunc_NativeEvent)  
{  
    P_FINISH;  
    P_NATIVE_BEGIN;  
    P_THIS->MyFunc_NativeEvent_Implementation();  
    P_NATIVE_END;  
}  
  
void AMyFunction_Default::MyFunc_NativeEvent()  
{  
  
    ProcessEvent(FindFunctionChecked(NAME_AMyFunction_Default_MyFunc_NativeEvent), NULL);  
}
```

# BlueprintGetter

---

- **Function Description:** Designates this function as a custom getter for the property attribute.
- **Metadata Type:** boolean
- **Engine Module:** Blueprint
- **Functionality Mechanism:** Includes BlueprintGetter in the Meta category and FUNC\_BlueprintCallable, FUNC\_BlueprintPure in the Function Flags
- **Commonly Used:** ★★

Designates this function as a custom getter for the property.

This specifier implicitly includes BlueprintPure and BlueprintCallable.

For further details, refer to the BlueprintGetter of UPROPERTY

# BlueprintSetter

---

- **Function Description:** Designates this function as a custom setter for the property attribute.
- **Metadata Type:** bool
- **Engine Module:** Blueprint
- **Functionality Mechanism:** Incorporates BlueprintSetter into the Meta and FUNC\_BlueprintCallable into the Function Flags
- **Commonly Used:** ★★

Designates this function as a custom setter for the property.

This specifier implicitly includes BlueprintCallable.

For further details, refer to the UPROPERTY BlueprintSetter

# Exec

---

- **Function description:** Register a function within a specific class as a console command that can accept arguments.
- **Metadata type:** bool
- **Engine module:** Behavior
- **Restriction type:** Specific several classes
- **Action mechanism:** Add FUNC\_Exec to FunctionFlags
- **Commonly used:** ★★★

Classes typically include: UPlayerInput, APlayerController, APawn, AHUD, AGameModeBase, ACheatManager, AGameStateBase, and subclasses of APlayerCameraManager.

When a console command is entered in the viewport, the execution sequence is as follows: UConsole::ConsoleCommand, then APlayerController::ConsoleCommand, followed by UPlayer::ConsoleCommand. In between, ViewportClient->Exec is attempted (which may handle some editor commands), and finally, ULocalPlayer::Exec is reached (which has already processed some custom commands).

UGameViewportClient, UGameInstance, and UPlayer inherit from FExec, hence they include four overloaded virtual functions: Exec, Exec\_Runtime, Exec\_Dev, and Exec\_Editor.

UEngine::Exec internally forwards the call to various modules for handling. The significant function here is StaticExec, which ultimately calls FSelfRegisteringExec::StaticExec(InWorld, Cmd, Ar) to invoke self-registered Execs.

If a command is executed in the editor with ~, FConsoleCommandExecutor::ExecInternal will ultimately lead to ULocalPlayer::Exec.

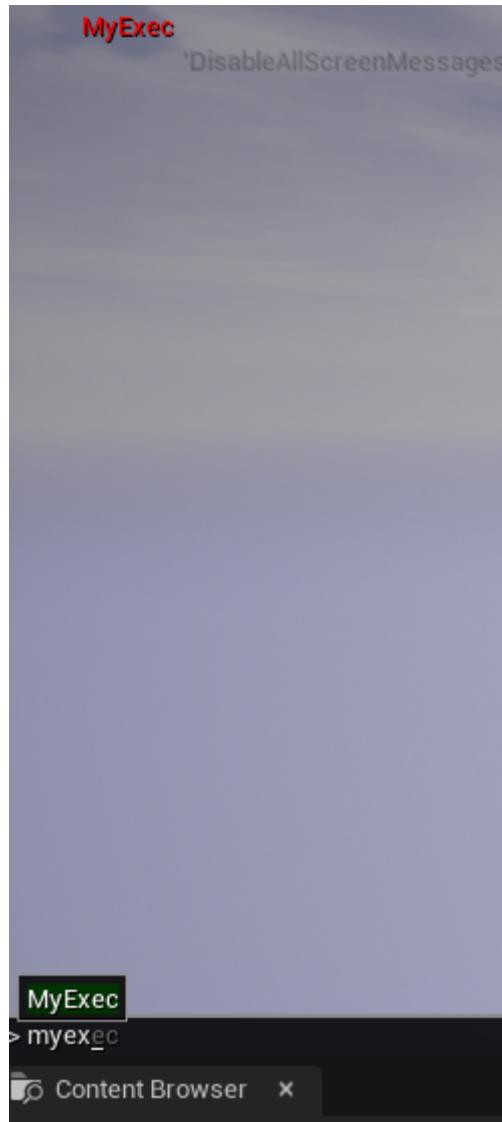
## Test code:

---

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Exec :public APawn
{
public:
    GENERATED_BODY()
public:
    //FunctionFlags: FUNC_Final | FUNC_Exec | FUNC_Native | FUNC_Public
    UFUNCTION(exec)
    void MyExec();
};

void AMyFunction_Exec::MyExec()
{
    GEngine->AddOnScreenDebugMessage(-1, 3.f, FColor::Red, "MyExec");
}
```

Results when opening the console and running during PIE:



## Principle:

According to the source code flow:

```
bool UGameViewportClient::Exec(Uworld* InWorld, const TCHAR* Cmd, FOutputDevice& Ar)
{
    //ULocalPlayer::Exec_Editor, Exec_Dev, and Exec_Runtime are checked in order
    //to determine if they correspond to certain commands
    if (FExec::Exec(InWorld, Cmd, Ar))
    {
        return true;
    }
    else if (ProcessConsoleExec(Cmd, Ar, NULL))
    {
        return true;
    }
    else if (GameInstance && (GameInstance->Exec(InWorld, Cmd, Ar) ||
        GameInstance->ProcessConsoleExec(Cmd, Ar, nullptr)))
    {
        return true;
    }
    else if (GEngine->Exec(InWorld, Cmd, Ar))
    {
```

```

        return true;
    }
    else
    {
        return false;
    }
}

bool UPlayer::Exec( Uworld* InWorld, const TCHAR* Cmd, FOutputDevice& Ar)
{
    // Route through Exec_Dev and Exec_Editor first
    //ULocalPlayer::Exec_Editor, Exec_Dev, and Exec_Runtime are checked in order
    to determine if they correspond to certain commands
    if (FExec::Exec(InWorld, Cmd, Ar))
    {
        return true;
    }

    AActor* ExecActor = PlayerController;
    if (!ExecActor)
    {
        UNetConnection* NetConn = Cast<UNetConnection>(this);
        ExecActor = (NetConn && NetConn->OwningActor) ? ToRawPtr(NetConn-
>OwningActor) : nullptr;
    }

    if (ExecActor)
    {
        // Since UGameViewportClient calls Exec on Uworld, we only need to
        explicitly
        // call Uworld::Exec if we either have a null GEngine or a null
        ViewportClient
        Uworld* world = ExecActor->GetWorld();
        check(world);
        check(InWorld == nullptr || InWorld == world);
        const bool bWorldNeedsExec = GEngine == nullptr || Cast<ULocalPlayer>
        (this) == nullptr || static_cast<ULocalPlayer*>(this)->ViewportClient == nullptr;
        APawn* PCPawn = PlayerController ? PlayerController->GetPawnOrSpectator()
        : nullptr;
        if (bWorldNeedsExec && world->Exec(world, Cmd, Ar))
        {
            return true;
        }
        else if (PlayerController && PlayerController->PlayerInput &&
        PlayerController->PlayerInput->ProcessConsoleExec(Cmd, Ar, PCPawn))
        {
            return true;
        }
        else if (ExecActor->ProcessConsoleExec(Cmd, Ar, PCPawn))
        {
            return true;
        }
        else if (PCPawn && PCPawn->ProcessConsoleExec(Cmd, Ar, PCPawn))
        {
            return true;
        }
    }
}

```

```

        else if (PlayerController && PlayerController->MyHUD && PlayerController-
>MyHUD->ProcessConsoleExec(Cmd, Ar, PCPawn))
    {
        return true;
    }
    else if (world->GetAuthGameMode() && world->GetAuthGameMode()-_
>ProcessConsoleExec(Cmd, Ar, PCPawn))
    {
        return true;
    }
    else if (PlayerController && PlayerController->CheatManager &&
PlayerController->CheatManager->ProcessConsoleExec(Cmd, Ar, PCPawn))
    {
        return true;
    }
    else if (world->GetGameState() && world->GetGameState()-_
>ProcessConsoleExec(Cmd, Ar, PCPawn))
    {
        return true;
    }
    else if (PlayerController && PlayerController->PlayerCameraManager &&
PlayerController->PlayerCameraManager->ProcessConsoleExec(Cmd, Ar, PCPawn))
    {
        return true;
    }
}
return false;
}

```

The order to search for Exec should be:

- UGameInstance::Exec, UGameInstance::ProcessConsoleExec
- GEngine->Exec(InWorld, Cmd, Ar)
- Uworld::Exec, when there is no LocalPlayer handling
- UPlayerInput::ProcessConsoleExec
- APlayerController::ProcessConsoleExec
- APawn::ProcessConsoleExec
- AHUD::ProcessConsoleExec
- AGameModeBase::ProcessConsoleExec
- ACheatManager::ProcessConsoleExec
- AGameStateBase::ProcessConsoleExec
- APlayerCameraManager::ProcessConsoleExec

ProcessConsoleExec internally calls the CallFunctionByNameWithArguments code, thus indeed limiting the way Execs declared in this manner to only the aforementioned classes

```

bool Uobject::CallFunctionByNameWithArguments(const TCHAR* Str, FOutputDevice&
Ar, UObject* Executor, bool bForceCallWithNonExec/*=false*/)
{
    UFunction* Function = FindFunction(Message); //Search for function
}

```

## SealedEvent

- **Function description:** This function cannot be overridden in subclasses. SealedEvent Keyword can only be used for events. For non-event functions, declare them as static or final to seal them.
- **Metadata Type:** bool
- **Engine Module:** Behavior
- **Mechanism of action:** Add FUNC\_Final in FunctionFlags

Search in the source code: It is found that they are all used in network-related functions

```

▲ Engine\Classes\GameFramework (8)
  ▲ HUD.h (3)
    UFUNCTION(reliable, client, SealedEvent)
    UFUNCTION(reliable, client, SealedEvent)
    UFUNCTION(reliable, client, SealedEvent)

  ▲ PlayerController.h (5)
    UFUNCTION(reliable, client, SealedEvent)
    UFUNCTION(reliable, client, SealedEvent)
    UFUNCTION(reliable, server, WithValidation, SealedEvent)
    UFUNCTION(reliable, server, WithValidation, SealedEvent)
    UFUNCTION(reliable, server, WithValidation, SealedEvent)

```

## Processing in UHT:

```

//First, identify the symbol
[UhtSpecifier(Extends = UhtTableNames.Function, valueType =
UhtSpecifierValueType.Legacy)]
private static void SealedEventSpecifier(UhtSpecifierContext specifierContext)
{
    UhtFunction function = (UhtFunction)specifierContext.Type;
    function.FunctionExportFlags |= UhtFunctionExportFlags.SealedEvent;
}

//Then, set the mark
// Handle the initial implicit/explicit final
// A user can still specify an explicit final after the parameter list as well.
if (automaticallyFinal ||
function.FunctionExportFlags.HasAnyFlags(UhtFunctionExportFlags.SealedEvent))
{
    function.FunctionFlags |= EFunctionFlags.Final;
    function.FunctionExportFlags |= UhtFunctionExportFlags.Final |
UhtFunctionExportFlags.AutoFinal;
}

```

再验证：限定只能用在Event上。

```
if (FunctionExportFlags.HasAnyFlags(UhtFunctionFlags.SealedEvent) &&
    !FunctionFlags.HasAnyFlags(EFunctionFlags.Event))
{
    this.LogError("SealedEvent may only be used on events");
}

if (FunctionExportFlags.HasAnyFlags(UhtFunctionFlags.SealedEvent) &&
    FunctionFlags.HasAnyFlags(EFunctionFlags.BlueprintEvent))
{
    this.LogError("SealedEvent cannot be used on Blueprint events");
}
```

## Test Code:

```
//Error: "SealedEvent may only be used on events"
UFUNCTION(SealedEvent)
void MyFunc_SealedEvent() {}

//Error: "SealedEvent cannot be used on Blueprint events"
UFUNCTION(BlueprintCallable, BlueprintImplementableEvent, SealedEvent)
void MyFunc_ImplementableEvent();

//Error: "SealedEvent cannot be used on Blueprint events"
UFUNCTION(BlueprintCallable, BlueprintNativeEvent, SealedEvent)
void MyFunc_NativeEvent();
```

Therefore, it is not applicable to regular functions, nor can it be used for events within blueprints. So, what is this that is both an event and not a BlueprintEvent? A look at the source code reveals that it pertains only to certain network functions.

Through comparison, we found that the difference between the Sealed functions is the addition of the FUNC\_Final mark. But FUNC\_Final does not necessarily have to be added with SealedEvent , exec or the ordinary BlueprintCallable function will be added. But if it is a virtual function, it will not be added. The principle in UHT is:

```
private static UhtParseResult ParseUFunction(UhtParsingsScope parentScope,
UhtToken token)
{
    if (function.FunctionFlags.HasAnyFlags(EFunctionFlags.Net))
    {
        // Network replicated functions are always events, and
        // are only final if sealed
        scopeName = "event";
        tokenContext.Reset(scopeName);
        automaticallyFinal = false;
    }

    // If virtual, remove the implicit final, the user can still specifying
    // an explicit final at the end of the declaration
```

```

if
(function.FunctionExportFlags.HasAnyFlags(UhtFunctionExportFlags.Virtual))
{
    automaticallyFinal = false;
}
// Handle the initial implicit/explicit final
// A user can still specify an explicit final after the parameter list as
well.
if (automaticallyFinal ||
function.FunctionExportFlags.HasAnyFlags(UhtFunctionExportFlags.SealedEvent))
{
    function.FunctionFlags |= EFunctionFlags.Final;
    function.FunctionExportFlags |=
UhtFunctionExportFlags.Final | UhtFunctionExportFlags.AutoFinal;
}

}

```

I tested it in my own C++ code and found that no compilation error would be triggered no matter how I inherited it in C++. Therefore, if you want to refuse to be inherited, you should still use the C++ standard final keyword. Add final at the end of the function.

E:\P4V\Engine\Source\Editor\KismetCompiler\Private\KismetCompiler.cpp

```

const uint32 overrideFlagsToCheck = (FUNC_FuncOverrideMatch &
~FUNC_AccessSpecifiers);
if ((Context.Function->FunctionFlags & overrideFlagsToCheck) !=
(OverriddenFunction->FunctionFlags & overrideFlagsToCheck))
{
    MessageLog.Error(*LOCTEXT("IncompatibleOverrideFlags_Error", "Overridden
function is not compatible with the parent function @@. Check flags: Exec, Final,
Static.").ToString(), Context.EntryPoint);
}

```

During compilation, it is detected whether the function of the overloaded parent class is overloaded, but because SealedEvent does not act on ordinary functions or BlueprintEvent, it feels that it can only be inherited in C++.

## BlueprintAuthorityOnly

- **Function description:** This function can only execute on a terminal with network access.
- **Metadata type:** bool
- **Engine module:** Network
- **Action mechanism:** Add FUNC\_BlueprintAuthorityOnly to the FunctionFlags
- **Commonality:** ★★★

This function can only be executed on a client with network permissions. HasAuthority::(GetLocalRole() == ROLE\_Authority). There are four types of NetRole: ROLE\_None (not replicated), ROLE\_SimulatedProxy (simulated proxy on the client), ROLE\_AutonomousProxy (anonymous proxy on the client, receiving player input), and ROLE\_Authority (the server with permissions).

Thus, BlueprintAuthorityOnly restricts this function to run only on the server, which can be an LS server, a DS server, or a standalone server (considered as a server without a client).

Note that the Actor must be set to Replicates for testing purposes.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Network :public AActor
{
public:
    GENERATED_BODY()
public:
    //FunctionFlags: FUNC_Final | FUNC_Native | FUNC_Public | FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable)
    void MyFunc_Default();

    //FunctionFlags: FUNC_Final | FUNC_BlueprintAuthorityOnly | FUNC_Native | FUNC_Public | FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable, BlueprintAuthorityOnly)
    void MyFunc_BlueprintAuthorityOnly();

    static void PrintFuncStatus(AActor* actor, FString funcName);
};

void AMyFunction_Network::MyFunc_Default()
{
    PrintFuncStatus(this, TEXT("MyFunc_Default"));
}

void AMyFunction_Network::MyFunc_BlueprintAuthorityOnly()
{
    PrintFuncStatus(this, TEXT("MyFunc_BlueprintAuthorityOnly"));
}

void AMyFunction_Network::PrintFuncStatus(AActor* actor, FString funcName)
{
    FString actorName = actor->GetName();

    FString localRoleStr;
    UEnum::GetValueAsString(actor->GetLocalRole(), localRoleStr);

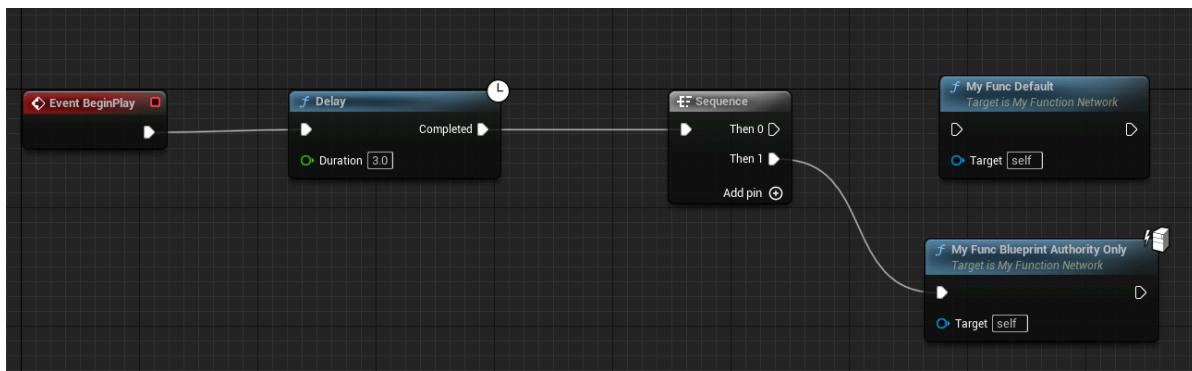
    FString remoteRoleStr;
    UEnum::GetValueAsString(actor->GetRemoteRole(), remoteRoleStr);

    FString netModeStr = Insider::NetModeToString(actor->GetNetMode());

    FString str = FString::Printf(TEXT("%s\t%s\t%s\tLocal:%s\tRemote:%s"),
        *funcName, *actorName, *netModeStr, *localRoleStr, *remoteRoleStr);
    GEngine->AddOnScreenDebugMessage(-1, 20.f, FColor::Red, str);

    UE_LOG(LogInsider, Display, TEXT("%s"), *str);
}
```

## Blueprint Code:



For non-replicated Actors:

```
MyFunc_Default BP_Network_C_1 NM_ListenServer Local:ROLE_Authority
  Remote:ROLE_None
MyFunc_Default BP_Network_C_1 NM_Client Local:ROLE_None Remote:ROLE_Authority
MyFunc_Default BP_Network_C_1 NM_Client Local:ROLE_None Remote:ROLE_Authority
```

For replicated Actors, with 1 Server and two Clients simultaneously, run the standard function:

```
MyFunc_Default BP_Network_C_1 NM_ListenServer Local:ROLE_Authority
  Remote:ROLE_SimulatedProxy
MyFunc_Default BP_Network_C_1 NM_Client Local:ROLE_SimulatedProxy
  Remote:ROLE_Authority
MyFunc_Default BP_Network_C_1 NM_Client Local:ROLE_SimulatedProxy
  Remote:ROLE_Authority
```

If the BlueprintAuthorityOnly function is permitted:

```
MyFunc_BlueprintAuthorityOnly BP_Network_C_1 NM_ListenServer
  Local:ROLE_Authority Remote:ROLE_SimulatedProxy
```

The results indicate that the Default function can run on all three ends, whereas BlueprintAuthorityOnly operates exclusively on the server and not on the client.

## Principle:

```
int32 AActor::GetFunctionCallspace( UFunction* Function, FFrame* Stack )
{
    FunctionCallspace::Type Callspace = (LocalRole < ROLE_Authority) && Function->HasAllFunctionFlags(FUNC_BlueprintAuthorityOnly) ? FunctionCallspace::Absorbed : FunctionCallspace::Local;
}
```

## BlueprintCosmetic

- **Function description:** This function is purely decorative and cannot be executed on the DS.
- **Metadata type:** bool
- **Engine module:** Network

- **Mechanism of action:** FUNC\_BlueprintCosmetic is added to the FunctionFlags
- **Commonly used:** ★★★

This function is decorative in nature, which means it is intended to present content unrelated to the logic, such as animations, sound effects, and visual effects. Since the DS lacks a visual output, these decorative functions are irrelevant to the DS and will be disregarded.

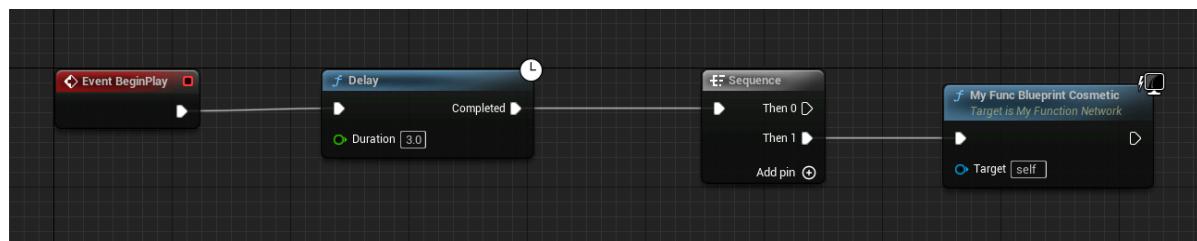
However, it is important to note that on a ListenServer or a Client, both are permitted to run. This is because both ends require a visual display.

## Test Code:

```
UFUNCTION(BlueprintCallable, BlueprintCosmetic)
void MyFunc_BlueprintCosmetic();
```

## Test Blueprint:

The computer tag on the node indicates that the function will execute exclusively on the client side.



Result output

```
MyFunc_BlueprintCosmetic      BP_Network_C_1  NM_ListenServer  Local:ROLE_Authority
                             Remote:ROLE_SimulatedProxy
MyFunc_BlueprintCosmetic      BP_Network_C_1  NM_Client       Local:ROLE_SimulatedProxy
                             Remote:ROLE_Authority
MyFunc_BlueprintCosmetic      BP_Network_C_1  NM_Client       Local:ROLE_SimulatedProxy
                             Remote:ROLE_Authority
```

## Principle:

```
int32 AActor::GetFunctionCallspace( UFunction* Function, FFrame* Stack )
{
// Dedicated servers don't care about "cosmetic" functions.
if (NetMode == NM_DedicatedServer && Function->HasAllFunctionFlags(FUNC_BlueprintCosmetic))
{
    DEBUG_CALLSPACE(TEXT("GetFunctionCallspace Blueprint Cosmetic Absorbed: %s"),
*Function->GetName());
    return FunctionCallspace::Absorbed;
}
}
```

# Client

- **Function Description:** Executes an RPC function on a Client-owned Actor (PlayerController or Pawn), which runs exclusively on the client side. The corresponding implementation function will have the \_Implementation suffix appended.
- **Metadata Type:** bool
- **Engine Module:** Network
- **Functionality Mechanism:** Adds FUNC\_Net and FUNC\_NetClient to the Function Flags
- **Usage Frequency:** ★★★★★

Executes an RPC function on a Client-owned Actor (PlayerController or Pawn), which runs exclusively on the client side. The corresponding implementation function will have the \_Implementation suffix appended.

Usually used to send an RPC from the Server to the Client, similar to the RunOnClient event in Blueprints.

For the definition of Client-owned, refer to the documentation:

<https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/Networking/Actors/RPCs/>

从服务器调用的 RPC

Actor 所有权	未复制	NetMulticast	Server	Client
Client-owned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在 actor 的所属客户端上运行
Server-owned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在服务器上运行
Unowned actor	在服务器上运行	在服务器和所有客户端上运行	在服务器上运行	在服务器上运行

从客户端调用的 RPC

Actor 所有权	未复制	NetMulticast	Server	Client
Owned by invoking client	在执行调用的客户端上运行	在执行调用的客户端上运行	在服务器上运行	在执行调用的客户端上运行
Owned by a different client	在执行调用的客户端上运行	在执行调用的客户端上运行	丢弃	在执行调用的客户端上运行
Server-owned actor	在执行调用的客户端上运行	在执行调用的客户端上运行	丢弃	在执行调用的客户端上运行
Unowned actor	在执行调用的客户端上运行	在执行调用的客户端上运行	丢弃	在执行调用的客户端上运行

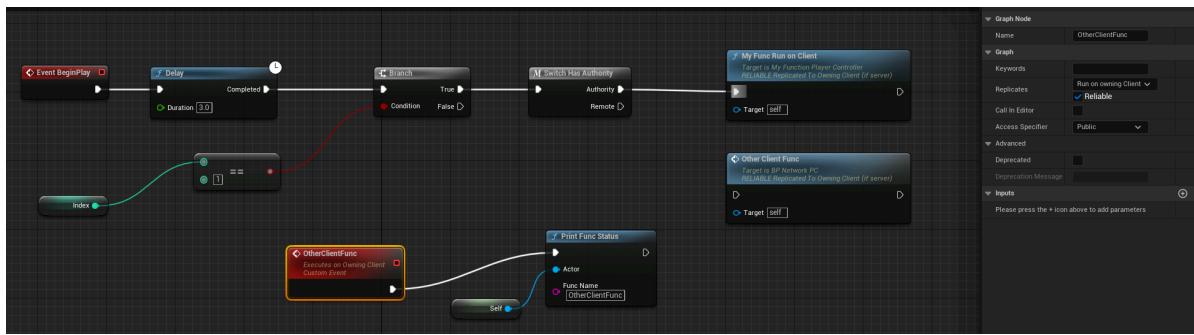
## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_PlayerController :public APlayerController
{
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintCallable, client, Reliable)
    void MyFunc_RunOnClient();

void AMyFunction_PlayerController::MyFunc_RunOnClient_Implementation()
{
    UInsiderLibrary::PrintFuncStatus(this,
    TEXT("MyFunc_RunOnClient_Implementation"));
}
```

Test setup: PIE mode, one ListenServer and two Clients



## Test Output Results:

```

MyFunc_Client_Implementation    BP_NetworkPC_C_0    NM_Client
Local:ROLE_AutonomousProxy   Remote:ROLE_Authority
OtherClientFunc  BP_NetworkPC_C_0    NM_Client    Local:ROLE_AutonomousProxy
                                         Remote:ROLE_Authority

```

It can be observed that the test code selects the second PC and initiates a Run on Client RPC call, which is successfully triggered on the Client. The function defined in C++ and the custom RunOnClient event added in the Blueprint have equivalent effects.

If this function is executed on a Server-owned Actor, it will only run on the Server and will not be propagated to the Client.

## Server

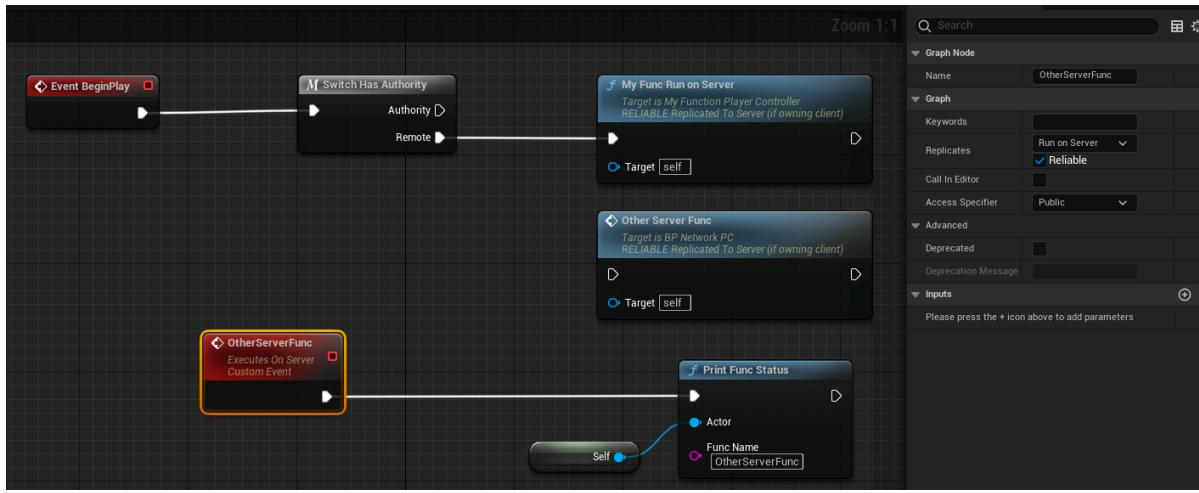
- **Function description:** Execute a RPC function on a client-owned Actor (PlayerController or Pawn). The function will only be executed on the server. The corresponding implementation function will be suffixed with "\_Implementation"
- **Metadata Type:** bool
- **Engine Module:** Network
- **Action Mechanism:** Add FUNC\_Net and FUNC\_NetServer to the Function Flags
- **Commonly Used:** ★★★★☆

Executes a RPC function on an Actor owned by the client (PlayerController or Pawn), which runs exclusively on the server. The corresponding implementation function will have the \_Implementation suffix appended.

Has the same effect as RunOnServer.

For the definition of Client-owned, refer to the documentation:

<https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/Networking/Actors/RPCs/>



## Test Code:

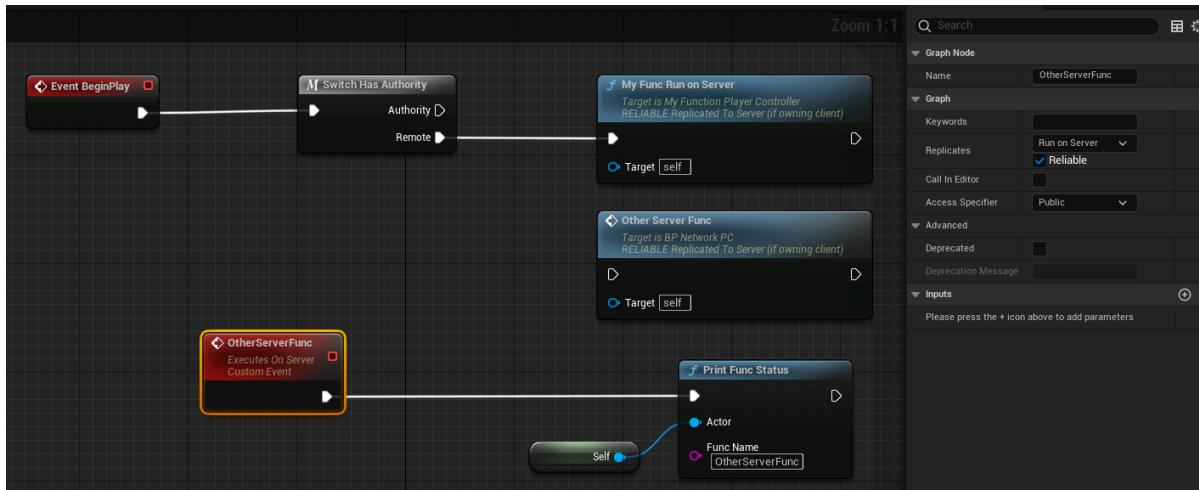
```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_PlayerController :public APlayerController
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, server, Reliable)
        void MyFunc_RunOnServer();
};

void AMyFunction_PlayerController::MyFunc_RunOnServer_Implementation()
{
    UInsiderLibrary::PrintFuncStatus(this,
    TEXT("MyFunc_RunOnServer_Implementation"));
}

```

Test Blueprint: PIE mode, with one ListenServer and two Clients



# Test Output Results:

```
LogInsider: Display: 5118b400    MyFunc_RunOnServer_Implementation
BP_NetworkPC_C_1      NM_ListenServer Local:ROLE_Authority
Remote:ROLE_AutonomousProxy
LogInsider: Display: 44ec3c00    MyFunc_RunOnServer_Implementation
BP_NetworkPC_C_2      NM_ListenServer Local:ROLE_Authority
Remote:ROLE_AutonomousProxy

LogInsider: Display: 49999000    OtherserverFunc BP_NetworkPC_C_1
NM_ListenServer Local:ROLE_Authority    Remote:ROLE_AutonomousProxy
LogInsider: Display: 4bcbd800    OtherserverFunc BP_NetworkPC_C_2
NM_ListenServer Local:ROLE_Authority    Remote:ROLE_AutonomousProxy
```

It can be observed that the test code takes the second PlayerController and initiates a Run on Server RPC call, which is successfully triggered on the Server. The function defined in C++ is equivalent in effect to the custom RunOnServer event added in the blueprint.

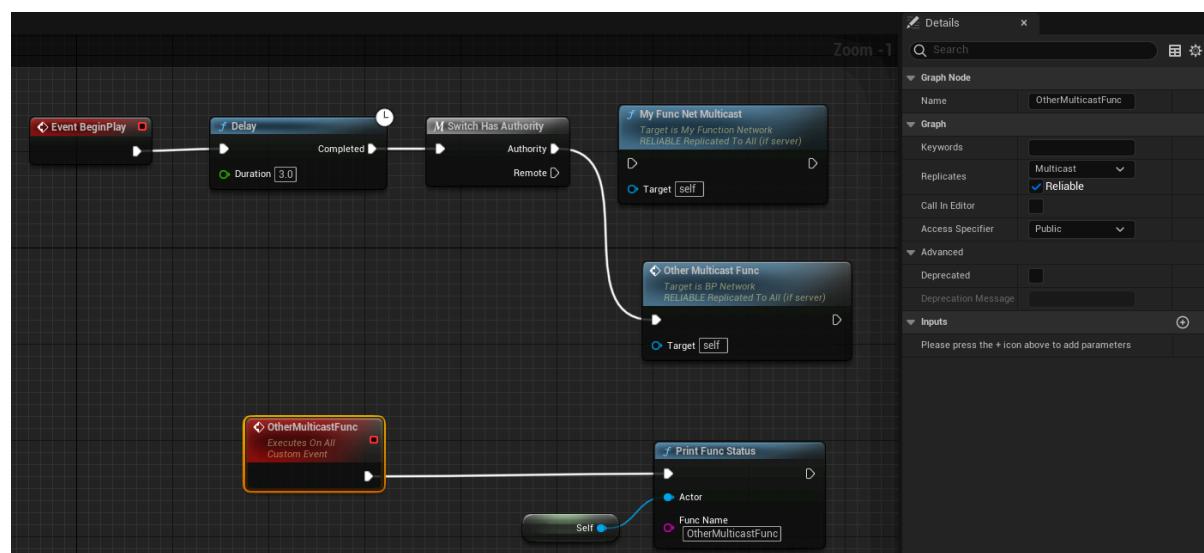
If this function is executed on a Server-owned Actor, it will only run on the server and will not be propagated to the clients.

## NetMulticast

- **Function Description:** Define a multicast RPC function that is executed on both the server and the client. The corresponding implementation function will have the \_Implementation suffix appended.
- **Metadata Type:** bool
- **Engine Module:** Network
- **Action Mechanism:** Add FUNC\_Net and FUNC\_NetMulticast to the Function Flags
- **Usage Frequency:** ★★★★☆

Define a multicast RPC function that is executed on both the server and the client. The corresponding implementation function will have the \_Implementation suffix appended.

RPC execution rules, refer to the documentation: <https://docs.unrealengine.com/4.27/zh-CN/InteractiveExperiences/Networking/Actors/RPCs/>



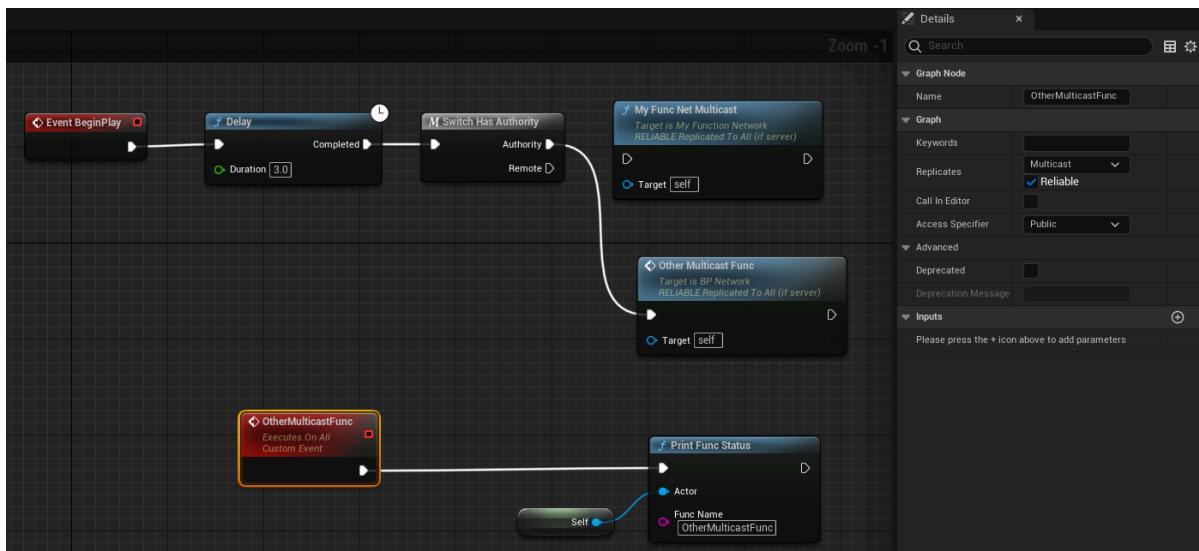
## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Network :public AActor
{
public:
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintCallable, NetMulticast, Reliable)
    void MyFunc_NetMulticast();
};

void AMyFunction_Network::MyFunc_NetMulticast_Implementation()
{
    UIInsiderLibrary::PrintFuncStatus(this,
    TEXT("MyFunc_NetMulticast_Implementation"));
}
```

Test Blueprint: PIE mode, with one ListenServer and two Clients



## Test Output Results:

```
LogInsider: Display: 46715a00      MyFunc_NetMulticast_Implementation
BP_Network_C_1  NM_ListenServer Local:ROLE_Authority
Remote:ROLE_SimulatedProxy
LogInsider: Display: 46e65000      MyFunc_NetMulticast_Implementation
BP_Network_C_1  NM_Client     Local:ROLE_SimulatedProxy  Remote:ROLE_Authority
LogInsider: Display: 29aaaa00      MyFunc_NetMulticast_Implementation
BP_Network_C_1  NM_Client     Local:ROLE_SimulatedProxy  Remote:ROLE_Authority

LogInsider: Display: 4ff44600      OtherMulticastFunc  BP_Network_C_1
NM_ListenServer Local:ROLE_Authority      Remote:ROLE_SimulatedProxy
LogInsider: Display: 3bf89b00      OtherMulticastFunc  BP_Network_C_1  NM_Client
Local:ROLE_SimulatedProxy      Remote:ROLE_Authority
LogInsider: Display: 29d68700      OtherMulticastFunc  BP_Network_C_1  NM_Client
Local:ROLE_SimulatedProxy      Remote:ROLE_Authority
```

On a Server-Owned Actor, initiating a Multicast RPC event call can be observed to be invoked on all three ends.

## Reliable

---

- **Function Description:** Designate a RPC function as "reliable," which will automatically retry upon encountering network errors to ensure delivery. Typically applied to functions critical to the logic.
- **Metadata Type:** boolean
- **Engine Module:** Network
- **Operation Mechanism:** Add FUNC\_NetReliable to the function flags in FunctionOptions
- **Common Usage:** ★★★★☆

Designate a RPC function as "reliable," which will automatically retry upon encountering network errors to ensure delivery. Typically applied to functions critical to the logic.

The underlying principle involves the logic of resending data packets.

## Unreliable

---

- **Function Description:** Designates an RPC function as "unreliable," meaning it will be abandoned in the event of a network error. Typically applied to functions where the communication effect is concerned, and it is acceptable if some calls are missed.
- **Metadata Type:** bool
- **Engine Module:** Network
- **Usage Frequency:** ★★★★☆

Designates an RPC function as "unreliable," which will be discarded if a network error occurs. This is usually used for functions that convey communication effects, and it is not critical if some instances are not processed.

## WithValidation

---

- **Function Description:** Specifies that a RPC function requires validation before execution, and it can only proceed if the validation is successful.
- **Metadata Type:** boolean
- **Engine Module:** Network
- **Action Mechanism:** Include FUNC\_NetValidate in the FunctionFlags
- **Common Usage:** ★★★★☆

A RPC function must be validated before execution, and it can only be executed if the validation passes.

WithValidation can actually be applied to the RPC functions of the Client, Server, and NetMulticast. However, it is most commonly employed on the Server, as it is typically the Server's data that holds the highest authority and is suitable for performing data validity checks.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_PlayerController :public APlayerController
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, Client, Reliable,withValidation)
    void MyFunc2_RunOnClient();

    UFUNCTION(BlueprintCallable, Server, Reliable,withValidation)
    void MyFunc2_RunOnServer();
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Network :public AActor
{
public:
    GENERATED_BODY()
    UFUNCTION(BlueprintCallable, NetMulticast, Reliable,withValidation)
    void MyFunc2_NetMulticast();
};

void AMyFunction_PlayerController::MyFunc2_RunOnServer_Implementation()
{
    UIInsiderLibrary::PrintFuncStatus(this,
TEXT("MyFunc2_RunOnServer_Implementation"));
}

bool AMyFunction_PlayerController::MyFunc2_RunOnServer_Validate()
{
    UIInsiderLibrary::PrintFuncStatus(this, TEXT("MyFunc2_RunOnServer_Validate"));
    return true;
}

bool AMyFunction_Network::MyFunc2_NetMulticast_Validate()
{
    UIInsiderLibrary::PrintFuncStatus(this,
TEXT("MyFunc2_NetMulticast_Validate"));
    return true;
}
```

## Test Results:

```
RunOnClient:
LogInsider: Display: 815f7800      MyFunc2_RunOnClient_Validate      BP_NetworkPC_C_0
            NM_Client   Local:ROLE_AutonomousProxy  Remote:ROLE_Authority
LogInsider: Display: 815f7800      MyFunc2_RunOnClient_Implementation
            BP_NetworkPC_C_0      NM_Client   Local:ROLE_AutonomousProxy
            Remote:ROLE_Authority

RunOnServer:
LogInsider: Display: 7fd11800      MyFunc2_RunOnServer_Validate      BP_NetworkPC_C_1
            NM_ListenServer Local:ROLE_Authority      Remote:ROLE_AutonomousProxy
```

```

LogInsider: Display: 7fd11800    MyFunc2_RunOnServer_Implementation
BP_NetworkPC_C_1      NM_ListenServer Local:ROLE_Authority
Remote:ROLE_AutonomousProxy

Multicast: ServerOwned
LogInsider: Display: 947e6400    MyFunc2_NetMulticast_Validate    BP_Network_C_1
NM_ListenServer Local:ROLE_Authority      Remote:ROLE_SimulatedProxy
LogInsider: Display: 947e6400    MyFunc2_NetMulticast_Implementation
BP_Network_C_1  NM_ListenServer Local:ROLE_Authority
Remote:ROLE_SimulatedProxy
LogInsider: Display: 8795eb00    MyFunc2_NetMulticast_Validate    BP_Network_C_1
NM_Client  Local:ROLE_SimulatedProxy      Remote:ROLE_Authority
LogInsider: Display: 8795eb00    MyFunc2_NetMulticast_Implementation
BP_Network_C_1  NM_Client  Local:ROLE_SimulatedProxy      Remote:ROLE_Authority
LogInsider: Display: 8f6a3700    MyFunc2_NetMulticast_Validate    BP_Network_C_1
NM_Client  Local:ROLE_SimulatedProxy      Remote:ROLE_Authority
LogInsider: Display: 8f6a3700    MyFunc2_NetMulticast_Implementation
BP_Network_C_1  NM_Client  Local:ROLE_SimulatedProxy      Remote:ROLE_Authority

```

## Principle:

When the WithValidation tag is included, during the generation of code by UHT, it will:

```

DEFINE_FUNCTION(AMyFunction_PlayerController::execMyFunc2_RunOnServer)
{
    P_FINISH;
    P_NATIVE_BEGIN;
    if (!P_THIS->MyFunc2_RunOnServer_Validate())
    {
        RPC_ValidateFailed(TEXT("MyFunc2_RunOnServer_Validate"));
        return;
    }
    P_THIS->MyFunc2_RunOnServer_Implementation();
    P_NATIVE_END;
}

DEFINE_FUNCTION(AMyFunction_PlayerController::execMyFunc2_RunOnClient)
{
    P_FINISH;
    P_NATIVE_BEGIN;
    if (!P_THIS->MyFunc2_RunOnClient_Validate())
    {
        RPC_ValidateFailed(TEXT("MyFunc2_RunOnClient_Validate"));
        return;
    }
    P_THIS->MyFunc2_RunOnClient_Implementation();
    P_NATIVE_END;
}

DEFINE_FUNCTION(AMyFunction_Network::execMyFunc2_NetMulticast)
{
    P_FINISH;
    P_NATIVE_BEGIN;
    if (!P_THIS->MyFunc2_NetMulticast_Validate())
    {

```

```

        RPC_ValidateFailed(TEXT("MyFunc2_NetMulticast_Validate"));
        return;
    }
P_THIS->MyFunc2_NetMulticast_Implementation();
P_NATIVE_END;
}

```

## ServiceRequest

- **Function Description:** This function serves as an RPC (Remote Procedure Call) service request. RPC service request
- **Metadata Type:** bool
- **Engine Module:** Network
- **Action Mechanism:** CustomThunk is added to Meta, and FUNC\_Net, FUNC\_Event, FUNC\_NetReliable, and FUNC\_NetRequest are added to FunctionFlags

No usage was found in the source code, only search results were obtained

```

UCLASS()
class
UTestReplicationStateDescriptor_TestFunctionwithNotReplicatedNonPODParameters :
public UObject
{
GENERATED_BODY()

protected:
    // Currently some features such as not replicating all parameters isn't
    // allowed on regular RPCs
    UFUNCTION(ServiceRequest(Iris))
    void FunctionWithNotReplicatedNonPODParameters(int Param0, bool Param1, int
Param2, UPARAM(NotReplicated) const
TArray<FTestReplicationStateDescriptor_TestStructWithRefCArray>&
NotReplicatedParam3);
    void FunctionWithNotReplicatedNonPODParameters_Implementation(int Param0,
bool Param1, int Param2, UPARAM(NotReplicated) const
TArray<FTestReplicationStateDescriptor_TestStructWithRefCArray>&
NotReplicatedParam3);
};

```

## UDN Response:

Alex: Those specifiers were added quite a while ago as a way to mark functions as RPC requests/responses to and from a backend service, the name of which would be given as part of the specifier: UFUNCTION(ServiceRequest()). However, the feature was never fully implemented, and since then the specifiers have only been used internally (and even then, I don't believe "ServiceResponse" is used at all anymore). This is why there isn't any public documentation or examples available, as they're not formally supported in the engine. You can check out ServiceRequestSpecifier and ServiceResponseSpecifier in UhtFunctionSpecifiers.cs to see how UHT handles these specifiers.

Mi: These two tags are used by us to freely expand our communication with our own servers (e.g., HTTP requests). For instance, we can provide our own NetDriver to handle the RPC for ServiceRequest with specific tags, serializing the corresponding parameters ourselves and sending them to our service.

"Does this mean that if the engine's default implementation is used, these two tags are ineffective? When I attempt to initiate a call to a ufunction with the ServiceRequest tag on the server or client, an error log is printed."

Yes, the default NetDriver for UE client and DS communication does not require these two keywords. Using them will result in the corresponding NetDriver implementation not being found.

An error occurs when calling on a Server-Owned Actor: LogNet: Warning:  
UNetDriver::ProcessRemoteFunction: No owning connection for actor BP\_Network\_C\_1. Function MyFunc\_ServiceRequest will not be processed.

An error also occurs when the server calls on a PC:

```
LogRep: Error: Rejected RPC function due to access rights. Object: BP_NetworkPC_C  
/Game/UEDPIE_0_StartMap.StartMap:PersistentLevel.BP_NetworkPC_C_1, Function:  
MyFunc_ServiceRequest  
LogNet: Error: UActorChannel::ProcessBunch: Replicator.ReceivedBunch failed. Closing  
connection. RepObj: BP_NetworkPC_C  
/Game/UEDPIE_0_StartMap.StartMap:PersistentLevel.BP_NetworkPC_C_1, Channel: 3
```

## ServiceResponse

---

- **Function Description:** This function serves as the response for the RPC service. RPC service reply
- **Metadata Type:** Boolean
- **Engine Module:** Network
- **Action Mechanism:** FUNC\_Net, FUNC\_Event, FUNC\_NetReliable, and FUNC\_NetResponse are added to the FunctionFlags

No instances of usage have been observed in the source code.

## BlueprintInternalUseOnly

---

- **Function Description:** Indicates that this function should not be disclosed to end users. It is called internally within blueprints and not exposed to users.
- **Metadata Type:** bool
- **Engine Module:** Blueprint, UHT
- **Functionality Mechanism:** Adds BlueprintInternalUseOnly and BlueprintType to the Meta section
- **Commonality:** ★★★

Indicates that this function should not be disclosed to end users. It is called internally within blueprints and not exposed to users.

Equivalent to setting BlueprintInternalUseOnly = true in the meta. By default, functions marked as BlueprintCallable/Pure generate UK2Node\_CallFunction for invocation. However, BlueprintInternalUseOnly prevents this from occurring.

There are two typical use cases:

Firstly, to hide the function within blueprints, while still allowing it to be invoked reflectively by name due to its UFUNCTION attribute. Although this usage is rare, it is still considered a valid application.

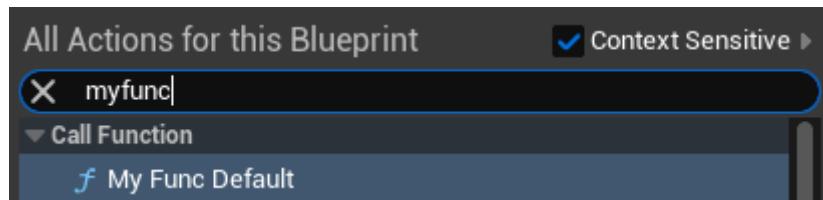
Secondly, the engine may declare another blueprint function node elsewhere for this function, following specific rules, which necessitates hiding the one created by default rules.

## Sample Code 1:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Internal :public AActor
{
public:
    GENERATED_BODY()
public:
    //BlueprintInternalUseOnly = true, BlueprintType = true, ModuleRelativePath
    = Function/MyFunction_Internal.h
    //FunctionFlags: FUNC_Final | FUNC_Native | FUNC_Public |
    FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable, BlueprintInternalUseOnly)
    void MyFunc_InternalOnly() {}

    //FunctionFlags: FUNC_Final | FUNC_Native | FUNC_Public |
    FUNC_BlueprintCallable
    UFUNCTION(BlueprintCallable)
    void MyFunc_Default() {}
};
```

Only MyFunc\_Default is callable within blueprints. This implies that the function is still exposed to blueprints but is hidden from direct user invocation, yet can be called indirectly through function name lookup in the code.



An example is found in the source code, where the GetLevelScriptActor function is not callable from blueprints but can be located by name, facilitating the generation of a UFunction to be injected elsewhere as a callback

```
ULevelStreaming:
UFUNCTION(BlueprintPure, meta = (BlueprintInternalUseOnly = "true"))
ENGINE_API ALevelScriptActor* GetLevelScriptActor();
```

然后发现：

```
GetLevelScriptActorNode->SetFromFunction(ULevelStreaming::StaticClass()-
>FindFunctionByName(GET_FUNCTION_NAME_CHECKED(ULevelStreaming,
GetLevelScriptActor)));
```

## Sample Code 2:

The implementation code is not provided here; you can refer to it in the project.

```
UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy =
MyAsyncObject, HasDedicatedAsyncNode))
class INSIDER_API UMyFunction_Async :public UCancellableAsyncAction
{
public:
    GENERATED_BODY()

public:
    UPROPERTY(BlueprintAssignable)
    FDelayOutputPin Loop;

    UPROPERTY(BlueprintAssignable)
    FDelayOutputPin Complete;

    UFUNCTION(BlueprintCallable, meta = (BlueprintInternalUseOnly = "true",
WorldContext = "WorldContextObject"), Category = "Flow Control")
        static UMyFunction_Async* DelayLoop(const UObject* WorldContextObject, const
float DelayInSeconds, const int Iterations);

    virtual void Activate() override;

    UFUNCTION()
    static void Test();

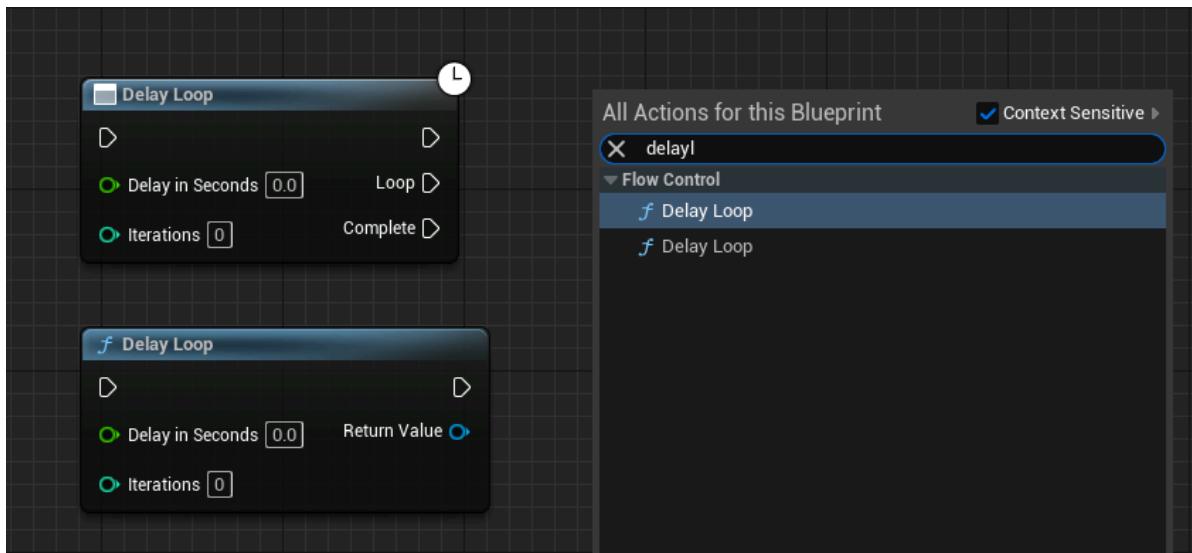
private:
    const UObject* WorldContextObject = nullptr;
    float MyDelay = 0.f;
    int MyIterations = 0;
    bool Active = false;

    UFUNCTION()
    void ExecuteLoop();

    UFUNCTION()
    void ExecuteComplete();
};
```

## Example Effect:

If the BlueprintInternalUseOnly attribute is commented out in the source code, two DelayLoop nodes will appear in the blueprint. The first one is generated according to UBlueprintAsyncActionBase rules, and the second one is generated according to regular blueprint function rules. Clearly, we do not want to present both to the user, causing confusion. Thus, BlueprintInternalUseOnly must be added to prevent the creation of the default blueprint node.



## Principle:

Regarding the use of UBlueprintAsyncActionBase, the function implementation in UK2Node\_BaseAsyncTask reflects the rules for writing classes derived from UBlueprintAsyncActionBase. In essence, it uses a static function as a factory function and analyzes the Delegate property of this Proxy class as a Pin.

If BlueprintInternalUseOnly is not set to "true," two functions will be generated. The one below is the generation of an ordinary static function, while the one above is generated by analyzing UBlueprintAsyncActionBase.

Among them, the process of identifying static function as FactoryFunction in UBlueprintAsyncActionBase is, BlueprintActionDatabaseImpl::GetNodeSpecificActions will trigger UK2Node\_AsyncAction::GetMenuActions , so ActionRegistrar.RegisterClassFactoryActions , the internal judgment RegisterClassFactoryActions\_Utils::IsFactoryMethod ( Function UBlueprintAsyncActionBase ) will pass (the judgment is a static function, and the return type is a subclass object of UBlueprintAsyncActionBase ), and then continue Create a factory method of nodeSpawner via callback UBlueprintFunctionNodeSpawner::Create ( FactoryFunc ); .

In summary, at this point, BlueprintInternalUseOnly serves to hide the node generated by default.

## CustomThunk

- **Function Description:** Instruct UHT not to generate an auxiliary function for blueprint invocation for this function, instead requiring the user to define it manually.
- **Metadata Type:** bool
- **Engine Module:** UHT
- **Functionality Mechanism:** Include CustomThunk in the Meta section
- **Common Usage:** ★★★

Specifies that UHT should not generate an auxiliary function for blueprint invocation for this function, necessitating user-defined implementation.

Thunk refers to a function analogous to execFoo, which requires user-defined implementation.

"CustomThunk" is typically employed in scenarios where function parameters are indeterminate, such as with various wildcard characters, or when a more nuanced, custom logic processing is necessary.

## Test Code:

```
UFUNCTION(BlueprintPure, CustomThunk)
static int32 MyFunc_CustomDivide(int32 A, int32 B = 1);

DECLARE_FUNCTION(execMyFunc_CustomDivide);

int32 UMyFunction_Custom::MyFunc_CustomDivide(int32 A, int32 B /*= 1*/)
{
    return 1;
}

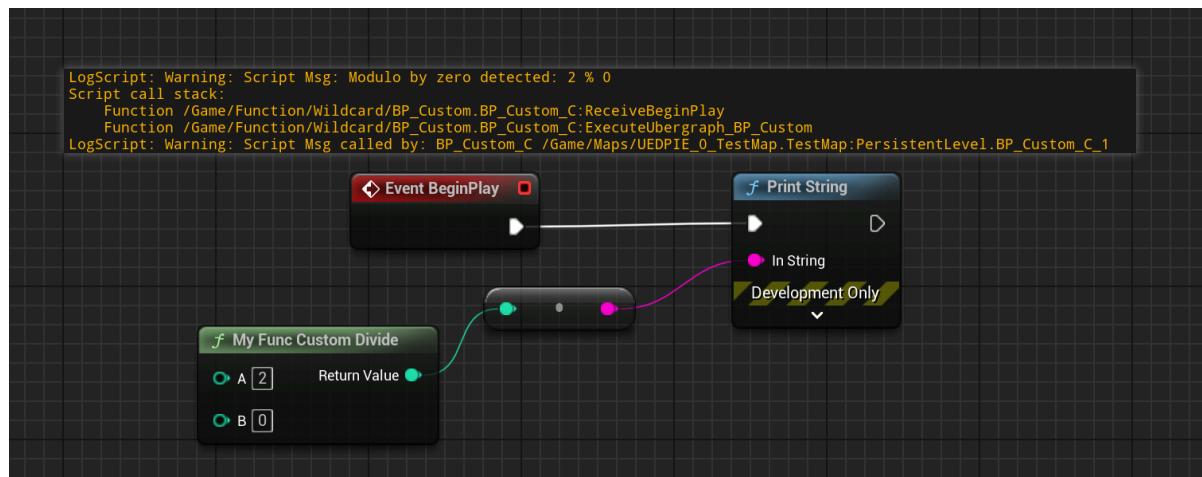
DEFINE_FUNCTION(UMyFunction_Custom::execMyFunc_CustomDivide)
{
    P_GET_PROPERTY(FIntProperty, A);
    P_GET_PROPERTY(FIntProperty, B);

    P_FINISH;

    if (B == 0)
    {
        FFrame::KismetExecutionMessage(*FString::Printf(TEXT("Modulo by zero
detected: %d %% 0\n%s"), A, *Stack.GetStackTrace()), ELogVerbosity::Warning);
        *(int32*)RESULT_PARAM = 0;
        return;
    }

    *(int32*)RESULT_PARAM = A/B;
}
```

## Blueprint Outcome:



It can be observed that even with a division by zero, a custom error message can be defined.

Significantly, examining .gen.cpp reveals that the execFoo function is no longer generated internally.

# Variadic

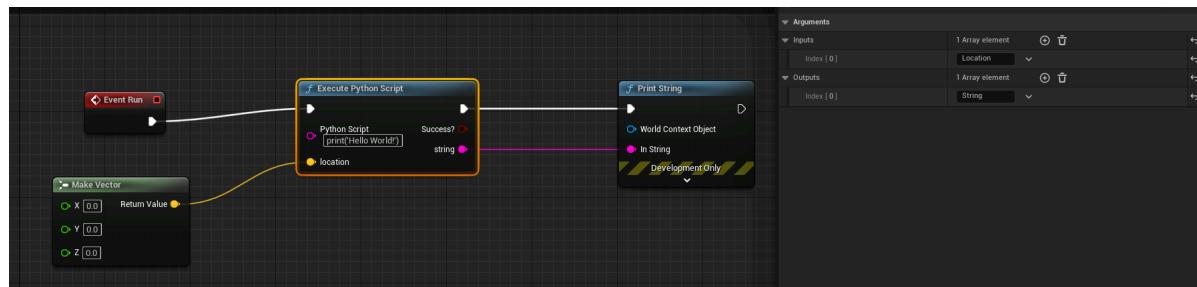
- **Function description:** Indicates that a function can accept multiple parameters of any type (including input/output)
- **Metadata type:** bool
- **Engine module:** Blueprint, UHT
- **Action mechanism:** Include Variadic in the Meta
- **Commonly used:** ★★★

Indicates that a function can accept multiple parameters of any type (including input/output)

Search for application in the source code: then coordinate with UK2Node\_ExecutePythonScript

```
UFUNCTION(BlueprintCallable, CustomThunk, Category = "Python|Execution", meta=(Variadic, BlueprintInternalUseOnly="true"))
    static bool ExecutePythonScript(UPARAM(meta=(MultiLine=True)) const FString& PythonScript, const TArray< FString>& PythonInputs, const TArray< FString>& PythonOutputs);
DECLARE_FUNCTION(execExecutePythonScript);
```

Blueprint effect:



## Example code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Variadic : public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    /*
        [PrintVariadicFields Function->Struct->Field->Object
        /Script/Insider.MyFunction_Variadic:PrintVariadicFields]
        (BlueprintInternalUseOnly = true, BlueprintType = true, CustomThunk =
        true, ModuleRelativePath = Function/Variadic/MyFunction_Variadic.h, Variadic = )
    */
    UFUNCTION(BlueprintCallable, CustomThunk, BlueprintInternalUseOnly, meta =
    (Variadic))
    static FString PrintVariadicFields(const TArray< FString>& Inputs, const
    TArray< FString>& Outputs);
    DECLARE_FUNCTION(execPrintVariadicFields);
};
```

```

FString UMyFunction_Variadic::PrintVariadicFields(const TArray<FString>& Inputs,
const TArray<FString>& Outputs)
{
    check(0);
    return TEXT("");
}

DEFINE_FUNCTION(UMyFunction_Variadic::execPrintVariadicFields)
{
    FString str;

    P_GET_TARRAY_REF(FString, Inputs);
    P_GET_TARRAY_REF(FString, Outputs);

    for (const FString& PythonInput : Inputs)
    {
        Stack.MostRecentPropertyAddress = nullptr;
        Stack.MostRecentProperty = nullptr;
        Stack.StepCompiledIn<FProperty>(nullptr);
        check(Stack.MostRecentProperty && Stack.MostRecentPropertyAddress);

        FProperty* p = CastField<FProperty>(Stack.MostRecentProperty);

        FString PropertyValueString;
        const void* PropertyValuePtr = p->ContainerPtrToValuePtr<const void*>
        (Stack.MostRecentPropertyContainer);

        p->ExportTextItem_Direct(PropertyValueString, PropertyValuePtr, nullptr,
        nullptr, PPF_None);

        str += FString::Printf(TEXT("%s:%s\n"), *p->GetFName().ToString(),
        *PropertyValueString);
    }
    P_FINISH;

    *(FString*)RESULT_PARAM = str;
}

```

## Example effect:



Print:

CallFunc\_MakeVector\_ReturnValue:(X=1.000000,Y=2.000000,Z=3.000000)  
 CallFunc\_MakeLiteralDouble\_ReturnValue:456.000000

# Principle:

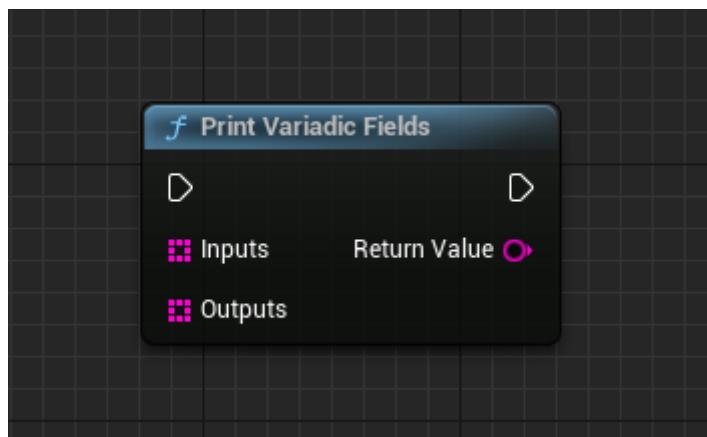
Standard CustomThunk functions have limitations; parameter names and counts are fixed in the UFunction and do not support dynamic counts.

Currently, to use the **Variadic** feature, it is necessary to create a custom blueprint node using C++ to add pins to **K2Node\_CallFunction**.

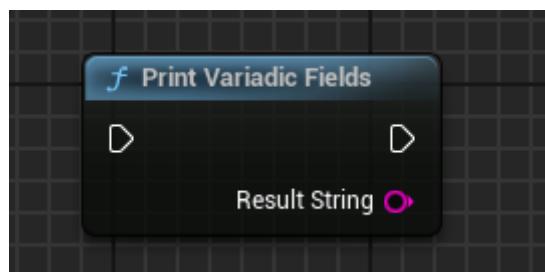
presumably to develop simultaneously to achieve **K2Node** and the corresponding **CustomThunk + Variadic** methods, ensuring safe usage.

BlueprintInternalUseOnly must also be included; otherwise, an ordinary blueprint function will be automatically generated, and the desired variadic effect will not be realized.

The following is the version automatically generated without the inclusion of BlueprintInternalUseOnly:



Actually, it should be: Then manually add the parameters.



Distinct from Wildcard, which allows parameters of any type but with a fixed number, BlueprintInternalUseOnly ensures that the number of parameters is variable



Officially added feature for interaction with **Python**: [Added a Blueprint node for calling Python with arguments](#)

Official submission:

<https://github.com/EpicGames/UnrealEngine/commit/61d0f65e1cded45ed94f0422eb931f44688e972>

## Notes:

Implemented variadic function support for Blueprints

Variadic functions are required to be a CustomThunk marked with the "Variadic" meta-data. They can then be used from a custom Blueprint node to accept arbitrary arguments at the end of their parameter list (any extra pins added to the node that aren't part of the main function definition will become the variadic payload).

Variadic arguments aren't type checked, so you need other function input to tell you how many to expect, and for a nativized function, also what type of arguments you're dealing with.

#jira UE-84932

#rb Dan.OConnor

[CL 10421401 by Jamie Dale in Dev-Editor branch]`

## FieldNotify

- **Function Description:** Establish a binding point for FieldNotify for this function.
- **Metadata Type:** bool
- **Engine Module:** UHT
- **Restriction Type:** Functions within the ViewModel
- **Common Usage:** ★★★

Establish a binding point for FieldNotify for this function.

It should be noted that if it is a Get function, when its return value changes, the event needs to be manually broadcast in other places that trigger the change. As the following code UE\_MVVM\_BROADCAST\_FIELD\_VALUE\_CHANGED ( GetHPPPercent ) ; does.

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyViewModel :public UMVVMViewModelBase
{
    GENERATED_BODY()
protected:
    UPROPERTY(BlueprintReadWrite, FieldNotify, Getter, Setter, BlueprintSetter = SetHP)
        float HP = 1.f;

    UPROPERTY(BlueprintReadWrite, FieldNotify, Getter, Setter, BlueprintSetter = SetMaxHP)
        float MaxHP = 100.f;
public:
    float GetHP()const { return HP; }
    UFUNCTION(BlueprintSetter)
    void SetHP(float val)
    {
        if (UE_MVVM_SET_PROPERTY_VALUE(HP, val))
    }
}
```

```

    {
        UE_MVVM_BROADCAST_FIELD_VALUE_CHANGED(GetHPPPercent);
    }
}

float GetMaxHP() const { return MaxHP; }
UFUNCTION(BlueprintSetter)
void SetMaxHP(float val)
{
    if (UE_MVVM_SET_PROPERTY_VALUE(MaxHP, val))
    {
        UE_MVVM_BROADCAST_FIELD_VALUE_CHANGED(GetHPPPercent);
    }
}

//You need to manually notify that GetHealthPercent changed when
CurrentHealth or MaxHealth changed.
UFUNCTION(BlueprintPure, FieldNotify)
float GetHPPPercent() const
{
    return (MaxHP != 0.f) ? HP / MaxHP : 0.f;
}
;

```

## Test Results:

It can be seen that GetHPPPercent generates a FIELD.

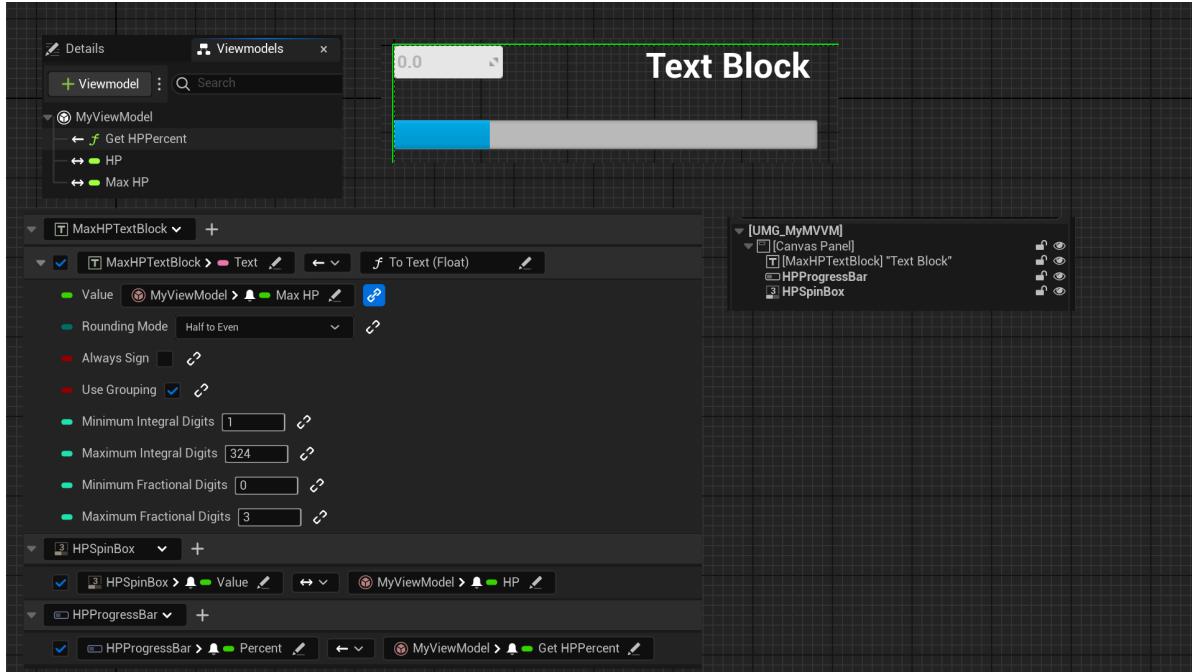
```

//MyViewModel.generated.h
#define
FID_Gitworkspace_Hello_Source_Insider_Property_MVVM_MyViewModel_h_12_FIELDNOTIFY \
\
UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_BEGIN(INSIDER_API ) \
UE_FIELD_NOTIFICATION_DECLARE_FIELD(HP) \
UE_FIELD_NOTIFICATION_DECLARE_FIELD(MaxHP) \
UE_FIELD_NOTIFICATION_DECLARE_FIELD(GetHPPPercent) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_BEGIN(HP) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(MaxHP) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(GetHPPPercent) \
UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_END() \
UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_END(); \
//MyViewModel.gen.cpp
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, HP)
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, MaxHP)
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, GetHPPPercent)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_BEGIN(UMyViewModel)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_ENUM_FIELD(UMyViewModel, HP)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_ENUM_FIELD(UMyViewModel, MaxHP)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_ENUM_FIELD(UMyViewModel, GetHPPPercent)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_END(UMyViewModel);

```

## Blueprint Effect:

Progress bars can be bound to GetHPPPercent.



## DisplayName

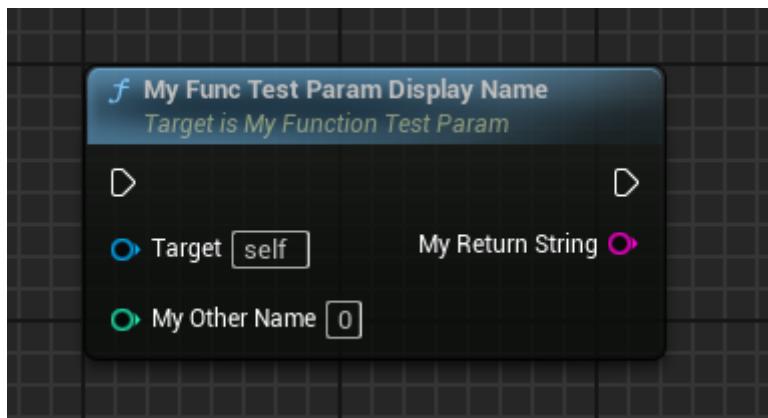
- **Function Description:** Change the display name of function parameters on blueprint nodes
- **Metadata Type:** string = "abc"
- **Engine Module:** Blueprint, Parameter
- **Action Mechanism:** Include DisplayName in the Meta data
- **Common Usage:** ★★★★☆

Note: The UPARAM can also be employed for return values, with the default being the ReturnValue.

## Test Code:

```
//(DisplayName = My Other Name)
UFUNCTION(BlueprintCallable)
UPARAM(DisplayName = "My Return String") FString
MyFuncTestParam_DisplayName(UPARAM(DisplayName = "My Other Name") int value);
```

## Blueprint Node:



## ref

- **Function Description:** Allows function parameters to be passed by reference
- **Metadata Type:** bool
- **Engine Module:** Blueprint, Parameter
- **Action Mechanism:** Add CPF\_ReferenceParm to PropertyFlags
- **Common Usage:** ★★★★☆

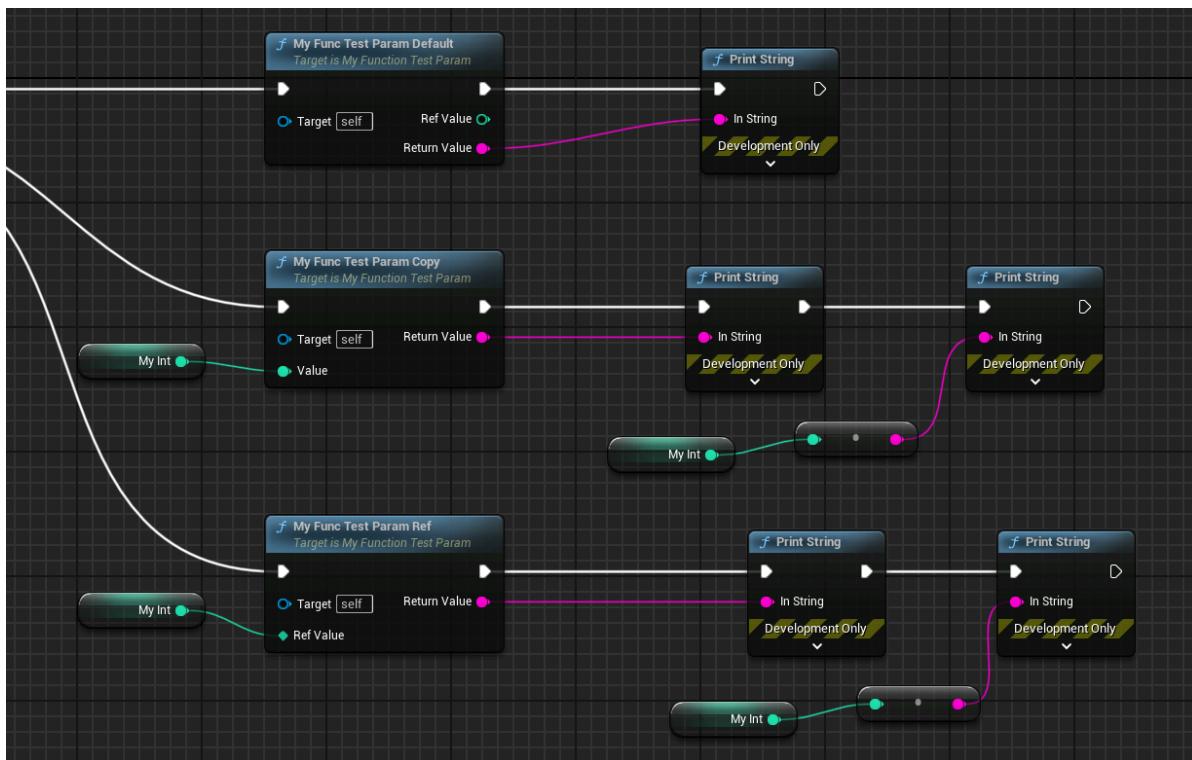
The distinction between regular parameters and reference parameters is that, when accessing parameters, the Ref type directly obtains a reference to the actual argument rather than a copy. This avoids the need for copying and allows modifications to be preserved.

A simple & parameter will be interpreted as an output return parameter; therefore, it must be further designated with ref.

## Test Code:

```
//PropertyFlags: CPF_Parm | CPF_OutParm | CPF_ZeroConstructor |  
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |  
CPF_NativeAccessSpecifierPublic  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_Default(int& refValue);  
  
//PropertyFlags: CPF_Parm | CPF_OutParm | CPF_ZeroConstructor |  
CPF_ReferenceParm | CPF_IsPlainOldData | CPF_NoDestructor |  
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_Ref(UPARAM(ref) int& refValue);  
  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_Copy(int value);
```

# Blueprint Code:



## Principle:

Reference parameters will be retrieved using P\_GET\_PROPERTY\_REF during UHT generation

```
#define P_GET_PROPERTY(PropertyType, ParamName)
\
    PropertyType::TCppType ParamName = PropertyType::GetDefaultValue();
\
    Stack.StepCompiledIn<PropertyType>(&ParamName);

#define P_GET_PROPERTY_REF(PropertyType, ParamName)
\
    PropertyType::TCppType ParamName##Temp =
    PropertyType::GetDefaultValue(); \
    PropertyType::TCppType& ParamName = Stack.StepCompiledInRef<PropertyType,
    PropertyType::TCppType>(&ParamName##Temp);
```

## Const

- **Function Description:** Specifies that the function parameters are immutable
- **Metadata Type:** bool
- **Engine Module:** Blueprint, Parameter
- **Action Mechanism:** Add CPF\_ConstParm to PropertyFlags and NativeConst to Meta
- **Commonality:** ★

Specifies that the function parameters are not modifiable.

If "const" is directly added to the parameters in the C++ code, it will be automatically recognized by UHT and the CPF\_ConstParm flag, as well as NativeConst metadata, will be added. However, you can also manually add UPARAM(const) to force UHT to add CPF\_ConstParm, as seen in the Out node in the blueprint below, which converts the output parameter into an input parameter.

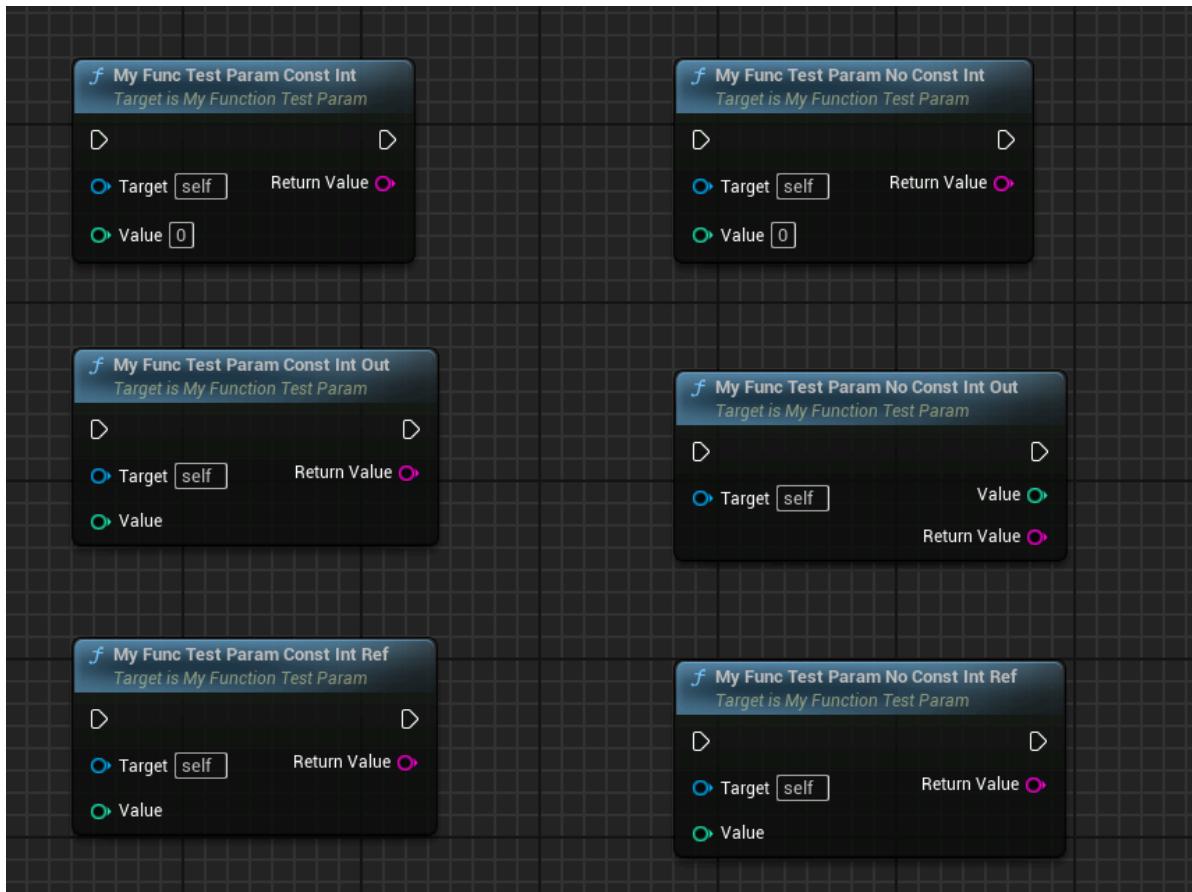
Although it is unclear when manual addition is necessary, no practical examples were found in the source code. The imagined use case is to make it a const input parameter at the blueprint level, while still being a mutable reference parameter in C++, facilitating the invocation of some non-const methods in C++.

## Test Code:

```
//PropertyFlags:    CPF_ConstParm | CPF_Parm | CPF_ZeroConstructor |  
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |  
CPF_NativeAccessSpecifierPublic  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_ConstInt(UPARAM(const) int value);  
  
//PropertyFlags:    CPF_ConstParm | CPF_Parm | CPF_OutParm |  
CPF_ZeroConstructor | CPF_ReferenceParm | CPF_IsPlainOldData | CPF_NoDestructor |  
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_ConstIntOut(UPARAM(const) int& value);  
  
//(NativeConst = )  
//PropertyFlags:    CPF_ConstParm | CPF_Parm | CPF_OutParm |  
CPF_ZeroConstructor | CPF_ReferenceParm | CPF_IsPlainOldData | CPF_NoDestructor |  
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_ConstIntRef(UPARAM(const) const int& value);  
  
//PropertyFlags:    CPF_Parm | CPF_ZeroConstructor | CPF_IsPlainOldData |  
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_NoConstInt(int value);  
  
//PropertyFlags:    CPF_Parm | CPF_OutParm | CPF_ZeroConstructor |  
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |  
CPF_NativeAccessSpecifierPublic  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_NoConstIntOut(int& value);  
  
//(NativeConst = )  
//PropertyFlags:    CPF_ConstParm | CPF_Parm | CPF_OutParm |  
CPF_ZeroConstructor | CPF_ReferenceParm | CPF_IsPlainOldData | CPF_NoDestructor |  
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic  
UFUNCTION(BlueprintCallable)  
FString MyFuncTestParam_NoConstIntRef(const int& value);
```

## Blueprint Node:

The output Value of MyFuncTestParam\_ConstIntOut has become an input Value because it cannot be altered.



## Principle Code:

When "const" is actually present in the code, the CPF\_ConstParam flag is added.

```
\Engine\Source\Programs\Shared\EpicGames.UHT\Parsers\UhtPropertyParser.cs 1030

if (propertySettings.PropertyCategory != UhtPropertyCategory.Member &&
!isTemplateArgument)
{
    // const before the variable type support (only for params)
    if (tokenReader.TryOptional("const"))
    {
        propertySettings.PropertyFlags |= EPropertyFlags.ConstParm;
        propertySettings.MetaData.Add(UhtNames.NativeConst, "");
    }
}
```

## Required

- **Function Description:** The parameter node for the specified function must be connected to supply a value
- **Metadata Type:** bool
- **Engine Module:** Blueprint, Parameter

- **Action Mechanism:** Include CPF\_RequiredParm in PropertyFlags
- **Common Usage:** ★★

The parameter node of the specified function must be connected to a variable to supply a value.

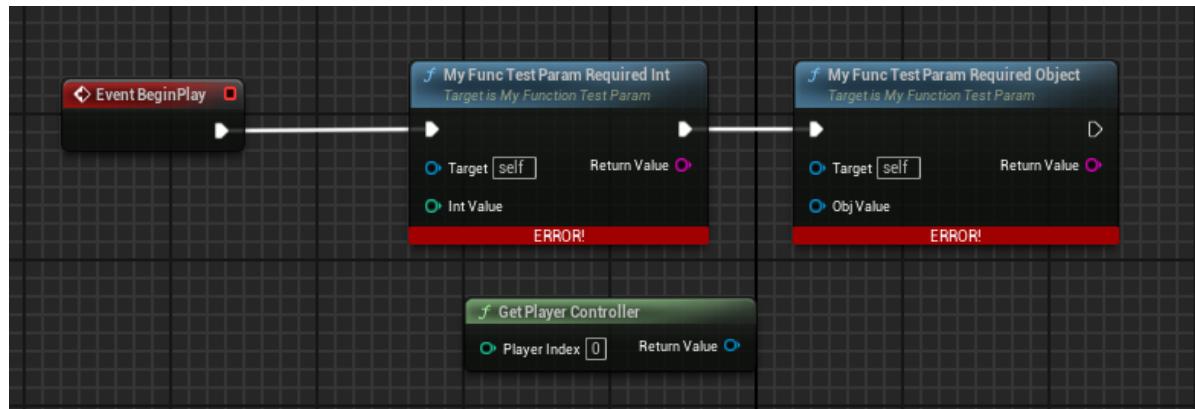
If a default value is provided for the parameter, this flag will still disregard the default value, treating it as if no value has been provided. A variable must still be connected.

## Test Code:

```
//PropertyFlags:    CPF_Parm | CPF_ZeroConstructor | CPF_RequiredParm |
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UFUNCTION(BlueprintCallable)
FString MyFuncTestParam_RequiredObject(UPARAM(Required) UObject* objValue);

//(CPP_Default_intValue = 123, ModuleRelativePath =
Function/Param/MyFunction_TestParam.h)
//PropertyFlags:    CPF_Parm | CPF_ZeroConstructor | CPF_RequiredParm |
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
CPF_NativeAccessSpecifierPublic
UFUNCTION(BlueprintCallable)
FString MyFuncTestParam_RequiredInt(UPARAM(Required) int intValue=123);
```

## Blueprint Node:



If a node is not connected, an error will be reported during compilation

Pin Int Value must be linked to another node (in My Func Test Param Required Int )

Pin Obj Value must be linked to another node (in My Func Test Param Required Object )

## Principle:

The judgment is based on this marker.

```
const bool bIsRequiredParam = Param->HasAnyPropertyFlags(CPF_RequiredParm);
// Don't let the user edit the default value if the parameter is required to
be explicit.
Pin->bDefaultValueIsIgnored |= bIsRequiredParam;
```

# NotReplicated

- **Engine modules:** Blueprint, Network, Parameter
- **Mechanism of action:** Add CPF\_RepSkip to PropertyFlags

Referring to UFUNCTION's ServiceRequest, this specifier is deprecated.

"Only parameters within service request functions can be marked as NotReplicated."

```
if (context.PropertySettings.PropertyCategory ==
UhtPropertyCategory.ReplicatedParameter)
{
    context.PropertySettings.PropertyCategory =
UhtPropertyCategory.RegularParameter;
    context.PropertySettings.PropertyFlags |= EPropertyFlags.RepSkip;
}
else
{
    context.MessageSite.LogError("Only parameters in service request
functions can be marked NotReplicated");
}
```

"The source code only has knowledge of this."

```
// Currently some features such as not replicating all parameters isn't allowed
on regular RPCs
UFUNCTION(ServiceRequest(Iris))
void FunctionWithNotReplicatedNonPODParameters(int Param0, bool Param1, int
Param2, UPARAM(NotReplicated) const
TArray<FTestReplicationStateDescriptor_TestStructWithRefCArray>&
NotReplicatedParam3);
void FunctionWithNotReplicatedNonPODParameters_Implementation(int Param0, bool
Param1, int Param2, UPARAM(NotReplicated) const
TArray<FTestReplicationStateDescriptor_TestStructWithRefCArray>&
NotReplicatedParam3);
```

# Export

- **Function description:** Determines that when exporting an Asset, the object of this class should export its internal attribute values rather than the object's path.
- **Metadata type:** bool
- **Engine module:** Serialization
- **Restriction type:** Object property, or an array of Objects
- **Mechanism of action:** Include CPF\_ExportObject in PropertyFlags
- **Frequency of use:** ★

When exporting an Asset, it is determined that the object of this class should export its internal attribute values rather than the object's path.

- When an Object is copied (e.g., during copy/paste operations), the Object assigned to this property should be exported as a complete sub-Object block (as seen in the examples below, this actually means also exporting the values of the internal attributes), rather than just exporting the Object reference itself.
- Applies exclusively to Object properties (or arrays of Objects), as it is used for object exports.
- Effectively, it's the difference between shallow and deep copying. Without the Export mark, it's a shallow copy, only outputting the object path. With the Export mark, it's a deep copy, also outputting the object's internal attributes.

## Sample Code:

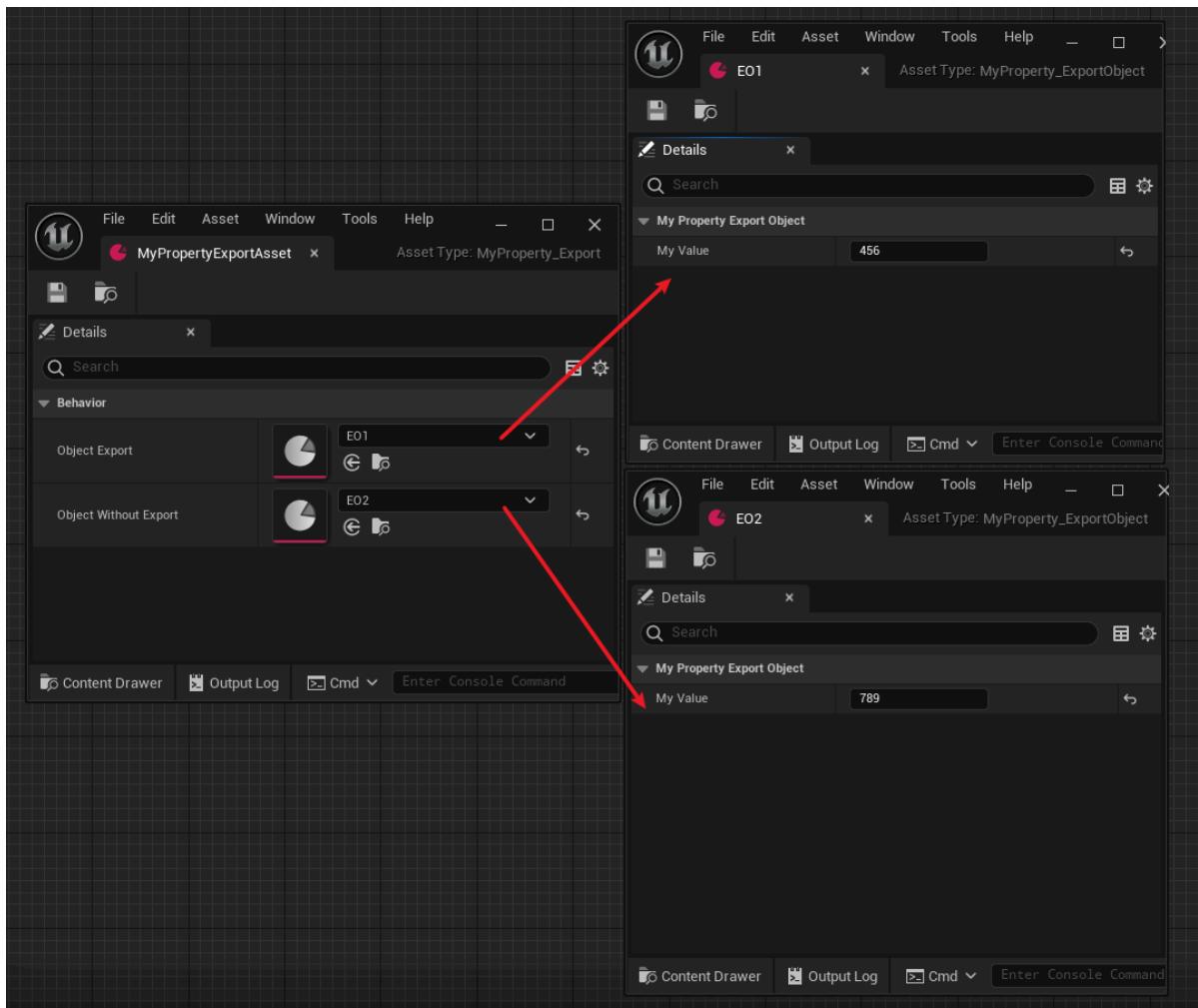
```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_ExportObject :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyValue = 123;
};

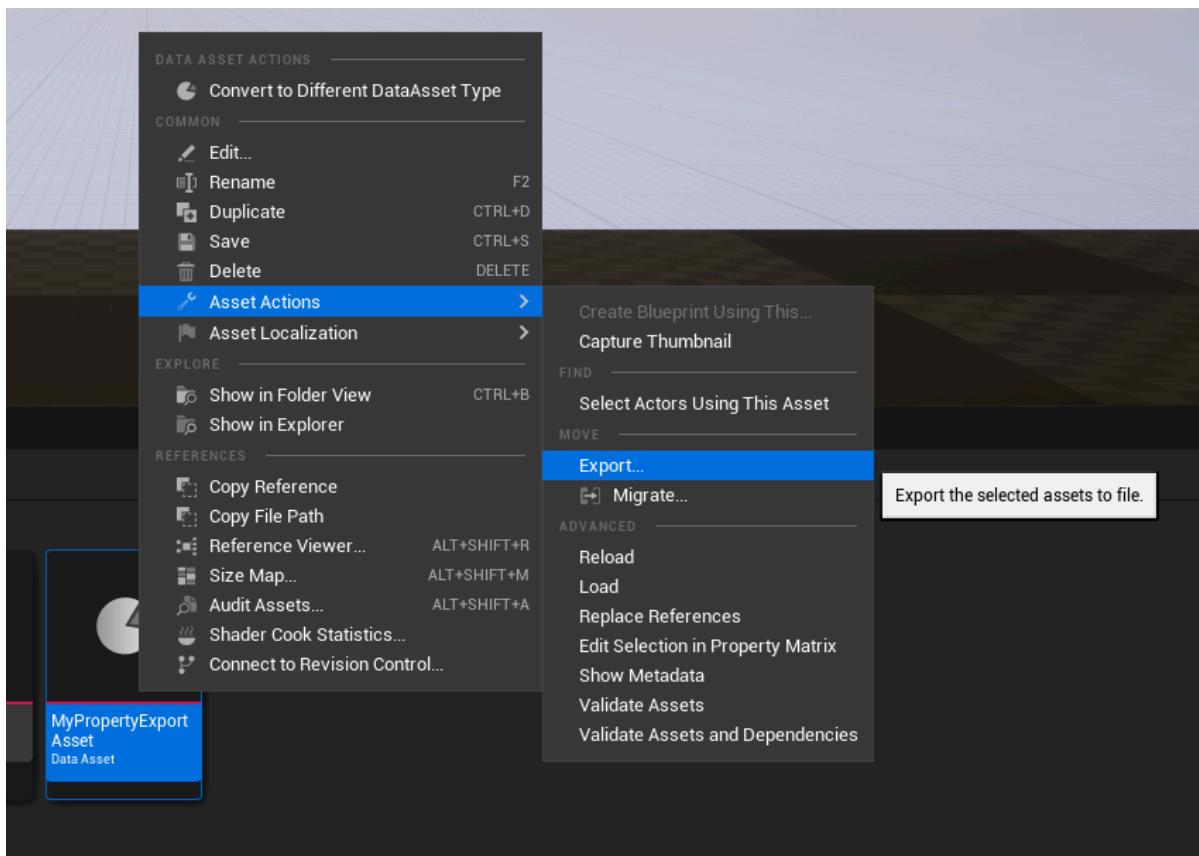
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Export :public UDataAsset
{
public:
public:
    GENERATED_BODY()
    UMyProperty_Export(const FObjectInitializer& ObjectInitializer =
FObjectInitializer::Get());
public:
    //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ExportObject |
CPF_ZeroConstructor | CPF_NoDestructor | CPF_HasGetValueTypeHash |
CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Export, Category = Behavior)
        UMyProperty_ExportObject* ObjectExport;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Behavior)
        UMyProperty_ExportObject* ObjectwithoutExport;
};

```

Configured Object Value:



Primarily used during export operations to determine how to export the content of an Object\* attribute. NoExport will only output the object reference path, while Export will output the internal attribute values of the object.



Exported Text:

```

Begin Object Class=/Script/Insider.MyProperty_Export Name="MyPropertyExportAsset"
ExportPath=/Script/Insider.MyProperty_Export"/Game/Property/MyPropertyExportAsset.MyPropertyExportAsset"
    Begin Object Class=/Script/Insider.MyProperty_ExportObject Name="EO1"
    ExportPath=/Script/Insider.MyProperty_ExportObject"/Game/Property/EO1.EO1"
        "MyValue"=456
    End Object

    "ObjectExport"/=Script/Insider.MyProperty_ExportObject"/Game/Property/EO1.EO1"

    "ObjectWithoutExport"/=Script/Insider.MyProperty_ExportObject"/Game/Property/EO2
        .EO2"
    End Object

```

You can see that objects with ObjectExport also export field values, whereas objects with ObjectWithoutExport only export paths.

## Principle:

In the source code, the function in question must be noted: for the Export tag to take effect in ExportProperties, the export tag cannot be applied to the sub-objects of the main object, or it will follow the ExportInnerObjects call route. In the example above, both ObjectExport and ObjectWithoutExport point to another external object, thus DataAsset is used to generate the assets.

```

void ExportProperties()
{
    FObjectPropertyParams* ExportObjectProp = (PropertyParams->PropertyFlags &
    CPF_ExportObject) != 0 ? CastField<FOBJECTPROPERTYBASE>(PropertyParams) : NULL;
}

```

## SaveGame

- **Function Description:** During SaveGame archiving, only attributes marked with SaveGame are serialized, while other attributes are not.
- **Metadata Type:** bool
- **Engine Module:** Serialization
- **Common Usage:** ★★★★

When archiving SaveGame, only attributes marked with SaveGame are serialized, excluding all others.

It specifically designates which properties are to be saved in the archive.

Substructure or subobject properties must also be tagged with SaveGame.

Many foundational structures within NoExportTypes.h are marked with SaveGame.

## Test Code:

```
struct FMySaveGameArchive : public FObjectAndNameAsStringProxyArchive
{
    FMySaveGameArchive (FArchive& InInnerArchive)
        : FObjectAndNameAsStringProxyArchive(InInnerArchive)
    {
        ArIsSaveGame = true;
    }
};

USTRUCT(BlueprintType)
struct FMySaveGameStruct
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        FString MyString_Default;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, SaveGame)
        FString MyString_SaveGame;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_SaveGame :public USaveGame
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Default = 123;
    //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
    CPF_SaveGame | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, SaveGame)
        int32 MyInt_SaveGame = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, SaveGame)
        FMySaveGameStruct MyStruct;
};

UMyProperty_SaveGame* UMyProperty_SaveGame_Test::LoadGameFromMemory(const
TArray<uint8>& InSaveData)
{
    FMemoryReader MemoryReader(InSaveData, true);

    FObjectAndNameAsStringProxyArchive Ar(MemoryReader, true);
    Ar.ArIsSaveGame = true;//This tag must be added manually

    UMyProperty_SaveGame* OutSaveGameObject = NewObject<UMyProperty_SaveGame>
    (GetTransientPackage(), UMyProperty_SaveGame::StaticClass());
    OutSaveGameObject->Serialize(Ar);

    return OutSaveGameObject;
}
```

```

bool UMyProperty_SaveGame_Test::SaveGameToMemory(UMyProperty_SaveGame*&
SaveGameObject, TArray<uint8>& OutSaveData)
{
    FMemoryWriter MemoryWriter(OutSaveData, true);

    // Then save the object state, replacing object refs and names with strings
    FObjectAndNameAsStringProxyArchive Ar(MemoryWriter, false);
    Ar.ArIsSaveGame = true; // This tag must be added manually
    SaveGameObject->Serialize(Ar);

    return true; // Not sure if there's a failure case here.
}

void UMyProperty_SaveGame_Test::RunTest()
{
    UMyProperty_SaveGame* saveGame = Cast<UMyProperty_SaveGame>
(UGameplayStatics::CreateSaveGameObject(UMyProperty_SaveGame::StaticClass()));
    saveGame->MyInt_Default = 456;
    saveGame->MyInt_SaveGame = 456;
    saveGame->MyStruct.MyString_Default = TEXT("Hello");
    saveGame->MyStruct.MyString_SaveGame = TEXT("Hello");

    TArray<uint8> outBytes;
    UMyProperty_SaveGame_Test::SaveGameToMemory(saveGame, outBytes);

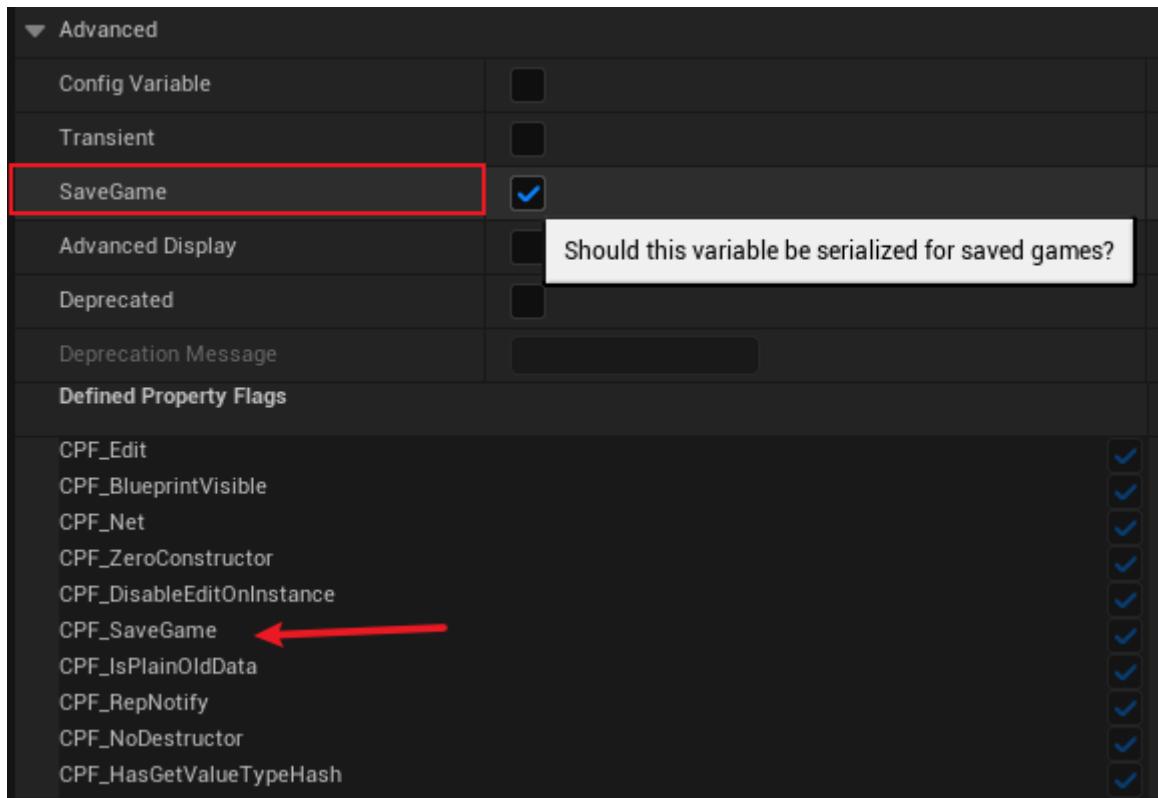
    UMyProperty_SaveGame* saveGame2 =
    UMyProperty_SaveGame_Test::LoadGameFromMemory(outBytes);
}

```

The test results indicate that only attributes marked with SaveGame are serialized.

Name	Value	Type
UMyProperty_SaveGame_Test::LoadGam...	0x00000760e109cbe0 (Name="MyProperty_SaveGame"_1)	UMyProperty_SaveGa...
outBytes	Num=205, Max=285 "xf 0 0 0MyInt_SaveGame 0 f 0 0 0Int... Q View ▾	TArray<unsigned char ...
saveGame	0x00000760e119d2c0 (Name="MyProperty_SaveGame"_0)	UMyProperty_SaveGa...
saveGame2	0x00000760e109cbe0 (Name="MyProperty_SaveGame"_1)	UMyProperty_SaveGa...
USaveGame	(Name="MyProperty_SaveGame"_1)	USaveGame
MyInt_Default	123	int
MyInt_SaveGame	456	int
MyStruct	{MyString_Default=Empty MyString_SaveGame=L"Hello"}	FMySaveGameStruct
MyString_Default	Empty	FString
MyString_SaveGame	L"Hello"	FString

Equivalent to indicating in the Blueprint Details panel:



## Principle:

This tag is only detected when `Ar.IsSaveGame`, which means that this tag is only used when detecting the sub-object structure properties of the `USaveGame` object. But `Ar.IsSaveGame` needs to be set to true manually, otherwise this mechanism will not work by default. One way to achieve this is to manually add `Ar.ArIsSaveGame = true;`, or customize a `FMySaveGameArchive` for serialization.

In the source code, `UEnhancedInputUserSettings` is found to inherit from `USaveGame` and is saved using the archiving method.

```

bool FProperty::ShouldSerializeValue(FArchive& Ar) const
{
    // Skip the property if the archive says we should
    if (Ar.ShouldSkipProperty(this))
    {
        return false;
    }

    // Skip non-SaveGame properties if we're saving game state
    if (!(PropertyFlags & CPF_SaveGame) && Ar.IsSaveGame())
    {
        return false;
    }

    const uint64 skipFlags = CPF_Transient | CPF_DuplicateTransient |
    CPF_NonPIEDuplicateTransient | CPF_NonTransactional | CPF_Deprecated |
    CPF_DevelopmentAssets | CPF_SkipSerialization;
    if (!(PropertyFlags & skipFlags))
    {
        return true;
    }
}

```

```

        // skip properties marked Transient when persisting an object, unless we're
        // saving an archetype
        if ((PropertyFlags & CPF_Transient) && Ar.IsPersistent() &&
!Ar.IsSerializingDefaults())
{
    return false;
}

        // skip properties marked DuplicateTransient when duplicating
        if ((PropertyFlags & CPF_DuplicateTransient) && (Ar.GetPortFlags() &
PPF_Duplicate))
{
    return false;
}

        // skip properties marked NonPIEDuplicateTransient when duplicating, but not
when we're duplicating for PIE
        if ((PropertyFlags & CPF_NonPIEDuplicateTransient) && !(Ar.GetPortFlags() &
PPF_DuplicateForPIE) && (Ar.GetPortFlags() & PPF_Duplicate))
{
    return false;
}

        // skip properties marked NonTransactional when transacting
        if ((PropertyFlags & CPF_NonTransactional) && Ar.IsTransacting())
{
    return false;
}

        // skip deprecated properties when saving or transacting, unless the archive
has explicitly requested them
        if ((PropertyFlags & CPF_Deprecated) &&
!Ar.HasAllPortFlags(PPF_UseDeprecatedProperties) && (Ar.IsSaving() ||
Ar.IsTransacting() || Ar.WantBinaryPropertySerialization()))
{
    return false;
}

        // skip properties marked skipSerialization, unless the archive is forcing
them
        if ((PropertyFlags & CPF_SkipSerialization) &&
(Ar.WantBinaryPropertySerialization() ||
!Ar.HasAllPortFlags(PPF_ForceTaggedSerialization())))
{
    return false;
}

        // skip editor-only properties when the archive is rejecting them
        if (IsEditorOnlyProperty() && Ar.IsFilterEditorOnly())
{
    return false;
}

        // otherwise serialize!
        return true;
}

```

# SkipSerialization

- **Function Description:** Skips serialization of this attribute during binary serialization, but it remains exportable when using ExportText.
- **Metadata Type:** bool
- **Engine Module:** Serialization
- **Action Mechanism:** Include CPF\_SkipSerialization in PropertyFlags
- **Commonly Used:** ★★★

During standard binary serialization, this marker prevents the attribute from being serialized, functioning similarly to Transient. However, it can still be exported if ExportText is used. Internally, ExportProperties is utilized.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_SerializationText :public Uobject
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Default = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
        CPF_NativeAccessSpecifierPublic | CPF_SkipSerialization
    UPROPERTY(EditAnywhere, BlueprintReadWrite,SkipSerialization)
        int32 MyInt_skipSerialization = 123;
};

void UMyProperty_SerializationText_Test::RunTest()
{
    UMyProperty_SerializationText* obj = NewObject<UMyProperty_SerializationText>(GetTransientPackage());

    obj->MyInt_Default = 456;
    obj->MyInt_skipSerialization = 456;

    //save obj
    TArray<uint8> outBytes;
    FMemoryWriter MemoryWriter(outBytes, true);
    FObjectAndNameAsStringProxyArchive Ar(MemoryWriter, false);
    obj->Serialize(Ar);

    //load
    FMemoryReader MemoryReader(outBytes, true);

    FObjectAndNameAsStringProxyArchive Ar2(MemoryReader, true);

    UMyProperty_SerializationText* obj2 =
NewObject<UMyProperty_SerializationText>(GetTransientPackage());
}
```

```
    obj2->Serialize(Ar2);  
}
```

As seen in the test results, the attribute has not been serialized.

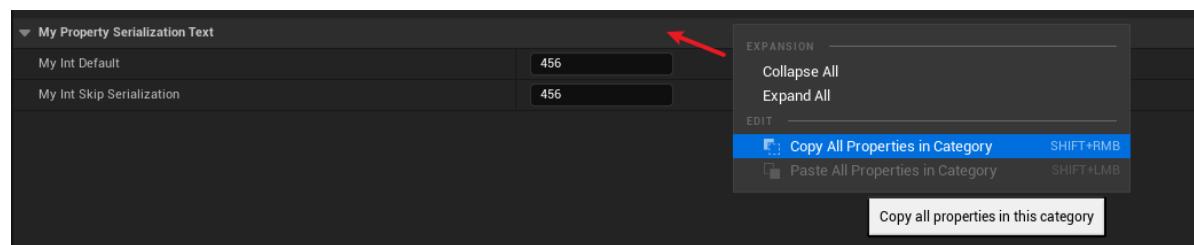
If ExportText is used for export: either T3D or COPY format is acceptable

```
UMyProperty_SerializationText* obj = NewObject<UMyProperty_SerializationText>(GetTransientPackage());  
  
obj->MyInt_Default = 456;  
obj->MyInt_skipSerialization = 456;  
  
FStringOutputDevice Ar;  
UExporter::ExportToOutputDevice(nullptr, obj, nullptr, Ar, TEXT("T3D"), 3);
```

The output result will be:

```
Begin Object Class=/Script/Insider.MyProperty_SerializationText
Name="MyProperty_SerializationText_0"
ExportPath=/Script/Insider.MyProperty_SerializationText'"/Engine/Transient.MyProperty_SerializationText_0'
    MyInt_Default=456
    MyInt_SkipSerialization=456
End Object
```

Moreover, if you right-click and copy in the editor,



It is also possible to generate a text export:

```

{
    "Tagged": [
        [
            "MyInt_Default",
            "456"
        ],
        [
            "MyInt_SkipSerialization",
            "456"
        ]
    ]
}

```

## Principle:

Note that when determining whether a Property should be serialized, the ShouldSerializeValue function is used for standard serialization, while ShouldPort is used during ExportText.

## TextExportTransient

- **Function Description:** When exporting to .COPY format using ExportText, this attribute is ignored.
- **Metadata Type:** bool
- **Engine Module:** Serialization
- **Action Mechanism:** Include CPF\_TextExportTransient in PropertyFlags
- **Common Usage:** ★

When exporting to .COPY format with ExportText, this attribute is ignored.

However, text export will still occur when copying properties with the mouse.

## \*Test Code:

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_SerializationText :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Default = 123;
        //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
        CPF_NativeAccessSpecifierPublic | CPF_SkipSerialization
    UPROPERTY(EditAnywhere, BlueprintReadWrite,SkipSerialization)
        int32 MyInt_SkipSerialization = 123;
        //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
        | CPF_IsPlainOldData | CPF_NoDestructor | CPF_TextExportTransient |
        CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite,TextExportTransient)
        int32 MyInt_TextExportTransient = 123;

```

```

};

void UMyProperty_SerializationText_Test::RunExportTest()
{
    UMyProperty_SerializationText* obj = NewObject<UMyProperty_SerializationText>(GetTransientPackage());

    obj->MyInt_Default = 456;
    obj->MyInt_SkipSerialization = 456;
    obj->MyInt_TextExportTransient = 456;

    FStringOutputDevice Ar;
    UExporter::ExportToOutputDevice(nullptr, obj, nullptr, Ar, TEXT("T3D"), 3);

    FStringOutputDevice Ar2;
    UExporter::ExportToOutputDevice(nullptr, obj, nullptr, Ar2, TEXT("COPY"), 3);

    FString str=Ar;
}

```

\*Exported Results:

```

T3D格式:
Begin Object Class=/Script/Insider.MyProperty_SerializationText
Name="BP_SerializationText"
ExportPath="/Script/Insider.MyProperty_SerializationText'/Game/Property/BP_SerializationText.BP_SerializationText'"
    MyInt_Default=456
    MyInt_SkipSerialization=456
    MyInt_TextExportTransient=456
End Object

COPY格式:
Begin Object Class=/Script/Insider.MyProperty_SerializationText
Name="BP_SerializationText"
ExportPath="/Script/Insider.MyProperty_SerializationText'/Game/Property/BP_SerializationText.BP_SerializationText'"
    MyInt_Default=456
    MyInt_SkipSerialization=456
End Object

```

\*The copy will still have the text in effect:

```

{
    "Tagged": [
        [
            "MyInt_Default",
            "456"
        ],
        [
            "MyInt_SkipSerialization",
            "456"
        ],
        [

```

```

        "MyInt_TextExportTransient",
        "456"
    ]
}

```

Thus, it can be observed that MyInt\_TextExportTransient is not exported in COPY format.

## \*Principle:

---

Note that when determining whether a Property should be serialized, the ShouldSerializeValue function is used for general serialization. For ExportText, the ShouldPort function is used for this purpose.

But if the serialization format is COPY, when setting PortFlags, an additional PPF\_Copy flag is added. Therefore, the CPF\_TextExportTransient check becomes effective in subsequent evaluations.

```

if ( FCString::Stricmp(FileType, TEXT("COPY")) == 0 )
{
    // some code which doesn't have access to the exporter's file type needs
    // to handle copy/paste differently than exporting to file,
    // so set the export flag accordingly
    PortFlags |= PPF_Copy;
}

// Return whether the property should be exported.
//
bool FProperty::ShouldPort( uint32 PortFlags/*=0*/ ) const
{
    // if no size, don't export
    if (GetSize() <= 0)
    {
        return false;
    }

    if (HasAnyPropertyFlags(CPF_Deprecated) && !(PortFlags &
(PPF_ParsingDefaultProperties | PPF_UseDeprecatedProperties)))
    {
        return false;
    }

    // if we're parsing default properties or the user indicated that transient
    // properties should be included
    if (HasAnyPropertyFlags(CPF_Transient) && !(PortFlags &
(PPF_ParsingDefaultProperties | PPF_IncludeTransient)))
    {
        return false;
    }

    // if we're copying, treat DuplicateTransient as transient
    if ((PortFlags & PPF_Copy) && HasAnyPropertyFlags(CPF_DuplicateTransient |
CPF_TextExportTransient) && !(PortFlags & (PPF_ParsingDefaultProperties |
PPF_IncludeTransient)))

```

```

    {
        return false;
    }

    // if we're not copying for PIE and NonPIETransient is set, don't export
    if (!(PortFlags & PPF_DuplicateForPIE) &&
HasAnyPropertyFlags(CPF_NonPIEDuplicateTransient))
    {
        return false;
    }

    // if we're only supposed to export components and this isn't a component
    // property, don't export
    if ((PortFlags & PPF_SubobjectsOnly) && !ContainsInstancedObjectProperty())
    {
        return false;
    }

    // hide non-Edit properties when we're exporting for the property window
    if ((PortFlags & PPF_Propertywindow) && !(PropertyFlags & CPF_Edit))
    {
        return false;
    }

    return true;
}

```

## Transient

---

- **Function Description:** Do not serialize this property; it will be initialized with 0.
- **Metadata Type:** bool
- **Engine Module:** Serialization
- **Mechanism:** Add CPF\_Transient to PropertyFlags
- **Common Usage:** ★★★★☆

During serialization, this property is skipped, and the default value is filled with 0.

Neither binary nor text serialization includes this property.

Usually used for temporary intermediate variables or variables holding calculated results.

## Sample Code:

---

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Serialization :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Default = 123;

```

```

        //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
| CPF_Transient | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
| CPF_NativeAccessSpecifierPublic
        UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
            int32 MyInt_Transient = 123;
        //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
| CPF_DuplicateTransient | CPF_IsPlainOldData | CPF_NoDestructor |
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
        UPROPERTY(EditAnywhere, BlueprintReadWrite, DuplicateTransient)
            int32 MyInt_DuplicateTransient = 123;
        //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
| CPF_IsPlainOldData | CPF_NoDestructor | CPF_NonPIEDuplicateTransient |
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
        UPROPERTY(EditAnywhere, BlueprintReadWrite, NonPIEDuplicateTransient)
            int32 MyInt_NonPIEDuplicateTransient = 123;
};

void UMyProperty_Serialization_Test::RunTest()
{
    UMyProperty_Serialization* obj = NewObject<UMyProperty_Serialization>
(GetTransientPackage());

    obj->MyInt_Default = 456;
    obj->MyInt_Transient = 456;
    obj->MyInt_DuplicateTransient = 456;
    obj->MyInt_NonPIEDuplicateTransient = 456;

    //save obj
    TArray<uint8> outBytes;
    FMemoryWriter MemoryWriter(outBytes, true);
    FObjectAndNameAsStringProxyArchive Ar(MemoryWriter, false);
    obj->Serialize(Ar);

    //load
    FMemoryReader MemoryReader(outBytes, true);

    FObjectAndNameAsStringProxyArchive Ar2(MemoryReader, true);

    UMyProperty_Serialization* obj2 = NewObject<UMyProperty_Serialization>
(GetTransientPackage());
    obj2->Serialize(Ar2);
}

```

Perform AssetActions→Export on such a BP DataAsset,

T3D Format:

```

Begin Object Class=/Script/Insider.MyProperty_Serialization
Name="BP_Serialization"
ExportPath="/Script/Insider.MyProperty_Serialization'/Game/Property/BP_Serializat
ion.BP_Serialization"
    MyInt_Default=456
    MyInt_DuplicateTransient=456
End Object

```

COPY Format:

```

Begin Object Class=/Script/Insider.MyProperty_Serialization
Name="BP_Serialization"
ExportPath="/Script/Insider.MyProperty_Serialization'/Game/Property/BP_Serialization.BP_Serialization"
MyInt_Default=456
End Object

```

If it is standard serialization:

It can be observed that obj2's MyInt\_Transient property does not receive the new serialized value of 456.

obj	0x000005965862a820 (Name="MyProperty_Serialization"_1)	UMyProperty_Serializa...
UDataAsset	(Name="MyProperty_Serialization"_1)	UDataAsset
MyInt_Default	456	int
MyInt_Transient	456	int
MyInt_DuplicateTransient	456	int
MyInt_NonPIEDuplicateTransient	456	int
obj2	0x000005965862a910 (Name="MyProperty_Serialization"_2)	UMyProperty_Serializa...
UDataAsset	(Name="MyProperty_Serialization"_2)	UDataAsset
MyInt_Default	456	int
MyInt_Transient	123	int
MyInt_DuplicateTransient	456	int
MyInt_NonPIEDuplicateTransient	456	int
outBytes	Num=182, Max=208 "xe\0\0\0MyInt_Default\0\f\0\0\0IntPro... Q View ▾	TArray<unsigned char...

## Principle Code:

The CPF\_Transient flag is effective only when IsPersistent() is true, and not when saving a CDO. The SetIsPersistent() function is called in many places, such as in MemoryReader/MemoryWriter, where it always checks IsPersistent.

Therefore, Transient properties are ignored during serialization.

When ExportText is performed, CPF\_Transient is checked, unless the PPF\_IncludeTransient flag is explicitly included

```

bool FProperty::ShouldSerializeValue(FArchive& Ar) const
{
    // Skip the property if the archive says we should
    if (Ar.ShouldSkipProperty(this))
    {
        return false;
    }

    // Skip non-SaveGame properties if we're saving game state
    if (!(PropertyFlags & CPF_SaveGame) && Ar.IsSaveGame())
    {
        return false;
    }

    const uint64 SkipFlags = CPF_Transient | CPF_DuplicateTransient |
    CPF_NonPIEDuplicateTransient | CPF_NonTransactional | CPF_Deprecated |
    CPF_DevelopmentAssets | CPF_SkipSerialization;
    if (!(PropertyFlags & SkipFlags))
    {
        return true;
    }
}

```

```

        // skip properties marked Transient when persisting an object, unless we're
        // saving an archetype
        if ((PropertyFlags & CPF_Transient) && Ar.IsPersistent() &&
!Ar.IsSerializingDefaults())
{
    return false;
}

        // skip properties marked DuplicateTransient when duplicating
        if ((PropertyFlags & CPF_DuplicateTransient) && (Ar.GetPortFlags() &
PPF_Duplicate))
{
    return false;
}

        // Skip properties marked NonPIEDuplicateTransient when duplicating, but not
when we're duplicating for PIE
        if ((PropertyFlags & CPF_NonPIEDuplicateTransient) && !(Ar.GetPortFlags() &
PPF_DuplicateForPIE) && (Ar.GetPortFlags() & PPF_Duplicate))
{
    return false;
}

        // skip properties marked NonTransactional when transacting
        if ((PropertyFlags & CPF_NonTransactional) && Ar.IsTransacting())
{
    return false;
}

        // skip deprecated properties when saving or transacting, unless the archive
has explicitly requested them
        if ((PropertyFlags & CPF_Deprecated) &&
!Ar.HasAllPortFlags(PPF_UseDeprecatedProperties) && (Ar.IsSaving() ||

Ar.IsTransacting() || Ar.WantBinaryPropertySerialization()))
{
    return false;
}

        // skip properties marked SkipSerialization, unless the archive is forcing
them
        if ((PropertyFlags & CPF_SkipSerialization) &&
(Ar.WantBinaryPropertySerialization() ||
!Ar.HasAllPortFlags(PPF_ForceTaggedSerialization)))
{
    return false;
}

        // skip editor-only properties when the archive is rejecting them
        if (IsEditorOnlyProperty() && Ar.IsFilterEditorOnly())
{
    return false;
}

        // otherwise serialize!
        return true;

```

```

}

///////////////////////////////
bool FProperty::ShouldPort( uint32 PortFlags/*=0*/ ) const
{
    // if no size, don't export
    if (GetSize() <= 0)
    {
        return false;
    }

    if (HasAnyPropertyFlags(CPF_Deprecated) && !(PortFlags &
(PPF_ParsingDefaultProperties | PPF_UseDeprecatedProperties)))
    {
        return false;
    }

    // if we're parsing default properties or the user indicated that transient
    // properties should be included
    if (HasAnyPropertyFlags(CPF_Transient) && !(PortFlags &
(PPF_ParsingDefaultProperties | PPF_IncludeTransient)))
    {
        return false;
    }

    // if we're copying, treat DuplicateTransient as transient
    if ((PortFlags & PPF_Copy) && HasAnyPropertyFlags(CPF_DuplicateTransient | 
CPF_TextExportTransient) && !(PortFlags & (PPF_ParsingDefaultProperties | 
PPF_IncludeTransient)))
    {
        return false;
    }

    // if we're not copying for PIE and NonPIETransient is set, don't export
    if (!(PortFlags & PPF_DuplicateForPIE) &&
HasAnyPropertyFlags(CPF_NonPIEDuplicateTransient))
    {
        return false;
    }

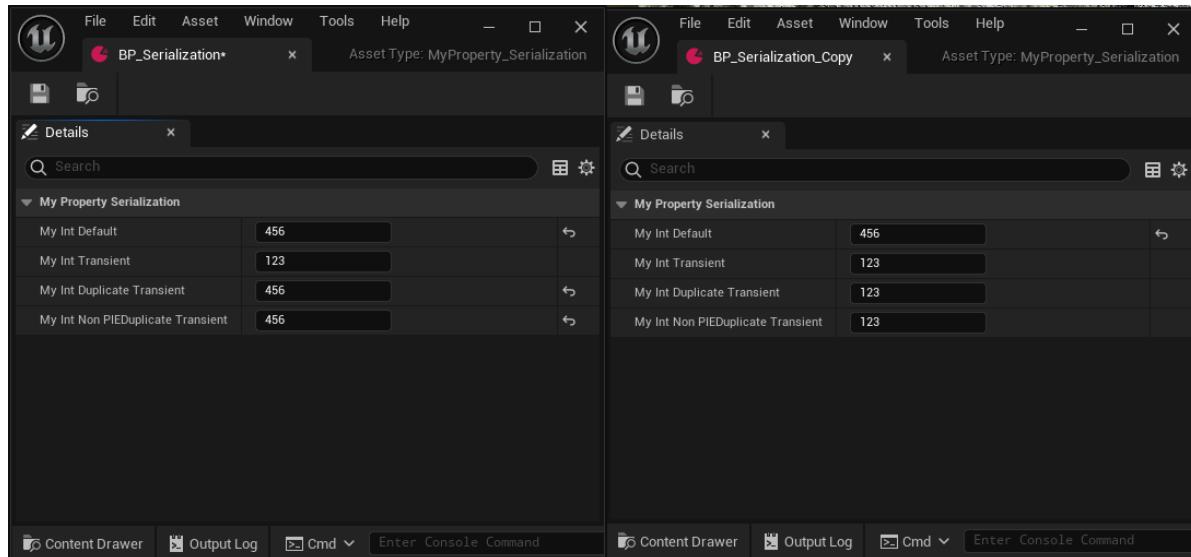
    // if we're only supposed to export components and this isn't a component
    // property, don't export
    if ((PortFlags & PPF_SubobjectsOnly) && !ContainsInstancedObjectProperty())
    {
        return false;
    }

    // hide non-Edit properties when we're exporting for the property window
    if ((PortFlags & PPF_Propertywindow) && !(PropertyFlags & CPF_Edit))
    {
        return false;
    }

    return true;
}

```

Because Transient properties are not serialized, any changes to their values are not saved. When the Asset is opened, it will still display the default value and will not be copied.



## DuplicateTransient

- **Function description:** This attribute is ignored during object duplication or when exporting in COPY format.
- **Metadata type:** bool
- **Engine module:** Serialization
- **Action mechanism:** Add CPF\_DuplicateTransient to PropertyFlags
- **Commonality:** ★★

This attribute is ignored when duplicating objects or exporting in COPY format.

## Sample Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Serialization :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Default = 123;
    //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
    | CPF_Transient | CPF_IsPlainOldDataOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
    | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
    int32 MyInt_Transient = 123;
    //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
    | CPF_DuplicateTransient | CPF_IsPlainOldDataOldData | CPF_NoDestructor |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, DuplicateTransient)
    int32 MyInt_DuplicateTransient = 123;
    //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
    | CPF_IsPlainOldDataOldData | CPF_NoDestructor | CPF_NonPIEDuplicateTransient |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, NonPIEDuplicateTransient)
    int32 MyInt_NonPIEDuplicateTransient = 123;
};

void UMyProperty_Serialization_Test::RunTest()
{
    UMyProperty_Serialization* obj = NewObject<UMyProperty_Serialization>(GetTransientPackage());

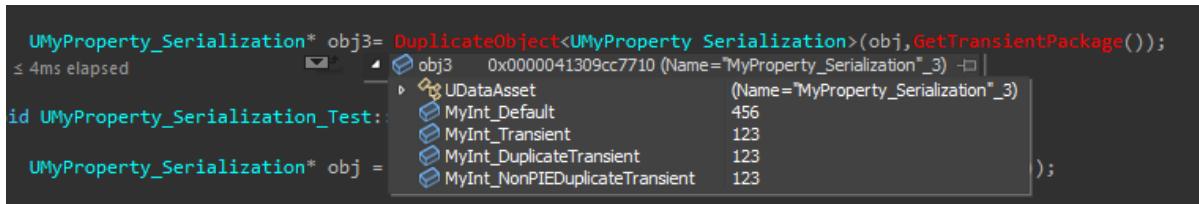
    obj->MyInt_Default = 456;
    obj->MyInt_Transient = 456;
    obj->MyInt_DuplicateTransient = 456;
    obj->MyInt_NonPIEDuplicateTransient = 456;

    UMyProperty_Serialization* obj3 = DuplicateObject<UMyProperty_Serialization>(obj, GetTransientPackage());
}

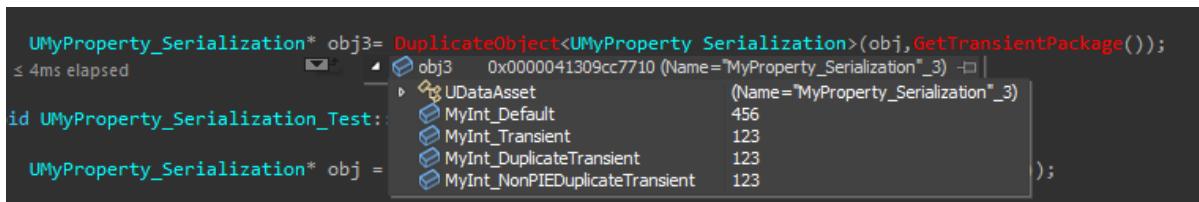
```

## Example Effect:

When duplicating a blueprint, it is evident that DuplicateTransient is not copied



When copying in C++, it is observed that MyInt\_DuplicateTransient is not duplicated; it remains 123 instead of 456.



## Principle:

During text export, if the format is T3D, the attribute will still be exported. However, if the format is COPY, it will not be exported.

```

bool FProperty::ShouldPort( uint32 PortFlags/*=0*/ ) const
{
    // if we're copying, treat DuplicateTransient as transient
    if ((PortFlags & PPF_Copy) && HasAnyPropertyParams(CPF_DuplicateTransient |
        CPF_TextExportTransient) && !(PortFlags & (PPF_ParsingDefaultProperties |
        PPF_IncludeTransient)))
    {
        return false;
    }
}

```

During binary serialization:

The attribute will only be skipped if PPF\_Duplicate is active (either during DuplicateObject or asset copy)

```
bool FProperty::ShouldSerializeValue(FArchive& Ar) const
{
// Skip properties marked DuplicateTransient when duplicating
if ((PropertyFlags & CPF_DuplicateTransient) && (Ar.GetPortFlags() &
PPF_Duplicate))
{
    return false;
}
}
```

When duplicating an asset, both DuplicateAsset and DuplicateObject occur. At this point, PortFlags equals PPF\_Duplicate, and ShouldSerializeValue is triggered for assessment. The attribute will be skipped at this stage

## NonPIEDuplicateTransient

- **Function description:** Ignore this property during object duplication, and in scenarios other than PIE.
- **Metadata type:** bool
- **Engine module:** Serialization
- **Mechanism of action:** Include CPF\_NonPIEDuplicateTransient in PropertyFlags
- **Commonly used:** ★

Ignore this property during object duplication, and in scenarios other than PIE.

- The difference between DuplicateTransient and NonPIEDuplicateTransient is that the former ignores this property during any object duplication, while the latter still duplicates the property during PIE (also during the object duplication process), with behavior consistent with the former in other duplication scenarios.
- PIE essentially involves copying the Actor from the current editing world to the PIE world, which triggers the duplication of the Actor.

## Sample Code:

Prepared a DataAsset and an Actor to separately verify the different duplication behaviors.

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Serialization :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Default = 123;
    //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
    | CPF_Transient | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
    | CPF_NativeAccessSpecifierPublic
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
    int32 MyInt_Transient = 123;
    //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
| CPF_DuplicateTransient | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UPROPERTY(EditAnywhere, BlueprintReadWrite, DuplicateTransient)
    int32 MyInt_DuplicateTransient = 123;
    //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
| CPF_IsPlainOldData | CPF_NoDestructor | CPF_NonPIEDuplicateTransient | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UPROPERTY(EditAnywhere, BlueprintReadWrite, NonPIEDuplicateTransient)
    int32 MyInt_NonPIEDuplicateTransient = 123;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Serialization_TestActor :public AActor
{
public:
    GENERATED_BODY()
protected:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyInt_Default = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
| CPF_Transient | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
| CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
        int32 MyInt_Transient = 123;
        //PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor
| CPF_DuplicateTransient | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, DuplicateTransient)
        int32 MyInt_DuplicateTransient = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, NonPIEDuplicateTransient)
        int32 MyInt_NonPIEDuplicateTransient = 123;
};

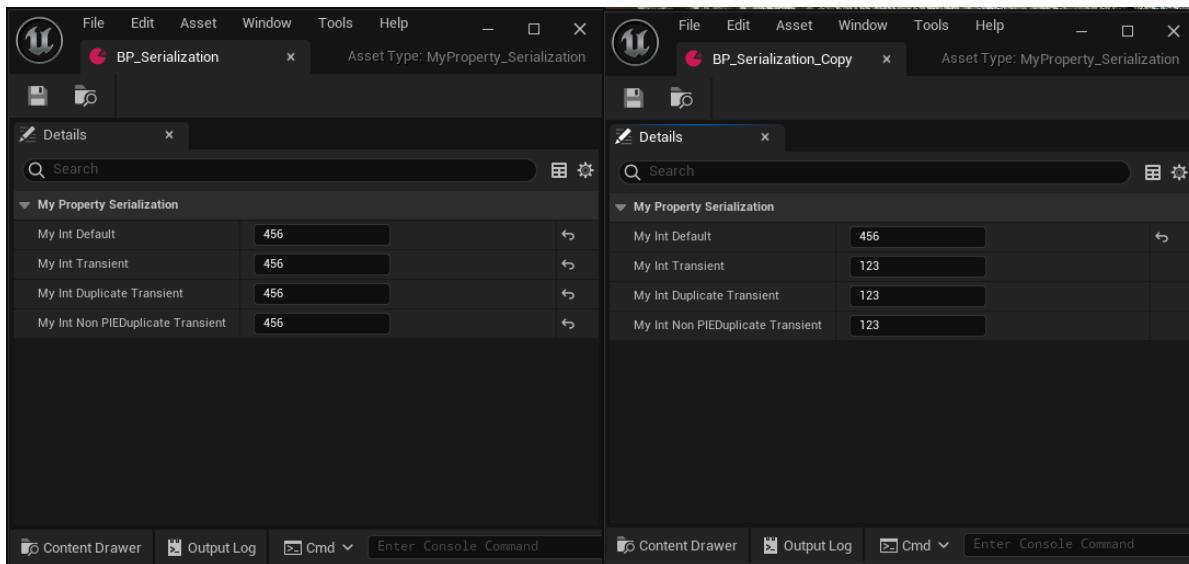
```

## Example Effect:

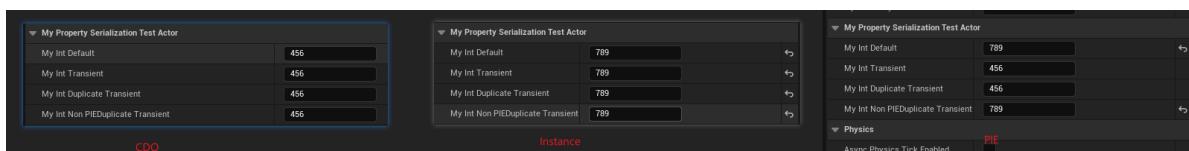
---

When duplicating an asset, `DuplicateAsset` and `DuplicateObject` occur, at which point `PortFlags = PPF_Duplicate`, and `ShouldSerializeValue` is triggered for judgment. This property is skipped at this time.

It can be observed that `NonPIEDuplicateTransient` is not duplicated.



When PIE is activated, it can be seen that NonPIEDuplicateTransient is indeed copied at this point. This is because PortFlags = PPF\_DuplicateForPIE & PPF\_Duplicate



In conclusion, this is used for certain cache data where serialization is not required during duplication, preventing two different Actors from using the same calculated temporary data. However, during PIE, Actors can each use their own set of data, because during PIE, the essence is copying an instance of the Actor from the current editing world to the PIE world, which triggers the duplication of the Actor.

## Principle:

During text export, this property is not serialized when not duplicating in PIE.

```
bool FProperty::shouldPort( uint32 PortFlags/*=0*/ ) const
{
    // if we're not copying for PIE and NonPIETransient is set, don't export
    if (!(PortFlags & PPF_DuplicateForPIE) &&
        HasAnyPropertyFlags(CPF_NonPIEDuplicateTransient))
    {
        return false;
    }
}
```

During binary serialization:

This property is skipped only when PPF\_Duplicate is effective (DuplicateObject? or asset duplication), but serialization must continue during PIE.

```

bool FProperty::ShouldSerializeValue(FArchive& Ar) const
{
// Skip properties marked NonPIEDuplicateTransient when duplicating, but not when
we're duplicating for PIE
    if ((PropertyFlags & CPF_NonPIEDuplicateTransient) && !(Ar.GetPortFlags() &
PPF_DuplicateForPIE) && (Ar.GetPortFlags() & PPF_Duplicate))
    {
        return false;
    }
}

```

## Interp

- **Function Description:** Specifies that the attribute value can be exposed for editing within the timeline, typically used in standard Timeline or UMG animations.
- **Metadata Type:** bool
- **Engine Module:** Sequencer
- **Action Mechanism:** Includes CPF\_Edit, CPF\_BlueprintVisible, CPF\_Interp in the PropertyFlags
- **Common Usage:** ★★★

The property can be revealed in the timeline, generally for animation editing purposes.

## Example Code:

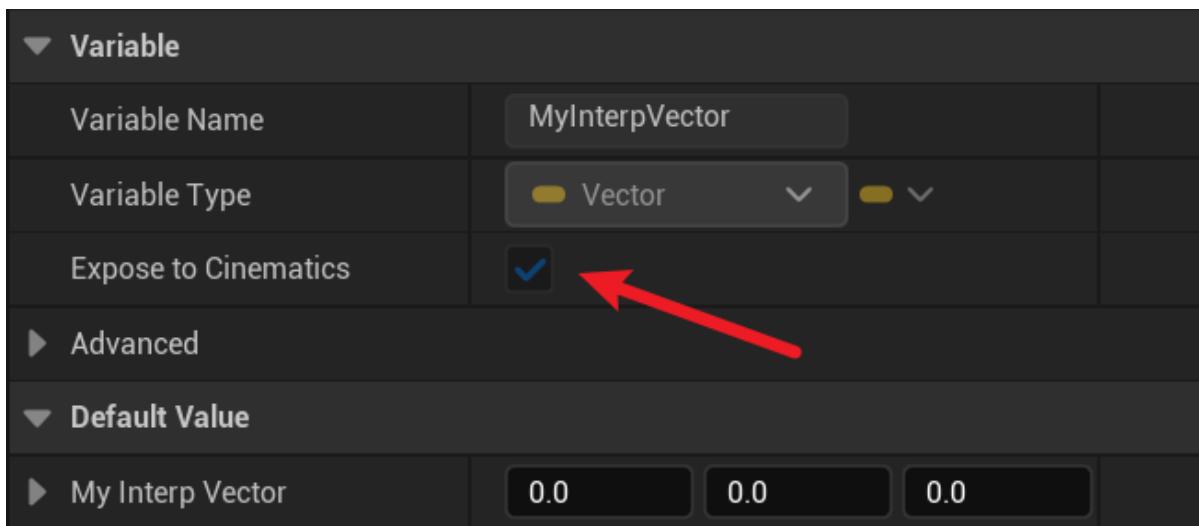
```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Interp :public AActor
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Interp, Category = Animation)
        FVector MyInterpVector;
};

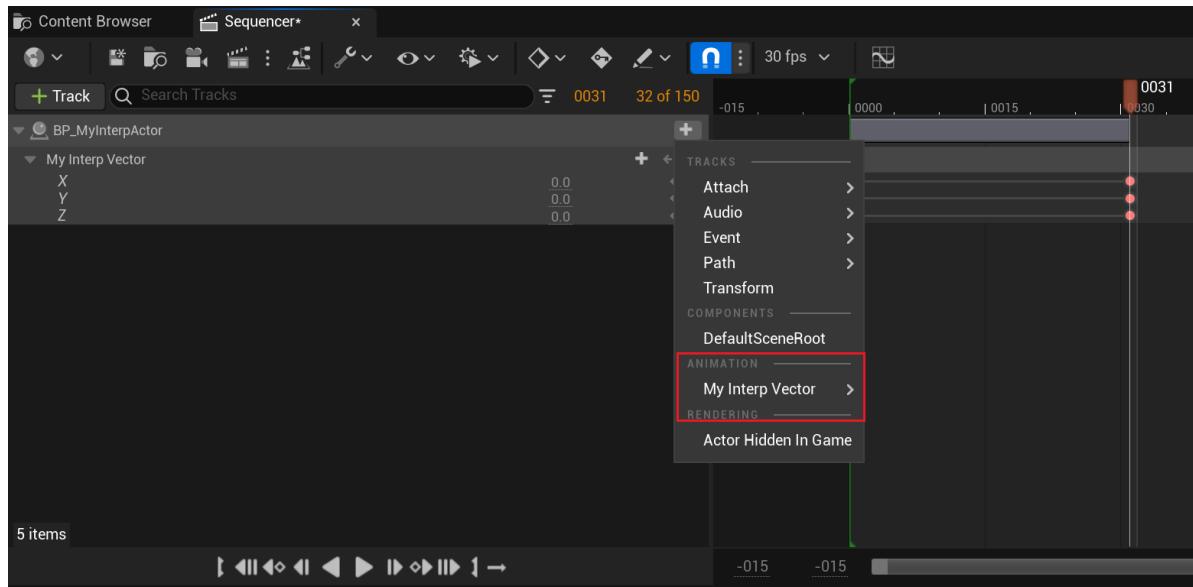
```

## Example Effect:

It affects the flag on the property



Thus, a Track can be added to this property within the Sequencer



## Replicated

- **Function Description:** Specifies that this property should be replicated along with the network.
- **Metadata Type:** bool
- **Engine Module:** Network
- **Action Mechanism:** Include CPF\_Net in PropertyFlags
- **Common Usage:** ★★★★☆

## Sample Code:

Remember to add the GetLifetimeReplicatedProps function accordingly in the cpp code

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Network :public AActor
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt_Default = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Replicated)
    int32 MyInt_Replicated = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Replicated)
    FMyReplicatedStruct MyStruct_Replicated;
};

void AMyProperty_Network::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);
    DOREPLIFETIME(AMyProperty_Network, MyInt_Replicated);
    DOREPLIFETIME(AMyProperty_Network, MyStruct_Replicated);
}
```

```
}
```

No need to demonstrate the sample effect; this is a fundamental network attribute.

## ReplicatedUsing

- **Function Description:** Defines a notification callback function that executes after an attribute is updated over the network.
- **Metadata Type:** string="abc"
- **Engine Module:** Network
- **Action Mechanism:** Include CPF\_Net, CPF\_RepNotify in PropertyFlags
- **Common Usage:** ★★★★☆

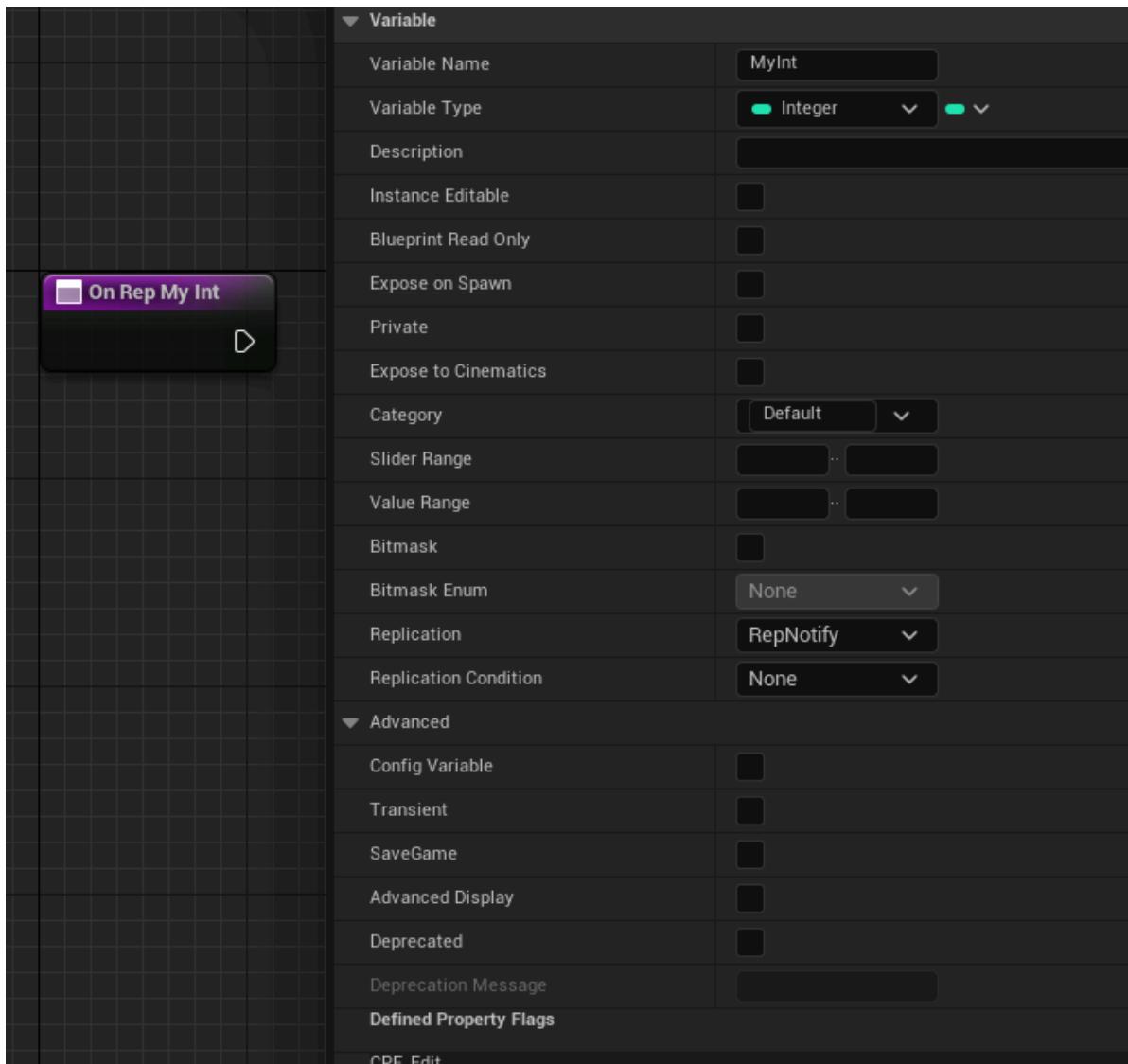
ReplicatedUsing can accept a function without parameters, or a function with one parameter that carries the old value. Typically, within the OnRep function, actions such as enabling or disabling are handled. For instance, replicating the 'enabled' property will trigger the subsequent logic accordingly.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Network :public AActor
{
public:
    GENERATED_BODY()
protected:
    UFUNCTION()
        void OnRep_MyInt(int32 oldValue);
UPROPERTY(EditAnywhere, BlueprintReadWrite, ReplicatedUsing = OnRep_MyInt)
    int32 MyInt_Replicatedusing = 123;
};

void AMyProperty_Network::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);
    DOREPLIFETIME(AMyProperty_Network, MyInt_Replicatedusing);
}
```

Functions equivalently to RepNotify within blueprints.



## NotReplicated

- **Function Description:** Skip the copy operation. This applies exclusively to the structure members and parameters within service request functions.
- **Metadata Type:** bool
- **Engine Module:** Network
- **Restriction Type:** Struct Members
- **Action Mechanism:** Include CPF\_RepSkip in PropertyFlags
- **Common Usage:** ★★★

Used only within struct members to specify that a certain attribute should not be copied; otherwise, all attributes will be copied by default. This is used to exclude a specific attribute from the structure.

## Sample Code:

```
USTRUCT(BlueprintType)
struct FMyReplicatedStruct
{
    GENERATED_BODY()
public:
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString MyString_Default;
UPROPERTY(EditAnywhere, BlueprintReadWrite, NotReplicated)
    FString MyString_NotReplicated;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Network :public AActor
{
public:
    GENERATED_BODY()
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Replicated)
        FMyReplicatedStruct MyStruct_Replicated;
};

```

MyStruct\_Replicated will be replicated, but MyString\_NotReplicated will not.

## RepRetry

---

- **Function Description:** Applicable solely to structure properties. Should this property fail to be transmitted in full (for instance, an Object reference that cannot yet be serialized over the network), an attempt to copy it will be made anew. This is the default selection for simple references; however, for structures, it results in bandwidth consumption and is not the optimal choice. Consequently, this feature is disabled by default until the tag is explicitly specified.
- **Metadata Type:** bool
- **Engine Module:** Network

## FieldNotify

---

- **Function Description:** Upon enabling the MVVM plugin, this attribute is transformed to support FieldNotify functionality.
- **Metadata Type:** bool
- **Engine Module:** MVVM, UHT
- **Restriction Type:** Attributes within ViewModel
- **Common Usage:** ★★★★

Enables the property to support FieldNotify after the MVVM plugin is activated.

## Test Code:

---

```

UCLASS(BlueprintType)
class INSIDER_API UMyViewModel :public UMVVMViewModelBase
{
    GENERATED_BODY()
protected:
    UPROPERTY(BlueprintReadWrite, FieldNotify, Getter, Setter, BlueprintSetter =
SetHP)
        float HP = 1.f;

```

```

    UPROPERTY(BlueprintReadWrite, FieldNotify, Getter, Setter, BlueprintSetter =
SetMaxHP)
    float MaxHP = 100.f;
public:
    float GetHP()const { return HP; }
    UFUNCTION(BlueprintSetter)
    void SetHP(float val)
    {
        if (UE_MVVM_SET_PROPERTY_VALUE(HP, val))
        {
            UE_MVVM_BROADCAST_FIELD_VALUE_CHANGED(GetHPPPercent);
        }
    }

    float GetMaxHP()const { return MaxHP; }
    UFUNCTION(BlueprintSetter)
    void SetMaxHP(float val)
    {
        if (UE_MVVM_SET_PROPERTY_VALUE(MaxHP, val))
        {
            UE_MVVM_BROADCAST_FIELD_VALUE_CHANGED(GetHPPPercent);
        }
    }

    //You need to manually notify that GetHealthPercent changed when
CurrentHealth or MaxHealth changed.
    UFUNCTION(BlueprintPure, FieldNotify)
    float GetHPPPercent() const
    {
        return (MaxHP != 0.f) ? HP / MaxHP : 0.f;
    }
};

```

## Test Results:

The effect is twofold: on one hand, it involves the code generated in the .generated.h and .gen.cpp files by UHT. The specifics of the macro operations will not be elaborated here; for more details, you may refer to other comprehensive MVVM-related articles. It suffices to know that UHT defines FFieldId and FFieldNotificationClassDescriptor for properties marked with FieldNotify, signifying a property that can receive notifications.

```

//MyViewModel.generated.h
#define
FID_Gitworkspace_Hello_Source_Insider_Property_MVVM_MyViewModel_h_12_FIELDNOTIFY \
\
    UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_BEGIN(INSIDER_API ) \
    UE_FIELD_NOTIFICATION_DECLARE_FIELD(HP) \
    UE_FIELD_NOTIFICATION_DECLARE_FIELD(MaxHP) \
    UE_FIELD_NOTIFICATION_DECLARE_FIELD(GetHPPPercent) \
    UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_BEGIN(HP) \
    UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(MaxHP) \
    UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD(GetHPPPercent) \
    UE_FIELD_NOTIFICATION_DECLARE_ENUM_FIELD_END() \

```

```

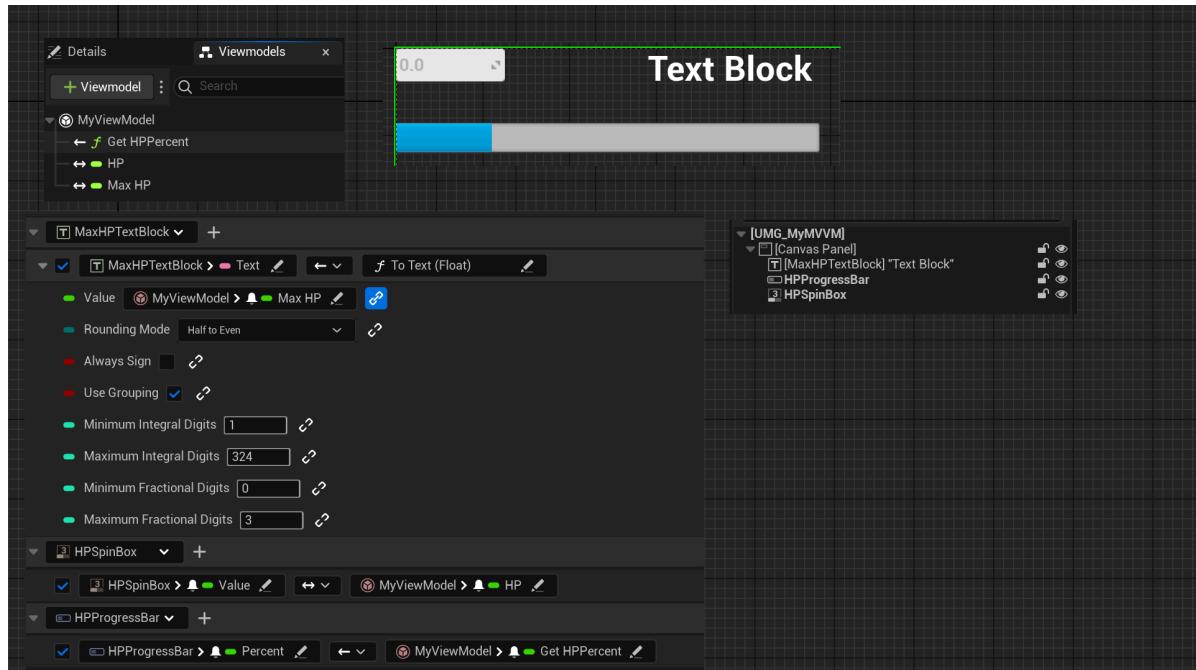
UE_FIELD_NOTIFICATION_DECLARE_CLASS_DESCRIPTOR_END();

//MyViewModel.gen.cpp
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, HP)
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, MaxHP)
UE_FIELD_NOTIFICATION_IMPLEMENT_FIELD(UMyViewModel, GetHPPercent)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_BEGIN(UMyViewModel)
UE_FIELD_NOTIFICATION_IMPLEMENT_ENUM_FIELD(UMyViewModel, HP)
UE_FIELD_NOTIFICATION_IMPLEMENT_ENUM_FIELD(UMyViewModel, MaxHP)
UE_FIELD_NOTIFICATION_IMPLEMENT_ENUM_FIELD(UMyViewModel, GetHPPercent)
UE_FIELD_NOTIFICATION_IMPLEMENTATION_END(UMyViewModel);

```

### Blueprint Effect:

These control properties can now be bound to the properties within the ViewModel.



## Instanced

- **Function Description:** Specifies that editing assignments to the properties of this object should create a new instance as a child object, rather than searching for an existing object reference.
- **Metadata Type:** bool
- **Engine Module:** Instance
- **Restriction Type:** UObject\*
- **Functionality Mechanism:** Include CPF\_PersistentInstance, CPF\_ExportObject, CPF\_InstancedReference in PropertyFlags, and EditInline in Meta
- **Commonly Used:** ★★★

Specifies that editing assignments to the properties of this object should create a new instance as a child object, rather than searching for an existing object reference.

- The effect of Instanced on a single attribute and DefaultToInstanced on UCLASS are somewhat similar, with the difference being that the former applies only to individual attributes, while the latter applies to all attributes of the class type.

- Often used in conjunction with EditInlineNew, allowing new instances to be created and edited for object properties within the details panel.
- Instanced implies EditInline and Export capabilities.

When setting a value for an Object\* attribute, if not marked with Instanced, only an object reference can be assigned. To create an actual object instance in the editor and assign it to the attribute, the Instanced marker must be added. However, Instanced alone is insufficient; the class also needs to include EditInlineNew to appear in the list of classes that can create new instances.

Of course, manually setting an object to this attribute in C++ is still possible. It is important to distinguish this from UCLASS(DefaultToInstanced), which indicates that all attributes of this class are Instanced by default, thus avoiding the need to manually set each attribute every time.

## Sample Code:

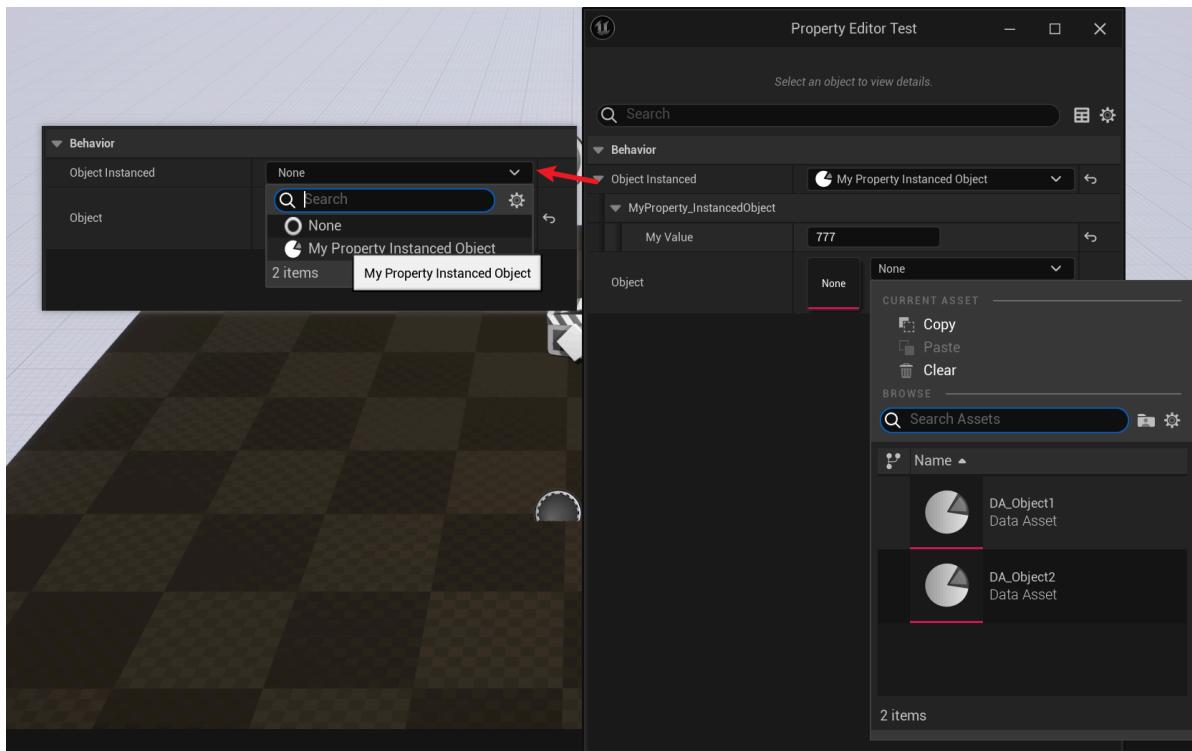
```
UCLASS(Blueprintable, BlueprintType, editinlinenew)
class INSIDER_API UMyProperty_InstancedObject :public UDataAsset
{
public:
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 MyValue = 123;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Instanced :public UObject
{
public:
    GENERATED_BODY()
    UMyProperty_Instanced(const FObjectInitializer& ObjectInitializer =
FObjectInitializer::Get());
public:
    //PropertyFlags:    CPF_Edit | CPF_BlueprintVisible | CPF_ExportObject | 
CPF_ZeroConstructor | CPF_InstancedReference | CPF_NoDestructor | 
CPF_PersistentInstance | CPF_HasGetValueTypeHash | 
CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced, Category = Behavior)
        UMyProperty_InstancedObject* ObjectInstanced;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Behavior)
        UMyProperty_InstancedObject* Object;
};
```

## Example Effect:

It is evident that the editing dialog boxes for ObjectInstanced and Object are different.



## NonTransactional

- **Function Description:** Any modifications made to this attribute will not be incorporated into the editor's Undo/Redo commands.
- **Metadata Type:** bool
- **Engine Module:** Editor
- **Action Mechanism:** Include CPF\_NonTransactional in PropertyFlags
- **Common Usage:** ★★

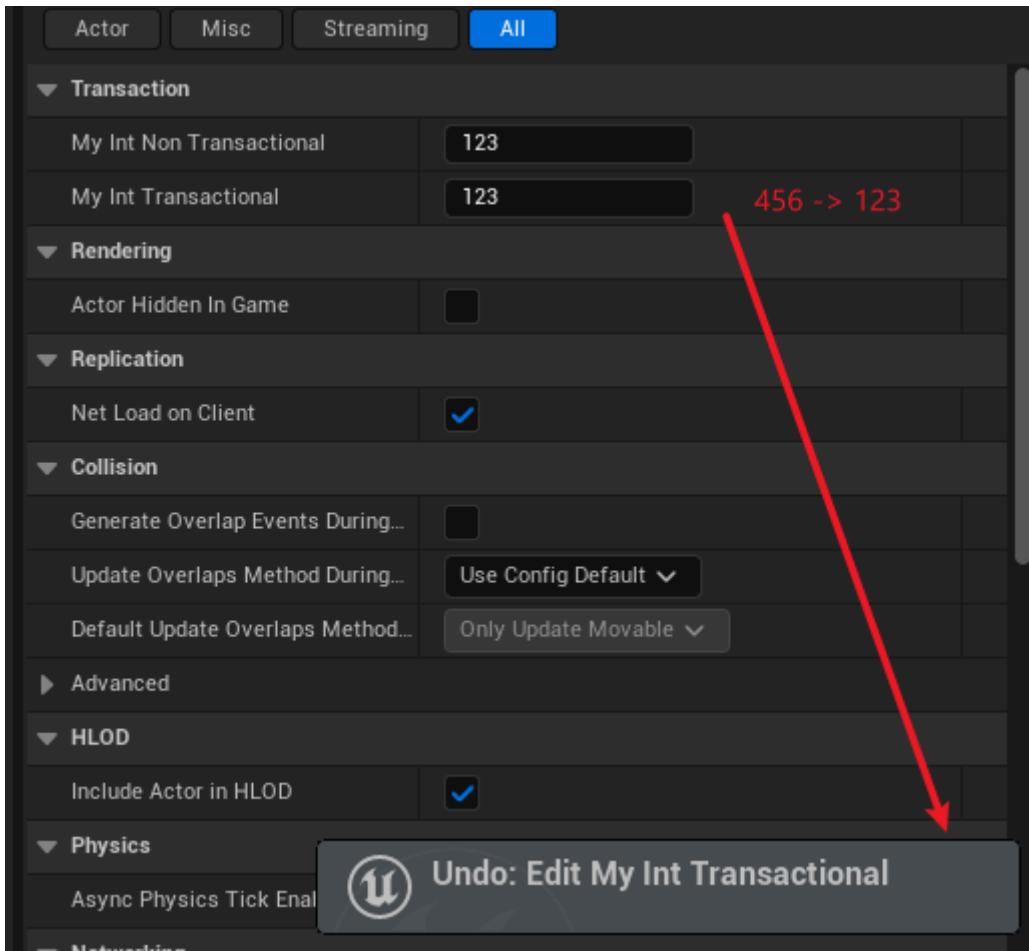
Changes to this attribute cannot be reversed using Ctrl+Z or redone using Ctrl+Y within the editor. This applies to Class Defaults in both Actors and Blueprints.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_Transaction :public AActor
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, NonTransactional, Category = Transaction)
        int32 MyInt_NonTransactional = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Transaction)
        int32 MyInt_Transactional = 123;
};
```

# Blueprint Performance:

Inputs on MyInt\_Transactional can be undone, whereas inputs on MyInt\_NonTransactional cannot be undone with Ctrl+Z.



## Category

- **Function description:** Specify the category of an attribute using | operators to define nested categories.
- **Metadata type:** strings = "a|b|c"
- **Engine module:** DetailsPanel, Editor
- **Action mechanism:** Add Category to the Meta
- **Commonly used:** ★★★★☆

Specify the category of an attribute using | operators to define nested categories.

## Sample Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Test :public UObject
{
    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_SimpleDisplay | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, SimpleDisplay, Category = Display)
        int32 MyInt_SimpleDisplay = 123;
```

```

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
CPF_NoDestructor | CPF_AdvancedDisplay | CPF_HasGetValueTypeHash | 
CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, AdvancedDisplay, Category = Display)
        int32 MyInt_AdvancedDisplay = 123;
public:
    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, Category = Edit)
        int32 MyInt_EditAnywhere = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | 
CPF_DisableEditOnInstance | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditDefaultsOnly, Category = Edit)
        int32 MyInt_EditDefaultsOnly = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | 
CPF_DisableEditOnTemplate | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditInstanceOnly, Category = Edit)
        int32 MyInt_EditInstanceOnly = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_EditConst | 
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | 
CPF_NativeAccessSpecifierPublic
    UPROPERTY(VisibleAnywhere, Category = Edit)
        int32 MyInt_VisibleAnywhere = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | 
CPF_DisableEditOnInstance | CPF_EditConst | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(VisibleDefaultsOnly, Category = Edit)
        int32 MyInt_VisibleDefaultsOnly = 123;
}

```

## Example Effect:



## Fundamental Principle:

The process is relatively simple, involving setting the value to the Category in the meta and then retrieving it for use.

## SimpleDisplay

- **Function Description:** Directly visible in the Details Panel without being collapsed into the Advanced section.
- **Metadata Type:** bool
- **Engine Module:** DetailsPanel, Editor
- **Action Mechanism:** Include CPF\_SimpleDisplay in PropertyFlags
- **Common Usage:** ★★★

Visible directly in the Details Panel without being collapsed into the Advanced section.

By default, it remains uncollapsed, but it can be used to override the AdvancedClassDisplay setting on the class. See the code and effects of AdvancedClassDisplay for specifics.

## Sample Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Test :public UObject
{
    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_SimpleDisplay | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, SimpleDisplay, Category = Display)
        int32 MyInt_SimpleDisplay = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_AdvancedDisplay | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, AdvancedDisplay, Category = Display)
        int32 MyInt_AdvancedDisplay = 123;
}
```

## Example Effect:



## Principle:

If CPF\_SimpleDisplay is present, then bAdvanced = false

```

void FPropertyNode::InitNode(const FPropertyParams& InitParams)
{
    // Property is advanced if it is marked advanced or the entire class is
    // advanced and the property not marked as simple
    static const FName Name_AdvancedClassDisplay("AdvancedClassDisplay");
    bool bAdvanced = Property.IsValid() ? (Property-
>HasAnyPropertyParams(CPF_AdvancedDisplay) || (!Property->HasAnyPropertyParams(
CPF_SimpleDisplay) && Property->GetOwnerClass() && Property->GetOwnerClass()->GetBoolMetaData(Name_AdvancedClassDisplay) ) ) : false;
}

```

## AdvancedDisplay

- Function Description:** Collapsed under the Advanced section and requires manual expansion. Typically used for attributes that are not frequently accessed.
- Metadata Type:** bool
- Engine Module:** DetailsPanel, Editor
- Mechanism of Action:** Include CPF\_AdvancedDisplay inPropertyParams
- Commonly Used:** ★★★★☆

Collapsed under the Advanced section and requires manual expansion. Typically used for attributes that are not frequently accessed.

## Sample Code:

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Test :public UObject
{
    //PropertyParams: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_SimpleDisplay | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, SimpleDisplay, Category = Display)
    int32 MyInt_SimpleDisplay = 123;

    //PropertyParams: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_AdvancedDisplay | CPF_HasGetValueTypeHash | 
    CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, AdvancedDisplay, Category = Display)
    int32 MyInt_AdvancedDisplay = 123;
}

```

## Example Effect:



# Principle:

If CPF\_AdvancedDisplay is set, then bAdvanced = true

```
void FPropertyNode::InitNode(const FPropertyParamsInitParams& InitParams)
{
    // Property is advanced if it is marked advanced or the entire class is
    // advanced and the property not marked as simple
    static const FName Name_AdvancedClassDisplay("AdvancedClassDisplay");
    bool bAdvanced = Property.IsValid() ? (Property-
>HasAnyPropertyParams(CPF_AdvancedDisplay) || (!Property->HasAnyPropertyParams(
CPF_SimpleDisplay) && Property->GetOwnerClass() && Property->GetOwnerClass()-
>GetBoolMetaData(Name_AdvancedClassDisplay)) ) : false;

}
```

# EditAnywhere

- **Function description:** Editable in both the default value and the instance's detail panel
- **Metadata type:** bool
- **Engine module:** DetailsPanel, Editor
- **Action mechanism:** Include CPF\_Edit inPropertyParams
- **Commonly used:** ★★★★☆

Editable in both the default value and the instance's detail panel.

# Sample Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Test :public UObject
{
public:
    //PropertyParams: CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor | CPF_DisableEditOnInstance | CPF_DisableEditOnTemplate | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, Category = Edit)
    int32 MyInt_EditAnywhere = 123;

    //PropertyParams: CPF_Edit | CPF_ZeroConstructor | CPF_DisableEditOnInstance | CPF_DisableEditOnTemplate | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditDefaultsOnly, Category = Edit)
    int32 MyInt_EditDefaultsonly = 123;

    //PropertyParams: CPF_Edit | CPF_ZeroConstructor | CPF_DisableEditOnInstance | CPF_DisableEditOnTemplate | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditInstanceOnly, Category = Edit)
    int32 MyInt_EditInstanceOnly = 123;

    //PropertyParams: CPF_Edit | CPF_ZeroConstructor | CPF_EditConst | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
```

```

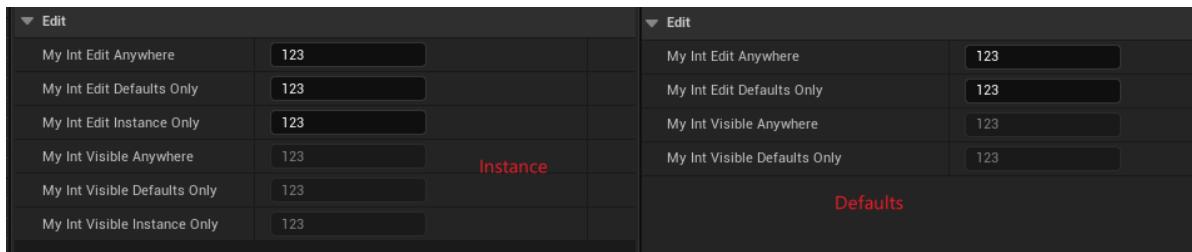
UPROPERTY(VisibleAnywhere, Category = Edit)
    int32 MyInt_VisibleAnywhere = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor |
CPF_DisableEditOnInstance | CPF_EditConst | CPF_IsPlainOldData | CPF_NoDestructor
| CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UPROPERTY(VisibleDefaultsOnly, Category = Edit)
    int32 MyInt_VisibleDefaultsOnly = 123;

    //PropertyFlags: CPF_Edit | CPF_ZeroConstructor |
CPF_DisableEditOnTemplate | CPF_EditConst | CPF_IsPlainOldData | CPF_NoDestructor
| CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UPROPERTY(VisibleInstanceOnly, Category = Edit)
    int32 MyInt_VisibleInstanceOnly = 123;
}

```

## Example Effect:



## Fundamental Principle:

CPF\_Edit is extensively used in the source code, determining the visibility and editability of attributes in numerous locations. Those interested can search for the usage of CPF\_Edit on their own.

## EditDefaultsOnly

- **Function Description:** Editing is restricted to the default value panel only
- **Metadata Type:** bool
- **Engine Modules:** DetailsPanel, Editor
- **Action Mechanism:** CPF\_Edit and CPF\_DisableEditOnInstance are included in PropertyFlags
- **Commonly Used:** ★★★★☆

Refer also to the sample code and effects in EditAnywhere for further examples.

## EditInstanceOnly

- **Function description:** This property can only be modified within the instance's detail panel
- **Metadata type:** boolean
- **Engine module:** DetailsPanel, Editor
- **Action mechanism:** CPF\_Edit and CPF\_DisableEditOnTemplate are included in PropertyFlags
- **Commonly used:** ★★★★☆

Refer also to the sample code and effects demonstrated in EditAnywhere.

# VisibleAnywhere

---

- **Function Description:** Visible in both the default value and instance detail panels, but not editable
- **Metadata Type:** bool
- **Engine Modules:** DetailsPanel, Editor
- **Action Mechanism:** CPF\_Edit and CPF\_EditConst are included in PropertyFlags
- **Commonly Used:** ★★★★

Refer also to the sample code and effects in EditAnywhere.

# VisibleDefaultsOnly

---

- **Function Description:** Visible in the default details panel, but not editable
- **Metadata Type:** bool
- **Engine Modules:** DetailsPanel, Editor
- **Action Mechanism:** CPF\_Edit and CPF\_DisableEditOnInstance are included in PropertyFlags
- **Common Usage:** ★★★★

Refer also to the sample code and effects in EditAnywhere.

# VisibleInstanceOnly

---

- **Function description:** Visible in the instance details panel but not editable
- **Metadata type:** bool
- **Engine module:** DetailsPanel, Editor
- **Action mechanism:** Included CPF\_Edit and CPF\_DisableEditOnTemplate in PropertyFlags
- **Commonly used:** ★★★★

Refer also to the sample code and effects in EditAnywhere.

# EditFixedSize

---

- **Function Description:** The number of elements in this container cannot be changed within the details panel.
- **Metadata Type:** bool
- **Engine Modules:** DetailsPanel, Editor
- **Restriction Types:** TArray, TSet, TMap
- **Action Mechanism:** Add CPF\_EditFixedSize to PropertyFlags
- **Commonly Used:** ★★

Changing the number of elements in this container is not allowed in the details panel.

Only applicable to containers. This prevents users from altering the number of elements in the container through the Unreal Editor's property window.

However, it can still be modified in C++ code and blueprints.

## Example Code:

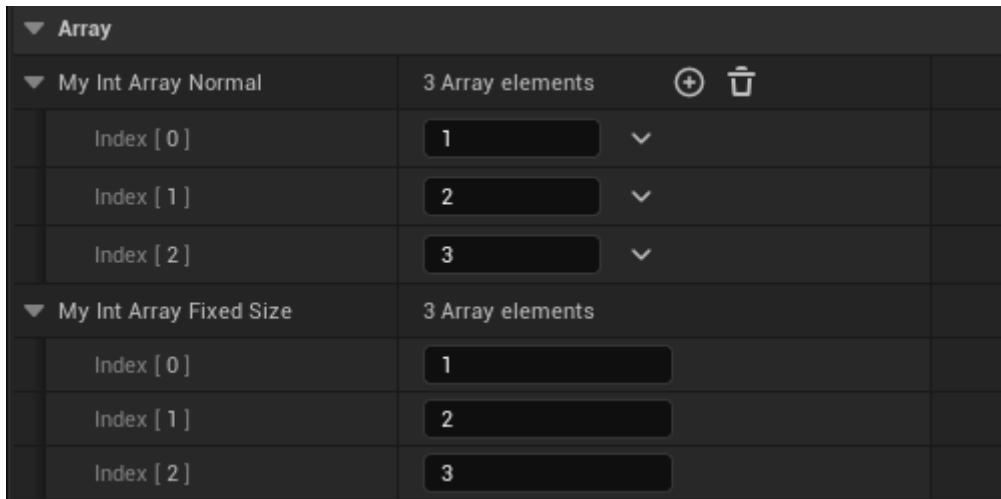
Consider TArray as an example; the same principle applies to others.

```
UPROPERTY(EditAnywhere, Category = Array)
TArray<int32> MyIntArray_Normal{1,2,3};

UPROPERTY(EditAnywhere, EditFixedSize,Category = Array)
TArray<int32> MyIntArray_FixedSize{1,2,3};
```

## Example Effect:

In blueprints, the former allows for dynamic addition of elements, whereas the latter does not.



## Principle:

If there is CPF\_EditFixedSize , + and empty buttons will not be added.

```
void PropertyEditorHelpers::GetRequiredPropertyButtons( TSharedRef<FPropertyName>
PropertyNode, TArray<EPropertyButton::Type>& OutRequiredButtons, bool
bUsingAssetPicker )
{
    // Handle a container property.
    if( NodeProperty->IsA(FArrayProperty::StaticClass()) || NodeProperty-
>IsA(F SetProperty::StaticClass()) || NodeProperty-
>IsA(FMapProperty::StaticClass()) )
    {
        if( !(NodeProperty->PropertyFlags & CPF_EditFixedSize) )
        {
            OutRequiredButtons.Add( EPropertyButton::Add );
            OutRequiredButtons.Add( EPropertyButton::Empty );
        }
    }
}
```

# NoClear

- **Function Description:** Specifies that the Clear button should not appear in the editing options for this attribute, and setting it to null is not permitted.
- **Metadata Type:** bool
- **Engine Module:** DetailsPanel, Editor
- **Restriction Type:** Reference type
- **Action Mechanism:** Include CPF\_NoClear in PropertyFlags
- **Common Usage:** ★★★

The Clear button will not appear in the editing options for this property.

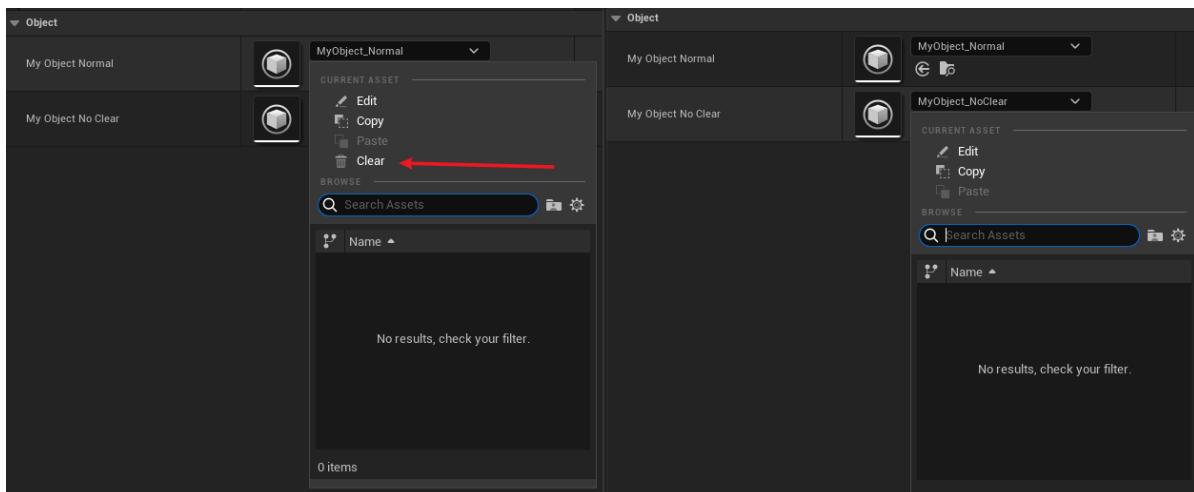
It prevents users from setting this Object reference to null within the editor panel. However, it can also be applied to other structures representing a reference type, such as FPrimaryAssetId, FInstancedStruct, FDataRegistryType, etc.

## Sample Code:

```
UPROPERTY(EditAnywhere, Category = Object)
class UMyClass_Default* MyObject_Normal;
//PropertyFlags: CPF_Edit | CPF_ZeroConstructor | CPF_NoClear |
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
UPROPERTY(EditAnywhere, NoClear, Category = Object)
class UMyClass_Default* MyObject_NoClear;

//Constructor Assignment:
MyObject_Normal = CreateDefaultSubobject<UMyClass_Default>("MyObject_Normal");
MyObject_NoClear = CreateDefaultSubobject<UMyClass_Default>("MyObject_NoClear");
```

## Example Effect:



## Principle:

CPF\_NoClear is extensively utilized within the engine.

```
const bool bAllowClear = !StructPropertyHandle->GetMetaDataProperty()->HasAnyPropertyFlags(CPF_NoClear);
```

# Config

- **Function description:** Specifies that this property is a configuration property, which can be serialized, read and written to ini file (the path is specified by the config tag of uclass ).
- **Metadata type:** bool
- **Engine module:** Config
- **Action mechanism:** CPF\_Config
- **Commonly used:** ★★★

Specifies that this property is a configuration property that can be serialized, read and written to the ini file (the path is specified by the config tag of uclass).

It will be automatically loaded from the INI file upon loading. If no write marker is added, the property will implicitly be set to ReadOnly.

See sample code and effects of the config tag in UCLASS.

# GlobalConfig

- **Function Description:** Similar to Config, this attribute can be designated for reading and writing as a configuration in INI files, but only the values of the base class specified in the configuration file will be read and written, not those of the subclass.
- **Metadata Type:** bool
- **Engine Module:** Config
- **Action Mechanism:** Include CPF\_GlobalConfig in PropertyFlags
- **Common Usage:** ★★★

Like Config, this property can be specified for reading and writing as a configuration in INI files, but only the values of the base class specified in the configuration file will be read, not those of the subclass.

However, the difference lies in the fact that during LoadConfig, this attribute will only read the base class's INI, not the subclass's INI. Since only the INI settings of the base class take effect, it is as if there is only one global configuration in effect, hence the name GlobalConfig.

## Sample Code:

```
UCLASS(Config = MyOtherGame)
class INSIDER_API UMyProperty_Config :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Config)
    int32 MyPropertyWithConfig = 123;
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, GlobalConfig)
int32 MyPropertywithGlobalConfig = 123;
};

UCLASS(Config = MyOtherGame)
class INSIDER_API UMyProperty_Config_Child :public UMyProperty_Config
{
    GENERATED_BODY()
public:
};

void UMyProperty_Config_Test::TestConfigSave()
{
    FString fileName = FPaths::ProjectConfigDir() / TEXT("MyOtherGame.ini");
    fileName = FConfigCacheIni::NormalizeConfigIniPath(fileName);

    {
        UMyProperty_Config* testObject = NewObject<UMyProperty_Config>
        (GetTransientPackage(), TEXT("testObject"));

        testObject->MyProperty = 777;
        testObject->MyPropertyWithConfig = 777;
        testObject->MyPropertywithGlobalConfig = 777;

        testObject->SaveConfig(CPF_Config, *fileName);
    }

    {
        UMyProperty_Config_Child* testObject =
        NewObject<UMyProperty_Config_Child>(GetTransientPackage(),
        TEXT("testObjectChild"));

        testObject->MyProperty = 888;
        testObject->MyPropertyWithConfig = 888;
        testObject->MyPropertywithGlobalConfig = 888;

        testObject->SaveConfig(CPF_Config, *fileName);
    }
}

void UMyProperty_Config_Test::TestConfigLoad()
{
    FString fileName = FPaths::ProjectConfigDir() / TEXT("MyOtherGame.ini");
    fileName = FConfigCacheIni::NormalizeConfigIniPath(fileName);

    UMyProperty_Config* testObject = NewObject<UMyProperty_Config>
    (GetTransientPackage(), TEXT("testObject"));
    testObject->LoadConfig(nullptr, *fileName);

    UMyProperty_Config_Child* testObjectChild =
    NewObject<UMyProperty_Config_Child>(GetTransientPackage(),
    TEXT("testObjectChild"));
    testObjectChild->LoadConfig(nullptr, *fileName);
}

```

## Example Effect:

After TestConfigSave, MyPropertyWithGlobalConfig=888, indicating that the value is saved only on the base class.

```
[/Script/Insider.MyProperty_Config]
MyPropertywithConfig=777
MyPropertywithGlobalConfig=888

[/Script/Insider.MyProperty_Config_Child]
MyPropertywithConfig=888
```

For testing purposes, if the value in the configuration is manually changed to:, then the TestConfigLoad test is performed

```
[/Script/Insider.MyProperty_Config]
MyPropertywithConfig=777
MyPropertywithGlobalConfig=888

[/Script/Insider.MyProperty_Config_Child]
MyPropertywithConfig=888
MyPropertywithGlobalConfig=999
```

Display Result:

It can be observed that the value of testObjectChild does not use the value of 999 under MyProperty\_Config\_Child in the INI file, but remains 888.

testObject	0x00000693ee020e40 (Name="testObject", InternalFlags=ReachabilityFlag0 (1))	UMyProperty_Config *
↳ UObject	(Name="testObject", InternalFlags=ReachabilityFlag0 (1))	UObject
↳ MyProperty	123	int
↳ MyPropertyWithConfig	777	int
↳ MyPropertyWithGlobalConfig	888	int
↳ testObjectChild	0x00000693ee020d80 (Name="testObjectChild", InternalFlags=ReachabilityFlag0 (1))	UMyProperty_Config_Child *
↳ UMyProperty_Config	(Name="testObjectChild", InternalFlags=ReachabilityFlag0 (1))	UMyProperty_Config
↳ UObject	(Name="testObjectChild", InternalFlags=ReachabilityFlag0 (1))	UObject
↳ MyProperty	123	int
↳ MyPropertyWithConfig	888	int
↳ MyPropertyWithGlobalCon...	888	int

## Working Principle:

If bGlobalConfig is set, the base class's settings will be adopted.

```
void UObject::LoadConfig( UClass* ConfigClass /*=NULL*/, const TCHAR* InFilename /*=NULL*/, uint32 PropagationFlags /*=LCPF_None*/, FProperty* PropertyToLoad /*=NULL*/ )
{
    const bool bGlobalConfig = (Property->PropertyFlags & CPF_GlobalConfig) != 0;
    UClass* OwnerClass = Property->GetOwnerClass();
```

```

UClass* BaseClass = bGlobalConfig ? ownerClass : ConfigClass;
if ( !bPerObject )
{
    ClassSection = BaseClass->GetPathName();
    LongCommitName = BaseClass->GetOutermost()->GetFName();

    // allow the class to override the expected section name
    overrideConfigSection(ClassSection);
}

// globalconfig properties should always use the owning class's config
file
// specifying a value for InFilename will override this behavior (as it
does with normal properties)
const FString& PropFileName = (bGlobalConfig && InFilename == NULL) ?
OwnerClass->GetConfigName() : filename;
}

```

## BlueprintAuthorityOnly

- **Function Description:** Bound exclusively to events with BlueprintAuthorityOnly, ensuring the multicast delegate only responds to events executed on the server side
- **Metadata Type:** boolean
- **Engine Module:** Blueprint, Network
- **Restriction Type:** Multicast Delegates
- **Action Mechanism:** Include CPF\_BlueprintAuthorityOnly in the PropertyFlags
- **Common Usage:** ★★★

## Testing Code:

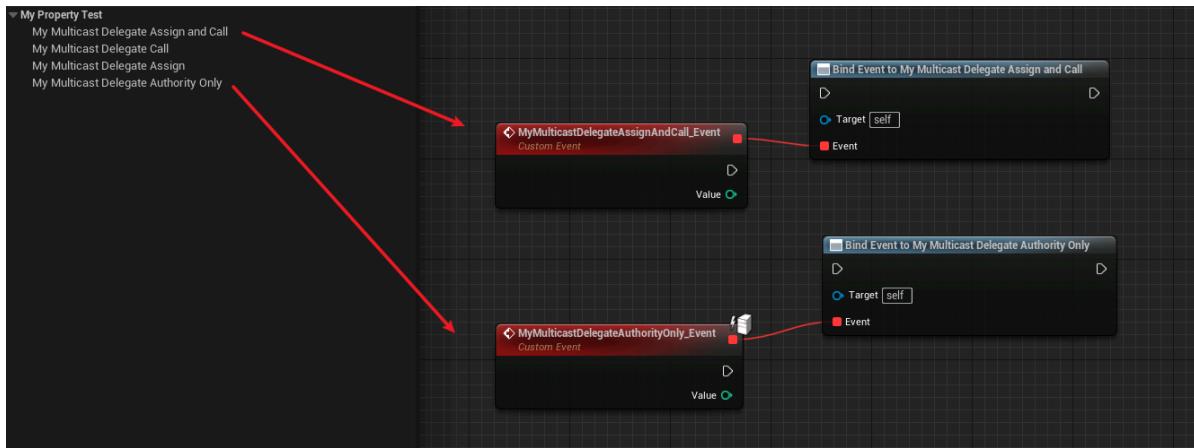
```

UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable,
BlueprintCallable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAssignAndCall;

UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable,
BlueprintCallable, BlueprintAuthorityOnly)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAuthorityOnly;

```

# Performance Manifestation in the Blueprint:



## BlueprintReadWrite

- Functional Description:** This property can be read from or written to within a Blueprint.
- Metadata type:** boolean
- Engine module:** Blueprint
- Action mechanism:** Include CPF\_BlueprintVisible in the PropertyFlags
- Commonly used:** ★★★★☆

This property can be read from or written to within a Blueprint.

This specifier is not compatible with the BlueprintReadOnly specifier.

## Sample Code:

```
public:  
    //PropertyFlags:    CPF_BlueprintVisible | CPF_ZeroConstructor |  
    CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |  
    CPF_NativeAccessSpecifierPublic  
    UPROPERTY(BlueprintReadWrite, Category = Blueprint)  
    int32 MyInt_Readwrite = 123;  
    //PropertyFlags:    CPF_BlueprintVisible | CPF_BlueprintReadOnly |  
    CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor |  
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic  
    UPROPERTY(BlueprintReadOnly, Category = Blueprint)  
    int32 MyInt_ReadOnly = 123;
```

## Example Effect:

Readable and writable in Blueprints:



## Fundamental Principle:

If CPF\_Edit or | CPF\_BlueprintVisible | CPF\_BlueprintAssignable is present, the property can be retrieved.

```
EPropertyAccessResultFlags PropertyAccessUtil::CanGetPropertyValue(const  
FProperty* InProp)  
{  
    if (!InProp->HasAnyPropertyFlags(CPF_Edit | CPF_BlueprintVisible |  
    CPF_BlueprintAssignable))  
    {  
        return EPropertyAccessResultFlags::PermissionDenied |  
    EPropertyAccessResultFlags::AccessProtected;  
    }  
  
    return EPropertyAccessResultFlags::Success;  
}
```

## BlueprintReadOnly

- **Functional Description:** This attribute can be read by Blueprints but is not modifiable.
- **Metadata Type:** bool
- **Engine Module:** Blueprint
- **Functionality Mechanism:** CPF\_BlueprintVisible and CPF\_BlueprintReadOnly are added to the PropertyFlags
- **Commonly Used:** ★★★★☆

This attribute can be read by Blueprints but is not modifiable. This specifier is incompatible with the BlueprintReadWrite specifier.

## Sample Code:

```
public:  
    //PropertyFlags:    CPF_BlueprintVisible | CPF_ZeroConstructor |  
    CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |  
    CPF_NativeAccessSpecifierPublic  
    UPROPERTY(BlueprintReadWrite, Category = Blueprint)  
        int32 MyInt_Readwrite = 123;  
    //PropertyFlags:    CPF_BlueprintVisible | CPF_BlueprintReadOnly |  
    CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor |  
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic  
    UPROPERTY(BlueprintReadOnly, Category = Blueprint)  
        int32 MyInt_ReadOnly = 123;
```

## Example Effect:

Specify the blueprint as read-only:



## Principle:

With CPF\_BlueprintVisible, access is granted

After adding CPF\_BlueprintReadOnly, modifications are not allowed.

```

EPropertyAccessResultFlags PropertyAccessUtil::CanGetPropertyValue(const
FProperty* InProp)
{
    if (!InProp->HasAnyPropertyFlags(CPF_Edit | CPF_BlueprintVisible |
CPF_BlueprintAssignable))
    {
        return EPropertyAccessResultFlags::PermissionDenied |
EPropertyAccessResultFlags::AccessProtected;
    }

    return EPropertyAccessResultFlags::Success;
}

FBlueprintEditorUtils::EPropertyWritableState
FBlueprintEditorUtils::IsPropertyWritableInBlueprint(const UBlueprint* Blueprint,
const FProperty* Property)
{
    if (Property)
    {
        if (!Property->HasAnyPropertyFlags(CPF_BlueprintVisible))
        {
            return EPropertyWritableState::NotBlueprintVisible;
        }
        if (Property->HasAnyPropertyFlags(CPF_BlueprintReadOnly))
        {
            return EPropertyWritableState::BlueprintReadOnly;
        }
        if (Property->GetBoolMetaData(FBlueprintMetadata::MD_Private))
        {
            const UClass* OwningClass = Property->GetOwnerChecked<UClass>();
            if (OwningClass->ClassGeneratedBy.Get() != Blueprint)
            {
                return EPropertyWritableState::Private;
            }
        }
    }
    return EPropertyWritableState::Writable;
}

```

# BlueprintGetter

- **Function Description:** Define a custom Get function for the attribute to read.
- **Metadata Type:** string="abc"
- **Engine Module:** Blueprint
- **Action Mechanism:** Include CPF\_BlueprintReadOnly and CPF\_BlueprintVisible in PropertyFlags, and add BlueprintGetter in Meta
- **Common Usage:** ★★★

Define a custom Get function for the attribute to read.

If BlueprintSetter or BlueprintReadWrite is not set, BlueprintReadOnly will be set by default, making this property read-only.

## Example Code:

```
public:  
    //((BlueprintGetter = , Category = Blueprint, ModuleRelativePath =  
    Property/MyProperty_Test.h)  
    UFUNCTION(BlueprintGetter, Category = Blueprint)      //or BlueprintPure  
    int32 MyInt_Getter()const { return MyInt_withGetter * 2; }  
  
    //((BlueprintSetter = , Category = Blueprint, ModuleRelativePath =  
    Property/MyProperty_Test.h)  
    UFUNCTION(BlueprintSetter, Category = Blueprint)      //or BlueprintCallable  
    void MyInt_Setter(int NewValue) { MyInt_withSetter = NewValue / 4; }  
private:  
    //((BlueprintGetter = MyInt_Getter, Category = Blueprint, ModuleRelativePath =  
    Property/MyProperty_Test.h)  
    //PropertyFlags:    CPF_BlueprintVisible | CPF_BlueprintReadOnly |  
    CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor |  
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPrivate  
    UPROPERTY(BlueprintGetter = MyInt_Getter, Category = Blueprint)  
    int32 MyInt_withGetter = 123;  
  
    //((BlueprintSetter = MyInt_Setter, Category = Blueprint, ModuleRelativePath =  
    Property/MyProperty_Test.h)  
    //PropertyFlags:    CPF_BlueprintVisible | CPF_ZeroConstructor |  
    CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |  
    CPF_NativeAccessSpecifierPrivate  
    UPROPERTY(BlueprintSetter = MyInt_Setter, Category = Blueprint)  
    int32 MyInt_withSetter = 123;
```

## Example Effect:

"MyInt\_WithGetter" is read-only.

"MyInt\_WithSetter" is readable and writable.



# Getter

- **Function Description:** Add a C++ Get function to the attribute, applicable only at the C++ level.
- **Metadata Type:** string = "abc"
- **Engine Module:** Blueprint
- **Commonality:** ★★★

Add a C++ Get function to the attribute, applicable only at the C++ level.

- Should a function name not be specified for the Getter, the default name of GetXXX will be used. Alternatively, a different function name can be provided.
- These Getter functions do not include the UFUNCTION decorator, which should be distinguished from BlueprintGetter.
- "NativeGetter" might be a more appropriate name.
- The GetXXX function must be manually implemented; otherwise, UHT will generate an error.
- Of course, we can also manually write GetSet functions without specifying Getter and Setter metadata. However, the benefit of including Getters and Setters is that if reflection is used to access and modify values elsewhere in the project, and if Getters and Setters are not marked, the values will be accessed directly from the properties, bypassing the custom Get/Set process we intend to implement.

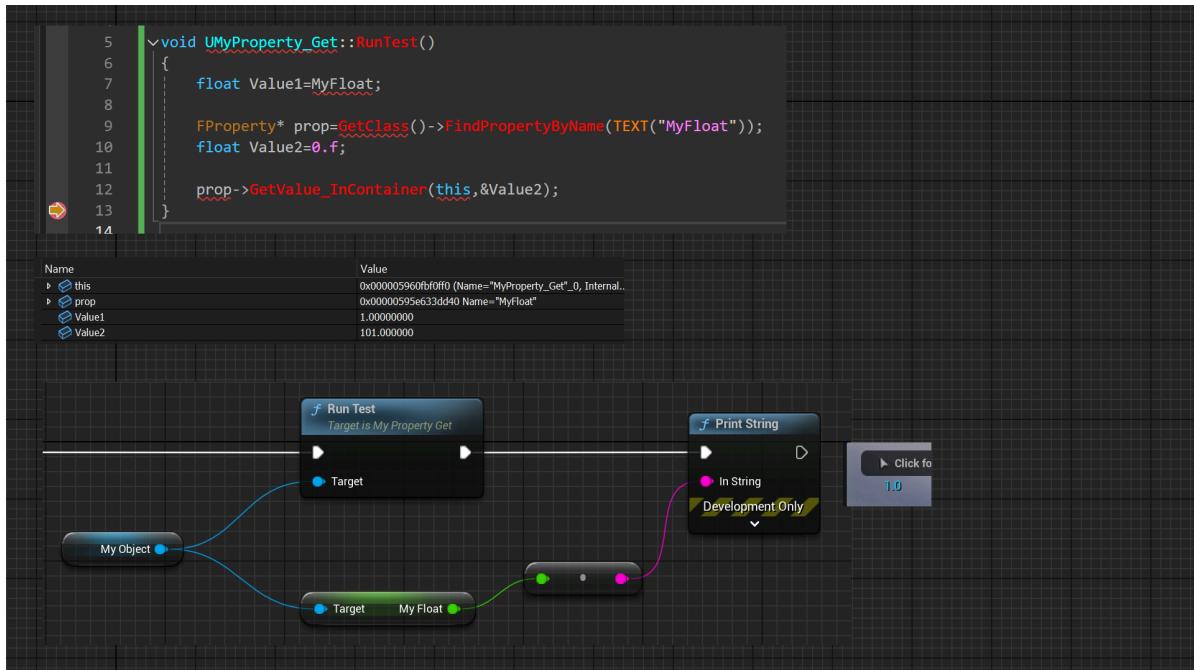
## Test Code:

```
public:  
    //GetterFunc: Has Native Getter  
    UPROPERTY(BlueprintReadWrite, Getter)  
    float MyFloat = 1.0f;  
  
    //GetterFunc: Has Native Getter  
    UPROPERTY(BlueprintReadWrite, Getter = GetMyCustomFloat)  
    float MyFloat2 = 1.0f;  
public:  
    float GetMyFloat()const { return MyFloat + 100.f; }  
  
    float GetMyCustomFloat()const { return MyFloat2 + 100.f; }  
  
void UMyProperty_Get::RunTest()  
{  
    float value1=MyFloat;  
  
    FProperty* prop=GetClass()->FindPropertyByName(TEXT("MyFloat"));  
    float value2=0.f;  
  
    prop->GetValue_InContainer(this,&value2);  
}
```

# Blueprint Performance:

During testing, it is evident that if the GetValue\_InContainer reflection method is used to retrieve the value, it will automatically invoke GetMyFloat, resulting in different values being returned.

Directly calling Get MyFloat in the blueprint still results in 1.



## Principle:

UHT, after analyzing the Getter marker, generates the corresponding function wrapper in gen.cpp. During the construction of FProperty, TPropertyWithSetterAndGetter is created, and CallGetter is invoked when GetSingleValue\_InContainer is called.

```
void UMyProperty_Get::GetMyFloat_WrapperImpl(const void* object, void* outValue)
{
    const UMyProperty_Get* Obj = (const UMyProperty_Get*)object;
    float& Result = *(float*)outValue;
    Result = (float)Obj->GetMyFloat();
}

const UECodeGen_Private::FFloatPropertyParams
Z_Construct_UClass_UMyProperty_Get_Statics::NewProp_MyFloat = { "MyFloat",
nullptr, (EPropertyFlags)0x0010000000000004,
UECodeGen_Private::EPropertyGenFlags::Float,
RF_Public|RF_Transient|RF_MarkAsNative, nullptr,
&UMyProperty_Get::GetMyFloat_WrapperImpl, 1, STRUCT_OFFSET(UMyProperty_Get,
MyFloat), METADATA_PARAMS(UE_ARRAY_COUNT(NewProp_MyFloat_MetaData),
NewProp_MyFloat_MetaData) };

template <typenamePropertyParamsType, typenamePropertyParamsType>
PropertyParams* NewFProperty(FFieldVariant Outer, constPropertyParamsBase&
PropBase)
{
    constPropertyParamsType& Prop = (constPropertyParamsType&)PropBase;
   PropertyParams* NewProp = nullptr;
```

```

    if (Prop.SetterFunc || Prop.GetterFunc)
    {
        NewProp = new TPropertyWithSetterAndGetter<.PropertyType>(Outer, Prop);
    }
    else
    {
        NewProp = new PropertyType(Outer, Prop);
    }
}

void FProperty::GetSinglevalue_InContainer(const void* InContainer, void*
OutValue, int32 ArrayIndex) const
{
    checkf(ArrayIndex <= ArrayDim, TEXT("ArrayIndex (%d) must be less than the
property %s array size (%d)", ArrayIndex, *GetFullName(), ArrayDim));
    if (!HasGetter())
    {
        // Fast path - direct memory access
        CopySinglevalue(OutValue,
ContainerVoidPtrToValuePtrInternal((void*)InContainer, ArrayIndex));
    }
    else
    {
        if (ArrayDim == 1)
        {
            // Slower but no mallocs. We can copy the value directly to the
resulting param
            CallGetter(InContainer, OutValue);
        }
        else
        {
            // Malloc a temp value that is the size of the array. Getter will
then copy the entire array to the temp value
            uint8* valueArray = (uint8*)AllocateAndInitializeValue();
            GetValue_InContainer(InContainer, valueArray);
            // Copy the item we care about and free the temp array
            CopySinglevalue(OutValue, valueArray + ArrayIndex * Elementsize);
            DestroyAndFreevalue(valueArray);
        }
    }
}

```

## Setter

- **Function Description:** Adds a C++ Set function to the attribute, applicable solely at the C++ level.
- **Metadata Type:** string="abc"
- **Engine Module:** Blueprint
- **Associated Items:** Getter
- **Usage Frequency:** ★★★

Adds a C++ Set function to the attribute, applicable solely at the C++ level.

- If no function name is provided for the Getter, the default name SetXXX will be used.  
Alternatively, a different function name can be specified.
- These Getter functions are not marked with UFUNCTION, differing from BlueprintGetters.
- A more fitting name might be NativeSetter.
- The SetXXX function must be manually written; otherwise, UHT will generate an error.
- We can also manually write GetSet functions without specifying Getter and Setter metadata.  
The benefit of explicitly defining Getters and Setters is that if reflection is used elsewhere in the project to access or modify values, the absence of these markings would result in direct property access, bypassing the custom Get/Set logic we intend to use.

## Test Code:

```

public:
    UPROPERTY(BlueprintReadWrite, Setter)
    float MyFloat = 1.0f;

    UPROPERTY(BlueprintReadWrite, Setter = SetMyCustomFloat)
    float MyFloat2 = 1.0f;
public:
    void SetMyFloat(float val) { MyFloat = val + 100.f; }
    void SetMyCustomFloat(float val) { MyFloat2 = val + 100.f; }

public:
    UFUNCTION(BlueprintCallable)
    void RunTest();
};

void UMyProperty_Set::RunTest()
{
    float OldValue=MyFloat;

    FProperty* prop=GetClass()->FindPropertyByName(TEXT("MyFloat"));
    const float Value2=20.f;

    prop->SetValue_InContainer(this,&Value2);

    float NewValue=MyFloat;
}

```

## Blueprint Behavior:

During testing, it is observed that using the reflection method SetValue\_InContainer to set a value automatically invokes SetMyFloat, effectively setting a different value.

```

5     void UMyProperty_Get::RunTest()
6     {
7         float Value1=MyFloat;
8
9         FProperty* prop=GetClass()->FindPropertyByName(TEXT("MyFloat"));
10        float Value2=0.f;
11
12        prop->GetValue_InContainer(this,&Value2);
13    }
14

```

Name Value

- this 0x000005960fbff0 (Name="MyProperty\_Get\_0", Internal..)
- prop 0x00000595e63dd40 Name="MyFloat"
- Value1 1.0000000
- Value2 10.000000

```

15    void UMyProperty_Set::RunTest()
16    {
17        float OldValue=MyFloat;
18
19        FProperty* prop=GetClass()->FindPropertyByName(TEXT("MyFloat"));
20        const float Value2=20.f;
21
22        prop->SetValue_InContainer(this,&Value2);
23
24        float NewValue=MyFloat;
25
26    }
27

```

Name Value

- this 0x000006d30d90aa0 (Name="MyProperty\_Set\_0", Internal..)
- NewValue 120.000000
- OldValue 1.0000000
- prop 0x000006d2f034e1a0 Name="MyFloat"
- Value2 20.000000

## Principle:

Upon analyzing Setter annotations, UHT generates the corresponding function wrappers in gen.cpp. When constructing FProperty, TPropertyWithSetterAndGetter is created, and during GetSingleValue\_InContainer, CallGetter is invoked.

```

void UMyProperty_Set::SetMyFloat_WrapperImpl(void* Object, const void* InValue)
{
    UMyProperty_Set* Obj = (UMyProperty_Set*)Object;
    float& value = *(float*)InValue;
    Obj->SetMyFloat(value);
}

```

```

const UECodeGen_Private::FFloatPropertyParams
Z_Construct_UClass_UMyProperty_Set_Statics::NewProp_MyFloat = { "MyFloat",
nullptr, (EPropertyFlags)0x0010000000000004,
UECodeGen_Private::EPropertyGenFlags::Float,
RF_Public|RF_Transient|RF_MarkAsNative, &UMyProperty_Set::SetMyFloat_WrapperImpl,
nullptr, 1, STRUCT_OFFSET(UMyProperty_Set, MyFloat),
METADATA_PARAMS(UE_ARRAY_COUNT(NewProp_MyFloat_MetaData),
NewProp_MyFloat_MetaData) };

template <typename PropertyType, typenamePropertyParamsType>
.PropertyType* NewFProperty(FFieldVariant Outer, const FPropertyParamsBase&
PropBase)
{
    constPropertyParamsType& Prop = (constPropertyParamsType&)PropBase;
    PropertyType* NewProp = nullptr;

    if (Prop.SetterFunc || Prop.GetterFunc)
    {
        NewProp = new TPropertyWithSetterAndGetter<.PropertyType>(Outer, Prop);
    }
    else
    {
        NewProp = new PropertyType(Outer, Prop);
    }
}

void FProperty::setSingleValue_InContainer(void* OutContainer, const void*
InValue, int32 ArrayIndex) const
{
    checkf(ArrayIndex <= ArrayDim, TEXT("ArrayIndex (%d) must be less than the
property %s array size (%d)"), ArrayIndex, *GetFullName(), ArrayDim);
    if (!HasSetter())
    {
        // Fast path - direct memory access
        CopySinglevalue(ContainerVoidPtrToValuePtrInternal((void*)OutContainer,
ArrayIndex), InValue);
    }
    else
    {
        if (ArrayDim == 1)
        {
            // Slower but no mallocs. We can copy the value directly to the
resulting param
            CallSetter(OutContainer, InValue);
        }
        else
        {
            // Malloc a temp value that is the size of the array. We will then
copy the entire array to the temp value
            uint8* ValueArray = (uint8*)AllocateAndInitializeValue();
            GetValue_InContainer(OutContainer, ValueArray);
            // Replace the value at the specified index in the temp array with
the InValue
            Copysinglevalue(ValueArray + ArrayIndex * ElementSize, InValue);
            // Now call a setter to replace the entire array and then destroy the
temp value
        }
    }
}

```

```

        CallSetter(OutContainer, valueArray);
        DestroyAndFreeValue(valueArray);
    }
}

```

## BlueprintSetter

- **Function Description:** Utilize a custom set function for reading.
- **Metadata Type:** string="abc"
- **Engine Module:** Blueprint
- **Action Mechanism:** Include CPF\_BlueprintVisible in PropertyFlags and BlueprintSetter in Meta
- **Common Usage:** ★★★

Utilize a custom set function for reading.

BlueprintReadWrite is set by default.

## Test Code:

```

public:
    //((BlueprintGetter = , Category = Blueprint, ModuleRelativePath =
Property/MyProperty_Test.h)
    UFUNCTION(BlueprintGetter, Category = Blueprint)      //or BlueprintPure
        int32 MyInt_Getter()const { return MyInt_withGetter * 2; }

    //((BlueprintSetter = , Category = Blueprint, ModuleRelativePath =
Property/MyProperty_Test.h)
    UFUNCTION(BlueprintSetter, Category = Blueprint)      //or BlueprintCallable
        void MyInt_Setter(int NewValue) { MyInt_withSetter = NewValue / 4; }

private:
    //((BlueprintGetter = MyInt_Getter, Category = Blueprint, ModuleRelativePath =
Property/MyProperty_Test.h)
    //PropertyFlags:   CPF_BlueprintVisible | CPF_BlueprintReadOnly |
CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor |
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPrivate
    UPROPERTY(BlueprintGetter = MyInt_Getter, Category = Blueprint)
        int32 MyInt_withGetter = 123;

    //((BlueprintSetter = MyInt_Setter, Category = Blueprint, ModuleRelativePath =
Property/MyProperty_Test.h)
    //PropertyFlags:   CPF_BlueprintVisible | CPF_ZeroConstructor |
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
CPF_NativeAccessSpecifierPrivate
    UPROPERTY(BlueprintSetter = MyInt_Setter, Category = Blueprint)
        int32 MyInt_withSetter = 123;

```

# Blueprint Implementation:



## Principle:

If MD\_PropertySetFunction exists, use it as the invocation for Set.

```
void UK2Node_VariableSet::ExpandNode(class FKismetCompilerContext&
CompilerContext, UEdGraph* SourceGraph)
{
    // If property has a BlueprintSetter accessor, then replace the variable
    // get node with a call function
    if (VariableProperty)
    {
        // todo check with BP team if we need to test if the variable has
        // native Setter
        const FString& SetFunctionName = VariableProperty-
>GetMetaData(FBlueprintMetadata::MD_PropertySetFunction);
        if (!SetFunctionName.IsEmpty())
        {
        }
    }
}
```

## BlueprintCallable

- **Function Description:** This multicast delegate can be invoked within blueprints
- **Metadata Type:** bool
- **Engine Module:** Blueprint
- **Restriction Type:** Multicast Delegates
- **Action Mechanism:** Include CPF\_BlueprintCallable in PropertyFlags
- **Common Usage:** ★★★

This multicast delegate can be invoked in blueprints.

## Sample Code:

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FMyDynamicMulticastDelegate_One,
int32, Value);

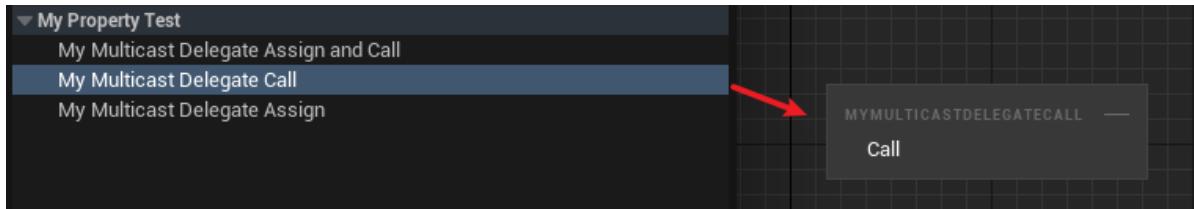
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable,
BlueprintCallable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAssignAndCall;

UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintCallable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateCall;

UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAssign;

UPROPERTY(EditAnywhere, BlueprintReadWrite)
FMyDynamicMulticastDelegate_One MyMulticastDelegate;
```

## Example Effect:



Please note that BlueprintAssignable and BlueprintCallable can only be used with multicast delegates

```
DECLARE_DYNAMIC_DELEGATE_OneParam(FMyDynamicSinglecastDelegate_One, int32,
value);

//Compilation Error: 'BlueprintCallable' is only allowed on a property when it is
//a multicast delegate
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintCallable)
FMyDynamicSinglecastDelegate_One MyMyDelegate4;

//Compilation Error: 'BlueprintAssignable' is only allowed on a property when
//it is a multicast delegate
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable)
FMyDynamicSinglecastDelegate_One MyMyDelegate5;
```

## BlueprintAssignable

- **Function Description:** This feature allows for binding events to the multicast delegate within a blueprint
- **Metadata Type:** bool
- **Engine Module:** Blueprint
- **Restriction Type:** Multicast Delegates
- **Action Mechanism:** Include CPF\_BlueprintAssignable in the PropertyFlags

- Common Usage: ★★★

## C++ Test Code:

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FMyDynamicMulticastDelegate_One,
int32, value);

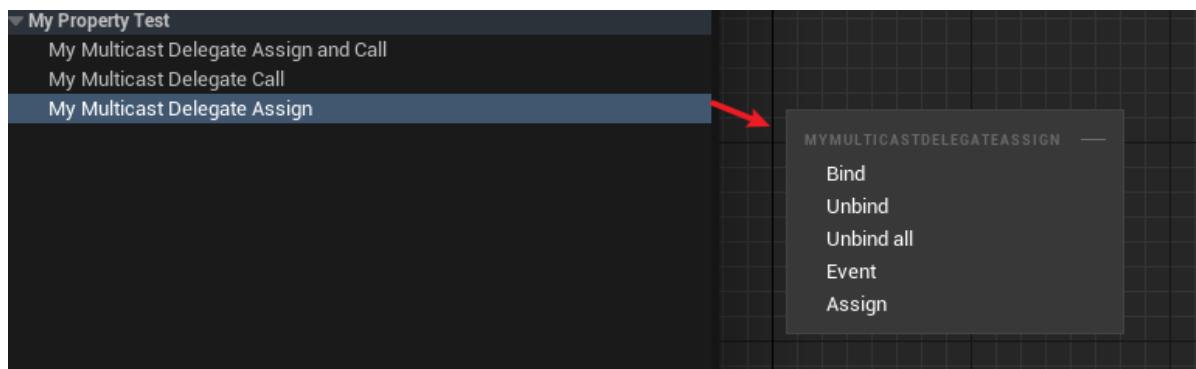
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable,
BlueprintCallable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAssignAndCall;

UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintCallable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateCall;

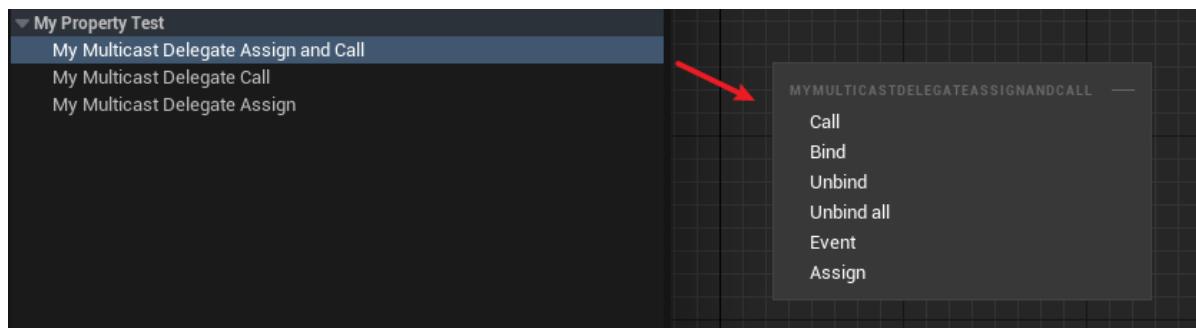
UPROPERTY(EditAnywhere, BlueprintReadWrite, BlueprintAssignable)
FMyDynamicMulticastDelegate_One MyMulticastDelegateAssign;

UPROPERTY(EditAnywhere, BlueprintReadWrite)
FMyDynamicMulticastDelegate_One MyMulticastDelegate;
```

## Blueprint Implementation:



Therefore, it is generally advised to add both tags accordingly



## Localized

- **Functional Description:** The value of this attribute will possess a predefined localized value. Frequently used for strings, it implies a ReadOnly characteristic. This value is associated with a localized value. It is most commonly annotated on strings
- **Metadata Type:** bool
- **Engine Module:** Behavior
- **Restriction Type:** FString

# Native

- **Functional Description:** The attribute is local: C++ code is responsible for its serialization and exposure to garbage collection.
- **Metadata Type:** bool
- **Engine Module:** Behavior

has been removed

## AssetRegistrySearchable

- **Function description:** Indicates that this attribute can be used as a Tag and Value in the AssetRegistry for filtering and searching assets
- **Metadata type:** bool
- **Engine module:** Asset
- **Action mechanism:** Include CPF\_AssetRegistrySearchable in PropertyFlags and RequiredAssetDataTags, DisallowedAssetDataTags in Meta
- **Commonly used:** ★★★

Cannot be used on struct properties.

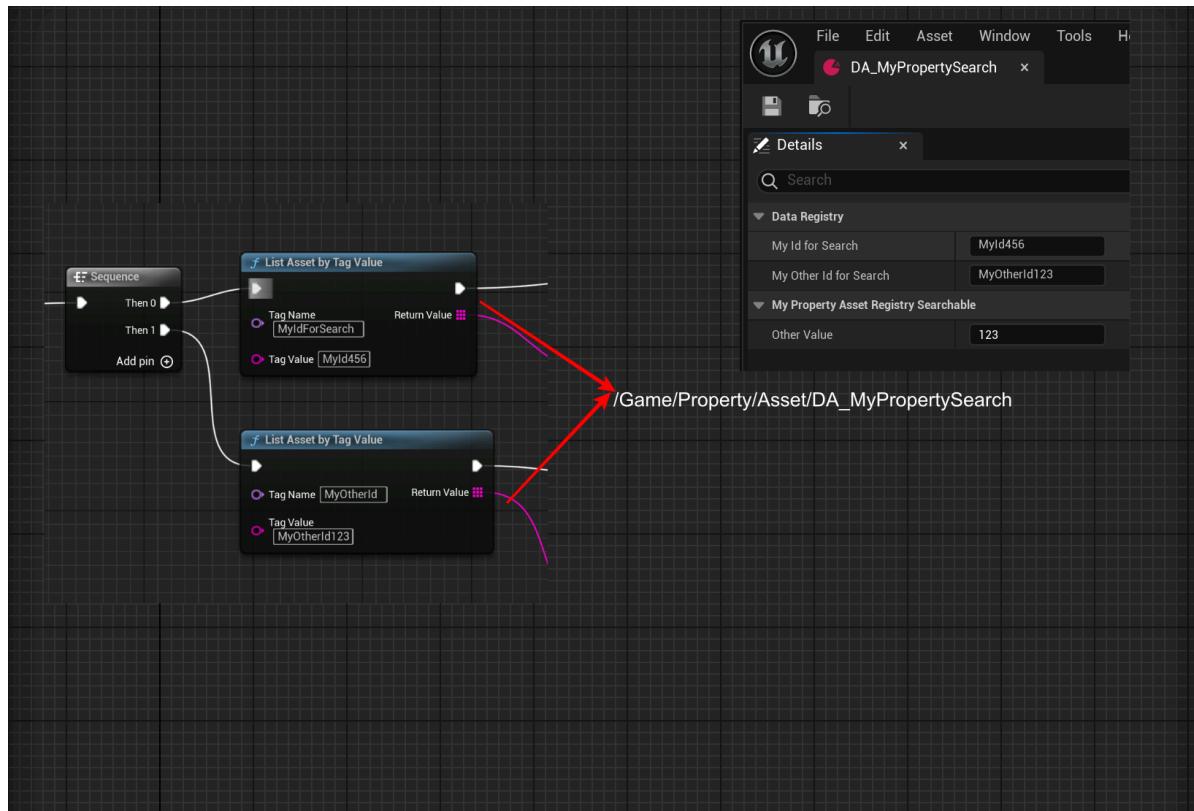
Subclasses can also override GetAssetRegistryTags to provide custom Tags.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_AssetRegistrySearchable : public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, AssetRegistrySearchable, Category = DataRegistry)
        FString MyIdForSearch;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int32 OtherValue = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DataRegistry)
        FString MyOtherIdForSearch;
public:
    virtual void GetAssetRegistryTags(FAssetRegistryTagsContext Context) const
override
    {
        //called on CDO and instances
        Super::GetAssetRegistryTags(Context);
        Context.AddTag(FAssetRegistryTag(TEXT("MyOtherID")), MyOtherIdForSearch,
UObject::FAssetRegistryTag::TT_Alphabetical));
    }
};
```

## Test Results:

When tested in EditorUtilityWidget, it is visible that ListAssetByTagValue can successfully search and find the Asset.



The test blueprint code can also use `IAssetRegistry::Get()>GetAssetsByTagValues(tagValues, outAssets);` for searching, but note that the search should occur after the AssetRegistry has been loaded. If AssetRegistry is Runtime, remember to serialize it to disk

```
//DefaultEngine.ini
[AssetRegistry]
bSerializeAssetRegistry=true
```

## Principle:

You can refer to the implementation and invocation of the `GetAssetRegistryTags` function. It is called and used in `UObject::GetAssetRegistryTags`, where the value of this property is provided to the AssetRegistry as the Tag for AssetData

## ChildCanTick

- **Functional Description:** Indicates that its Blueprint subclasses are permitted to receive the Tick event
- **Usage Location:** UCLASS
- **Engine Module:** Actor
- **Metadata Type:** bool
- **Restriction Type:** Subclasses of Actor or ActorComponent
- **Associated Items:** ChildCannotTick
- **Commonality:** ★★★

To override the Tick event function in a Blueprint, this check is performed only at compile time.

```
//(BlueprintType = true, ChildCannotTick = , IncludePath =
Class/Blueprint/MyActor_ChildTick.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Blueprint/MyActor_ChildTick.h)
UCLASS(Blueprintable,meta=(ChildCanTick))
class INSIDER_API AMyActor_ChildCanTick : public AActor
{
    GENERATED_BODY()
public:
    AMyActor_ChildCanTick()
    {
        PrimaryActorTick.bCanEverTick = false;
    }
};

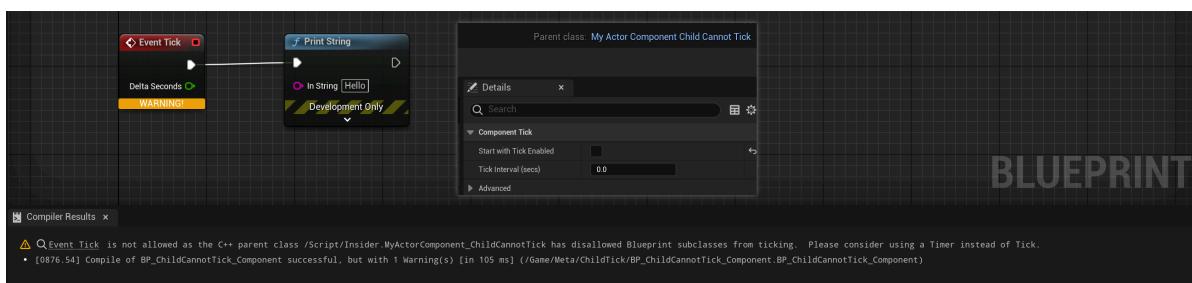
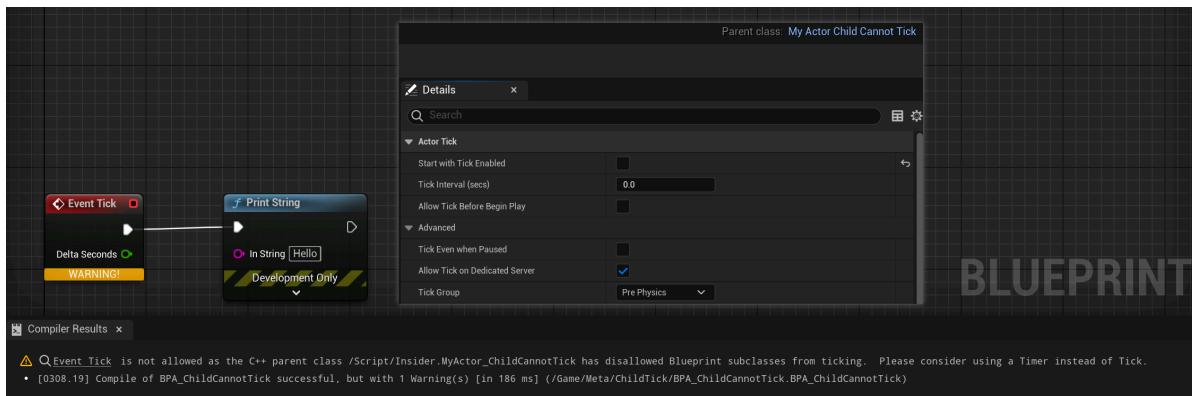
//(BlueprintType = true, ChildCanTick = , IncludePath =
Class/Blueprint/MyActor_ChildTick.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Blueprint/MyActor_ChildTick.h)
UCLASS(Blueprintable,meta=(ChildCanTick))
class INSIDER_API UMyActorComponent_ChildCanTick : public UActorComponent
{
    GENERATED_BODY()
public:
};

//(BlueprintType = true, ChildCannotTick = , IncludePath =
Class/Blueprint/MyActor_ChildTick.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Blueprint/MyActor_ChildTick.h)
UCLASS(Blueprintable,meta=(ChildCannotTick))
class INSIDER_API AMyActor_ChildCannotTick : public AActor
{
    GENERATED_BODY()
public:
};

//(BlueprintType = true, ChildCannotTick = , IncludePath =
Class/Blueprint/MyActor_ChildTick.h, IsBlueprintBase = true, ModuleRelativePath =
Class/Blueprint/MyActor_ChildTick.h)
UCLASS(Blueprintable,meta=(ChildCannotTick))
class INSIDER_API UMyActorComponent_ChildCannotTick : public UActorComponent
{
    GENERATED_BODY()
public:
};
```

Test within the Blueprint Actor or ActorComponent:

Also, be aware that this assessment is unrelated to whether the Tick is enabled in the blueprint.



Even though PrimaryActorTick.bCanEverTick is manually disabled in the AMyActor\_ChildCanTick class, subclasses can still tick normally (the bCanEverTick can be re-enabled internally during compilation).



## The logical reasoning within the source code:

There are three conditions for enabling bCanEverTick to be true: first, EngineSettings->bCanBlueprintsTickByDefault; second, the parent class is AActor or UActorComponent itself; and third, there is a ChildCanTick attribute on the C++ base class.

```
void FKismetCompilerContext::SetCanEverTick() const
{
    // RECEIVE TICK
    if (!TickFunction->bCanEverTick)
    {
        // Make sure that both AActor and UActorComponent have the same name for
        // their tick method
        static FName ReceiveTickName(GET_FUNCTION_NAME_CHECKED(AActor, ReceiveTick));
        static FName
        ComponentReceiveTickName(GET_FUNCTION_NAME_CHECKED(UActorComponent,
        ReceiveTick));

        if (const UFunction* ReceiveTickEvent =
            FKismetCompilerUtilities::FindOverriddenImplementableEvent(ReceiveTickName,
            NewClass))
        {
    
```

```

// We have a tick node, but are we allowed to?

const UEngine* EngineSettings = GetDefault<UEngine>();
const bool bAllowTickingByDefault = EngineSettings->bCanBlueprintsTickByDefault;

const UClass* FirstNativeClass =
FBlueprintEditorUtils::FindFirstNativeClass(NewClass);
const bool bHasCanTickMetadata = (FirstNativeClass != nullptr) &&
FirstNativeClass->HasMetaData(FBlueprintMetadata::MD_ChildCanTick);
const bool bHasCannotTickMetadata = (FirstNativeClass != nullptr) &&
FirstNativeClass->HasMetaData(FBlueprintMetadata::MD_ChildCannotTick);
const bool bHasUniversalParent = (FirstNativeClass != nullptr) &&
(AActor::StaticClass() == FirstNativeClass) || (UActorComponent::StaticClass()
== FirstNativeClass);

if (bHasCanTickMetadata && bHasCannotTickMetadata)
{
    // User error: The C++ class has conflicting metadata
    const FString ConflictingMetadataWarning = FText::Format(
        LOCTEXT("HasBothCanAndCannotMetadataFmt", "Native class %s has
both '{0}' and '{1}' metadata specified, they are mutually exclusive and '{1}'
will win."),
        FText::FromString(FirstNativeClass->GetPathName()),
        FText::FromName(FBlueprintMetadata::MD_ChildCanTick),
        FText::FromName(FBlueprintMetadata::MD_ChildCannotTick)
    ).ToString();
    MessageLog.Warning(*ConflictingMetadataWarning);
}

if (bHasCannotTickMetadata)
{
    // This could only happen if someone adds bad metadata to AActor or
    // UActorComponent directly
    check(!bHasUniversalParent);

    // Parent class has forbidden us to tick
    const FString NativeClassSaidNo = FText::Format(
        LOCTEXT("NativeClassProhibitsTickingFmt", "@@ is not allowed as
the C++ parent class {0} has disallowed Blueprint subclasses from ticking.
Please consider using a Timer instead of Tick."),
        FText::FromString(FirstNativeClass->GetPathName())
    ).ToString();
    MessageLog.Warning(*NativeClassSaidNo,
FindLocalEntryPoint(ReceiveTickEvent));
}

else
{
    if (bAllowTickingByDefault || bHasUniversalParent ||

bHasCanTickMetadata)
    {
        // We're allowed to tick for one reason or another
        TickFunction->bCanEverTick = true;
    }
    else
    {

```

```

        // Nothing allowing us to tick
        const FString ReceiveTickEventWarning = FText::Format(
            LOCTEXT("ReceiveTick_CanNeverTickFmt", "@@ is not allowed for
blueprints based on the C++ parent class {0}, so it will never Tick!"),
            FText::FromText(FirstNativeClass ? *FirstNativeClass-
>GetPathName() : TEXT("<null>"))
            .ToString();
        MessageLog.Warning(*ReceiveTickEventWarning,
FindLocalEntryPoint(ReceiveTickEvent));

        const FString ReceiveTickEventRemedies = FText::Format(
            LOCTEXT("ReceiveTick_CanNeverTickRemediesFmt", "You can solve
this in several ways:\n 1) Consider using a Timer instead of Tick.\n 2) Add
meta={0}) to the parent C++ class\n 3) Reparent the Blueprint to AActor or
UActorComponent, which can always tick."),
            FText::FromName(FBlueprintMetadata::MD_ChildCanTick)
            .ToString();
        MessageLog.Warning(*ReceiveTickEventRemedies);
    }
}
}
}
}

```

## ChildCannotTick

- **Function Description:** Indicates that for subclasses of Actor or ActorComponent, blueprint subclasses are not allowed to receive the Tick event response, even if the parent class is capable of receiving Tick events
- **Usage Location:** UCLASS
- **Metadata Type:** boolean
- **Restriction Type:** Actor class
- **Associated Items:** ChildCanTick
- **Commonality:** ★★

## AnimNotifyBoneName

- **Function Description:** Enables the FName attribute under UAnimNotify or UAnimNotifyState to function as BoneName.
- **Usage Location:** UPROPERTY
- **Engine Module:** AnimationGraph
- **Metadata Type:** bool
- **Restriction Type:** FName attribute under UAnimNotify or UAnimNotifyState subclass
- **Commonality:** ★★

Enables the FName attribute under UAnimNotify or UAnimNotifyState to function as BoneName.

In animation notifications, there is often a need to pass a bone name parameter, which cannot be sufficiently customized using standard string parameters. Therefore, marking an FName attribute with AnimNotifyBoneName allows for the creation of a dedicated UI in the corresponding detail panel customization, facilitating easier input of BoneName.

## Example in Source Code:

```
UCLASS(const, hidecategories = Object, collapsecategories, meta = (DisplayName =
"Play Niagara Particle Effect"), MinimalAPI)
class UAnimNotify_PlayNiagaraEffect : public UAnimNotify
{
    // SocketName to attach to
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AnimNotify", meta =
(AnimNotifyBoneName = "true"))
    FName SocketName;
}

UCLASS(Blueprintable, meta = (DisplayName = "Timed Niagara Effect"), MinimalAPI)
class UAnimNotifyState_TimedNiagaraEffect : public UAnimNotifyState
{
    // The socket within our mesh component to attach to when we spawn the
    Niagara component
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = NiagaraSystem, meta =
(ToolTip = "The socket or bone to attach the system to", AnimNotifyBoneName =
"true"))
    FName SocketName;
}
```

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UAnimNotify_MyTest:public UAnimNotify
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FName MyName;

    UPROPERTY(EditAnywhere, BlueprintReadWrite,meta=(AnimNotifyBoneName="true"))
    FName MyName_Bone;
};

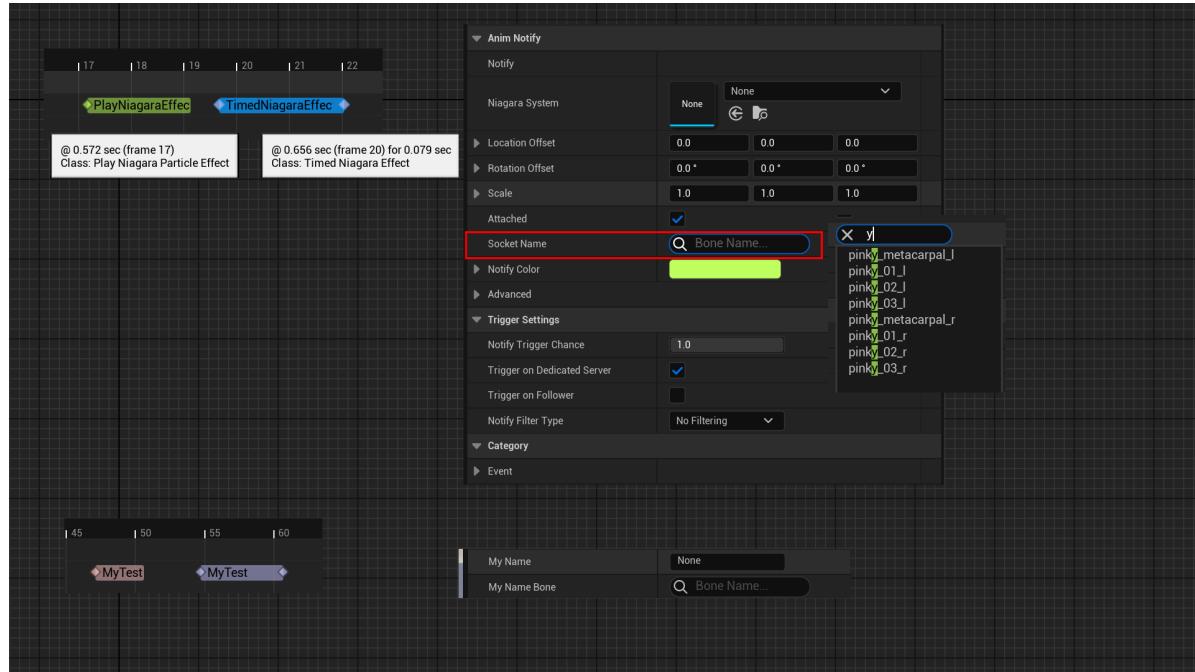
UCLASS(BlueprintType)
class INSIDER_API UAnimNotifyState_MyTest:public UAnimNotifyState
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FName MyName;

    UPROPERTY(EditAnywhere, BlueprintReadWrite,meta=(AnimNotifyBoneName="true"))
    FName MyName_Bone;
};
```

## Test Results:

In an animation sequence, animation notifications can be added in two forms: UAnimNotify or UAnimNotifyState. The built-in examples UAnimNotify\_PlayNiagaraEffect and UAnimNotifyState\_TimedNiagaraEffect in the engine show that the UI for SocketName in the details panel on the right is not a regular string.

Our custom animation notification MyBoneName can achieve the same effect. MyName\_Bone, with AnimNotifyBoneName added, differs from the standard MyName.



## Principle:

During customization, a special UI is generated based on whether the attributes under AnimNotify have this tag.

```
bool FAnimNotifyDetails::CustomizeProperty(IDetailCategoryBuilder& CategoryBuilder, UObject* Notify, TSharedPtr<IPropertyHandle> Property)
{
    else if (InPropertyParams->GetBoolMetaData(TEXT("AnimNotifyBoneName")))
    {
        // Convert this property to a bone name property
        AddBoneNameProperty(CategoryBuilder, Notify, InPropertyParams);
    }

    if (bIsBoneName)
    {
        AddBoneNameProperty(CategoryBuilder, Notify, Property);
        return true;
    }
}
```

# AnimBlueprintFunction

---

- **Function Description:** Indicates an internal pure stub function within the Animation Blueprint, utilized exclusively during the compilation of the Animation Blueprint
- **Usage Location:** UFUNCTION
- **Engine Module:** AnimationGraph
- **Metadata Type:** bool
- **Restriction Type:** Anim BP

Used solely internally, this is set during the compilation of the Animation Blueprint but is not explicitly written into the code.

# CustomizeProperty

---

- **Function Description:** Used on member properties of FAnimNode to instruct the editor not to generate the default Details panel control for them. Custom controls will be created later in DetailsCustomization.
- **Usage Location:** UPROPERTY
- **Engine Module:** AnimationGraph
- **Metadata Type:** bool
- **Restricted Type:** Attributes within FAnimNode
- **Commonality:** ★

Applied to member properties of FAnimNode, this tells the editor not to generate the default Details panel control for them. Custom controls will be created in DetailsCustomization subsequently.

Its function is somewhat similar to AllowEditInlineCustomization, both serving as markers to indicate that the editor will customize elsewhere, thus no default Details panel control needs to be generated for them.

## Example in Source Code:

---

There are numerous examples in the source code. Commonly, properties on nodes in AnimBP require specific customization in the Details panel. The most typical example is the Slot node, where SlotName is merely a FString type, but it is displayed as a ComboBoxString in the Details panel. This is because it is tagged with CustomizeProperty, instructing the default animation node Details panel generator (FAnimGraphNodeDetails) not to create an edit control for this property initially, and instead, a custom UI will be created for SlotName within its own customization (FAnimGraphNodeSlotDetails).

```
struct FAnimNode_Slot : public FAnimNode_Base
{
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Settings, meta=(CustomizeProperty))
    FName SlotName;
}
```

```

void FPersonaModule::CustomizeBlueprintEditorDetails(const TSharedRef<class
IDetailsView>& InDetailsView, FOnInvokeTab InOnInvokeTab)
{
    InDetailsView-
>RegisterInstancedCustomPropertyLayout(UAnimGraphNode_Slot::StaticClass(),

FOnGetDetailCustomizationInstance::CreateStatic(&FAnimGraphNodeSlotDetails::MakeI
nstance, InOnInvokeTab));

    InDetailsView->SetExtensionHandler(MakeShared<FAAnimGraphNodeBindingExtension>
());
}

```

## Test Code:

```

USTRUCT(BlueprintInternalUseOnly)
struct INSIDER_API FAAnimNode_MyCustomProperty : public FAAnimNode_Base
{
    GENERATED_USTRUCT_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = CustomProperty)
    FString MyString_Default;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = CustomProperty, meta =
(CustomizeProperty))
    FString MyString_CustomizeProperty;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = CustomProperty, meta =
(CustomizeProperty))
    FString MyString_CustomizeProperty_Other;
};

UCLASS()
class INSIDEREDITOR_API UAnimGraphNode_MyCustomProperty : public
UAnimGraphNode_Base
{
    GENERATED_UCLASS_BODY()

    UPROPERTY(EditAnywhere, Category = Settings)
    FAAnimNode_MyCustomProperty Node;
};

//Create another customization to generate a custom UI
void FMyAnimNode_MyCustomPropertyCustomization::CustomizeDetails(class
IDetailLayoutBuilder& DetailBuilder)
{
    TSharedPtr<IPropertyHandle> PropertyHandle =
DetailBuilder.GetProperty(TEXT("Node.MyString_CustomProperty"));

    //Just for test
    ComboBoxItems.Empty();
    ComboBoxItems.Add(MakeShareable(new FString(TEXT("First"))));
    ComboBoxItems.Add(MakeShareable(new FString(TEXT("Second"))));
    ComboBoxItems.Add(MakeShareable(new FString(TEXT("Third"))));
    const TSharedPtr<FString> ComboBoxSelectedItem = ComboBoxItems[0];
}

```

```

IDetailCategoryBuilder& Group =
DetailBuilder.EditCategory(TEXT("CustomProperty"));
Group.AddCustomRow(InvText("CustomProperty"))
.NameContent()
[
    PropertyHandle->CreatePropertyNameWidget()
]
.valueContent()
[
    SNew(STextComboBox)
.OptionsSource(&ComboListItems)
.InitiallySelectedItem(ComboBoxSelectedItem)
.ContentPadding(2, 10)
.ToolTipText(FText::FromString(*ComboBoxSelectedItem))
];
}

//Register the customization
PropertyModule.RegisterCustomClassLayout(TEXT("AnimGraphNode_MyCustomProperty"),
FOnGetDetailCustomizationInstance::CreateStatic(&FMyAnimNode_MyCustomPropertyCustomization::MakeInstance));

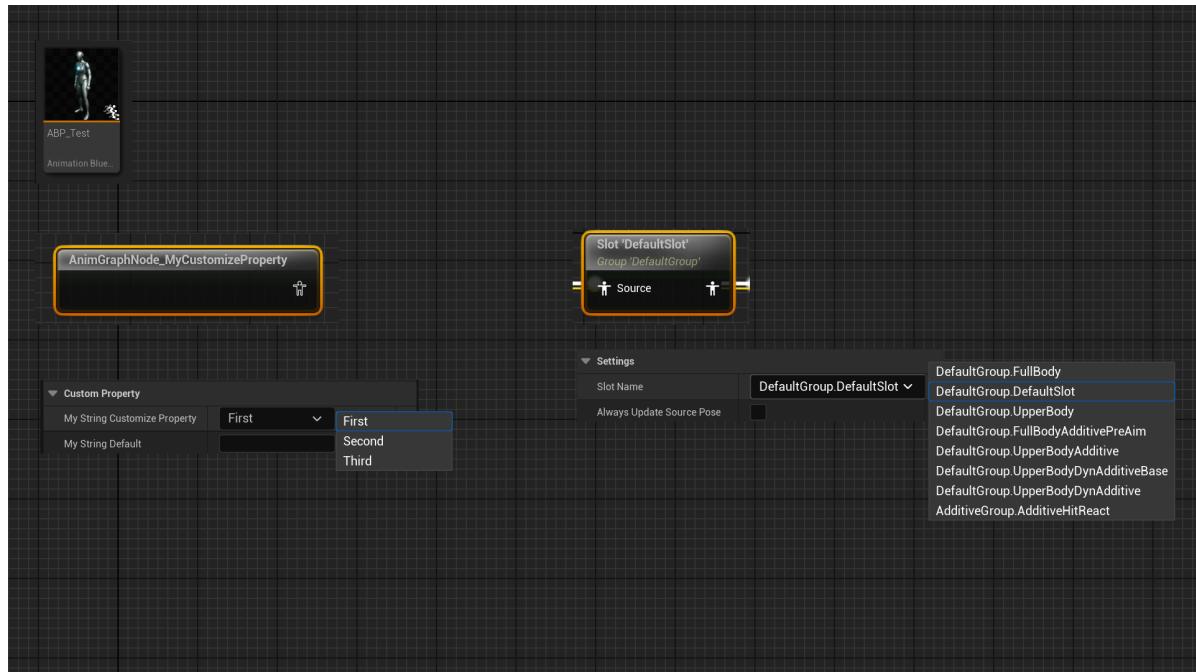
```

## Test Results:

The effect of SlotName is as shown on the right.

Our mimicked example shows that MyString\_Default remains just a default String, while MyString\_CustomizeProperty has a custom UI created for it.

For comparison, MyString\_CustomizeProperty\_Other is marked with CustomizeProperty but no UI is created for it, so it does not appear, indicating that the engine's default mechanism skips the UI creation process for it.



# Principle:

CustomizeProperty actually changes the Pin's bPropertyIsCustomized attribute (as reflected in GetRecordDefaults), and during the creation process, it does not create a default widget, which can be deduced from the bPropertyIsCustomized check in CustomizeDetails.

```
void FAnimBlueprintNodeOptionalPinManager::GetRecordDefaults(FProperty*  
TestProperty, FOptionalPinFromProperty& Record) const  
{  
    const UAnimationGraphSchema* Schema = GetDefault<UAnimationGraphSchema>();  
  
    // Determine if this is a pose or array of poses  
    FArrayProperty* ArrayProp = CastField<FArrayProperty>(TestProperty);  
    FStructProperty* StructProp = CastField<FStructProperty>(ArrayProp ?  
        ArrayProp->Inner : TestProperty);  
    const bool bIsPoseInput = (StructProp && StructProp->Struct-  
        >IsChildOf(FPoseLinkBase::StaticStruct()));  
  
    //TODO: Error if they specified two or more of these flags  
    const bool bAlwaysShow = TestProperty->HasMetaData(Schema->NAME_AlwaysAsPin)  
    || bIsPoseInput;  
    const bool bOptional_ShowByDefault = TestProperty->HasMetaData(Schema-  
        >NAME_PinShownByDefault);  
    const bool bOptional_HideByDefault = TestProperty->HasMetaData(Schema-  
        >NAME_PinHiddenByDefault);  
    const bool bNeverShow = TestProperty->HasMetaData(Schema->NAME_NeverAsPin);  
    const bool bPropertyIsCustomized = TestProperty->HasMetaData(Schema-  
        >NAME_CustomizeProperty);  
    const bool bCanTreatPropertyAsOptional =  
        CanTreatPropertyAsOptional(TestProperty);  
  
    Record.bCanToggleVisibility = bCanTreatPropertyAsOptional &&  
        (bOptional_ShowByDefault || bOptional_HideByDefault);  
    Record.bShowPin = bAlwaysShow || bOptional_ShowByDefault;  
    Record.bPropertyIsCustomized = bPropertyIsCustomized;  
}  
  
//This is the property that appears when a node is selected in AnimBP and then  
//viewed in the Details panel on the right  
void FAnimGraphNodeDetails::CustomizeDetails(class IDetailLayoutBuilder&  
DetailBuilder)  
{  
    // sometimes because of order of customization  
    // this gets called first for the node you'd like to customize  
    // then the above statement won't work  
    // so you can mark certain property to have meta data "CustomizeProperty"  
    // which will trigger below statement  
    if (OptionalPin.bPropertyIsCustomized)  
    {  
        continue;  
    }  
    TSharedRef<SWidget> InternalCustomWidget =  
        CreatePropertyWidget(TargetProperty, TargetPropertyHandle.ToSharedRef(),  
        AnimGraphNode->GetClass());
```

```
}
```

#AnimNotifyExpand

- **Function Description:** Allows properties under UAnimNotify or UAnimNotifyState to be directly expanded in the Details panel.
- **Usage Location:** UPROPERTY
- **Engine Module:** AnimationGraph
- **Metadata Type:** bool
- **Restriction Type:** Properties under UAnimNotify or UAnimNotifyState subclasses, specifically the FName attribute

Enables properties under UAnimNotify or UAnimNotifyState to be directly expanded in the Details panel.

No application examples were found in the source code.

## Principle:

Reviewing the source code reveals that it is hardcoded to only work with a few built-in engine classes, which is why custom test code does not take effect.

It is indeed not ideal for the code to be so rigidly written. I hope for future improvements, at which point there should be examples included in the source code.

```
PropertyModule.RegisterCustomClassLayout( "EditorNotifyObject",
FOnGetDetailCustomizationInstance::CreateStatic(&FAAnimNotifyDetails::MakeInstance
));

bool FAAnimNotifyDetails::CustomizeProperty(IDetailCategoryBuilder&
CategoryBuilder, UObject* Notify, TSharedPtr<IPropertyHandle> Property)
{
    if(Notify && Notify->GetClass() && Property->IsValidHandle())
    {
        FString ClassName = Notify->GetClass()->GetName();
        FStringPropertyName = Property->GetProperty()->GetName();
        bool bIsBoneName = Property->GetBoolMetaData(TEXT("AnimNotifyBoneName"));

        if(ClassName.Find(TEXT("AnimNotify_PlayParticleEffect")) != INDEX_NONE &&
PropertyName == TEXT("SocketName"))
        {
            AddBoneNameProperty(CategoryBuilder, Notify, Property);
            return true;
        }
        else if(ClassName.Find(TEXT("AnimNotifyState_TimedParticleEffect")) != INDEX_NONE &&
PropertyName == TEXT("SocketName"))
        {
            AddBoneNameProperty(CategoryBuilder, Notify, Property);
            return true;
        }
        else if(ClassName.Find(TEXT("AnimNotify_PlaySound")) != INDEX_NONE &&
PropertyName == TEXT("AttachName"))
        {
    }
```

```

        AddBoneNameProperty(CategoryBuilder, Notify, Property);
        return true;
    }
    else if (ClassName.Find(TEXT("_SoundLibrary")) != INDEX_NONE &&PropertyName
== TEXT("SoundContext"))
    {
        CategoryBuilder.AddProperty(Property);
        FixBoneNamePropertyRecurse(Property);
        return true;
    }
    else if (ClassName.Find(TEXT("AnimNotifyState_Trail")) != INDEX_NONE)
    {
        if(PropertyName == TEXT("FirstSocketName") || PropertyName ==
TEXT("SecondSocketName"))
        {
            AddBoneNameProperty(CategoryBuilder, Notify, Property);
            return true;
        }
        else if(PropertyName == TEXT("WidthScaleCurve"))
        {
            AddCurveNameProperty(CategoryBuilder, Notify, Property);
            return true;
        }
    }
    else if (bIsBoneName)
    {
        AddBoneNameProperty(CategoryBuilder, Notify, Property);
        return true;
    }
}
}

```

## OnEvaluate

- **Usage Location:** UPROPERTY
- **Engine Module:** AnimationGraph

## Principle:

It has been discovered in the source code, indicating that OnEvaluate has been abandoned.

```

// Dynamic value that needs to be wired up and evaluated each frame
const FString& EvaluationHandlerStr = SourcePinProperty-
>GetMetaData(AnimGraphDefaultSchema->NAME_OnEvaluate);
FName EvaluationHandlerName(*EvaluationHandlerStr);
if (EvaluationHandlerName != NAME_None)
{
    // warn that NAME_OnEvaluate is deprecated:
    InCompilationContext.GetMessageLog().Warning(*LOCTEXT("OnEvaluateDeprecated",
"OnEvaluate meta data is deprecated, found on @@").ToString(),
SourcePinProperty);
}

```

# FoldProperty

---

- **Function description:** Allows a certain attribute of an animation node to be designated as a FoldProperty within an animation blueprint.
- **Use location:** UPROPERTY
- **Engine module:** AnimationGraph
- **Metadata type:** bool
- **Restriction type:** Attributes under FAnimNode\_Base
- **Commonly used:** ★

Enables a specific attribute of an animation node to be marked as a FoldProperty within an animation blueprint.

- While PinHiddenByDefault also hides properties in the UI, FoldProperty behaves differently in terms of functionality.
- FoldProperty refers to attributes that are frequently enclosed with WITH\_EDITORONLY\_DATA, storing information specific to the editor's state. For instance, the PlayRate data under FAnimNode\_SequencePlayer is data that exists only in the editor's context. Alternatively, it may simply be information related to the animation blueprint itself, remaining consistent across all instances of the blueprint. Such attributes are suitable for being designated as FoldProperties.
- These attributes need to be edited on the node without being exposed as pins, hence they are placed in the details panel on the right side, similar to PinHiddenByDefault in appearance.

All "Constant/Fold" attribute information used by the instances of an animation node is stored in FAnimNodeData\* FAnimNode\_Base::NodeData. An animation blueprint may have multiple instances within a game, yet only one copy of the constant information and FoldProperty data for the animation node is maintained among these instances. Therefore, the actual data for properties marked as FoldProperty is stored in the TArray UAnimBlueprintGeneratedClass::AnimNodeData. This existence within the class is akin to the concept of CDO. One clear advantage of this approach is the conservation of memory.

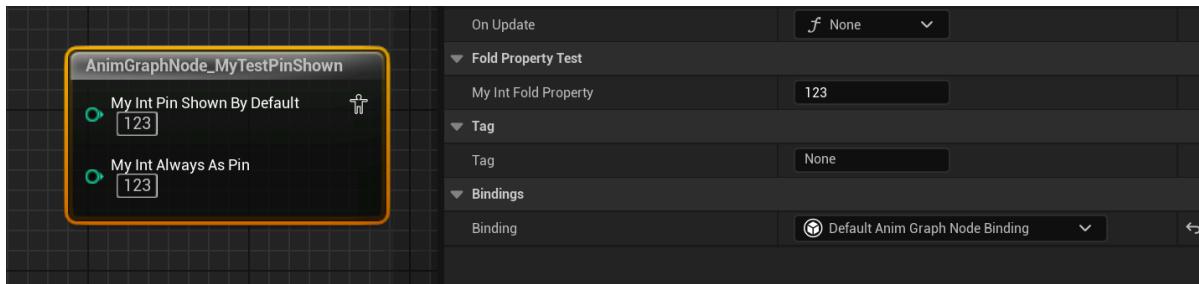
Naturally, with different storage methods, different access approaches are required. Hence, these FoldProperties are accessed using GET\_ANIM\_NODE\_DATA.

## Test Code:

---

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FoldPropertyTest, meta  
= (FoldProperty))  
    int32 MyInt_FoldProperty = 123;
```

# Test Results:



## Principle:

During compilation, the FoldProperty is added to FoldRecords. If the property is not dynamic and is not exposed for pin connections, it will be treated as a constant.

```
void FAnimBlueprintCompilerContext::GatherFoldRecordsForAnimationNode(const UScriptStruct* InNodeType, FStructProperty* InNodeProperty, UAnimGraphNode_Base* InVisualAnimNode)
{
    if(SubProperty->HasMetaData(NAME_FoldProperty))
    {
        // Add folding candidate
        AddFoldedPropertyRecord(InVisualAnimNode, InNodeProperty, SubProperty,
bAllPinsExposed, !bAllPinsDisconnected, bAlwaysDynamic);
    }
}

void FAnimBlueprintCompilerContext::AddFoldedPropertyRecord(UAnimGraphNode_Base* InAnimGraphNode, FStructProperty* InAnimNodeProperty, FProperty* InProperty, bool bInExposedOnPin, bool bInPinConnected, bool bInAlwaysDynamic)
{
    const bool bConstant = !bInAlwaysDynamic && (!bInExposedOnPin ||
(bInExposedOnPin && !bInPinConnected));

    if(!InProperty->HasAnyPropertyFlags(CPF_EditorOnly))
    {
        MessageLog.Warning(*FString::Printf(TEXT("Property %s on @@ is foldable, but not editor only"), *InProperty->GetName()), InAnimGraphNode);
    }

    // Create record and add it our lookup map
    TSharedRef<IAnimBlueprintCompilationContext::FFoldedPropertyRecord> Record =
MakeShared<IAnimBlueprintCompilationContext::FFoldedPropertyRecord>
(InAnimGraphNode, InAnimNodeProperty, InProperty, bConstant);
    TArray<TSharedRef<IAnimBlueprintCompilationContext::FFoldedPropertyRecord>>&
Array = NodeToFoldedPropertyRecordMap.FindOrAdd(InAnimGraphNode);
    Array.Add(Record);

    // Record it in the appropriate data area
    if(bConstant)
    {
        ConstantPropertyRecords.Add(Record);
    }
    else

```

```

    {
        MutablePropertyRecords.Add(Record);
    }
}

```

# BlueprintCompilerGeneratedDefaults

- **Function Description:** Indicates that the value of this attribute is generated by the compiler, thus it does not need to be copied post-compilation, which can enhance some aspects of compilation performance.
- **Usage Location:** UPROPERTY
- **Engine Module:** AnimationGraph
- **Metadata Type:** bool
- **Restricted Type:** Attributes within FAnimNode

Indicating that the value of this property is compiler-generated, it is unnecessary to copy it after compilation, which can improve some compilation performance.

Searching for examples in the source code reveals that these properties are primarily used under FAnimNode. After an animation blueprint is compiled,

UEngine::CopyPropertiesForUnrelatedObjects is invoked to copy the values from the old object compiled previously to the new object. The bSkipCompilerGeneratedDefaults parameter in FCopyPropertiesForUnrelatedObjectsParams determines whether to assign the value of this property. If marked, it signifies that the property should not be copied. This value is filled in by the compiler elsewhere.

The UAnimGraphNode\_Base::OnProcessDuringCompilation function is the callback function called after compilation.

## Test Code:

```

USTRUCT(BlueprintInternalUseOnly)
struct INSIDER_API FAnimNode_MyCompilerDefaults : public FAnimNode_Base
{
    GENERATED_USTRUCT_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Links)
    FPoseLink Source;
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = CompilerDefaultsTest)
    FString MyString;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = CompilerDefaultsTest,
meta = (BlueprintCompilerGeneratedDefaults))
    FString MyString_CompilerDefaults;
};

UCLASS()
class INSIDEREDITOR_API UAnimGraphNode_MyCompilerDefaults : public
UAnimGraphNode_Base
{
    GENERATED_UCLASS_BODY()
}

```

```

public:
    ~UAnimGraphNode_MyCompilerDefaults();

    UPROPERTY(EditAnywhere, Category = Settings)
    FAnimNode_MyCompilerDefaults Node;

protected:
    virtual void OnProcessDuringCompilation(IAnimBlueprintCompilationContext&
InCompilationContext, IAnimBlueprintGeneratedClassCompiledData& outCompiledData)
    {
        Node.MyString=TEXT("This is generated by compiler.");
        Node.MyString_CompilerDefaults=TEXT("This is generated by compiler.");
    }
};

```

## Test Effects:

As this is a serialization process, there is no直观 diagram available.

Verifiable results can be observed in FCPFUOWriter::ShouldSkipProperty, where the MyString\_CompilerDefaults property is skipped during copying.

## Principle:

The essence of the blueprint compilation process is to generate a new Graph object and then copy the attributes and objects from the result object of the last compilation into this new object. This operation is performed using UEngine::CopyPropertiesForUnrelatedObjects, and internally, FCPFUOWriter::ShouldSkipProperty is used to decide whether to copy a particular property. For some attributes, their values are only temporary and generated by the compiler; they will be generated again in the next compilation, so there is no need to copy them. Marking them can slightly enhance performance, although the improvement is marginal.

```

void
UK2Node_PropertyAccess::CreateClassVariablesFromBlueprint(IAnimBlueprintVariableC
reationContext& InCreationContext)
{
    GeneratedPropertyName = NAME_None;

    const bool bRequiresCachedVariable = !bWasResolvedThreadSafe ||
UAnimBlueprintExtension_PropertyAccess::ContextRequiresCachedVariable(ContextId);

    if(ResolvedPinType != FEdGraphPinType() && ResolvedPinType.PinCategory !=
UEdGraphSchema_K2::PC_Wildcard && bRequiresCachedVariable)
    {
        // Create internal generated destination property (only if we were not
identified as thread safe)
        if(FProperty* DestProperty = InCreationContext.CreateUniqueVariable(this,
ResolvedPinType))
        {
            GeneratedPropertyName = DestProperty->GetFName();
            DestProperty->SetMetaData(TEXT("BlueprintCompilerGeneratedDefaults"),
TEXT("true"));
        }
    }
}

```

```

        }

    }

/* Serializes and stores property data from a specified 'source' object. Only
stores data compatible with a target destination object. */
struct FCPFUOWriter : public FObjectWriter, public FCPFUOArchive
{
#if WITH_EDITOR
    virtual bool ShouldSkipProperty(const class FProperty* InProperty) const
override
    {
        return (bskipCompilerGeneratedDefaults && InProperty-
>HasMetaData(BlueprintCompilerGeneratedDefaultsName));
    }
#endif
}

```

## CustomWidget

---

- **Usage Location:** UFUNCTION, UPROPERTY
- **Engine Module:** AnimationGraph
- **Metadata Type:** string = "abc"

Also applicable when used as an attribute

```

// @param Scope   Scopes corresponding to an existing scope in a schedule, or
"None". Passing "None" will apply the parameter to the whole schedule.
// @param Ordering where to apply the parameter in relation to the supplied
scope. Ignored for scope "None".
// @param Name     The name of the parameter to apply
// @param Value    The value to set the parameter to
UFUNCTION(BlueprintCallable, Category = "AnimNext", CustomThunk, meta =
(CustomStructureParam = Value, UnsafeDuringActorConstruction))
void SetParameterInScope(UPARAM(meta = (CustomWidget = "ParamName",
AllowedParamType = "FAnimNextScope")) FName Scope,
EAnimNextParameterScopeOrdering Ordering, UPARAM(meta = (CustomWidget =
"ParamName")) FName Name, int32 Value);

```

Used exclusively in AnimNext and RigVM.

## AllowedParamType

---

- **Usage location:** UFUNCTION
- **Engine module:** AnimationGraph
- **Metadata type:** string = "abc"

```

// Sets a parameter's value in the supplied scope.
// @param Scope Scopes corresponding to an existing scope in a schedule, or
// "None". Passing "None" will apply the parameter to the whole schedule.
// @param Ordering where to apply the parameter in relation to the supplied
// scope. Ignored for scope "None".
// @param Name The name of the parameter to apply
// @param Value The value to set the parameter to
UFUNCTION(BlueprintCallable, Category = "AnimNext", CustomThunk, meta =
(CustomStructureParam = Value, UnsafeDuringActorConstruction))
void SetParameterInScope(UPARAM(meta = (CustomWidget = "ParamName",
AllowedParamType = "FAnimNextScope")) FName Scope,
EAnimNextParameterScopeOrdering Ordering, UPARAM(meta = (CustomWidget =
"ParamName")) FName Name, int32 Value);

```

Checked and found it is used exclusively in AnimNext.

## PinShownByDefault

- **Function description:** Exposes a specific attribute of an animation node as a pin by default within an animation blueprint, though this can be modified.
- **Use location:** UPROPERTY
- **Engine module:** AnimationGraph
- **Metadata type:** bool
- **Restriction type:** FAnimNode\_Base
- **Associated items:** AlwaysAsPin, NeverAsPin
- **Commonality:** ★★★

In the animation blueprint, a specific attribute of an animation node is made visible as a pin from the start.

Unlike conventional blueprints, attributes within FAnimNode\_Base are not displayed on the node by default, hence the need for this meta to explicitly indicate which attributes should be made explicit.

PinShownByDefault is currently applicable only to animation blueprint nodes.

Conversely, PinHiddenByDefault can be used to hide attributes as pins.

## Test Code:

```

USTRUCT(BlueprintInternalUseOnly)
struct INSIDEREDITOR_API FAnimNode_MyTestPinShown : public FAnimNode_Base
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest)
    int32 MyInt_NotShown = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (PinShownByDefault))
    int32 MyInt_PinShownByDefault = 123;

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (AlwaysAsPin))
int32 MyInt_AlwaysAsPin = 123;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (NeverAsPin))
int32 MyInt_NeverAsPin = 123;
};

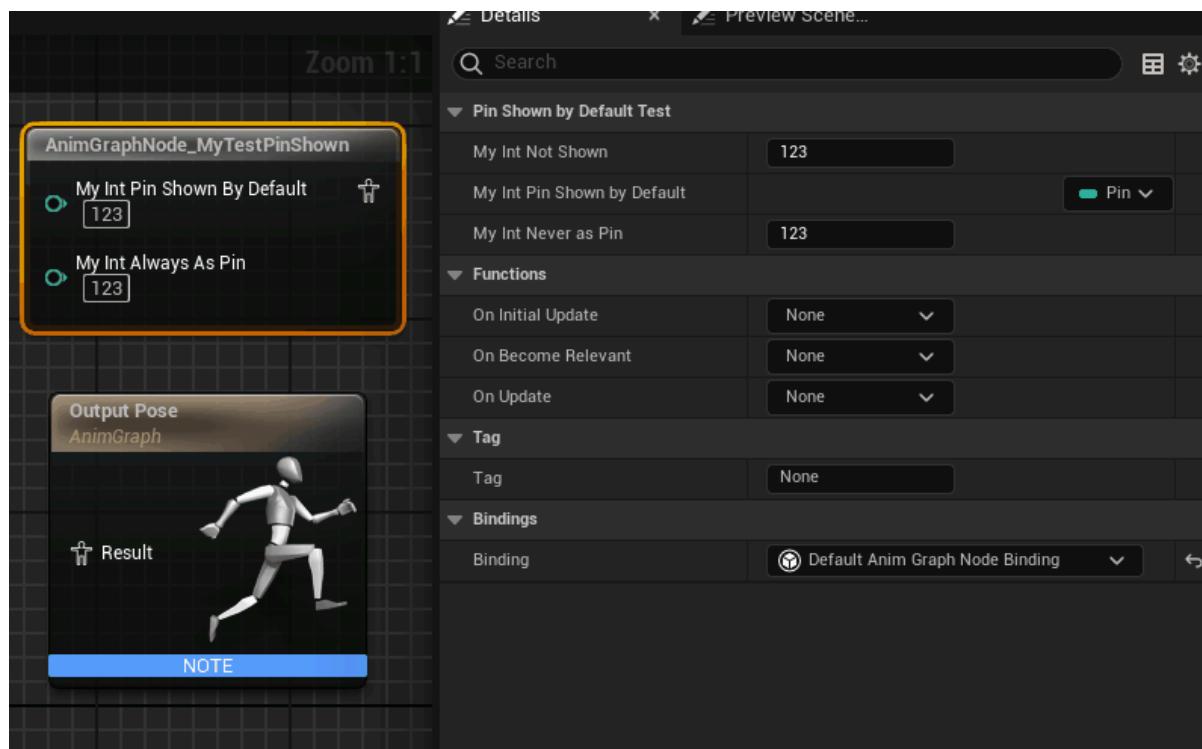
UCLASS()
class INSIDEREDITOR_API UAnimGraphNode_MyTestPinShown : public
UAnimGraphNode_Base
{
GENERATED_UCLASS_BODY()

UPROPERTY(EditAnywhere, Category = Settings)
FAnimNode_MyTestPinShown Node;
};

```

## Test Results:

It is evident that the two properties, MyInt\_NotShown does not appear as a node by default and can only be edited in the details panel, whereas MyInt\_PinShownByDefault is shown as a pin by default. PinShownByDefault can also be altered to remove the pin functionality.



## Principle:

The sole usage in the source code is within FAnimBlueprintNodeOptionalPinManager, which is responsible for handling the display of pins on animation blueprint nodes.

```

void FAnimBlueprintNodeOptionalPinManager::GetRecordDefaults(FProperty* TestProperty, FOptionalPinFromProperty& Record) const
{
    const UAnimationGraphSchema* Schema = GetDefault<UAnimationGraphSchema>();
}

```

```

// Determine if this is a pose or array of poses
FArrayProperty* ArrayProp = CastField<FArrayProperty>(TestProperty);
FStructProperty* StructProp = CastField<FStructProperty>(ArrayProp ? 
    ArrayProp->Inner : TestProperty);
    const bool bIsPoseInput = (StructProp && StructProp->Struct-
>IsChildOf(FPoseLinkBase::StaticStruct()));

//@TODO: Error if they specified two or more of these flags
const bool bAlwaysShow = TestProperty->HasMetaData(Schema->NAME_AlwaysAsPin)
|| bIsPoseInput;
    const bool bOptional_ShowByDefault = TestProperty->HasMetaData(Schema-
>NAME_PinShownByDefault);
    const bool bOptional_HideByDefault = TestProperty->HasMetaData(Schema-
>NAME_PinHiddenByDefault);
    const bool bNeverShow = TestProperty->HasMetaData(Schema->NAME_NeverAsPin);
    const bool bPropertyIsCustomized = TestProperty->HasMetaData(Schema-
>NAME_CustomizeProperty);
    const bool bCanTreatPropertyAsOptional =
CanTreatPropertyAsOptional(TestProperty);

    Record.bCanToggleVisibility = bCanTreatPropertyAsOptional &&
(bOptional_ShowByDefault || bOptional_HideByDefault);
    Record.bShowPin = bAlwaysShow || bOptional_ShowByDefault;
    Record.bPropertyIsCustomized = bPropertyIsCustomized;
}

```

## AnimGetter

---

- **Function Description:** Designates the function of specified UAnimInstance and its subclasses as an AnimGetter function.
- **Usage Location:** UFUNCTION
- **Engine Module:** AnimationGraph
- **Metadata Type:** bool
- **Restricted Types:** Functions of UAnimInstance and its subclasses
- **Associated Items:** GetterContext
- **Commonly Used:** ★★★

Designates the function of specified UAnimInstance and its subclasses as an AnimGetter function.

- In certain scenarios, UAnimInstance is inherited to create a custom animation blueprint subclass, where optimizations or additional functional methods can be implemented.
- An AnimGetter is essentially a function that is recognized and wrapped by the UK2Node\_AnimGetter, within the scope of UAnimInstance and its subclasses (i.e., animation blueprints).
- AnimGetter also offers two additional features: it automatically populates the AssetPlayerIndex, MachineIndex, StateIndex, TransitionIndex, and parameters within the function based on the current context. It also restricts the function to be callable only within certain blueprints, based on the GetterContext. These conveniences and checks are not available in standard blueprint functions, making them less intelligent.

- To qualify as an AnimGetter, it must also possess:
  - AnimGetter, naturally
  - BlueprintThreadSafe, to be callable within animation blueprints and thread-safe
  - BlueprintPure, to function as a value-getting function
  - BlueprintInternalUseOnly = "true", to prevent generating a default blueprint node and only use the one wrapped by UK2Node\_AnimGetter.

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyAnimInstance :public UAnimInstance
{
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
    (BlueprintInternalUseOnly = "true", AnimGetter, BlueprintThreadSafe))
        float MyGetAnimationLength_AnimGetter(int32 AssetPlayerIndex);

    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
    (BlueprintThreadsafe))
        float MyGetAnimationLength(int32 AssetPlayerIndex);
public:
    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
    (BlueprintInternalUseOnly = "true", AnimGetter, BlueprintThreadSafe))
        float MyGetStateweight_AnimGetter(int32 MachineIndex, int32 StateIndex);

    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
    (BlueprintThreadSafe))
        float MyGetStateweight(int32 MachineIndex, int32 StateIndex);
public:
    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
    (BlueprintInternalUseOnly = "true", AnimGetter, BlueprintThreadsafe))
        float MyGetTransitionTimeElapsed_AnimGetter(int32 MachineIndex, int32
    TransitionIndex);

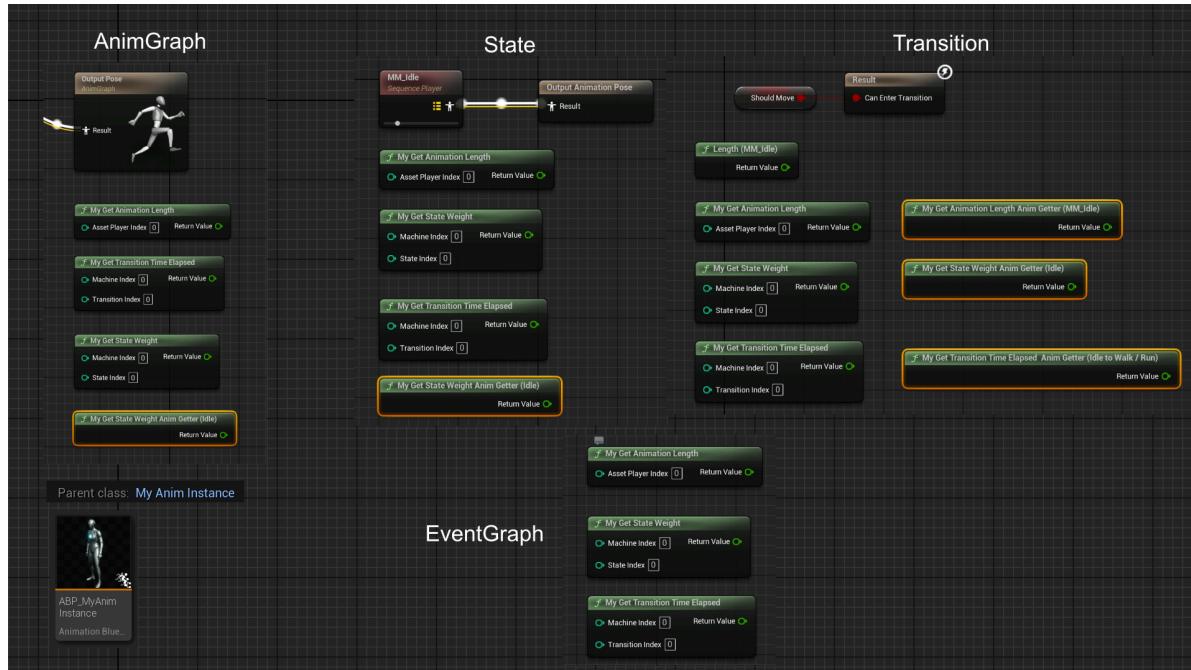
    UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
    (BlueprintThreadSafe))
        float MyGetTransitionTimeElapsed(int32 MachineIndex, int32 TransitionIndex);
};
```

## Test Results:

Define AnimGetter functions that use AssetPlayerIndex, MachineIndex, StateIndex, TransitionIndex, as well as ordinary blueprint functions for comparison. Examine their usage in various scopes within animation blueprints.

- It can be seen that no matter what scope, ordinary blueprint functions can be called (after all, there is no Context check). In addition, parameters such as AssetPlayerIndex are not automatically filled in, which is almost useless because users don't really know how to fill in these Indexes by hand. It is best to leave it to the compiler.

- The highlighted functions in the diagram are the callable AnimGetter functions. A closer examination reveals that the rule is that only those functions that can correctly populate parameters like AssetPlayerIndex can be called. Thus, the most callable are within Transitions, as this is the leaf node with animations, state machines, and Transition nodes.



## Principle:

The functionality of analyzing the AnimGetter tag on a function and generating blueprint nodes is primarily located within the UK2Node\_AnimGetter class. Feel free to inspect it yourself.

```
void UK2Node_AnimGetter::GetMenuActions(FBlueprintActionDatabaseRegistrar&
ActionRegistrar) const
{
    TArray<UFunction*> AnimGetters;
    for(TFieldIterator<UFunction> FuncIter(BPClass) ; FuncIter ;
++FuncIter)
    {
        UFunction* Func = *FuncIter;

        if(Func->HasMetaData(TEXT("AnimGetter")) && Func-
>HasAnyFunctionFlags(FUNC_Native))
        {
            AnimGetters.Add(Func);
        }
    }
}
```

## GetterContext

- Function Description:** Further specifies where the AnimGetter function can be utilized; if left blank, it is assumed to be available everywhere.
- Usage Location:** UFUNCTION
- Engine Module:** AnimationGraph
- Metadata Type:** string="abc"

- **Restriction Type:** AnimGetter function of UAnimInstance and its subclasses
- **Associated Items:** AnimGetter
- **Commonality:** ★★

Continuing to specify where the AnimGetter function can be used, if not filled, it can be used universally.

Options include: Transition, CustomBlend, AnimGraph.

## Source Code Comments:

```
* A context string can be provided in the GetterContext metadata and can
contain any (or none) of the
* following entries separated by a pipe (|)
* Transition - Only available in a transition rule
* AnimGraph - Only available in an animgraph (also covers state anim
graphs)
* CustomBlend - Only available in a custom blend graph
```

## Test Code:

```
UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
(BlueprintThreadSafe))
float MyGetStateWeight(int32 MachineIndex, int32 StateIndex);
public:
UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
(BlueprintInternalUseOnly = "true", AnimGetter, GetterContext = "Transition",
BlueprintThreadSafe))
float MyGetStateWeight_AnimGetter_OnlyTransition(int32 MachineIndex, int32
StateIndex);

UFUNCTION(BlueprintPure, Category = "Animation|Insider", meta =
(BlueprintInternalUseOnly = "true", AnimGetter, GetterContext = "CustomBlend",
BlueprintThreadSafe))
float MyGetTransitionTimeElapsed_AnimGetter_OnlyCustomBlend(int32 MachineIndex,
int32 TransitionIndex);
```

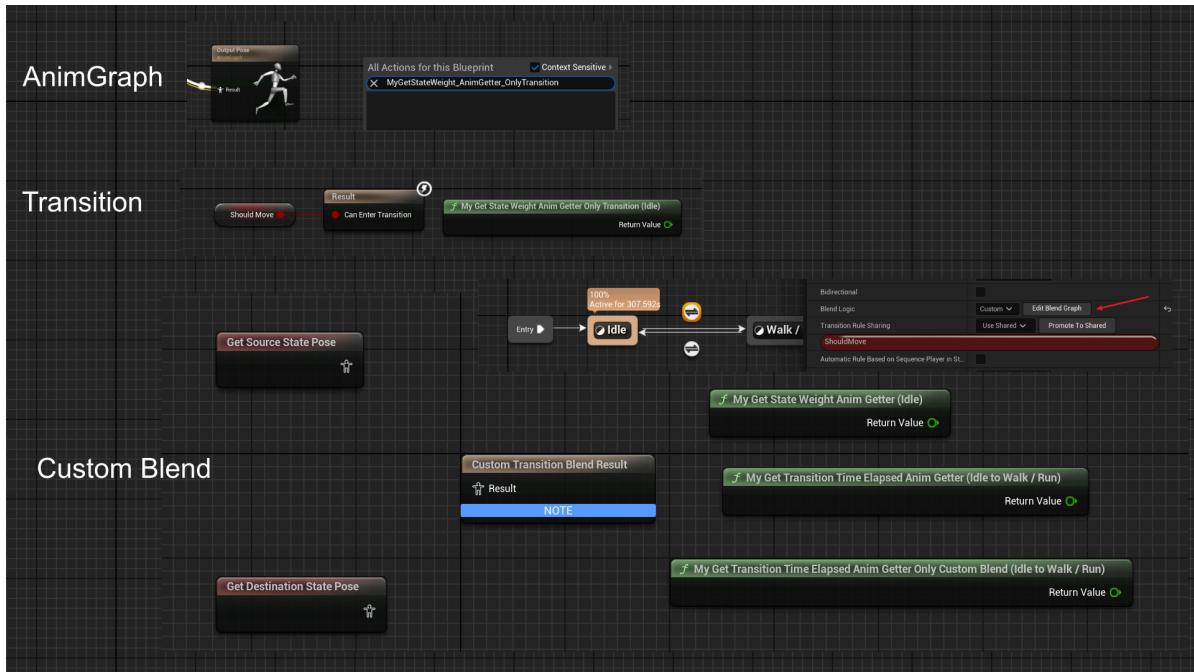
## Test Results:

This diagram should be compared with the one in AnimGetter for review.

The first point of focus is on MyGetStateWeight\_AnimGetter\_OnlyTransition within AnimGraph. If GetterContext is not marked, it can be invoked, but with the marking, it can only be called within Transition. It is also noted that this function cannot be called in CustomBlend.

The second point is in CustomBlend. The steps to operate are to change the detail panel on the right side of the Rule to Custom and then enter the CustomBlend blueprint. In this blueprint, MyGetStateWeight can be invoked as GetterContext is not specified.

MyGetTransitionTimeElapsed\_AnimGetter\_OnlyCustomBlend can now be called.



## Principle:

The function to determine whether it can be called is as follows.

```

bool UK2Node_AnimGetter::IsContextValidForSchema(const UEdGraphSchema* Schema)
{
    if(Contexts.Num() == 0)
    {
        // Valid in all graphs
        return true;
    }

    for(const FString& Context : Contexts)
    {
        UClass* ClassToCheck = nullptr;
        if(Context == TEXT("CustomBlend"))
        {
            ClassToCheck = UAnimationCustomTransitionSchema::StaticClass();
        }

        if(Context == TEXT("Transition"))
        {
            ClassToCheck = UAnimationTransitionSchema::StaticClass();
        }

        if(Context == TEXT("AnimGraph"))
        {
            ClassToCheck = UAnimationGraphSchema::StaticClass();
        }
    }

    return Schema->GetClass() == ClassToCheck;
}

return false;
}

```

# RequiredAssetDataTags

- **Function description:** Specifies tags on the UObject\* attribute for filtering; an object must possess these tags to be selected.
- **Usage location:** UPROPERTY
- **Engine module:** Asset Property
- **Metadata type:** strings = "a=b, c=d, e=f"
- **Restriction type:** UObject\*
- **Associated items:** DisallowedAssetDataTags, AssetRegistrySearchable
- **Commonly used:** ★★

Specifies tags on the UObject\* attribute for filtering; an object must possess these tags to be selected.

Refer to the AssetRegistrySearchable specifier and the GetAssetRegistryTags method for related information.

## Test Code:

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyTableRow_Required :public FTableRowBase
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt = 123;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyTableRow_Disallowed :public FTableRowBase
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat = 123.f;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    UTexture2D* MyTexture;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_AssetDataTags :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object)
    TObjectPtr<UObject> MyAsset_Default;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object, meta =
    (RequiredAssetDataTags = "MyIdForSearch=MyId456"))
};
```

```

TObjectPtr<UObject> MyAsset_RequiredAssetDataTags;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object, meta =
(DisallowAssetDataTags = "MyOtherId=MyOtherId789"))
TObjectPtr<UObject> MyAsset_DisallowedAssetDataTags;
public:
    UPROPERTY(EditAnywhere, Category = DataTable)
    TObjectPtr<class UDataTable> MyDataTable_Default;

    UPROPERTY(EditAnywhere, Category = DataTable, meta = (RequiredAssetDataTags =
"RowStructure=/Script/Insider.MyTableRow_Required"))
    TObjectPtr<class UDataTable> MyDataTable_RequiredAssetDataTags;

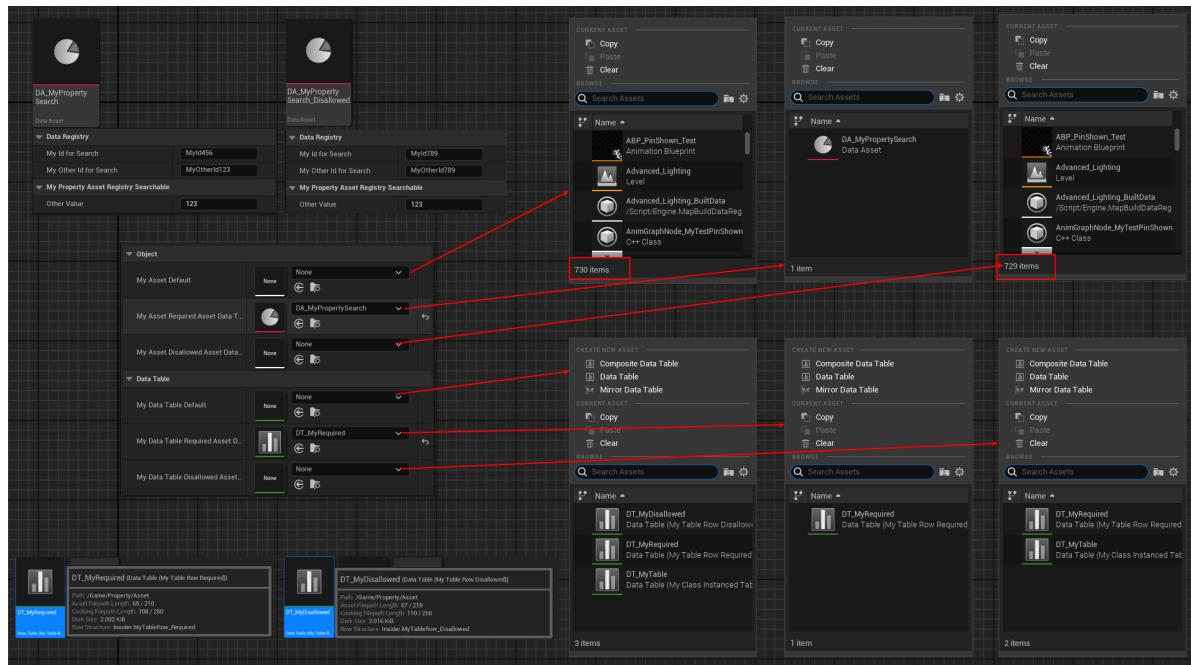
    UPROPERTY(EditAnywhere, Category = DataTable, meta = (DisallowAssetDataTags =
"RowStructure=/Script/Insider.MyTableRow_Disallowed"))
    TObjectPtr<class UDataTable> MyDataTable_DisallowedAssetDataTags;
};

```

## Test Results:

As seen in the code above, two different types of FTableRowBase are defined, and two DataTables are created. There are also two DataAssets (structures defined in the AssetRegistrySearchable example) that both have the MyIdForSearch and MyOtherId tags, but with different values to distinguish them.

- MyAsset\_Default can filter out all objects, with 730 examples shown in the image.
- MyAsset\_RequiredAssetDataTags filters out all but DA\_MyPropertySearch, because MyIdForSearch equals MyId456.
- MyAsset\_DisallowedAssetDataTags filters out DA\_MyPropertySearch\_Disallowed, because the configured MyOtherId equals MyOtherId789, leaving only 729.
- The same logic applies to DataTable. MyDataTable\_Default can retrieve all DataTables (there are three), while MyDataTable\_RequiredAssetDataTags restricts RowStructure to only FMyTableRow\_Required (thus filtering out one). MyDataTable\_DisallowedAssetDataTags excludes any RowStructure that is FMyTableRow\_Disallowed, leaving only two.



## Example in Source Code:

```
UPROPERTY(Category="StateTree", EditAnywhere, meta=
(RequiredAssetDataTags="Schema=/Script/MassAIBehavior.MassStateTreeSchema"))
TObjectPtr<UStateTree> StateTree;

UPROPERTY(EditAnywhere, Category=Appearance, meta = (RequiredAssetDataTags =
"RowStructure=/Script/UMG.RichImageRow"))
TObjectPtr<class UDataTable> ImageSet;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Compositing, meta =
(AllowPrivateAccess, RequiredAssetDataTags = "IsSourceValid=True"), Setter =
SetCompositeTexture, Getter = GetCompositeTexture)
TObjectPtr<class UTexture> CompositeTexture;
```

## Principle:

The configuration of RequiredAssetDataTags and DisallowedAssetDataTags on the UObject\* attribute is parsed and extracted into its member variables, RequiredAssetDataTags and DisallowedAssetDataTags, during the attribute editor (SPropertyEditorAsset) initialization. Essentially, it's a key-value pair. When performing asset filtering later (during the call to IsAssetFiltered), the Tags in FAssetData are matched against the attribute's Tags requirements. Assets with Disallowed Tags are excluded, while those lacking Required Tags are also filtered out. This achieves the desired filtering effect.

For information about the tags in FAssetData, refer to the AssetRegistrySearchable specifier and the usage of the GetAssetRegistryTags method. In essence, there is a method on the object that actively provides tags to the AssetRegistry.

The reason why DataTables can be filtered by RowStructure can be understood by examining the GetAssetRegistryTags method in DataTables, which actively registers the RowStructure tags.

```
FAssetDataTagMapBase=TSortedMap<FName, FString, FDefaultAllocator,
FNameFastLess>;

SPropertyEditorAsset::
/** Tags (and eventually values) that can NOT be used with this property */
TSharedPtr<FAssetDataTagMap> DisallowedAssetDataTags;

/** Tags and values that must be present for an asset to be used with this
property */
TSharedPtr<FAssetDataTagMap> RequiredAssetDataTags;

void SPropertyEditorAsset::InitializeAssetDataTags(const FProperty* Property)
{
    if (Property == nullptr)
    {
        return;
    }

    const FProperty* MetadataProperty = GetActualMetadataProperty(Property);
    const FString DisallowedAssetDataTagsFilterString = MetadataProperty-
>GetMetaData("DisallowedAssetDataTags");
    if (!DisallowedAssetDataTagsFilterString.IsEmpty())
```

```

{
    TArray< FString> DisallowedAssetDataTagsAndValues;

DisallowedAssetDataTagsFilterString.ParseIntoArray(DisallowedAssetDataTagsAndValues,
es, TEXT(","), true);

    for (const FString& TagAndOptionalValueString :
DisallowedAssetDataTagsAndValues)
    {
        TArray< FString> TagAndOptionalValue;
        TagAndOptionalValueString.ParseIntoArray(TagAndOptionalValue, TEXT(":"), true);
        size_t NumStrings = TagAndOptionalValue.Num();
        check((NumStrings == 1) || (NumStrings == 2)); // there should be a
single '=' within a tag/value pair

        if (!DisallowedAssetDataTags.IsValid())
        {
            DisallowedAssetDataTags = MakeShared< FAssetDataTagMap>();
        }
        DisallowedAssetDataTags->Add(FName(*TagAndOptionalValue[0]), (NumStrings
> 1) ? TagAndOptionalValue[1] : FString());
    }
}

const FString RequiredAssetDataTagsFilterString = MetadataProperty-
>GetMetaData("RequiredAssetDataTags");
if (!RequiredAssetDataTagsFilterString.IsEmpty())
{
    TArray< FString> RequiredAssetDataTagsAndValues;

RequiredAssetDataTagsFilterString.ParseIntoArray(RequiredAssetDataTagsAndValues,
TEXT(","), true);

    for (const FString& TagAndOptionalValueString :
RequiredAssetDataTagsAndValues)
    {
        TArray< FString> TagAndOptionalValue;
        TagAndOptionalValueString.ParseIntoArray(TagAndOptionalValue, TEXT(":"), true);
        size_t NumStrings = TagAndOptionalValue.Num();
        check((NumStrings == 1) || (NumStrings == 2)); // there should be a
single '=' within a tag/value pair

        if (!RequiredAssetDataTags.IsValid())
        {
            RequiredAssetDataTags = MakeShared< FAssetDataTagMap>();
        }
        RequiredAssetDataTags->Add(FName(*TagAndOptionalValue[0]), (NumStrings >
1) ? TagAndOptionalValue[1] : FString());
    }
}
}

bool SPropertyEditorAsset::IsAssetFiltered(const FAssetData& InAssetData)
{

```

```

//Judgment is made that the presence of tags does not meet the criteria
if (DisallowedAssetDataTags.IsValid())
{
    for (const auto& DisallowedTagAndValue : *DisallowedAssetDataTags.Get())
    {
        if
(InAssetData.TagsAndValues.ContainsKeyValue(DisallowedTagAndValue.Key,
DisallowedTagAndValue.value))
        {
            return true;
        }
    }
}
//Judgment is made that the presence of tags is required to avoid being
filtered out
if (RequiredAssetDataTags.IsValid())
{
    for (const auto& RequiredTagAndValue : *RequiredAssetDataTags.Get())
    {
        if
(!InAssetData.TagsAndValues.ContainsKeyValue(RequiredTagAndValue.Key,
RequiredTagAndValue.value))
        {
            // For backwards compatibility compare against short name version
            // of the tag value.
            if (!FPackageName::IsShortPackageName(RequiredTagAndValue.value)
&&
InAssetData.TagsAndValues.ContainsKeyValue(RequiredTagAndValue.Key,
FPackageName::ObjectPathToObjectName(RequiredTagAndValue.value)))
            {
                continue;
            }
            return true;
        }
    }
    return false;
}

void UDataTable::GetAssetRegistryTags(FAssetRegistryTagsContext Context) const
{
    if (AssetImportData)
    {
        Context.AddTag( FAssetRegistryTag(SourceFileTagName(), AssetImportData-
>GetSourceData().ToJson(), FAssetRegistryTag::TT_Hidden) );
    }

    // Add the row structure tag
    {
        static const FName RowStructureTag = "RowStructure";
        Context.AddTag( FAssetRegistryTag(RowStructureTag,
GetRowStructPathName().ToString(), FAssetRegistryTag::TT_Alphabetical) );
    }

    Super::GetAssetRegistryTags(Context);
}

```

```
}
```

## DisallowedAssetDataTags

- **Function Description:** Specifies tags on UObject\* properties for filtering purposes; an asset must not possess these tags to be eligible for selection.
- **Usage Location:** UPROPERTY
- **Engine Module:** Asset Property
- **Metadata Type:** strings="a=b, c=d, e=f"
- **Restriction Type:** UObject\*
- **Associated Items:** RequiredAssetDataTags, AssetRegistrySearchable
- **Commonality:** ★★

## ForceShowEngineContent

- **Function Description:** Forces the built-in engine resources to be optionally selectable in the resource selection list for a UObject\* attribute
- **Usage Location:** UPROPERTY
- **Engine Module:** Asset Property
- **Metadata Type:** bool
- **Restriction Type:** UObject\*
- **Associated Items:** ForceShowPluginContent
- **Commonliness:** ★★

Enforces the option to select the built-in resources of the engine in the resource selection list for the specified UObject\* attribute.

### Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_ShowContent :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object)
    TObjectPtr<UObject> MyAsset_Default;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object, meta =
(ForceShowEngineContent))
    TObjectPtr<UObject> MyAsset_ForceShowEngineContent;

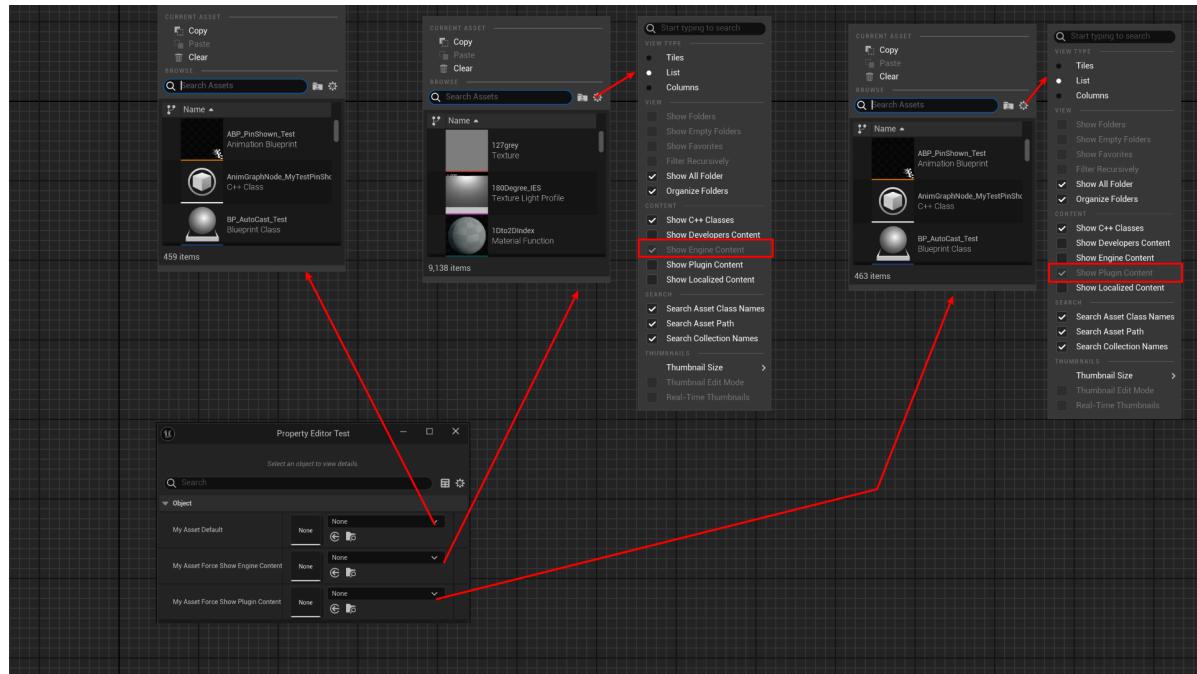
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Object, meta =
(ForceShowPluginContent))
    TObjectPtr<UObject> MyAsset_ForceShowPluginContent;
};
```

# Test Results:

By default, MyAsset\_Default is visible, containing only the resources of the current project.

The function of MyAsset\_ForceShowEngineContent is essentially to check the "ShowEngineContent" option in the tab, which results in a significantly larger number of optional resources being displayed.

The function of MyAsset\_ForceShowPluginContent is similarly to check the "ShowPluginContent" option in the tab, allowing selection of resources from other plugins.



## Principle:

In the resource selector for the property, the system will look for ForceShowEngineContent and ForceShowPluginContent, and then set them in AssetPickerConfig to alter the type of resources that can be optionally selected.

```
void SPropertyMenuAssetPicker::Construct( const FArguments& InArgs )
{
    const bool bForceShowEngineContent = PropertyHandle ? PropertyHandle->HasMetaData(TEXT("ForceShowEngineContent")) : false;
    const bool bForceShowPluginContent = PropertyHandle ? PropertyHandle->HasMetaData(TEXT("ForceshowPluginContent")) : false;

    FAssetPickerConfig AssetPickerConfig;
    // Force show engine content if meta data says so
    AssetPickerConfig.bForceShowEngineContent = bForceShowEngineContent;
    // Force show plugin content if meta data says so
    AssetPickerConfig.bForceShowPluginContent = bForceShowPluginContent;
}
```

# ForceShowPluginContent

- **Function Description:** Forces the optional resource list of a specified UObject\* attribute to include built-in resources from other plugins
- **Usage Location:** UPROPERTY
- **Engine Module:** Asset Property
- **Metadata Type:** bool
- **Restriction Type:** UObject\*
- **Associated Items:** ForceShowEngineContent

# GetAssetFilter

- **Function Description:** Specifies a UFUNCTION to filter out and exclude optional resources for a UObject\* attribute.
- **Usage Location:** UPROPERTY
- **Engine Module:** Asset Property
- **Metadata Type:** string="abc"
- **Restriction Type:** UObject\*
- **Commonliness:** ★★★

Specifies a UFUNCTION to filter out and exclude optional resources for a UObject\* attribute.

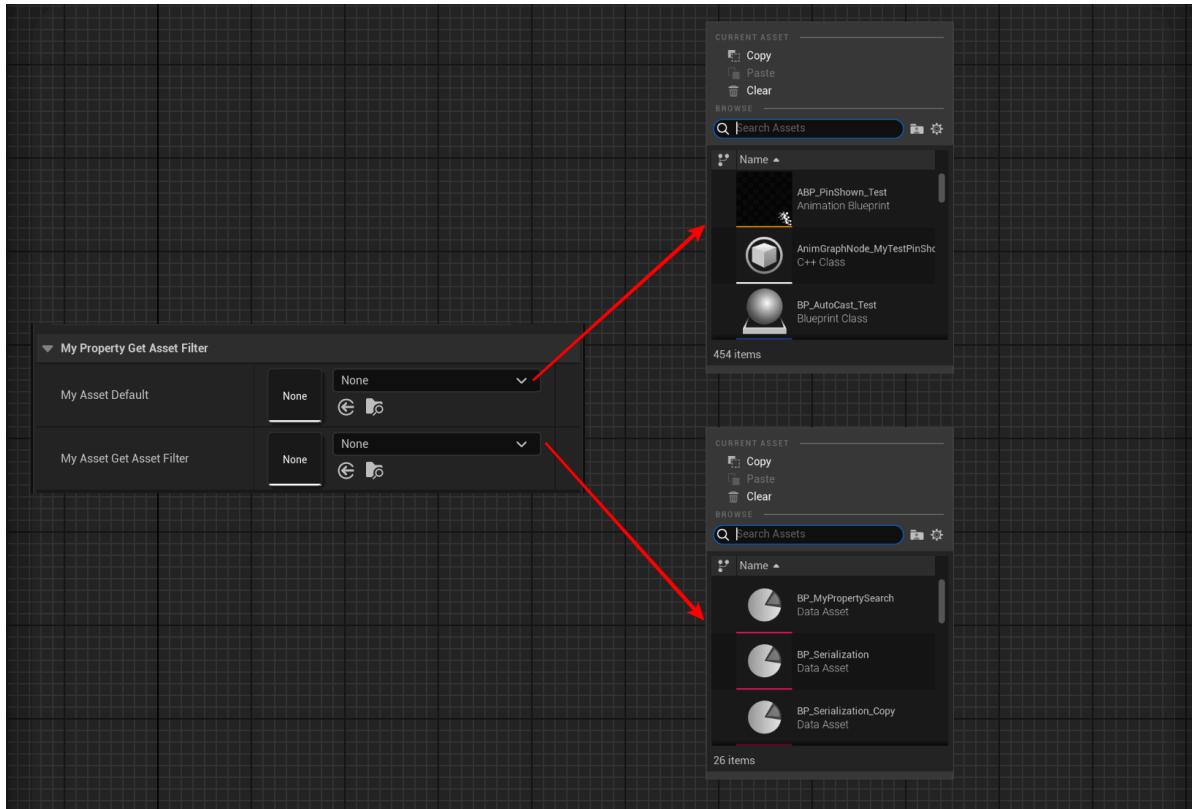
- The specified function name must be a UFUNCTION defined within this class.
- The prototype of the function is bool FuncName ( const FAssetData& AssetData ) const; and returns true to exclude the asset.
- This provides a method for users to customize asset filtering.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_GetAssetFilter :public UDataAsset
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UObject* MyAsset_Default;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (GetAssetFilter =
"IsShouldFilterAsset"))
    UObject* MyAsset_GetAssetFilter;
public:
    UFUNCTION()
    bool IsShouldFilterAsset(const FAssetData& AssetData)
    {
        return !AssetData.IsInstanceOf<UDataAsset>();
    }
};
```

## Test Effects:

It can be observed that after MyAsset\_GetAssetFilter performs the filtering, only assets of type DataAsset are allowed.



## Principle:

In SPropertyEditorAsset (which corresponds to properties of type UObject), there is a check on the meta for GetAssetFilter, from which the function obtained, it is attached to the callback for asset exclusion.

```
void SPropertyEditorAsset::Construct(const FArguments& InArgs, const TSharedPtr<FPropertyEditor>& InPropertyEditor)
{
    if (Property && Property->GetOwnerProperty()->HasMetaData("GetAssetFilter"))
    {
        // Add MetaData asset filter
        const FString GetAssetFilterFunctionName = Property->GetOwnerProperty()->GetMetaData("GetAssetFilter");
        if (!GetAssetFilterFunctionName.IsEmpty())
        {
            TArray<UObject*> ObjectList;
            if (PropertyEditor.IsValid())
            {
                PropertyEditor->GetPropertyHandle()->GetOuterObjects(ObjectList);
            }
            else if (PropertyHandle.IsValid())
            {
                PropertyHandle->GetOuterObjects(ObjectList);
            }
            for (UObject* object : ObjectList)
            {

```

```

        const UFunction* GetAssetFilterFunction = Object ? Object-
>FindFunction(*GetAssetFilterFunctionName) : nullptr;
        if (GetAssetFilterFunction)
        {

AppendOnShouldFilterAssetCallback(FOnShouldFilterAsset::Create(Object,
GetAssetFilterFunction->GetFName())));
        }
    }
}
}

```

## IgnoreTypePromotion

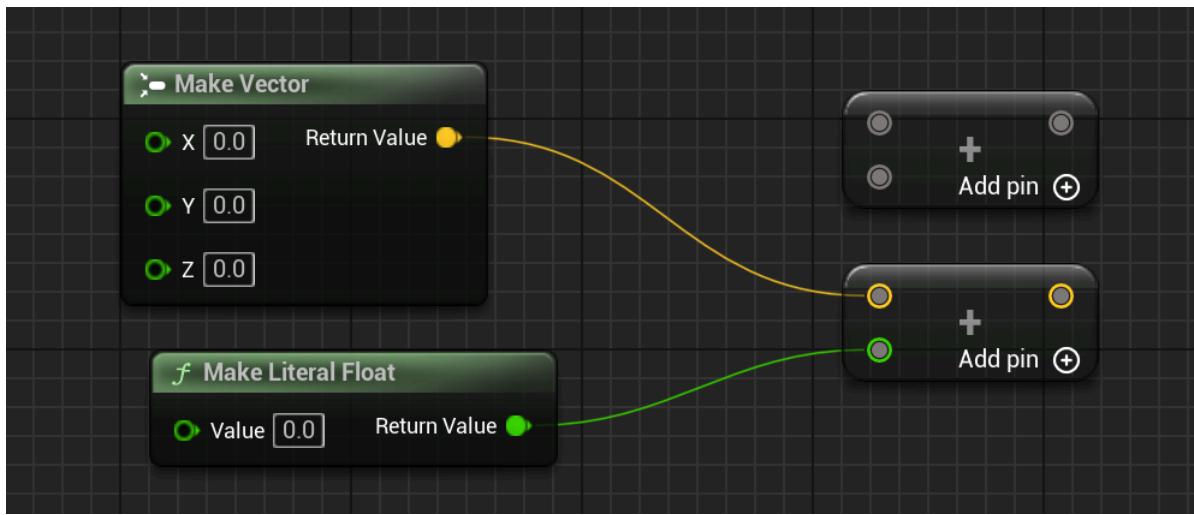
- **Function Description:** Marks this function as excluded from the type promotion function library
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Restriction Type:** BlueprintPure functions within UBlueprintFunctionLibrary, formatted as OP\_XXX
- **Commonality:** ★★

Mark this function as excluded from the type promotion function library.

## There are three key points to consider:

First, what type of function is this? Or, what is the nature of a type promotion function? According to the IsPromotableFunction source code definition, the function must be defined within UBlueprintFunctionLibrary, must be of type BlueprintPure, and must be named in the format of an operator "OP\_XXX," where the OP name is visible within the OperatorNames namespace. Examples of such functions can be found in abundance within KismetMathLibrary.

Second, what is a type promotion function library? The source code contains a class named FTypePromotion, which includes an OperatorTable that records a map from operator names to lists of functions, such as multiple Add\_Vector, Add\_Float, etc., that support the Add (+) operation. When right-clicking on a + or Add node in a blueprint, a generic + node is initially presented. Then, upon connecting to a specific variable type, the blueprint system will search the FTypePromotion::OperatorTable for the most compatible function to call, or perform type promotion automatically within the system. For instance, the + in the figure below ultimately calls UKismetMathLibrary::Add\_VectorFloat. This generic operator call facilitates a more convenient and unified approach to basic type operations in blueprint node creation, as well as direct Add Pin and Pin conversion to compatible types.



Third, why do some functions opt out of being included in FTypePromotion? Upon searching the source code, it is found that only FDateTime in KismetMathLibrary is marked with the IgnoreTypePromotion attribute. Although FDateTime defines a series of operator functions, such as Add, Subtract, and various comparison operators, FDateTime differs from other basic types in that FDateTime + float or FDateTime + vector hold no meaningful operation. FDateTime only allows operations with +FDateTime or +FTimeSpan. Therefore, for types like FDateTime that do not wish to participate in type promotion relationships with other types, and prefer to operate within their own limited scope, the IgnoreTypePromotion attribute can be added to opt out of the FTypePromotion system.

## Test Code:

Suppose we have a structure called FGameProp that defines combat attributes (HP, Attack, Defense) within the game. In the game, operations such as equipping and adding buffs typically calculate final attributes. For this FGameProp structure, we can define a series of basic operation functions and add the IgnoreTypePromotion attribute, as it is certainly not intended to participate in type promotion with other basic types (float, vector, etc.).

For comparison, the code also defines an identical structure, FGameProp2, with the same operation functions, the only difference being the absence of IgnoreTypePromotion. The purpose is to observe the differences in the resulting blueprint nodes.

```
USTRUCT(BlueprintType)
struct FGameProp
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    double HP;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    double Attack;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    double Defense;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_IgnoreTypePromotion : public
UBlueprintFunctionLibrary
{
public:
```

```

GENERATED_BODY()
public:
    /** Makes a GameProp struct */
    UFUNCTION(BlueprintPure, Category = "Math|GameProp", meta =
(IgnoreTypePromotion, NativeMakeFunc))
    static FGameProp MakeGameProp(double HP, double Attack, double Defense) {
return FGameProp(); }

    /** Breaks a GameProp into its components */
    UFUNCTION(BlueprintPure, Category = "Math|GameProp", meta =
(IgnoreTypePromotion, NativeBreakFunc))
    static void BreakGameProp(FGameProp InGameProp, double& HP, double& Attack,
double& Defense) {}

    /** Addition (A + B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
+ GameProp", CompactNodeTitle = "+", Keywords = "+ add plus"), Category =
"Math|GameProp")
    static FGameProp Add_GameProp(FGameProp A, FGameProp B);

    /** Subtraction (A - B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
- GameProp", CompactNodeTitle = "-", Keywords = "- subtract minus"), Category =
"Math|GameProp")
    static FGameProp Subtract_GameProp(FGameProp A, FGameProp B) { return
FGameProp(); }

    /** Returns true if the values are equal (A == B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "Equal
(GameProp)", CompactNodeTitle = "==", Keywords = "== equal"), Category =
"Math|GameProp")
    static bool EqualEqual_GameProp(FGameProp A, FGameProp B) { return true; }

    /** Returns true if the values are not equal (A != B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "Not
Equal (GameProp)", CompactNodeTitle = "!="), Keywords = "!= not equal"), Category =
"Math|GameProp")
    static bool NotEqual_GameProp(FGameProp A, FGameProp B) { return true; }

    /** Returns true if A is greater than B (A > B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
> GameProp", CompactNodeTitle = ">", Keywords = "> greater"), Category =
"Math|GameProp")
    static bool Greater_GameProp(FGameProp A, FGameProp B) { return true; }

    /** Returns true if A is greater than or equal to B (A >= B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
>= GameProp", CompactNodeTitle = ">=", Keywords = ">= greater"), Category =
"Math|GameProp")
    static bool GreaterEqual_GameProp(FGameProp A, FGameProp B) { return true; }

    /** Returns true if A is less than B (A < B) */
    UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
< GameProp", CompactNodeTitle = "<", Keywords = "< less"), Category =
"Math|GameProp")
    static bool Less_GameProp(FGameProp A, FGameProp B) { return true; }

```

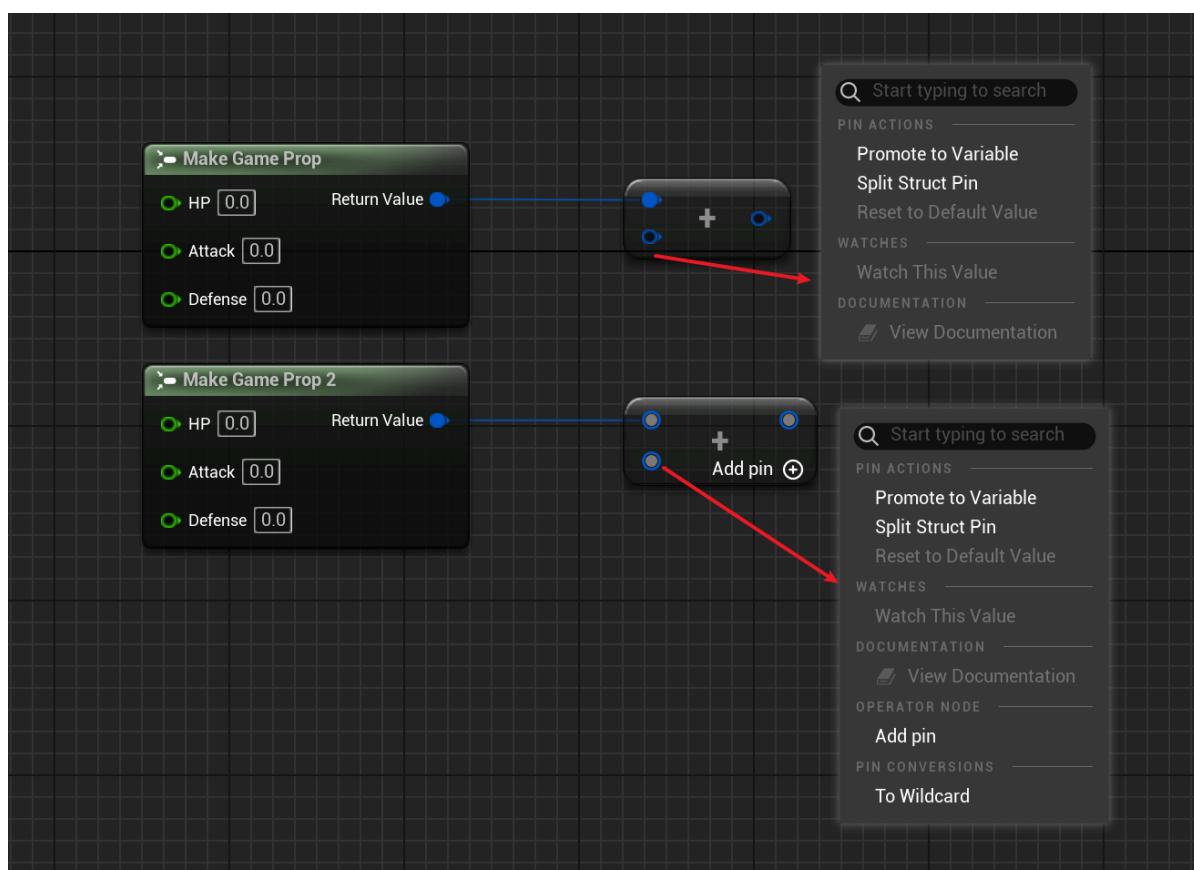
```

/** Returns true if A is less than or equal to B (A <= B) */
UFUNCTION(BlueprintPure, meta = (IgnoreTypePromotion, DisplayName = "GameProp
<= GameProp", CompactNodeTitle = "<=", Keywords = "<= less"), Category =
"Math|GameProp")
static bool LessEqual_GameProp(FGameProp A, FGameProp B) { return true; }
;

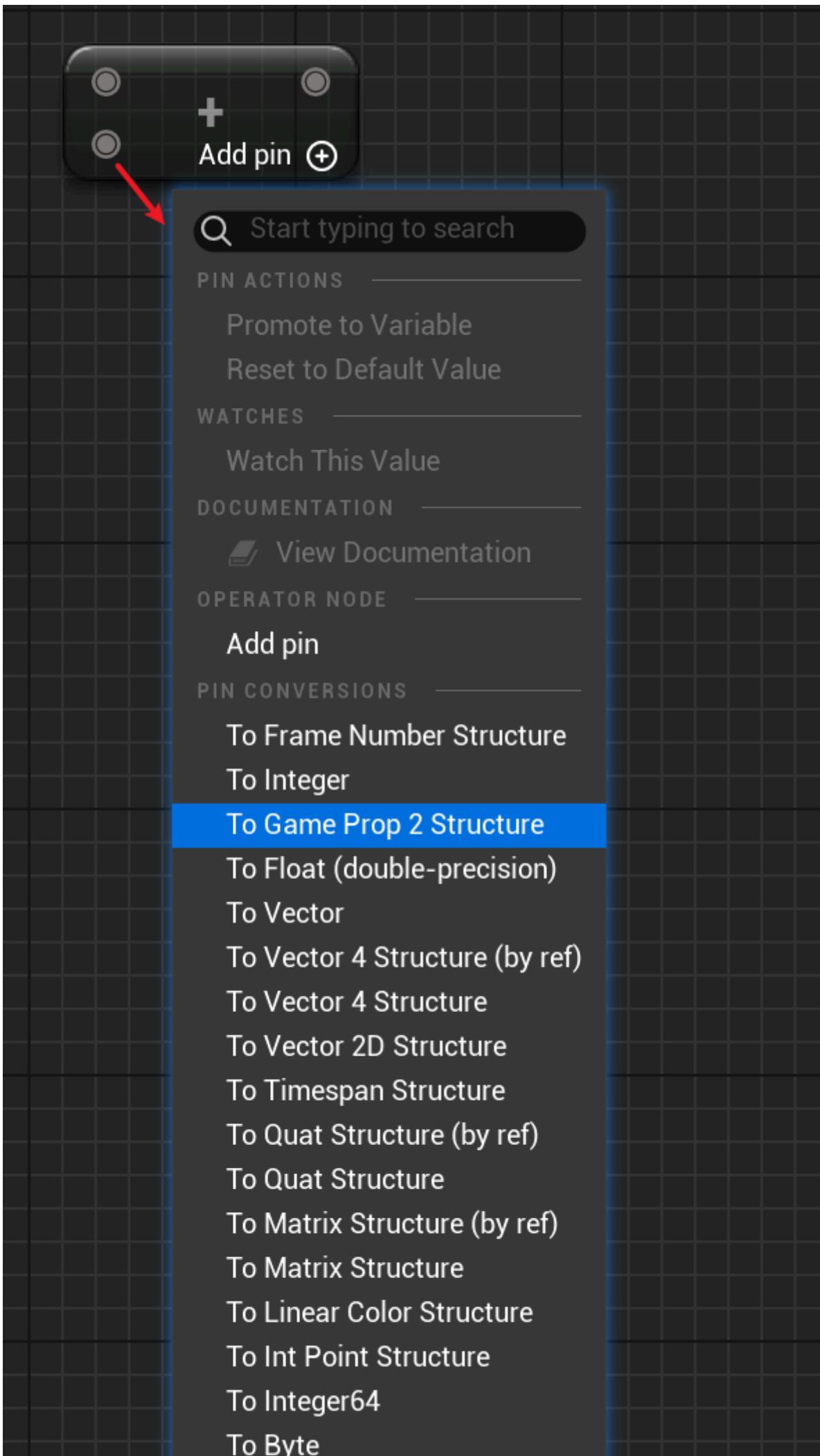
```

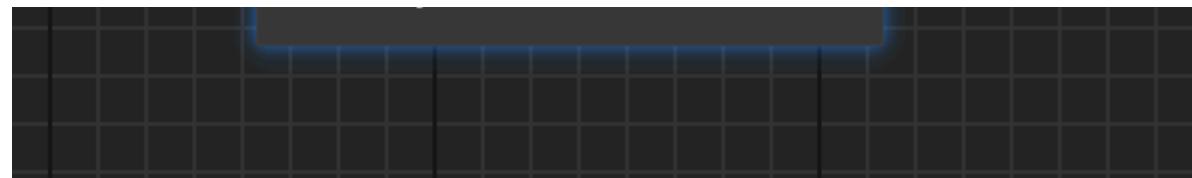
## Blueprint Effect:

With IgnoreTypePromotion added, the FGameProp Add operation results in the direct use of the original Add\_GameProp node. Without IgnoreTypePromotion, FGameProp2's Add operation generates a generic + node, which can be further used for Add Pin, and even attempts to find conversions to other types on the Pin (though no conversion is found here because we have not defined operation functions for FGameProp2 with other types).



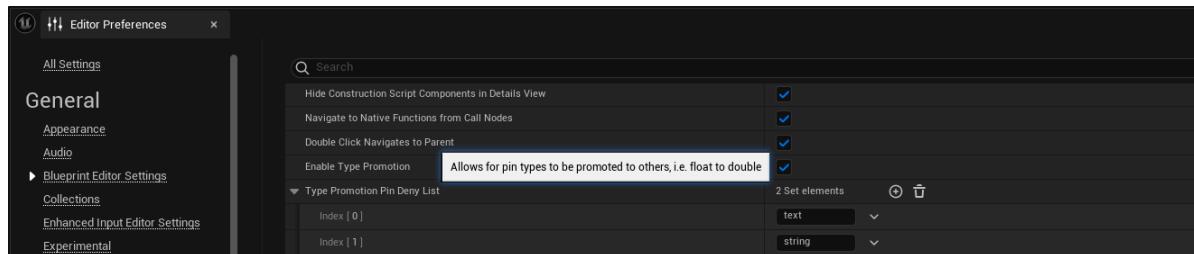
Another observation is that right-clicking on an empty generic Add node reveals an option to convert to FGameProp2 (but not FGameProp), indicating that FGameProp2 is part of the TypePromotion system. However, we do not want FGameProp2 to appear here, as the combat attributes in this gameplay have their own set of operation rules and should not be mixed with basic type mathematical operations.





## Principle:

In the editor settings, there is an option called `EnableTypePromotion`. When activated, `FTypePromotion` begins to collect all functions defined within the engine and determines whether they qualify as type promotion functions.



If a function name includes an operator prefix (as defined in `OperatorNames`), such as `Add_XXX`, the operator is extracted. Functions that are registered and added to the `FTypePromotion::OperatorTable` mapping are those that will be found when right-clicking on operators (like `+`) in blueprints.

```
namespace OperatorNames
{
    static const FName NoOp           = TEXT("NO_OP");

    static const FName Add            = TEXT("Add");
    static const FName Multiply      = TEXT("Multiply");
    static const FName Subtract      = TEXT("Subtract");
    static const FName Divide        = TEXT("Divide");

    static const FName Greater       = TEXT("Greater");
    static const FName GreaterEq     = TEXT("GreaterEqual");
    static const FName Less          = TEXT("Less");
    static const FName LessEq        = TEXT("LessEqual");
    static const FName NotEq         = TEXT("NotEqual");
    static const FName Equal         = TEXT("EqualEqual");
}

bool const bIsPromotableFunction = TypePromoDebug::IsTypePromoEnabled() &&
FTypePromotion::IsFunctionPromotionReady(Function);
if (bIsPromotableFunction)
{
    NodeClass = UK2Node_PromotableOperator::StaticClass();
}

bool FTypePromotion::IsPromotableFunction(const UFunction* Function)
{
    TRACE_CUPROFILER_EVENT_SCOPE(FTypePromotion::IsPromotableFunction);

    // Ensure that we don't have an invalid OpName as well for extra safety when
    // this function
    // is called outside of this class, not during the OpTable creation process
}
```

```

FName OpName = GetOpNameFromFunction(Function);
return Function &&
       Function->HasAnyFunctionFlags(FUNC_BlueprintPure) &&
       Function->GetReturnProperty() &&
       OpName != OperatorNames::NoOp &&
       !IsPinTypeDeniedForTypePromotion(Function) &&
       // Users can deny specific functions from being considered for type
       promotion
       !Function->HasMetaData(FBlueprintMetadata::MD_IgnoreTypePromotion);
}

```

The contents of the OperatorTable collected by FTypePromotion:

Index	Operator	Value
[0]	NotEqual	31
[1]	EqualEqual	34
[2]	Subtract	18
[3]	Multiply	23
[4]	Divide	15
[5]	Add	20
[6]	LessEqual	6
[7]	Less	6
[8]	GreaterEqual	6
[9]	Greater	6
[Raw View]	...	

If a function is identified as IsPromotableFunction, it will use UK2Node\_PromotableOperator as the blueprint node (instead of the default UK2Node\_CallFunction) when called.

UK2Node\_PromotableOperator is a typical binary operator used for wildcard generics. As shown in the figure below, Add (+) can trigger a type conversion menu from Wildcard to a specific type because the structure has defined Add\_XXX functions and does not have IgnoreTypePromotion, thus being included in the TypePromotion mapping.

The Pin Conversion menu mentioned above is collected in UK2Node\_PromotableOperator::CreateConversionMenu.

## Variadic

- **Function Description:** Indicates that the function can accept a variable number of arguments
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Associated Items:**
  - UFUNCTION: 可变参数
- **Commonality:** ★★★

## ForceAsFunction

- **Function description:** Enforce the conversion of events defined with BlueprintImplementableEvent or NativeEvent in C++ to functions that must be overridden in subclasses.
- **Use location:** UFUNCTION

- **Engine module:** Blueprint
- **Metadata type:** bool
- **Frequency:** ★★★

Force events, defined with BlueprintImplementableEvent or NativeEvent in C++, to be converted into functions that are overridden in subclasses.

When is it necessary to convert an Event into a function?

- Once converted into a function, internal local variables can be defined within its implementation. However, this also means losing the capability to use delay functions such as Delay.
- Events cannot have output parameters, but if a function with an output is desired to be overridden in a blueprint class (requiring BlueprintImplementableEvent or NativeEvent), the default event-style overload is insufficient. Thus, forcing the event to be a function allows for proper overriding.
- Events with output or return parameters will be automatically converted to functions, even without the ForceAsFunction metadata.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_ForceAsFunction :public AActor
{
public:
    GENERATED_BODY()
public:
//FUNC_Native | FUNC_Event | FUNC_Public | FUNC_BlueprintCallable | 
FUNC_BlueprintEvent
    UFUNCTION(BlueprintCallable, BlueprintNativeEvent)
    void MyNativeEvent_Default(const FString& name);

//FUNC_Event | FUNC_Public | FUNC_BlueprintCallable | FUNC_BlueprintEvent
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
    void MyImplementableEvent_Default(const FString& name);

public:
    //((ForceAsFunction = , ModuleRelativePath =
Function/MyFunction_ForceAsFunction.h)
    //FUNC_Native | FUNC_Event | FUNC_Public | FUNC_BlueprintCallable | 
FUNC_BlueprintEvent
    UFUNCTION(BlueprintCallable, BlueprintNativeEvent, meta = (ForceAsFunction))
    void MyNativeEvent_ForceAsFunction(const FString& name);

    ////(ForceAsFunction = , ModuleRelativePath =
Function/MyFunction_ForceAsFunction.h)
    //FUNC_Event | FUNC_Public | FUNC_BlueprintCallable | FUNC_BlueprintEvent
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent, meta =
(ForceAsFunction))
    void MyImplementableEvent_ForceAsFunction(const FString& name);

public:
```

```

//FUNC_Native | FUNC_Event | FUNC_Public | FUNC_HasOutParms | 
FUNC_BlueprintCallable | FUNC_BlueprintEvent
UFUNCTION(BlueprintCallable, BlueprintNativeEvent)
bool MyNativeEvent_Output(const FString& name, int32& outValue);

//FUNC_Event | FUNC_Public | FUNC_HasOutParms | FUNC_BlueprintCallable | 
FUNC_BlueprintEvent
UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
bool MyImplementableEvent_Output(const FString& name, int32& outValue);

//(ForceAsFunction = , ModuleRelativePath =
Function/MyFunction_ForceAsFunction.h)
//FUNC_Native | FUNC_Event | FUNC_Public | FUNC_HasOutParms | 
FUNC_BlueprintCallable | FUNC_BlueprintEvent
UFUNCTION(BlueprintCallable, BlueprintNativeEvent, meta = (ForceAsFunction))
bool MyNativeEvent_Output_ForceAsFunction(const FString& name, int32&
outValue);

//(ForceAsFunction = , ModuleRelativePath =
Function/MyFunction_ForceAsFunction.h)
//FUNC_Event | FUNC_Public | FUNC_HasOutParms | FUNC_BlueprintCallable | 
FUNC_BlueprintEvent
UFUNCTION(BlueprintCallable, BlueprintImplementableEvent, meta =
(ForceAsFunction))
bool MyImplementableEvent_Output_ForceAsFunction(const FString& name, int32&
outValue);
};

```

## Effect in Blueprint:

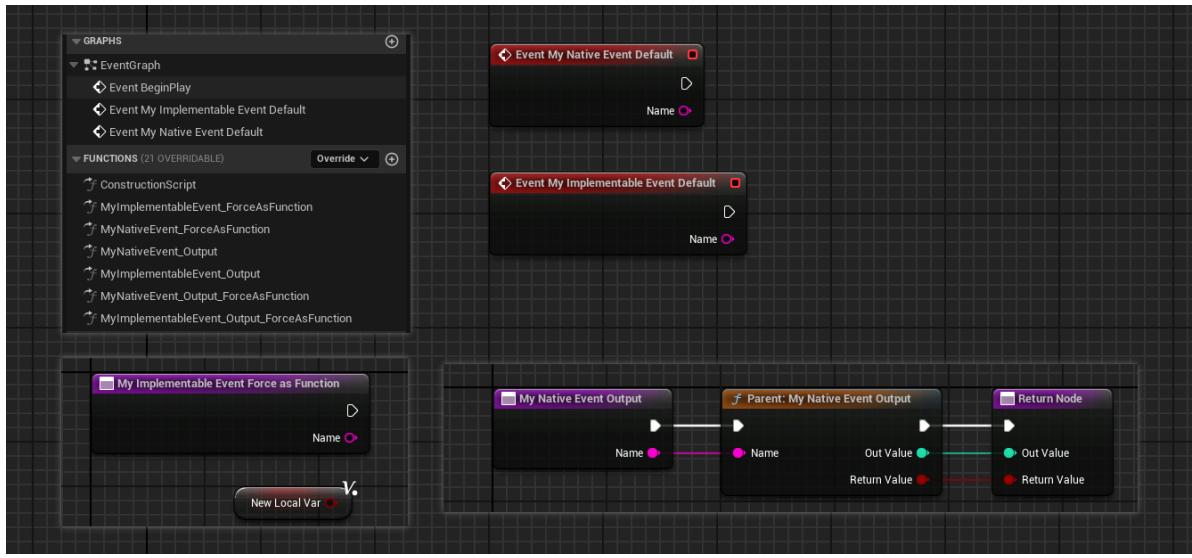
---

When functions are overridden, you will notice that only MyNativeEvent\_Default and MyImplementableEvent\_Default are overridden as events by default, while the rest are overridden as functions.

The diagram illustrates that after MyImplementableEvent\_ForceAsFunction is converted to a function, local variables can be defined within it.

It also shows the function body of MyNativeEvent\_Output, an event with output parameters, after it is overwritten as a function.

Regardless of whether it is overridden as an event or a function, the usage remains the same when called.



## Principle:

The logic for determining whether a function is an event is based on the following function:

Primarily, the second 'if' condition and the final judgment are considered. Functions marked with BlueprintImplementableEvent or NativeEvent will have the FUNC\_BlueprintEvent tag added. Therefore, if they have ForceAsFunction metadata or output parameters (including return values), they can only be displayed as functions.

```
bool UEdGraphSchema_K2::FunctionCanBePlacedAsEvent(const UFunction* InFunction)
{
    // First check we are override-able, non-static, non-const and not marked
    // thread safe
    if (!InFunction || !CanKismetOverrideFunction(InFunction) || InFunction-
>HasAnyFunctionFlags(FUNC_Static|FUNC_Const) ||
        FBlueprintEditorUtils::HasFunctionBlueprintThreadSafeMetaData(InFunction))
    {
        return false;
    }

    // Check if meta data has been set to force this to appear as blueprint
    // function even if it doesn't return a value.
    if (InFunction->HasAllFunctionFlags(FUNC_BlueprintEvent) && InFunction-
>HasMetaData(FBlueprintMetadata::MD_ForceAsFunction))
    {
        return false;
    }

    // Then look to see if we have any output, return, or reference params
    return !HasFunctionAnyOutputParameter(InFunction);
}
```

## CannotImplementInterfaceInBlueprint

- **Function Description:** Specifies that this interface cannot be implemented within a Blueprint
- **Engine Module:** Blueprint
- **Metadata Type:** bool

- **Associated Items:**

UINTERFACE: NotBlueprintable

- **Commonality:** ★★★

Functions similarly to UINTERFACE(NotBlueprintable), indicating that it is not possible to inherit this in a Blueprint

## CallInEditor

---

- **Function Description:** Can be invoked as a button on the details panel of an Actor.

- **Usage Location:** UFUNCTION

- **Engine Module:** Blueprint

- **Metadata Type:** bool

- **Associated Items:**

UFUNCTION: Call In Editor

- **Commonality:** ★★★★☆

## BlueprintProtected

---

- **Function description:** Specifies that the function or attribute can only be called or accessed for reading and writing within this class and its subclasses, akin to the protected scope restriction in C++. It is not accessible from other Blueprint classes.

- **Usage location:** UFUNCTION, UPROPERTY

- **Engine module:** Blueprint

- **Metadata type:** bool

- **Related items:** BlueprintPrivate, AllowPrivateAccess

- **Commonality:** ★★★

Effect on functions:

Indicates that the function can only be invoked within this class and its subclasses, similar to the scope restriction of protected functions in C++. It cannot be invoked from other Blueprint classes.

When applied to an attribute, it signifies that the attribute can only be read and written within this class or its derived classes, but not accessible from other Blueprint classes.

Specifies that the function or attribute can only be called or accessed for reading and writing within this class and its subclasses, similar to the protected scope restriction in C++. It is not accessible from other Blueprint classes.

## Test code:

---

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Access :public AActor
{
public:
    GENERATED_BODY()
public:
```

```

//((BlueprintProtected = true, ModuleRelativePath =
Function/MyFunction_Access.h)
//FUNC_Final | FUNC_Native | FUNC_Public | FUNC_BlueprintCallable
UFUNCTION(BlueprintCallable, meta = (BlueprintProtected = "true"))
void MyNative_HasProtected() {}

//((BlueprintPrivate = true, ModuleRelativePath =
Function/MyFunction_Access.h)
//FUNC_Final | FUNC_Native | FUNC_Public | FUNC_BlueprintCallable
UFUNCTION(BlueprintCallable, meta = (BlueprintPrivate = "true"))
void MyNative_HasPrivate() {}

public:
//FUNC_Final | FUNC_Native | FUNC_Public | FUNC_BlueprintCallable
UFUNCTION(BlueprintCallable)
void MyNative_NativePublic() {}

protected:
//FUNC_Final | FUNC_Native | FUNC_Protected | FUNC_BlueprintCallable
UFUNCTION(BlueprintCallable)
void MyNative_NativeProtected() {}

private:
//FUNC_Final | FUNC_Native | FUNC_Private | FUNC_BlueprintCallable
UFUNCTION(BlueprintCallable)
void MyNative_NativePrivate() {}

;

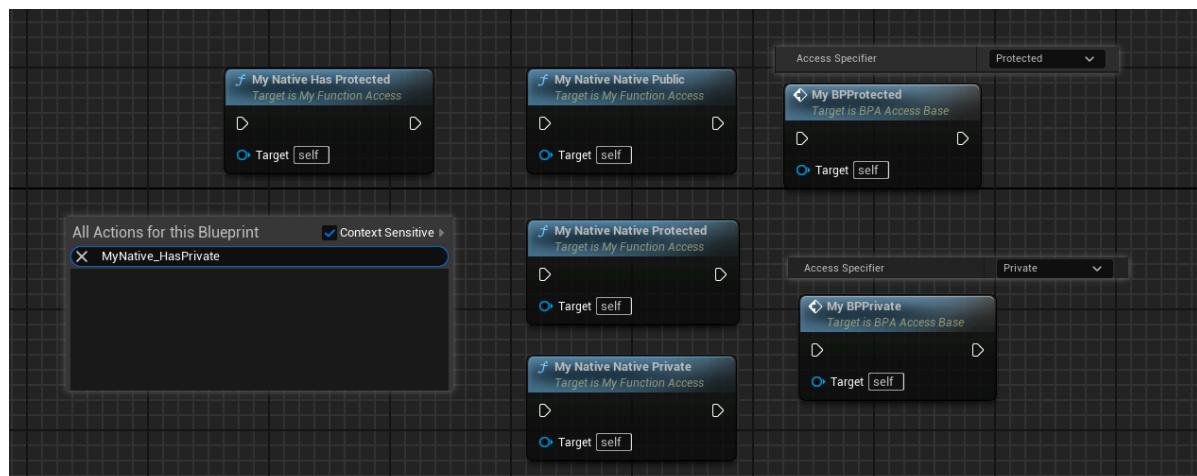
```

## Test results:

Effect in Blueprint subclass (BPA\_Access\_Base inheriting from AMyFunction\_Access):

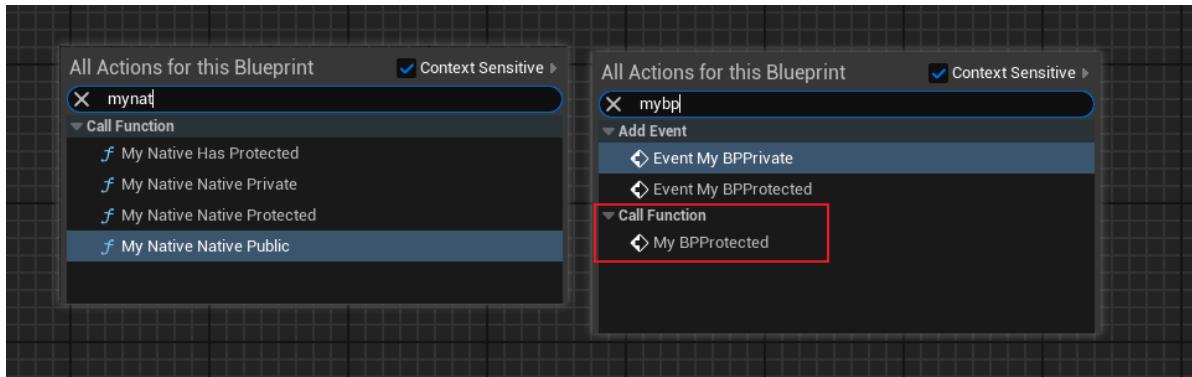
BlueprintProtected can be invoked by subclasses, but BlueprintPrivate can only be called within the class itself (those defined in C++ classes can only be called in C++, and those defined in Blueprints can only be called within the Blueprint class itself). Functions marked with protected or private in C++ will accordingly increase FUNC\_Protected or FUNC\_Private, but they do not actually take effect. This is because the design intent of the mechanism is as such (see detailed explanation below).

MyBPProtected and MyBPPPrivate, defined directly in BPA\_Access\_Base, can be called within the class by setting the AccessSpecifier directly in the function details panel, but MyBPPPrivate cannot be called in further Blueprint subclasses.

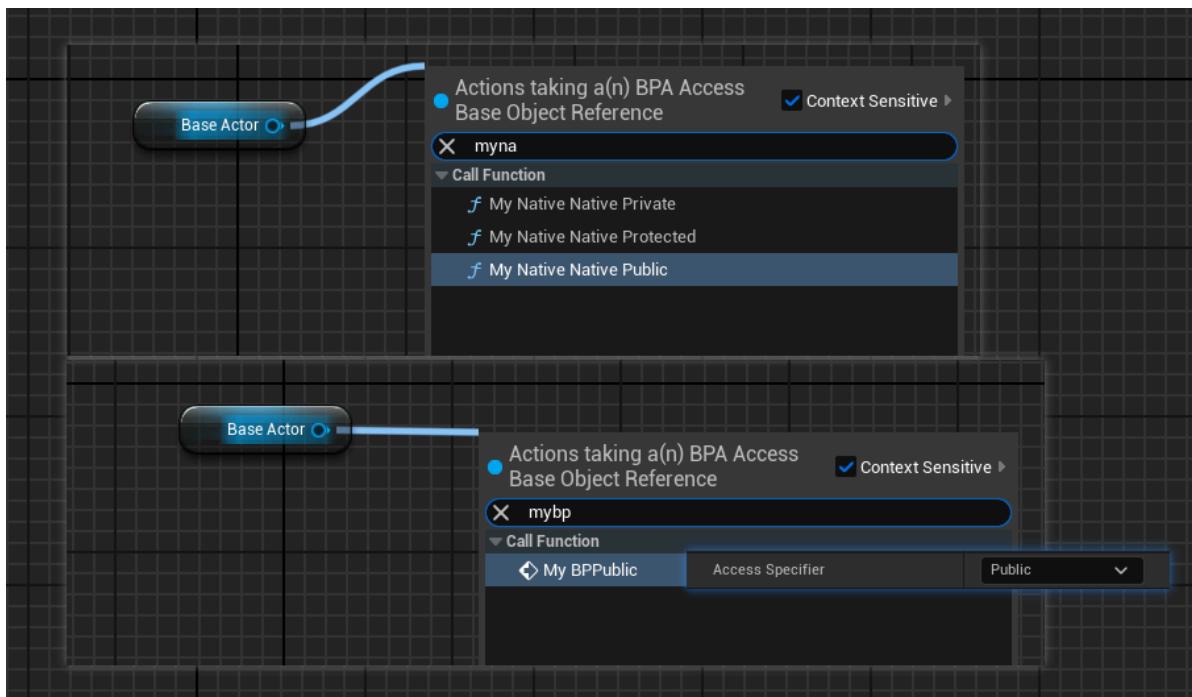


Effect in Blueprint subclass (BPA\_Access\_Child inheriting from BPA\_Access\_Base):

It can be observed that the access to MyNative functions remains the same. MyBPPrivate cannot be called, aligning with our expected rules.



In external classes (BPA\_Access\_Other, inheriting from Actor), when accessing functions through BPA\_Access\_Base or BPA\_Access\_Child object instances, it is found that neither BlueprintProtected nor BlueprintPrivate can be called. Only functions with the default Public AccessSpecifier can be called. This rule also aligns with expectations.



## Principle:

Is it possible to select the filtering logic for this function by right-clicking in the Blueprint:

If it is a static function, it is always possible. Otherwise, there must be no BlueprintProtected or BlueprintPrivate for it to be selectable as Public.

If it is Private, the external class must be the class itself.

If it is Protected, the external class only needs to be the class itself or a subclass.

```
static bool BlueprintActionFilterImpl::IsFieldInaccessible(FBlueprintActionFilter
const& Filter, FBlueprintActionInfo& BlueprintAction)
{
    bool const bIsProtected =
        Field.HasMetaData(FBlueprintMetadata::MD_Protected);
    bool const bIsPrivate   =
        Field.HasMetaData(FBlueprintMetadata::MD_Private);
```

```

        bool const bIsPublic    = !bIsPrivate && !bIsProtected;

        if( !bIsPublic )
        {
            UClass const* ActionOwner = BlueprintAction.GetOwnerClass();
            for (UBlueprint const* Blueprint : FilterContext.Blueprints)
            {
                UClass const* BpClass =
GetAuthoritativeBlueprintClass(Blueprint);
                if (!ensureMsgf(BpClass != nullptr
                    , TEXT("Unable to resolve IsFieldInaccessible() - Blueprint
(%s) missing an authoritative class (skel: %s, generated: %s, parent: %s)")
                    , *Blueprint->GetName()
                    , Blueprint->SkeletonGeneratedClass ? *Blueprint-
>SkeletonGeneratedClass->GetName() : TEXT("[NULL]")
                    , Blueprint->GeneratedClass ? *Blueprint->GeneratedClass-
>GetName() : TEXT("[NULL]")
                    , Blueprint->ParentClass ? *Blueprint->ParentClass->GetName()
: TEXT("[NULL]")))
                {
                    continue;
                }

                // private functions are only accessible from the class they
belong to
                if (bIsPrivate && !IsClassOfType(BpClass, ActionOwner,
/*bNeedsExactMatch =*/true))
                {
                    bIsFilteredOut = true;
                    break;
                }
                else if (bIsProtected && !IsClassOfType(BpClass, ActionOwner))
                {
                    bIsFilteredOut = true;
                    break;
                }
            }
        }
    }

bool UEdGraphSchema_K2::ClassHasBlueprintAccessibleMembers(const UClass* InClass)
const
{
    // @TODO Don't show other blueprints yet...
    UBlueprint* ClassBlueprint = UBlueprint::GetBlueprintFromClass(InClass);
    if (!InClass->HasAnyClassFlags(CLASS_Deprecated | CLASS_NewerVersionExists)
&& (ClassBlueprint == NULL))
    {
        // Find functions
        for (TFieldIterator<UFunction> FunctionIt(InClass,
EFieldIteratorFlags::IncludeSuper); FunctionIt; ++FunctionIt)
        {
            UFunction* Function = *FunctionIt;
            const bool bIsBlueprintProtected = Function-
>GetBoolMetaData(FBlueprintMetadata::MD_Protected);

```

```

        const bool bHidden =
FObjectEditorUtils::IsFunctionHiddenFromClass(Function, InClass);
        if (UEdGraphSchema_K2::CanUserKismetCallFunction(Function) &&
!bIsBlueprintProtected && !bHidden)
    {
        return true;
    }
}

// Find vars
for (TFieldIterator<FProperty> PropertyIt(InClass,
EFieldIteratorFlags::IncludeSuper); PropertyIt; ++PropertyIt)
{
    FProperty* Property = *PropertyIt;
    if (CanUserKismetAccessVariable(Property, InClass, CannotBeDelegate))
    {
        return true;
    }
}

return false;
}

```

If a function defined in BP is set to Protected or Private through AccessSpecifier, the function will be added accordingly with FUNC\_Protected or FUNC\_Private. This actually affects the function's scope. However, many checks in the source code will first determine if it is a Native function, and if so, no further restrictions are applied. Therefore, we can understand that this is an intentional design choice by UE, not considering C++ protected and private scopes, and requiring explicit manual marking of BlueprintProtected or BlueprintPrivate to avoid ambiguity.

```

bool UEdGraphSchema_K2::CanFunctionBeUsedInGraph(const UClass* InClass, const
UFunction* InFunction, const UEdGraph* InDestGraph, uint32
InAllowedFunctionTypes, bool bInCalledForEach, FText* OutReason) const
{
    const bool bIsNotNative =
!FBlueprintEditorUtils::IsNativeSignature(InFunction);
    if(bIsNotNative)
    {
        // Blueprint functions visibility flags can be enforced in blueprints -
native functions
        // are often using these flags to only hide functionality from other
native functions:
        const bool bIsProtected = (InFunction->FunctionFlags & FUNC_Protected) !=
0;
    }

    bool bIsFilteredOut = false;
    for(UEdGraph* TargetGraph : Filter.Context.Graphs)
    {
        bIsFilteredOut |= !CanPasteHere(TargetGraph);
    }
}

```

```

    }

    if(const UFunction* TargetFunction = GetTargetFunction())
    {
        const bool bIsProtected = (TargetFunction->FunctionFlags &
FUNC_Protected) != 0;
        const bool bIsPrivate = (TargetFunction->FunctionFlags & FUNC_Private) != 0;
        const UClass* OwningClass = TargetFunction->GetOwnerClass();
        if( (bIsProtected || bIsPrivate) &&
!FBlueprintEditorUtils::IsNativeSignature(TargetFunction) && OwningClass)
        {
            OwningClass = OwningClass->GetAuthoritativeClass();
            // we can filter private and protected blueprints that are unrelated:
            bool bAccessibleInAll = true;
            for (const UBlueprint* Blueprint : Filter.Context.Blueprints)
            {
                UClass* AuthoritativeClass = Blueprint->GeneratedClass;
                if(!AuthoritativeClass)
                {
                    continue;
                }

                if(bIsPrivate)
                {
                    bAccessibleInAll = bAccessibleInAll && AuthoritativeClass ==
OwningClass;
                }
                else if(bIsProtected)
                {
                    bAccessibleInAll = bAccessibleInAll && AuthoritativeClass-
>IsChildOf(OwningClass);
                }
            }

            if(!bAccessibleInAll)
            {
                bIsFilteredOut = true;
            }
        }
    }

    return bIsFilteredOut;
}

```

## Effect on attributes:

When applied to an attribute, it indicates that the attribute can only be read and written within this class or its derived classes, but not accessible from other Blueprint classes.

Test code:

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_Access :public AActor
{

```

```

public:
    GENERATED_BODY()
public:
    // (BlueprintProtected = true, Category = MyFunction_Access,
ModuleRelativePath = Function/MyFunction_Access.h)
    // CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadWrite, meta = (BlueprintProtected = "true"))
    int32 MyNativeInt_HasProtected;

    // (BlueprintPrivate = true, Category = MyFunction_Access, ModuleRelativePath
= Function/MyFunction_Access.h)
    // CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadWrite, meta = (BlueprintPrivate = "true"))
    int32 MyNativeInt_HasPrivate;

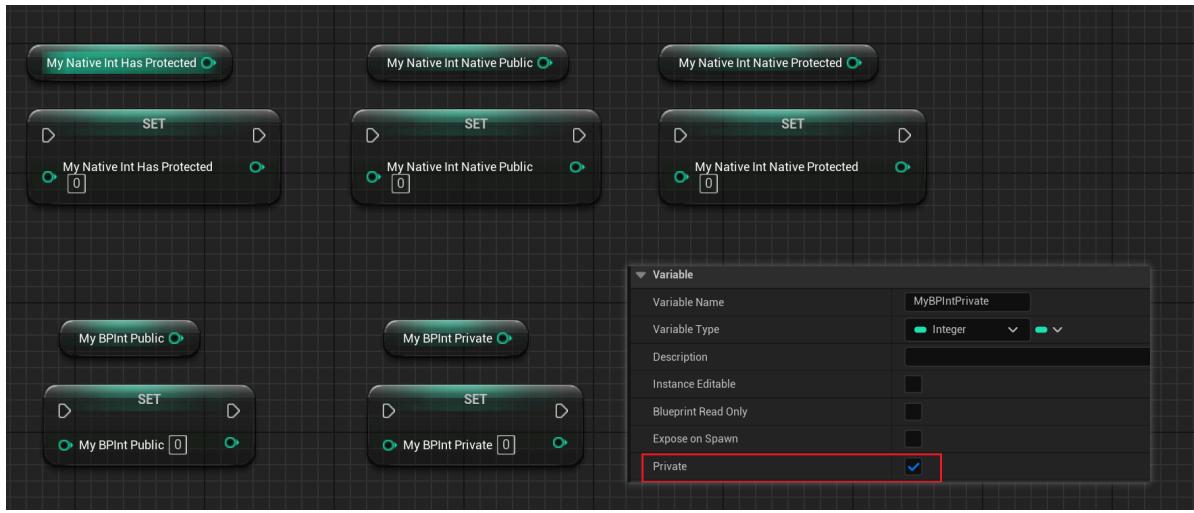
public:
// CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadWrite)
    int32 MyNativeInt_NativePublic;
protected:
    // CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
CPF_NoDestructor | CPF_Protected | CPF_HasGetValueTypeHash | 
CPF_NativeAccessSpecifierProtected
    UPROPERTY(BlueprintReadOnly)
    int32 MyNativeInt_NativeProtected;
private:
    // CPF_Edit | CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPrivate
    // error : BlueprintReadWrite should not be used on private members
    UPROPERTY(EditAnywhere)
    int32 MyNativeInt_NativePrivate;
};

```

Blueprint effect:

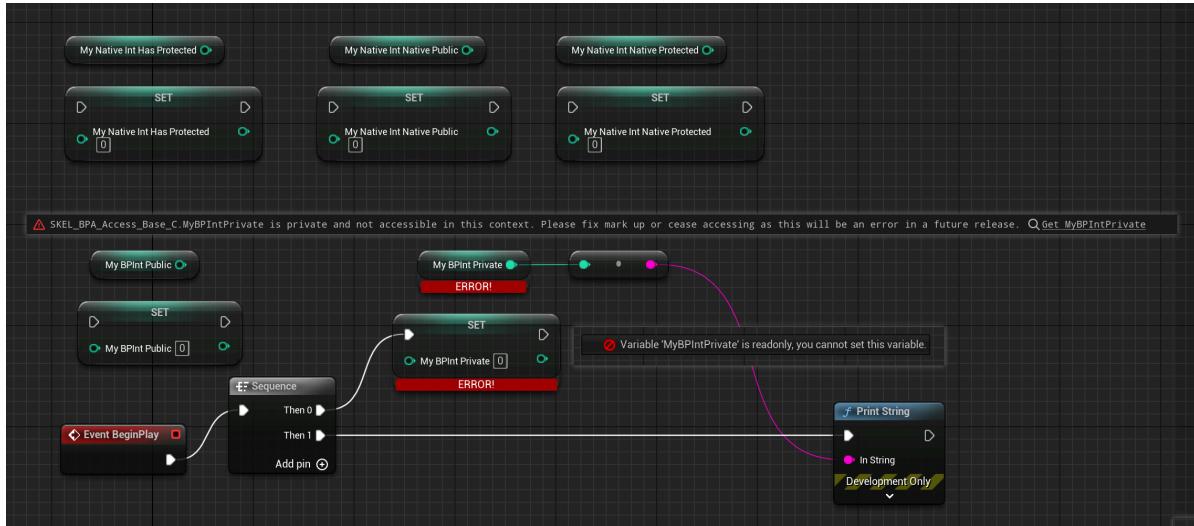
In testing with the subclass BPA\_Access\_Base, it was found that all attributes except MyNativeInt\_HasPrivate can be accessed. This is logical, as the meaning of Private is that it can only be accessed within the class itself.

Because MyBPIntPrivate, defined in this Blueprint class, is checked as Private, the property will add the meta BlueprintPrivate = true, but since it is defined within this class, it can still be read and written within this class.



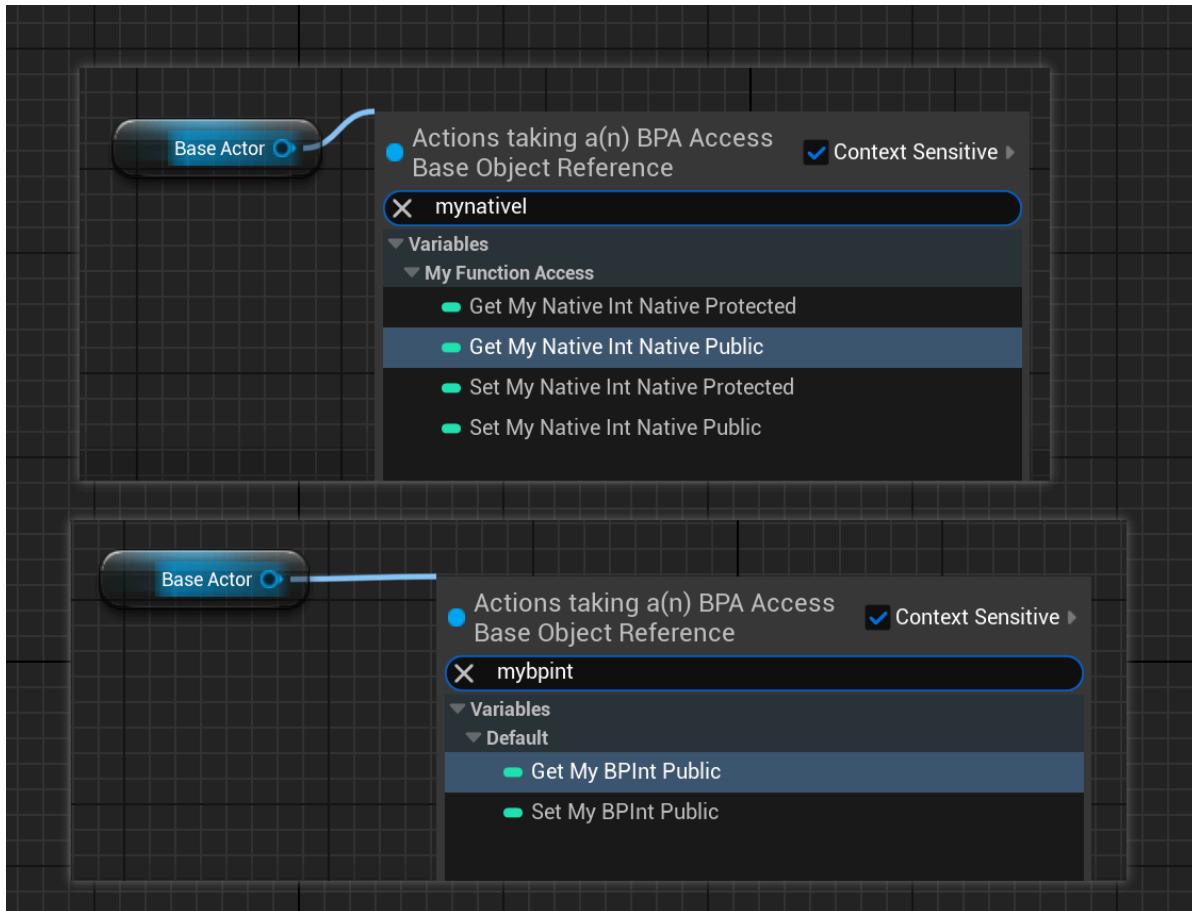
Continuing with the subclass effect in the Blueprint (BPA\_Access\_Child inheriting from BPA\_Access\_Base):

Protected attributes are still accessible, but the MyBPIntPrivate attribute cannot be read or written because it is Private. If the node is forcibly pasted, an error will be reported during compilation. Private means that it can only be accessed within the class itself.



In external classes (BPA\_Access\_Other, inheriting from Actor), when accessing properties through BPA\_Access\_Base or BPA\_Access\_Child object instances: neither BlueprintProtected nor BlueprintPrivate can be accessed. C++ protected attributes have no effect.

MyBPIntPrivate cannot be accessed because it is Private.



## Principle:

Searching the source code for CPF\_NativeAccessSpecifierProtected reveals no usage.

And CPF\_NativeAccessSpecifierPrivate is only referenced in IsPropertyPrivate, and the latter is only detected during thread safety checks in Blueprint compilation. Therefore, CPF\_NativeAccessSpecifierPrivate is not actually used as a scope restriction.

Combining the two, this is also the reason why protected and private in C++ do not affect Blueprints. However, UHT will prevent BlueprintReadWrite or BlueprintReadOnly on private variables, making them effectively inaccessible in Blueprints, achieving the effect that C++ base class private variables cannot be accessed in Blueprint subclasses.

Therefore, in fact, the variable scope control in the blueprint uses metadata BlueprintProtected and BlueprintPrivate . The logic of whether the attribute read-write node can be created by right-clicking on the blueprint is reflected in the BlueprintActionFilterImpl::IsFieldInaccessible function above. When compiling, the logic to determine whether an attribute is readable and writable lies in these two functions IsPropertyWritableInBlueprint and IsPropertyReadableInBlueprint . If the final status result is Private , it means it is inaccessible. The UK2Node\_VariableGet UK2Node\_VariableSet ValidateNodeDuringCompilation will be detected and an error will be reported.

```
bool FBlueprintEditorUtils::IsPropertyPrivate(const FProperty* Property)
{
    return Property->HasAnyPropertyParams(CPF_NativeAccessSpecifierPrivate) ||
    Property->GetBoolMetaData(FBlueprintMetadata::MD_Private);
}
```

```

FBlueprintEditorUtils::EPropertywritableState
FBlueprintEditorUtils::IsPropertywritableInBlueprint(const UBlueprint* Blueprint,
const FProperty* Property)
{
    if (Property)
    {
        if (!Property->HasAnyPropertyFlags(CPF_BlueprintVisible))
        {
            return EPropertywritableState::NotBlueprintvisible;
        }
        if (Property->HasAnyPropertyFlags(CPF_BlueprintReadOnly))
        {
            return EPropertywritableState::BlueprintReadOnly;
        }
        if (Property->GetBoolMetaData(FBlueprintMetadata::MD_Private))
        {
            const UClass* OwningClass = Property->GetOwnerChecked<UClass>();
            if (OwningClass->ClassGeneratedBy.Get() != Blueprint)
            {
                return EPropertywritableState::Private;
            }
        }
    }
    return EPropertywritableState::Writable;
}

FBlueprintEditorUtils::EPropertyReadableState
FBlueprintEditorUtils::IsPropertyReadableInBlueprint(const UBlueprint* Blueprint,
const FProperty* Property)
{
    if (Property)
    {
        if (!Property->HasAnyPropertyFlags(CPF_BlueprintVisible))
        {
            return EPropertyReadableState::NotBlueprintvisible;
        }
        if (Property->GetBoolMetaData(FBlueprintMetadata::MD_Private))
        {
            const UClass* OwningClass = Property->GetOwnerChecked<UClass>();
            if (OwningClass->ClassGeneratedBy.Get() != Blueprint)
            {
                return EPropertyReadableState::Private;
            }
        }
    }
    return EPropertyReadableState::Readable;
}

```

## AllowPrivateAccess

- **Function Description:** Allows a C++ private attribute to be accessible in Blueprints.
- **Usage Location:** UPROPERTY
- **Metadata Type:** bool

- **Associated Items:** BlueprintProtected

- **Commonality:** ★★★★☆

Allows a C++ private attribute to be accessible in Blueprints.

The purpose of AllowPrivateAccess is to permit the attribute to remain private in C++ and inaccessible to C++ subclasses, while still exposing it for access in Blueprints.

## Test Code:

```
public:
    //CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(BlueprintReadWrite)
    int32 MyNativeInt_NativePublic;

private:
    //CPF_ZeroConstructor | CPF_IsPlainOldData | CPF_NoDestructor | 
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPrivate
    //error : BlueprintReadWrite should not be used on private members
    UPROPERTY()
    int32 MyNativeInt_NativePrivate;

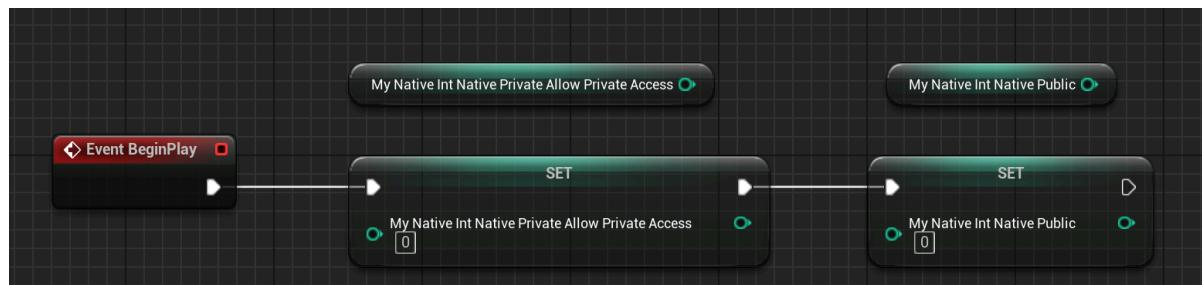
    //((AllowPrivateAccess = TRUE, Category = MyFunction_Access,
    ModuleRelativePath = Function/MyFunction_Access.h)
    //CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_IsPlainOldData | 
    CPF_NoDestructor | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPrivate
    UPROPERTY(BlueprintReadWrite, meta = (AllowPrivateAccess = true))
    int32 MyNativeInt_NativePrivate_AllowPrivateAccess;
```

Attempting to add BlueprintReadWrite or BlueprintReadOnly to MyNativeInt\_NativePrivate will trigger a UHT compilation error.

## Blueprint Effects:

By default, the access permissions for MyNativeInt\_NativePrivate\_AllowPrivateAccess in Blueprints are consistent with MyNativeInt\_NativePublic.

If users wish to modify the access permissions of the property in Blueprints, they can combine it with BlueprintProtected and BlueprintPrivate.



## Principle:

When UHT identifies the BlueprintReadWrite or BlueprintReadOnly specifiers for an attribute, it simultaneously checks for the presence of AllowPrivateAccess. If it is absent, an error is triggered.

Therefore, the role of AllowPrivateAccess is actually to prevent UHT from reporting an error. Once this error check is bypassed, the BlueprintReadWrite or BlueprintReadOnly on the attribute will be recognized and take effect, enabling access within Blueprints.

```
private static void BlueprintReadWriteSpecifier(UhtSpecifierContext specifierContext)
{
    bool allowPrivateAccess =
context.MetaData.TryGetValue(UhtNames.AllowPrivateAccess, out string? privateAccessMD) && !privateAccessMD.Equals("false",
StringComparison.OrdinalIgnoreCase);
    if (specifierContext.AccessSpecifier == UhtAccessSpecifier.Private &&
!allowPrivateAccess)
    {
        context.MessageSite.LogError("BlueprintReadWrite should not be used on private members");
    }
}

private static void BlueprintReadOnlySpecifier(UhtSpecifierContext specifierContext)
{
    bool allowPrivateAccess =
context.MetaData.TryGetValue(UhtNames.AllowPrivateAccess, out string? privateAccessMD) && !privateAccessMD.Equals("false",
StringComparison.OrdinalIgnoreCase);
    if (specifierContext.AccessSpecifier == UhtAccessSpecifier.Private &&
!allowPrivateAccess)
    {
        context.MessageSite.LogError("BlueprintReadOnly should not be used on private members");
    }
}
```

## BlueprintPrivate

- **Function Description:** Specifies that this function or property is accessible exclusively within the class it is defined, akin to the private access specifier in C++. It cannot be accessed from other Blueprint classes.
- **Usage Locations:** UFUNCTION, UPROPERTY
- **Metadata Type:** bool
- **Related Item:** BlueprintProtected
- **Commonality:** ★★

Access permissions for the function can be configured in the function details panel:

Description	<input type="text"/>
Category	Default <input type="button" value="▼"/>
Keywords	<input type="text"/>
Compact Node Title	<input type="text"/>
Pure	<input type="checkbox"/>
Call In Editor	<input type="checkbox"/>
Access Specifier	Private <input type="button" value="▼"/>

As a result, "BlueprintPrivate=true" is added to the function

Properties can also be configured in the details panel:

▼ Variable	
Variable Name	<input type="text"/> MyBPIntPrivate
Variable Type	<input type="button" value="Integer"/> <input type="button" value="▼"/> <input type="button" value="▼"/>
Description	<input type="text"/>
Instance Editable	<input type="checkbox"/>
Blueprint Read Only	<input type="checkbox"/>
Expose on Spawn	<input type="checkbox"/>
Private	<input checked="" type="checkbox"/>

As a result, "BlueprintPrivate=true" is added to the property

## CommutativeAssociativeBinaryOperator

- **Function Description:** Indicates that a binary operation function adheres to the commutative and associative laws, adding a "+" pin to the blueprint node that enables dynamic addition of multiple input values.
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Commonality:** ★★★★

Marking a binary operation function with this indicates support for the commutative and associative laws. A "+" pin is added to the blueprint node, allowing for the dynamic direct addition of multiple input values without the need to manually create several function nodes for computation. This is one of the convenient features provided by blueprints.

The restrictions of CommutativeAssociativeBinaryOperator are that the function must be BlueprintPure and have two parameters; otherwise, compilation errors or functional failures may occur.

## Test Code:

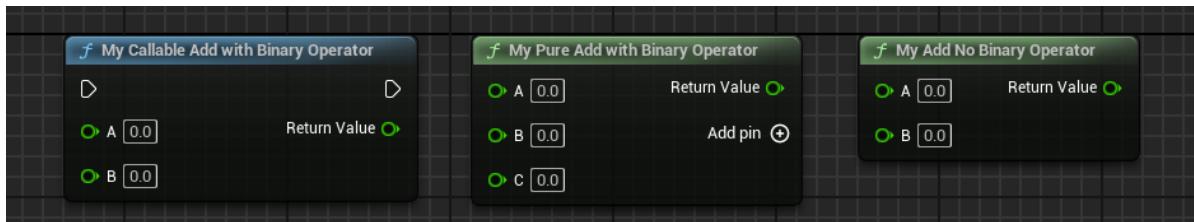
```
UFUNCTION(BlueprintCallable, meta = (CommutativeAssociativeBinaryOperator))
static float My_CallableAdd_WithBinaryOperator(float A, float B) { return A +
B; }

UFUNCTION(BlueprintPure, meta = (CommutativeAssociativeBinaryOperator))
static float My_PureAdd_WithBinaryOperator(float A, float B) { return A + B;
}

UFUNCTION(BlueprintPure, meta = ())
static float My_Add_NoBinaryOperator(float A, float B) { return A + B; }

// error : Commutative associative binary operators must have exactly 2
parameters of the same type and a return value.
//UFUNCTION(BlueprintPure, meta = (CommutativeAssociativeBinaryOperator))
// static float My_PureAdd3_WithBinaryOperator(float A, float B, float C) {
// return A + B+C; }
```

## Blueprint Effect:



## Principle:

Functions marked with CommutativeAssociativeBinaryOperator are generated using UK2Node\_CollectiveAssociativeBinaryOperator nodes. This binary operation satisfies the commutative and associative laws, allowing for the computation of multiple input values through multiple invocations of the function. When UK2Node\_CollectiveAssociativeBinaryOperator is expanded, several intermediate UK2Node\_CollectiveAssociativeBinaryOperator nodes are created to form a calling sequence.

Applications in the source code include various binary operations, with extensive use in UKismetMathLibrary, such as operations involving FVector.

```
void
UK2Node_CollectiveAssociativeBinaryOperator::ExpandNode(FKismetCompilerContext&
CompilerContext, UEdGraph* SourceGraph)
{
    Super::ExpandNode(CompilerContext, SourceGraph);

    if (NumAdditionalInputs > 0)
    {
        const UEdGraphSchema_K2* Schema = CompilerContext.GetSchema();
```

```

UEdGraphPin* LastOutPin = NULL;
const UFunction* const Function = GetTargetFunction();

const UEdGraphPin* SrcOutPin = FindOutPin();
const UEdGraphPin* SrcSelfPin = FindSelfPin();
UEdGraphPin* SrcFirstInput = GetInputPin(0);
check(SrcFirstInput);

for(int32 PinIndex = 0; PinIndex < Pins.Num(); PinIndex++)
{
    UEdGraphPin* CurrentPin = Pins[PinIndex];
    if( (CurrentPin == SrcFirstInput) || (CurrentPin == SrcOutPin) ||
    (SrcSelfPin == CurrentPin) )
    {
        continue;
    }

    UK2Node_CollectiveAssociativeBinaryOperator* NewOperator =
SourceGraph->CreateIntermediateNode<UK2Node_CollectiveAssociativeBinaryOperator>()
;
    NewOperator->SetFromFunction(Function);
    NewOperator->AllocateDefaultPins();

CompilerContext.MessageLog.NotifyIntermediateObjectCreation(NewOperator, this);

    UEdGraphPin* NewOperatorInputA = NewOperator->GetInputPin(0);
    check(NewOperatorInputA);
    if(LastOutPin)
    {
        Schema->TryCreateConnection(LastOutPin, NewOperatorInputA);
    }
    else
    {
        // handle first created node (SrcFirstInput is skipped, and has
no own node).
        CompilerContext.MovePinLinksToIntermediate(*SrcFirstInput,
*NewOperatorInputA);
    }

    UEdGraphPin* NewOperatorInputB = NewOperator->GetInputPin(1);
    check(NewOperatorInputB);
    CompilerContext.MovePinLinksToIntermediate(*CurrentPin,
*NewOperatorInputB);

    LastOutPin = NewOperator->FindOutPin();
}

check(LastOutPin);

UEdGraphPin* TrueOutPin = FindOutPin();
check(TrueOutPin);
CompilerContext.MovePinLinksToIntermediate(*TrueOutPin, *LastOutPin);

BreakAllNodeLinks();
}
}

```

# CompactNodeTitle

- **Function Description:** Enables the function to be displayed in a compact mode, while assigning a new abbreviated name
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Commonality:** ★★★

Enables the function to be displayed in a compact mode, and assigns a new abbreviated name. Note that in this mode, the DisplayName data is overlooked.

## Testing Code:

```
UFUNCTION(BlueprintCallable, meta = (CompactNodeTitle =
"MyCompact", DisplayName="AnotherName"))
static int32 MyFunc_HasCompactNodeTitle(FString Name) {return 0;}

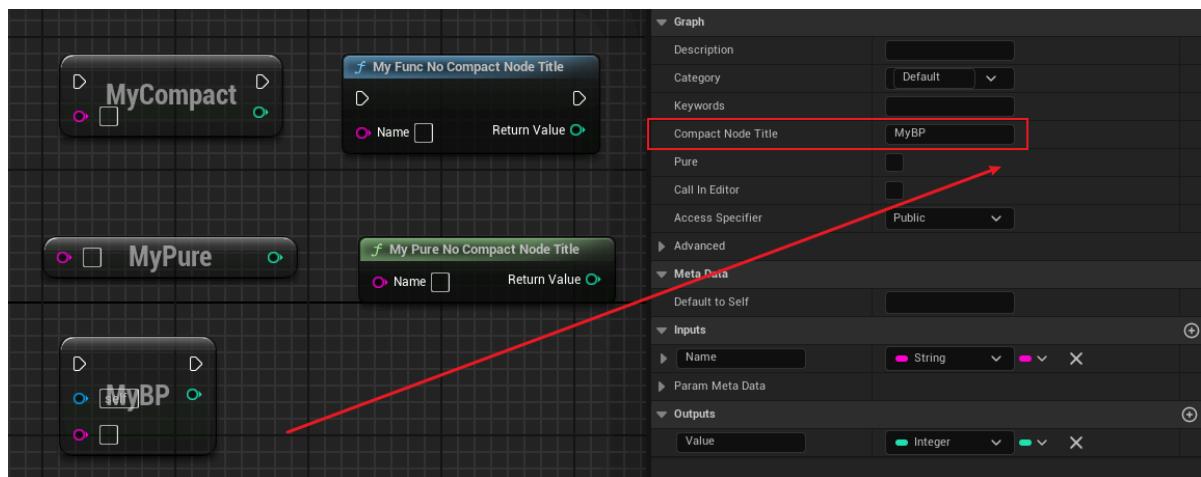
UFUNCTION(BlueprintCallable, meta = ())
static int32 MyFunc_NoCompactNodeTitle(FString Name) {return 0;}

UFUNCTION(BlueprintPure, meta = (CompactNodeTitle =
"MyPure", DisplayName="AnotherName"))
static int32 MyPure_HasCompactNodeTitle(FString Name) {return 0;}

UFUNCTION(BlueprintPure, meta = ())
static int32 MyPure_NoCompactNodeTitle(FString Name) {return 0;}
```

## Blueprint Effect:

The display effect has significantly changed. Moreover, functions defined within the blueprint can also be transformed into a compact display mode through settings on the detail panel.



## Principle:

```
bool UK2Node_CallFunction::ShouldDrawCompact(const UFunction* Function)
{
    return (Function != NULL) && Function->HasMetaData(FBlueprintMetadata::MD_CompactNodeTitle);
}

FString UK2Node_CallFunction::GetCompactNodeTitle(const UFunction* Function)
{
    static const FString ProgrammerMultiplicationSymbol = TEXT("*");
    static const FString CommonMultiplicationSymbol = TEXT("\xD7");

    static const FString ProgrammerDivisionSymbol = TEXT("/");
    static const FString CommonDivisionSymbol = TEXT("\xF7");

    static const FString ProgrammerConversionSymbol = TEXT("->");
    static const FString CommonConversionSymbol = TEXT("\x2022");

    const FString& OperatorTitle = Function->GetMetaData(FBlueprintMetadata::MD_CompactNodeTitle);
    if (!OperatorTitle.IsEmpty())
    {
        if (OperatorTitle == ProgrammerMultiplicationSymbol)
        {
            return CommonMultiplicationSymbol;
        }
        else if (OperatorTitle == ProgrammerDivisionSymbol)
        {
            return CommonDivisionSymbol;
        }
        else if (OperatorTitle == ProgrammerConversionSymbol)
        {
            return CommonConversionSymbol;
        }
        else
        {
            return OperatorTitle;
        }
    }

    return Function->GetName();
}
```

## CustomStructureParam

- **Function Description:** Parameters marked with CustomStructureParam will be transformed into wildcard parameters, with their pin types matching the type of the connected variables.
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** strings="a, b, c"
- **Commonly Used:** ★★★★☆

Multiple function parameters marked with CustomStructureParam will be converted into wildcard parameters, with their pin types corresponding to the type of the connected variables.

CustomStructureParam is always used in conjunction with CustomThunk to enable the handling of generic parameter types within the function body.

```
UFUNCTION(BlueprintCallable, CustomThunk, meta = (DisplayName =
"PrintStructFields", CustomStructureParam = "inputStruct"))
static FString PrintStructFields(const int32& inputStruct) { return TEXT(""); }

DECLARE_FUNCTION(execPrintStructFields);
static FString Generic_PrintStructFields(const UScriptStruct* ScriptStruct, const
void* StructData);

DEFINE_FUNCTION(UMyFunction_Custom::execPrintStructFields)
{
    FString result;
    Stack.MostRecentPropertyAddress = nullptr;
    Stack.StepCompiledIn<FStructProperty>(nullptr);

    void* StructData = Stack.MostRecentPropertyAddress;
    FStructProperty* StructProperty = CastField<FStructProperty>
(Stack.MostRecentProperty);
    UScriptStruct* ScriptStruct = StructProperty->Struct;
    P_FINISH;
    P_NATIVE_BEGIN;

    result = Generic_PrintStructFields(ScriptStruct, StructData);

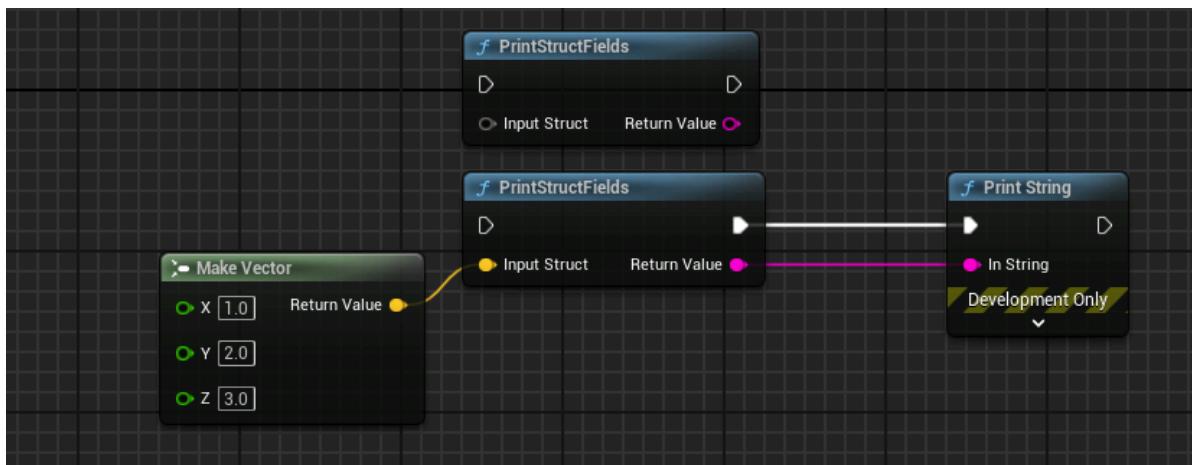
    P_NATIVE_END;
    *(FString*)RESULT_PARAM = result;
}

FString UMyFunction_Custom::Generic_PrintStructFields(const UScriptStruct*
ScriptStruct, const void* StructData)
{
    FString str;
    for (TFieldIterator<FProperty> i(ScriptStruct); i; ++i)
    {
        FString PropertyValueString;
        const void* PropertyValuePtr = i->ContainerPtrToValuePtr<const void*>
(StructData);
        i->ExportTextItem_Direct(PropertyValueString, PropertyValuePtr, nullptr,
(UObject*)ScriptStruct, PPF_None);

        str += FString::Printf(TEXT("%s:%s\n"), *i->GetFName().ToString(),
*PropertyValueString);
    }

    return str;
}
```

## Effect in Blueprint:



You can observe a node that accepts generic structure parameters, which then prints out all its internal attributes. CustomStructureParam designates that the function parameters are of custom types.

An illustrative example in the source code can be found here:

```
UFUNCTION(BlueprintCallable, CustomThunk, Category = "DataTable", meta=CustomStructureParam = "OutRow", BlueprintInternalUseOnly="true"))
static ENGINE_API bool GetDataTableRowFromName(UDataTable* Table, FName RowName,
FTableRowBase& OutRow);
```

## Principle:

First, parameters with CustomStructureParam are recognized as wildcard attributes. Then, FCustomStructureParamHelper is used to control Pin->PinType = LinkedTo->PinType; thereby changing the actual type of the pin.

```
bool UEdGraphSchema_K2::IsWildcardProperty(const FPropertyParams*PropertyParams)
{
    UFunction* Function =PropertyParams->GetOwner<UFunction>();

    return Function && ( UK2Node_CallArrayFunction::IsWildcardProperty(Function,PropertyParams)
        || UK2Node_CallFunction::IsStructureWildcardProperty(Function,PropertyParams->GetFName())
        || UK2Node_CallFunction::IsWildcardProperty(Function,PropertyParams)
        || FEdGraphUtilities::IsArrayDependentParam(Function,PropertyParams->GetFName()) );
}

static void FCustomStructureParamHelper::HandleSinglePin(UEdGraphPin* Pin)
{
    if (Pin)
    {
        if (Pin->LinkedTo.Num() > 0)
        {
            UEdGraphPin* LinkedTo = Pin->LinkedTo[0];
            check(LinkedTo);
        }
    }
}
```

```

    if (UK2Node* Node = Cast<UK2Node>(Pin->GetOwningNode()))
    {
        ensure(
            !LinkedTo->PinType.IsContainer() ||
            Node->DoesWildcardPinAcceptContainer(Pin)
        );
    }
    else
    {
        ensure( !LinkedTo->PinType.IsContainer() );
    }

    Pin->PinType = LinkedTo->PinType;
}
else
{
    // constness and refness are controlled by our declaration
    // but everything else needs to be reset to default wildcard:
    const bool bWasRef = Pin->PinType.bIsReference;
    const bool bWasConst = Pin->PinType.bIsConst;

    Pin->PinType = FEdGraphPinType();
    Pin->PinType.bIsReference = bWasRef;
    Pin->PinType.bIsConst = bWasConst;
    Pin->PinType.PinCategory = UEdGraphSchema_K2::PC_Wildcard;
    Pin->PinType.PinSubCategory = NAME_None;
    Pin->PinType.PinSubCategoryObject = nullptr;
}
}
}

```

## DefaultToSelf

- **Function Description:** Used on functions to specify a default value of Self for a parameter
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Commonly Used:** ★★★★☆

Enables more convenient calls within blueprints, depending on the function's requirements.

## Test Code:

```

UCLASS()
class INSIDER_API UMyFunctionLibrary_SelfPinTest :public
UBlueprintFunctionLibrary
{
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintCallable)
    static FString PrintProperty_Default(UObject* myOwner, FName propertyName);

```

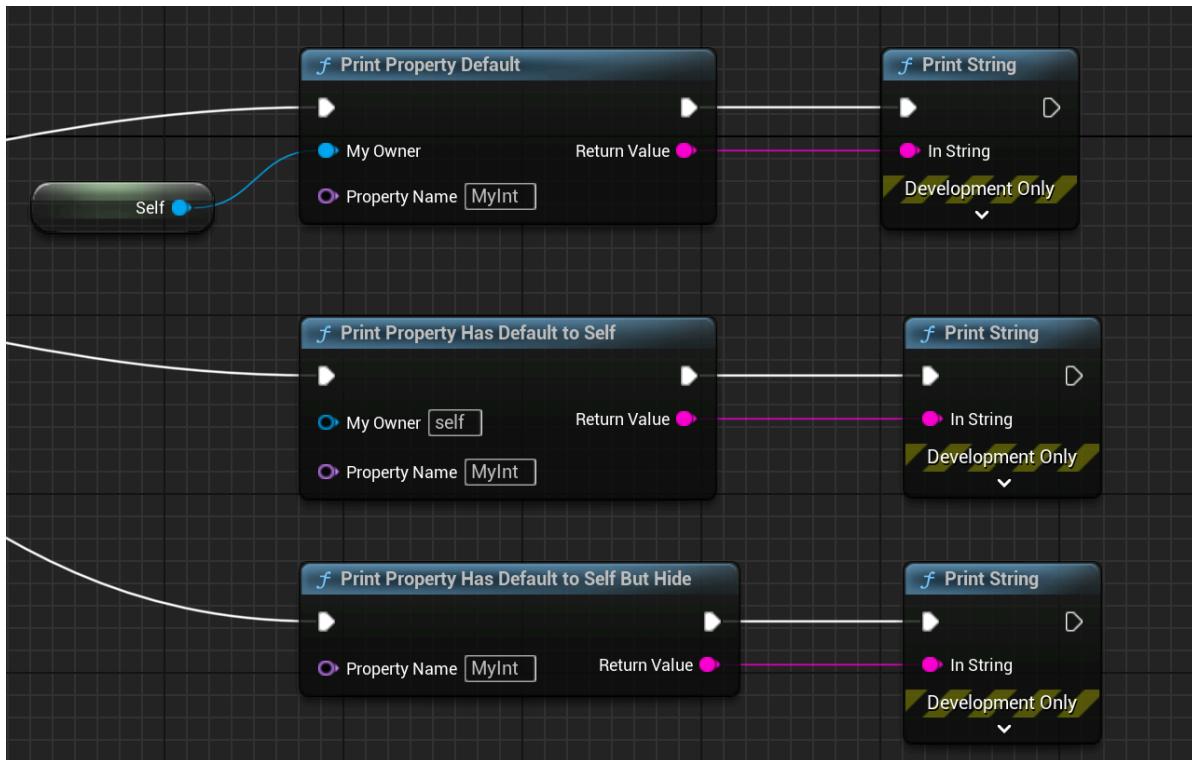
```

UFUNCTION(BlueprintCallable,meta=(DefaultToSelf="myOwner"))
static FString PrintProperty_HasDefaultToSelf(Uobject* myOwner,FName
propertyName);

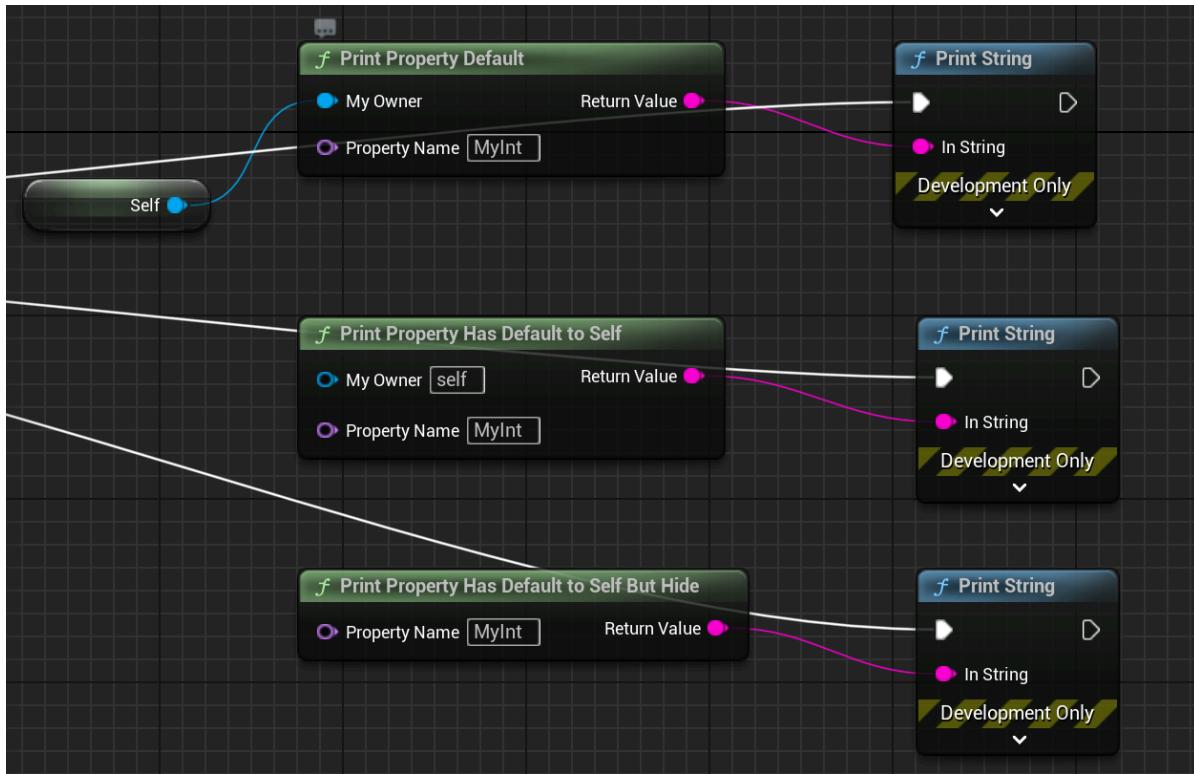
UFUNCTION(BlueprintCallable,meta=(DefaultToSelf="myOwner",hidePin="myOwner"))
static FString PrintProperty_HasDefaultToSelf_ButHide(Uobject* myOwner,FName
propertyName);
};

```

In blueprint nodes, it can be observed that the blueprint compiler automatically assigns the function parameter designated by DefaultToSelf to Self. This is fundamentally the same as manually linking to Self. Additionally, using HidePin can hide this function parameter, making the blueprint object (Self) where the node is located the default first function parameter, thus appearing more streamlined.



Also applicable if it is a BlueprintPure:



## Principle:

When the function call in the blueprint is compiled, SelfPin (named Target) will be automatically created. If the function is a static function or the mark of HideSelfPin, SelfPin will be hidden. The value of its SelfPin is the value of the current blueprint runtime object. Therefore, the effect of DefaultToSelf is that the blueprint system will automatically assign the value of this parameter to the blueprint runtime object where it is called, which is equivalent to the effect of the C++ this pointer

```

bool UK2Node_CallFunction::CreatePinsForFunctionCall(const UFunction* Function)
{
    UEdGraphPin* SelfPin = CreateSelfPin(Function);
    // Renamed self pin to target
    SelfPin->PinFriendlyName = LOCTEXT("Target", "Target");
}

UEdGraphPin* FBlueprintNodeStatics::CreateSelfPin(UK2Node* Node, const UFunction* Function)
{
    // Chase up the function's Super chain, the function can be called on any
    // object that is at least that specific
    const UFunction* FirstDeclaredFunction = Function;
    while (FirstDeclaredFunction->GetSuperFunction() != nullptr)
    {
        FirstDeclaredFunction = FirstDeclaredFunction->GetSuperFunction();
    }

    // Create the self pin
    UClass* FunctionClass = CastChecked<UClass>(FirstDeclaredFunction-
>GetOuter());
    // we don't want blueprint-function target pins to be formed from the
    // skeleton class (otherwise, they could be incompatible with other pins
    // that represent the same type)... this here could lead to a compiler
}

```

```

// warning (the GeneratedClass could not have the function yet), but in
// that, the user would be reminded to compile the other blueprint
if (FunctionClass->ClassGeneratedBy)
{
    FunctionClass = FunctionClass->GetAuthoritativeClass();
}

UEdGraphPin* SelfPin = NULL;
if (FunctionClass == Node->GetBlueprint()->GeneratedClass)
{
    // This means the function is defined within the blueprint, so the pin
    // should be a true "self" pin
    SelfPin = Node->CreatePin(EGPD_Input, UEdGraphSchema_K2::PC_Object,
        UEdGraphSchema_K2::PSC_Self, nullptr, UEdGraphSchema_K2::PN_Self);
}
else if (FunctionClass->IsChildof(UInterface::StaticClass()))
{
    SelfPin = Node->CreatePin(EGPD_Input, UEdGraphSchema_K2::PC_Interface,
        FunctionClass, UEdGraphSchema_K2::PN_Self);
}
else
{
    // This means that the function is declared in an external class, and
    // should reference that class
    SelfPin = Node->CreatePin(EGPD_Input, UEdGraphSchema_K2::PC_Object,
        FunctionClass, UEdGraphSchema_K2::PN_Self);
}
check(SelfPin != nullptr);

return SelfPin;
}

```

## ExpandEnumAsExecs

- **Function description:** Specifies multiple enum or bool type function parameters, automatically generating corresponding multiple input or output execution pins based on the entries, and altering the control flow according to different actual parameter values.
- **Usage location:** UFUNCTION
- **Engine module:** Blueprint
- **Metadata type:** strings = "a, b, c"
- **Related items:** ExpandBoolAsExecs
- **Commonly used:** ★★★★☆

Specifies multiple enum or bool type function parameters, automatically generating corresponding multiple input or output execution pins based on the entries, and altering the control flow according to different actual parameter values.

Supports changing input and output Execs; only one input Exec is allowed, but multiple output ExecEnum pins can be generated. However, it cannot be used on BlueprintPure (as there are no Exec pins).

You can also use single quotes ‘|’ to separate values.

Three parameter types are supported: enum class, TEnumAsByte`<EMyExecPins2::Type>`, and bool. Enums must be marked with UENUM.

Reference type parameters and return values are used as output pins, while value type parameters are used as input pins.

The name "ReturnValue" can be used to specify the utilization of return value parameters.

If there are multiple output Enum parameters, they will be arranged in a sequence after the function call to trigger the output Execs one by one based on the values of the output Enums.

## Test Code:

```
UENUM(BlueprintType)
enum class EMyExecPins : uint8
{
    First,
    Second,
    Third,
};

UENUM(BlueprintType)
namespace EMyExecPins2
{
    enum Type : int
    {
        Found,
        NotFound,
    };
}

UENUM(BlueprintType)
enum class EMyExecAnimalPins : uint8
{
    Cat,
    Dog,
};

public:
    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "Pins"))
    static int32 MyEnumAsExec_Output(FString Name, EMyExecPins& Pins) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "Pins"))
    static int32 MyEnumAsExec_Input(FString Name, TEnumAsByte<EMyExecPins2::Type> Pins) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "ReturnValue"))
    static EMyExecPins MyEnumAsExec_Return(FString Name) { return
        EMyExecPins::First; }

public:
    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "Pins"))
    static int32 MyBoolAsExec_Output(FString Name, bool& Pins) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "Pins"))
    static int32 MyBoolAsExec_Input(FString Name, bool Pins) { return 0; }
```

```

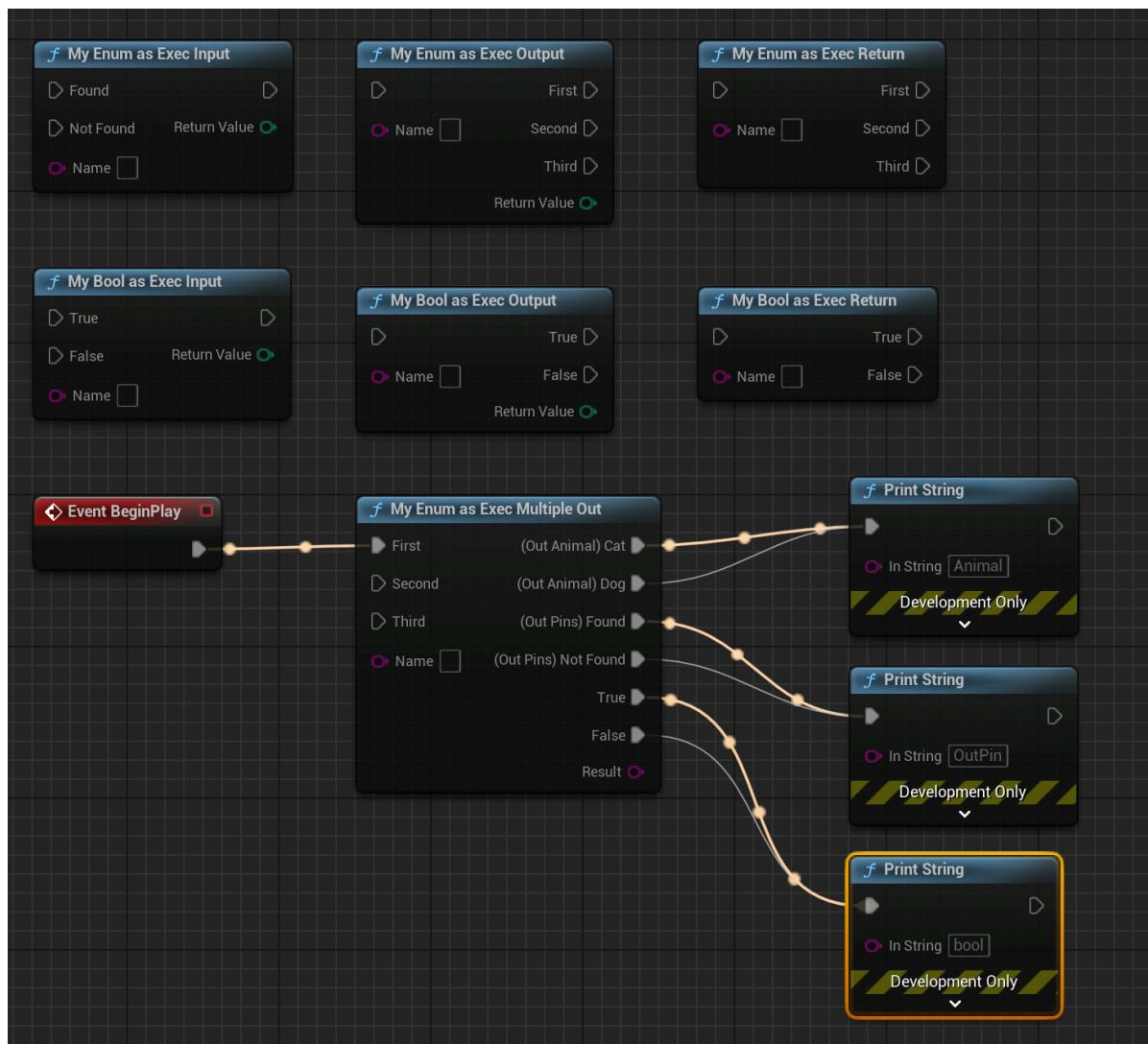
UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs = "ReturnValue"))
static bool MyBoolAsExec_Return(FString Name) { return false; }

public:
UFUNCTION(BlueprintCallable, meta = (ExpandEnumAsExecs =
"InPins|OutAnimal|OutPins|ReturnValue"))
static bool MyEnumAsExec_MultipleOut(FString Name, EMyExecPins InPins,
EMyExecAnimal Pins& OutAnimal, TEnumAsByte<EMyExecPins2::Type>& OutPins, FString&
Result);

```

## Blueprint Effect:

By comparing the function prototypes and blueprint nodes mentioned above, you can find that ExpandEnumAsExecs executes 3 parameter types. It also verified that when there are multiple output Enum parameters at the same time (OutAnimal in the code | OutPins | ReturnValue), the output will be executed three times in sequence, as if connected together by Sequence nodes.



## Principle:

The real creation Pin is in void UK2Node\_CallFunction::CreateExecPinsForFunctionCall ( const UFunction\* Function ), and then the logic of these new ExecPin and matching assignment of input parameter values, and execution of different outputs based on output parameters ExecPin is in UK2Node\_CallFunction::ExpandNode . There is too much code so I won't post it.

The original function parameter pins are hidden, thus only the generated Exec pins are exposed.

The original parameter Pin of the function will be hidden, thus only the generated Exec Pin will be exposed.

## ExpandBoolAsExecs

---

- **Function description:** An alias for ExpandEnumAsExecs, offering identical functionality.
- **Usage location:** UFUNCTION
- **Metadata type:** string="abc"
- **Related items:** ExpandEnumAsExecs
- **Frequency:** ★★★★

## ArrayParm

---

- **Function description:** Specifies a function to use an Array<\*> with the array element type being a wildcard generic.
- **Use location:** UFUNCTION
- **Engine module:** Blueprint
- **Metadata type:** strings="a, b, c"
- **Associated items:** ArrayTypeDependentParams
- **Commonly used:** ★★★

Specifies a function to use an Array<\*> with the array element type being a wildcard generic.

The internal logic processing difference is that functions with ArrayParm will use UK2Node\_CallArrayFunction to generate nodes instead of UK2Node\_CallFunction.

Multiple ArrayParams can be specified, separated by commas.

In the source code, it is only used in UKismetArrayLibrary, but if you wish to perform array operations, you can also add ArrayParam.

Because the array element type is a wildcard generic, when implementing in C++, it is necessary to use CustomThunk and write some custom blueprint logic glue code to handle different array types correctly. This can be emulated by referring to the example in the UKismetArrayLibrary source code.

## Test code:

---

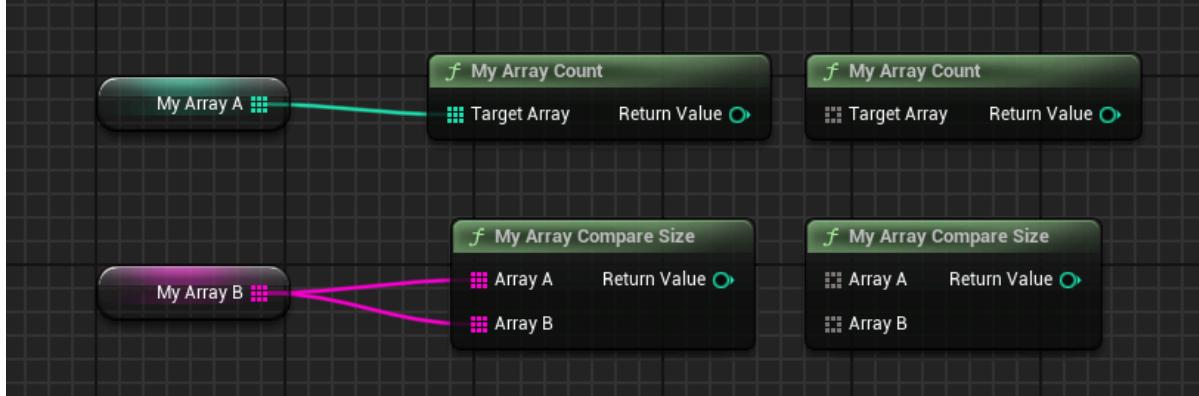
```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Param :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
//Array
    UFUNCTION(BlueprintPure, CustomThunk, meta = (ArrayParm = "TargetArray"))
    static int32 MyArray_Count(const TArray<int32>& TargetArray);
    static int32 GenericMyArray_Count(const void* TargetArray, const
FArrayProperty* ArrayProp);
    DECLARE_FUNCTION(execMyArray_Count);
```

```

UFUNCTION(BlueprintPure, CustomThunk, meta = (ArrayParm = "ArrayA,ArrayB",
ArrayTypeDependentParams = "ArrayB"))
    static int32 MyArray_CompareSize(const TArray<int32>& ArrayA, const
TArray<int32>& ArrayB);
    static int32 GenericMyArray_CompareSize(void* ArrayA, const FArrayProperty* ArrayAProp, void* ArrayB, const FArrayProperty* ArrayBProp);
DECLARE_FUNCTION(execMyArray_CompareSize);
};

```

## Blueprint effect:



As seen, when no specific array type is connected, the Array is displayed as a gray wildcard type. When different array types are connected, the Array parameter pins will automatically change to the corresponding types. This logic is implemented within UK2Node\_CallArrayFunction. Those interested can refer to it by reviewing the code.

## ArrayTypeDependentParams

- **Function description:** Specifies which array parameter types should be updated when a function, designated by ArryParam, has two or more array parameters.
- **Use location:** UFUNCTION
- **Metadata type:** string="abc"
- **Related items:** ArrayParm

When a function specified by ArryParam has two or more array parameters, the types of the specified array parameters should also be updated accordingly.

Indicates the type of a parameter, used to determine the value type of ArrayParam

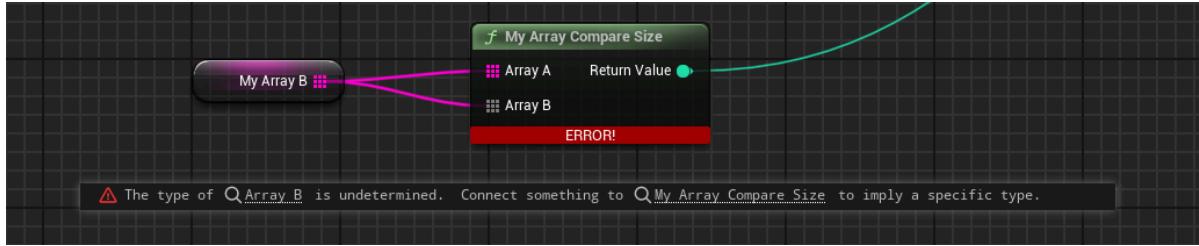
```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Param :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
//Array

    UFUNCTION(BlueprintPure, CustomThunk, meta = (ArrayParm = "ArrayA,ArrayB",
ArrayTypeDependentParams = "ArrayB"))
        static int32 MyArray_CompareSize(const TArray<int32>& ArrayA, const
TArray<int32>& ArrayB);
        static int32 GenericMyArray_CompareSize(void* ArrayA, const FArrayProperty*
ArrayAProp, void* ArrayB, const FArrayProperty* ArrayBProp);
    DECLARE_FUNCTION(execMyArray_CompareSize);
};

```

If there are no ArrayTypeDependentParams, the type of ArrayB remains undetermined after connecting ArrayA, even if it is connected. This is a limitation of the engine's implementation, and compilation will result in errors.



Therefore, ArrayTypeDependentParams can specify an additional array parameter, whose type will be determined by the type of the other (first) array actual parameter, i.e., `typeof(ArrayB)=typeof(ArrayA)`. As demonstrated in the sample code, by adding ArrayB as ArrayTypeDependentParams, MyArrayB can trigger a change to a consistent array type, whether it is connected to ArrayA or ArrayB first. This is because ArrayA, as the first parameter, has already implemented dynamic type changes in real-time within the engine. Thus, adding ArrayB is sufficient.

## Principle:

The engine has already implemented real-time dynamic type changes for the first parameter:

```

void UK2Node_CallArrayFunction::AllocateDefaultPins()
{
    Super::AllocateDefaultPins();

    UEdGraphPin* TargetArrayPin = GetTargetArrayPin();
    if (ensureMsgf(TargetArrayPin, TEXT("%s"), *GetFullName()))
    {
        TargetArrayPin->PinType.ContainerType = EPinContainerType::Array;
        TargetArrayPin->PinType.bIsReference = true;
        TargetArrayPin->PinType.PinCategory = UEdGraphSchema_K2::PC_Wildcard;
        TargetArrayPin->PinType.PinSubCategory = NAME_None;
        TargetArrayPin->PinType.PinSubCategoryObject = nullptr;
    }
}

```

```

TArray< FArrayPropertyPinCombo > ArrayPins;
GetArrayPins(ArrayPins);
for(auto Iter = ArrayPins.CreateConstIterator(); Iter; ++Iter)
{
    if(Iter->ArrayPropPin)
    {
        Iter->ArrayPropPin->bHidden = true;
        Iter->ArrayPropPin->bNotConnectable = true;
        Iter->ArrayPropPin->bDefaultValueIsReadOnly = true;
    }
}

PropagateArrayTypeinfo(TargetArrayPin);
}

```

For the mechanism of ArrayDependentParam, you can refer to the implementation of the functions NotifyPinConnectionListChanged and PropagateArrayTypeinfo in UK2Node\_CallArrayFunction. It can be observed that the types of other array parameter pins are dynamically changed to match the type of the SourcePin.

## AdvancedDisplay

- **Function Description:** Hides some function parameters from view by collapsing them; users must manually click the dropdown arrow to expand and edit them.
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** strings = "a, b, c"
- **Commonly Used:** ★★★★☆

Parameters of the function are hidden by collapsing them; users must manually click the dropdown arrow to expand and edit them.

AdvancedDisplay supports two formats: one uses "Parameter1, Parameter2, .." to explicitly specify the names of parameters to be collapsed, which is suitable for parameters that are not consecutive or located in the middle of the function parameter list. The other format uses "N" to specify a numerical sequence number, with all parameters following the Nth parameter displayed as advanced pins.

## Test Code:

```

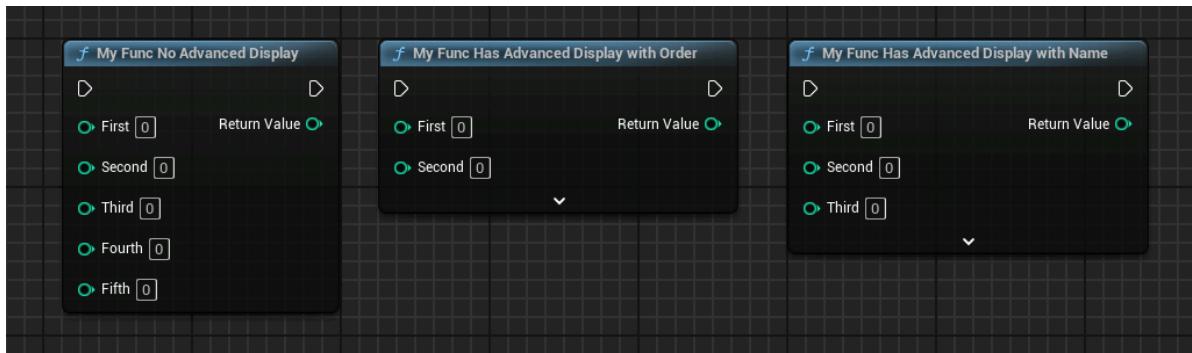
UFUNCTION(BlueprintCallable, meta = (AdvancedDisplay = "2"))
static int32 MyFunc_HasAdvancedDisplay_WithOrder(int32 First, int32 Second,
int32 Third, int32 Fourth, int32 Fifth) { return 0; }

UFUNCTION(BlueprintCallable, meta = (AdvancedDisplay = "Fourth,Fifth"))
static int32 MyFunc_HasAdvancedDisplay_WithName(int32 First, int32 Second,
int32 Third, int32 Fourth, int32 Fifth) { return 0; }

UFUNCTION(BlueprintCallable, meta = ())
static int32 MyFunc_NoAdvancedDisplay(int32 First, int32 Second, int32 Third,
int32 Fourth, int32 Fifth) { return 0; }

```

# Blueprint Effect:



In the source code, a typical example is `PrintString`, where all parameters after the second parameter are collapsed.

```
UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject",
CallableWithoutWorldContext, Keywords = "log print", AdvancedDisplay = "2",
DevelopmentOnly), Category="Development")
static ENGINE_API void PrintString(const UObject* WorldContextObject, const
FString& InString = FString(TEXT("Hello")), bool bPrintToScreen = true, bool
bPrintToLog = true, FLinearColor TextColor = FLinearColor(0.0f, 0.66f, 1.0f),
float Duration = 2.f, const FName Key = NAME_None);
```

## Principle:

AdvancedDisplay adds the `EPropertyFlags.AdvancedDisplay` mark to the function parameters it annotates, causing them to be collapsed. This logic is set when UHT parses the function.

```
//Supports both parameter name and numerical sequence number modes
if (_metaData.TryGetValue(UhtNames.AdvancedDisplay, out string? foundString))
{
    _parameterNames = foundString.ToString().Split(',', StringSplitOptions.RemoveEmptyEntries);
    for (int index = 0, endIndex = _parameterNames.Length; index < endIndex;
++index)
    {
        _parameterNames[index] = _parameterNames[index].Trim();
    }
    if (_parameterNames.Length == 1)
    {
        _bUseNumber = Int32.TryParse(_parameterNames[0], out
_numberLeaveUnmarked);
    }
}

//Set EPropertyFlags.AdvancedDisplay
private static void UhtFunctionParser::ParseParameterList(UhtParsingScope
topScope, UhtPropertyParseOptions options)
{
    UhtAdvancedDisplayParameterHandler advancedDisplay =
new(topScope.ScopeType.MetaData);

    topScope.TokenReader.RequireList(')', ',', false, () =>
{
```

```

topScope.HeaderParser.GetCachedPropertyParser().Parse(topScope, disallowFlags,
options, propertyCategory,
    (UhtParsingScope topScope, UhtProperty property, ref
UhtToken nameToken, UhtLayoutMacroType layoutMacroType) =>
{
    property.PropertyFlags |= EPropertyFlags.Parm;
    if (advancedDisplay.CanMarkMore() &&
advancedDisplay.ShouldMarkParameter(property.EngineName))
    {
        property.PropertyFlags |=
EPropertyFlags.AdvancedDisplay;
    }
}

```

## SetParam

- **Function Description:** Specifies a function to utilize Set with a wildcard generic element type.
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** string = "A" | B | C"
- **Restriction Type:** TSet
- **Commonliness:** ★★★

The source code is located in UBlueprintSetLibrary.

SetParam supports multiple Set and element parameters, separated by '|', and then the Pin pin can be passed '|2'    Test Code: | ItemA, SetB | 3    The effect in the blueprint:

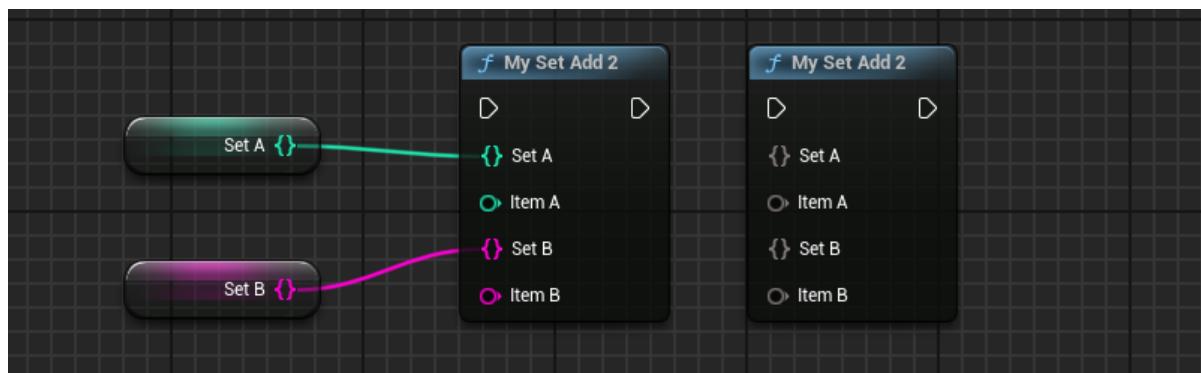
## 4 Principle:

```

UFUNCTION(BlueprintCallable, CustomThunk, meta = (SetParam =
"SetA|ItemA,SetB|ItemB"))
    static void MySet_Add2(const TSet<int32>& SetA, const int32& ItemA, const
TSet<int32>& SetB, const int32& ItemB);
    static void GenericMySet_Add2(const void* TargetSet, const FSetPropertyParams*
SetA, const void* ItemA, const FSetPropertyParams* SetB, const void* ItemB);
DECLARE_FUNCTION(execMySet_Add2);

```

## Effect in the Blueprint:



# Principle:

Groups are separated by ',', and ' ' is used within the group | All parameters Pin separated by '...' within a group are of the same type.

```
void UK2Node_CallFunction::ConformContainerPins()
{
    // find any pins marked as SetParam
    const FString& SetPinMetaData = TargetFunction-
>GetMetaData(FBlueprintMetadata::MD_SetParam);

    // useless copies/allocates in this code, could be an optimization
target...
    TArray<FString> SetParamPinGroups;
    {
        SetPinMetaData.ParseIntoArray(SetParamPinGroups, TEXT(", "), true);
    }

    for (FString& Entry : SetParamPinGroups)
    {
        // split the group:
        TArray<FString> GroupEntries;
        Entry.ParseIntoArray(GroupEntries, TEXT(" | "), true);
        // resolve pins
        TArray<UEdGraphPin*> ResolvedPins;
        for(UEdGraphPin* Pin : Pins)
        {
            if (GroupEntries.Contains(Pin->GetName()))
            {
                ResolvedPins.Add(Pin);
            }
        }

        // if nothing is connected (or non-default), reset to wildcard
        // else, find the first type and propagate to everyone else::
        bool bReadyToPropagatSetType = false;
        FEdGraphTerminalType TypeToPropagate;
        for (UEdGraphPin* Pin : ResolvedPins)
        {
            TryReadTypeToPropagate(Pin, bReadyToPropagatSetType,
TypeToPropagate);
            if(bReadyToPropagatSetType)
            {
                break;
            }
        }

        for (UEdGraphPin* Pin : ResolvedPins)
        {
            TryPropagateType( Pin, TypeToPropagate, bReadyToPropagatSetType
);
        }
    }
}
```

# MapParam

- **Function Description:** Designates a function to use `TMap< TKey, TValue >` with a wildcard generic element type.
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Restriction Type:** TMap
- **Associated Items:** MapKeyParam, MapValueParam
- **Commonality:** ★★★

Designates a function to use `TMap< TKey, TValue >` with a wildcard generic element type.

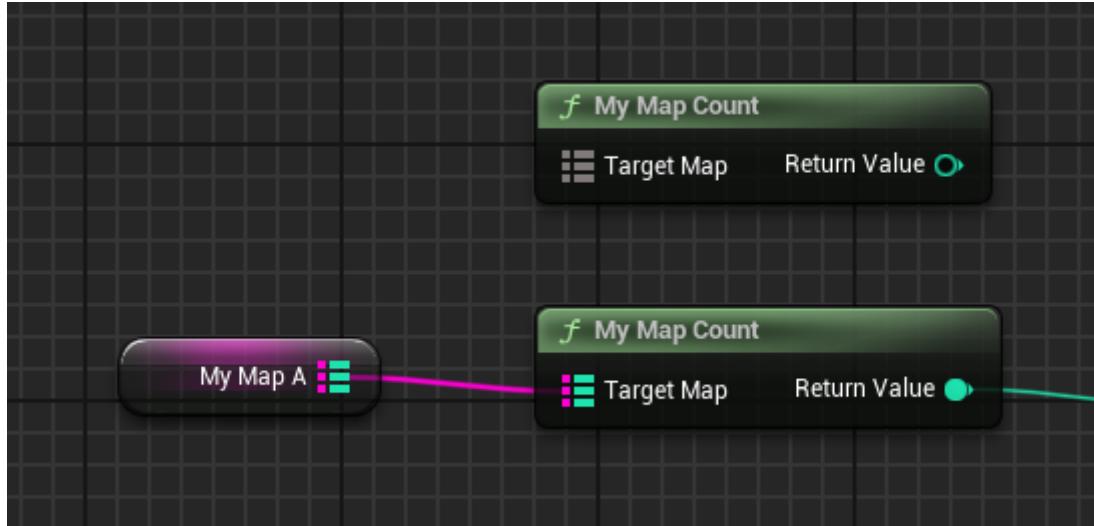
Only one MapParam is supported; the source code implementation finds a pin based on a single name.

Examples in the source code are all found within `UBlueprintMapLibrary`.

## Test Code 1:

```
UFUNCTION(BlueprintPure, CustomThunk, meta = (MapParam = "TargetMap"))
static int32 MyMap_Count(const TMap<int32, int32>& TargetMap);
static int32 GenericMyMap_Count(const void* TargetMap, const FMapPropertyParams);
DECLARE_FUNCTION(execMyMap_Count);
```

## Blueprint Effect 1:

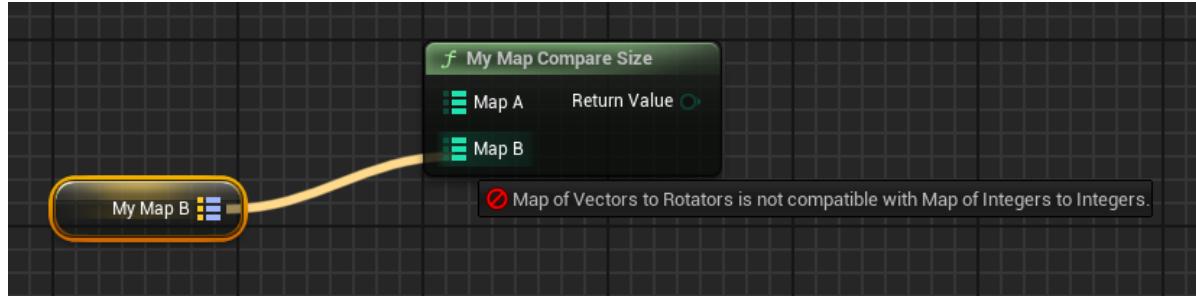


Due to the support for only one MapParam, if you write code like this.

## Test Code 2:

```
UFUNCTION(BlueprintPure, CustomThunk, meta = (MapParam = "MapA,MapB"))
static int32 MyMap_CompareSize(const TMap<int32, int32>& MapA, const
TMap<int32, int32>& MapB);
static int32 GenericMyMap_CompareSize(void* MapA, const FMapProperty*>
MapAProp, void* MapB, const FMapProperty*> MapBProp);
DECLARE_FUNCTION(execMyMap_CompareSize);
```

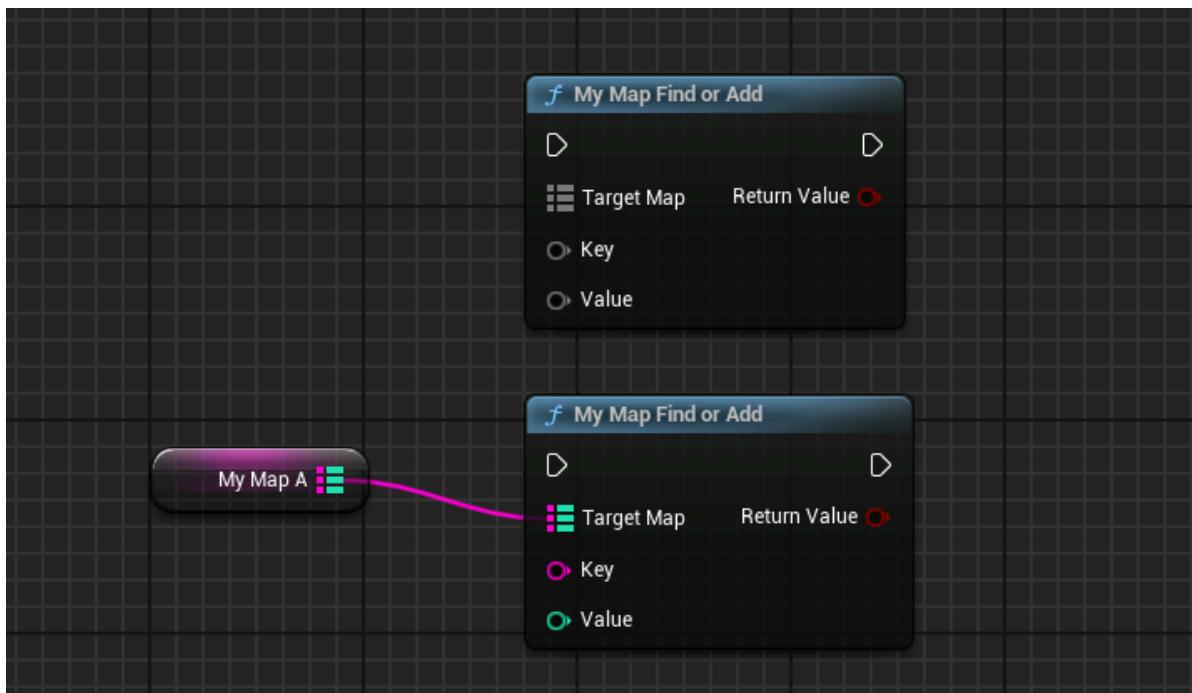
This will result in MapParam being unable to find the pin, thus losing the wildcard functionality.



To implement features similar to Add, where the pin types for Key and Value can dynamically change according to the Map type, you need to add MapKeyParam and MapValueParam to specify additional function parameters. This allows the correct pins to be located, thus enabling the dynamic adjustment of the KeyValue pin types based on the Map type. The parameters specified by MapKeyParam and MapValueParam can also be containers such as arrays, as seen with the Keys and Values parameters in UBlueprintMapLibrary.

```
UFUNCTION(BlueprintCallable, CustomThunk, meta = (MapParam =
"TargetMap", MapKeyParam = "Key", MapValueParam = "Value"))
static bool MyMap_FindOrAdd(const TMap<int32, int32>& TargetMap, const int32&
Key, const int32& value);
static bool GenericMyMap_FindOrAdd(const void* TargetMap, const FMapProperty*>
MapProperty, const void* KeyPtr, const void* ValuePtr);
DECLARE_FUNCTION(execMyMap_FindOrAdd);
```

## Blueprint Effect 2:



## Principle Code:

```
void UK2Node_CallFunction::ConformContainerPins()
{
    //Detect the container pin within this
    const FString& MapPinMetaData = TargetFunction->GetMetaData(FBlueprintMetadata::MD_MapParam);
    const FString& MapKeyPinMetaData = TargetFunction->GetMetaData(FBlueprintMetadata::MD_MapKeyParam);
    const FString& MapValuePinMetaData = TargetFunction->GetMetaData(FBlueprintMetadata::MD_MapValueParam);

    if(!MapPinMetaData.IsEmpty() || !MapKeyPinMetaData.IsEmpty() || !MapValuePinMetaData.IsEmpty())
    {
        // if the map pin has a connection infer from that, otherwise use the
        // information on the key param and value param:
        bool bReadyToPropagateKeyType = false;
        FEdGraphTerminalType KeyTypeToPropagate;
        bool bReadyToPropagateValueType = false;
        FEdGraphTerminalType ValueTypeToPropagate;

        UEdGraphPin* MapPin = MapPinMetaData.IsEmpty() ? nullptr :
            FindPin(MapPinMetaData);
        UEdGraphPin* MapKeyPin = MapKeyPinMetaData.IsEmpty() ? nullptr :
            FindPin(MapKeyPinMetaData);
        UEdGraphPin* MapValuePin = MapValuePinMetaData.IsEmpty() ? nullptr :
            FindPin(MapValuePinMetaData);

        TryReadTypeToPropagate(MapPin, bReadyToPropagateKeyType,
            KeyTypeToPropagate); //Read the connection type of the MapPin's Key
        TryReadValueTypeToPropagate(MapPin, bReadyToPropagateValueType,
            ValueTypeToPropagate); //Read the Map Value type connected to the MapPin
    }
}
```

```

        TryReadTypeToPropagate(MapKeyPin, bReadyToPropagateKeyType,
KeyTypeToPropagate); //Read the connection type on the KeyPin
        TryReadTypeToPropagate(MapValuePin, bReadyToPropagateValueType,
valueTypeToPropagate); //Read the connection type on the valuePin

        TryPropagateType(MapPin, KeyTypeToPropagate,
bReadyToPropagateKeyType); //Change the current type of the MapPin's Key
        TryPropagateType(MapKeyPin, KeyTypeToPropagate,
bReadyToPropagateKeyType); //Change the current type of the KeyPin

        TryPropagateValueType(MapPin, valueTypeToPropagate,
bReadyToPropagateValueType); //Change the current type of the MapPin's value
        TryPropagateType(MapValuePin, valueTypeToPropagate,
bReadyToPropagateValueType); //Change the current type of the valuePin
    }
}

```

## MapKeyParam

---

- **Function Description:** Specifies a function parameter as the key of a map, which dynamically adjusts based on the key type of the actual map parameter indicated by MapParam.
- **Usage Location:** UFUNCTION
- **Metadata Type:** string="abc"
- **Restriction Type:** TMap
- **Associated Item:** MapParam
- **Commonality:** ★★★

## MapValueParam

---

- **Function Description:** Specifies a function parameter as the value of a map, which dynamically adjusts based on the value type of the actual map parameter indicated by MapParam.
- **Usage Location:** UFUNCTION
- **Metadata Type:** string="abc"
- **Restriction Type:** TMap
- **Associated Item:** MapParam
- **Commonality:** ★★★

## Keywords

---

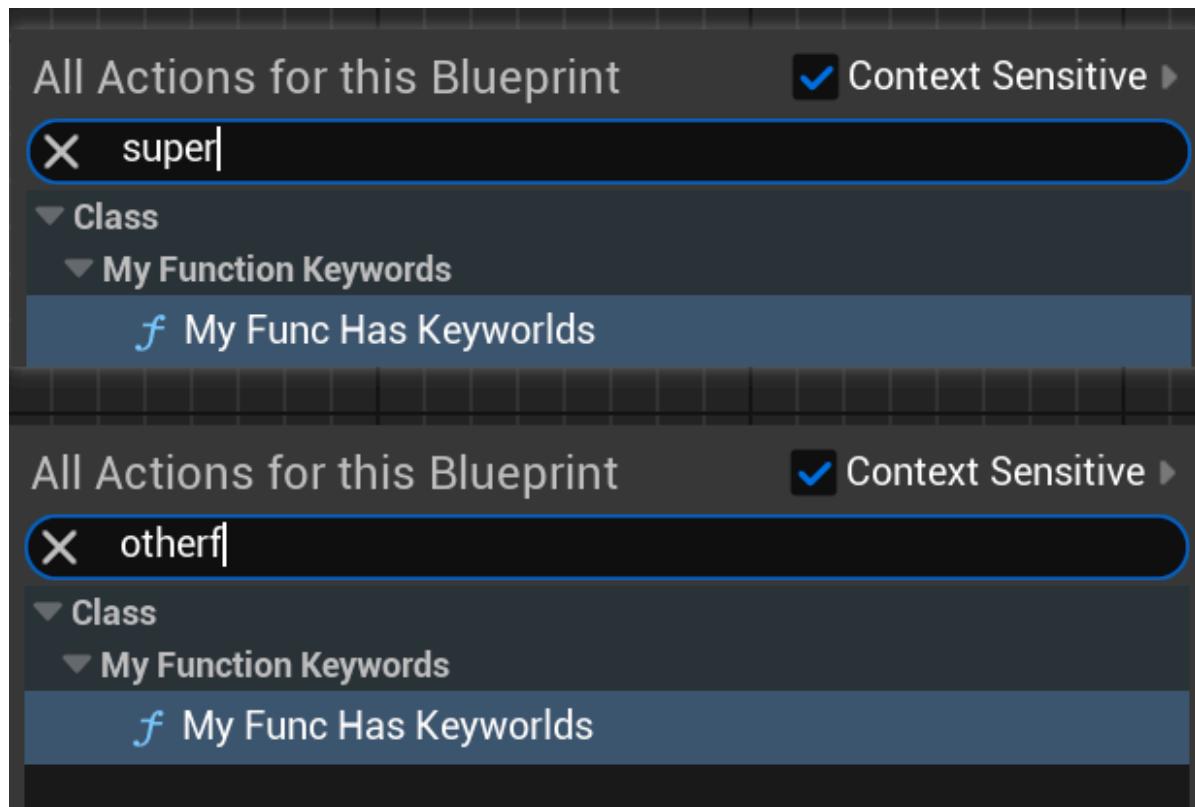
- **Function Description:** Specifies a set of keywords to locate the function within the blueprint via a right-click search
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Commonality:** ★★★★★

Keywords may be separated by spaces or commas. The text within will be subject to string matching search.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Keywords :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta=(Keywords="This is a SuperFunc,OtherFunc"))
        static void MyFunc_HasKeyworlds();
};
```

## Blueprint Effect:



## Principle:

The content of the Keywords will ultimately be utilized by FEdGraphSchemaAction for text search in the blueprint's right-click menu.

In addition, each K2Node is capable of returning a Keywords entry. The effect should be identical to that of Keywords on a function.

```
FText UEdGraphNode::GetKeywords() const
{
    return GetClass()->GetMetaDataText(TEXT("Keywords"), TEXT("UObjectKeywords")),
    GetClass()->GetFullName(false));
}
```

# Latent

- **Function Description:** Indicates that a function is a latent asynchronous operation
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Associated Items:** LatentInfo, NeedsLatentFixup, LatentCallbackTarget
- **Commonality:** ★★★★☆

Indicates that a function is a latent asynchronous operation and requires the use of LatentInfo for proper functionality.

It results in the manual triggering of the Then (also referred to as Complete) pin during logical execution (triggered internally by the engine), and a clock icon is added to the top right corner of the function.

## Test Code:

```
class FMySleepAction : public FPendingLatentAction
{
public:
    float TimeRemaining;
    FName ExecutionFunction;
    int32 OutputLink;
    FWeakObjectPtr CallbackTarget;

    FMySleepAction(float Duration, const FLatentActionInfo& LatentInfo)
        : TimeRemaining(Duration)
        , ExecutionFunction(LatentInfo.ExecutionFunction)
        , OutputLink(LatentInfo.Linkage)
        , CallbackTarget(LatentInfo.CallbackTarget)
    {
    }

    virtual void UpdateOperation(FLatentResponse& Response) override
    {
        TimeRemaining -= Response.ElapsedTime();
        Response.FinishAndTriggerIf(TimeRemaining <= 0.0f, ExecutionFunction,
        OutputLink, CallbackTarget);
    }
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Latent : public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta = (Latent, WorldContext =
    "WorldContextObject", LatentInfo = "LatentInfo", Duration = "5"))
    static void MySleep(const UObject* WorldContextObject, float Duration,
    FLatentActionInfo LatentInfo)
```

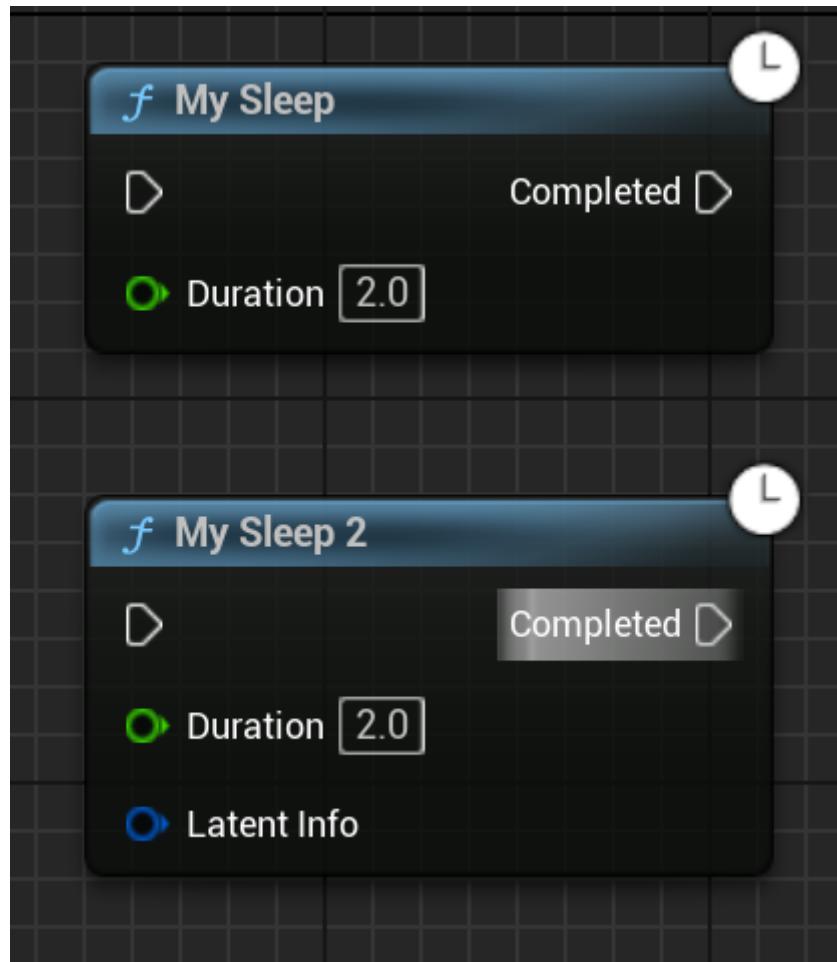
```

{
    if (UWorld* World = GEngine-
>GetWorldFromContextObject(WorldContextObject,
EGetWorldErrorMode::LogAndReturnNull))
    {
        FLatentActionManager& LatentActionManager = World-
>GetLatentActionManager();
        if (LatentActionManager.FindExistingAction<FMySleepAction>
(LatentInfo.CallbackTarget, LatentInfo.UUID) == NULL)
        {
            LatentActionManager.AddNewAction(LatentInfo.CallbackTarget,
LatentInfo.UUID, new FMySleepAction(Duration, LatentInfo));
        }
    }
}

UFUNCTION(BlueprintCallable, meta = (Latent, WorldContext =
"WorldContextObject", Duration = "5"))
static void MySleep2(const UObject* WorldContextObject, float Duration,
FLatentActionInfo LatentInfo);
};

```

## Blueprint Effect:



MySleep functions normally like Delay. However, MySleep2 does not have LatentInfo marked, so the LatentInfo function parameters are not assigned by the blueprint system, rendering it inoperable.

Latent is frequently used in the source code, with the most common example being:

```

UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject", Latent =
 "", LatentInfo = "LatentInfo", DisplayName = "Load Stream Level (by Name)", Category="Game")
static ENGINE_API void LoadStreamLevel(const UObject* WorldContextObject, FName LevelName, bool bMakeVisibleAfterLoad, bool bShouldBlockOnLoad, FLatentActionInfo LatentInfo);

UFUNCTION(BlueprintCallable, meta = (Latent, LatentInfo = "LatentInfo", worldContext = "WorldContextObject", BlueprintInternalUseOnly = "true"), Category = "Utilities")
static ENGINE_API void LoadAsset(const UObject* WorldContextObject, TSoftObjectPtr<UObject> Asset, FOnAssetLoaded OnLoaded, FLatentActionInfo LatentInfo);

UFUNCTION(BlueprintCallable, Category="Utilities|FlowControl", meta=(Latent, worldContext="WorldContextObject", LatentInfo="LatentInfo", Duration="0.2", Keywords="sleep"))
static ENGINE_API void Delay(const UObject* WorldContextObject, float Duration, struct FLatentActionInfo LatentInfo );

```

For the differences between using Latent and inheriting from UBlueprintAsyncActionBase to create blueprint asynchronous nodes, further information can be found in other articles online.

#NeedsLatentFixup

- **Function Description:** Used on the FLatentActionInfo::Linkage attribute to instruct the Blueprint VM to generate jump information
- **Usage Location:** UPROPERTY
- **Metadata Type:** boolean
- **Associated Items:** Latent
- **Commonality:** ★

## Applications in the Source Code:

```

USTRUCT(BlueprintInternalUseOnly)
struct FLatentActionInfo
{
    GENERATED_USTRUCT_BODY()

    /** The resume point within the function to execute */
    UPROPERTY(meta=(NeedsLatentFixup = true))
    int32 Linkage;

    //...
};

```

# Source Code Implementation:

Appears to handle the Linkage attribute separately, used for specialized jumps within the JumpTargetFixupMap

```
void EmitLatentInfoTerm(FBPTerminal* Term, FProperty* LatentInfoProperty,
FBlueprintCompiledStatement* TargetLabel)
{
    // Special case of the struct property emitter. Needs to emit a linkage
    // property for fixup
    FStructProperty* StructProperty = CastFieldChecked<FStructProperty>
    (LatentInfoProperty);
    check(StructProperty->Struct == LatentInfoStruct);

    int32 StructSize = LatentInfoStruct->GetStructureSize();
    uint8* StructData = (uint8*)FMemory_Alloca(StructSize);
    StructProperty->InitializeValue(StructData);

    // Assume that any errors on the import of the name string have been caught
    // in the function call generation
    StructProperty->ImportText_Direct(*Term->Name, StructData, NULL, 0, GLog);

    Writer << EX_StructConst;
    Writer << LatentInfoStruct;
    Writer << StructSize;

    checksSlow(Schema);
    for (FProperty* Prop = LatentInfoStruct->PropertyLink; Prop; Prop = Prop-
>PropertyLinkNext)
    {
        if (TargetLabel && Prop-
>GetBoolMetaData(FBlueprintMetadata::MD_NeedsLatentFixup))
        {
            // Emit the literal and queue a fixup to correct it once the address
            // is known
            Writer << EX_SkipOffsetConst;
            CodeSkipSizeType PatchUpNeededAtOffset =
            Writer.EmitPlaceholderskip();
            JumpTargetFixupMap.Add(PatchUpNeededAtOffset,
FCodeskipInfo(FCodeskipInfo::Fixup, TargetLabel));
        }
        else if (Prop-
>GetBoolMetaData(FBlueprintMetadata::MD_LatentCallbackTarget))
        {
            FBPTerminal CallbackTargetTerm;
            CallbackTargetTerm.bIsLiteral = true;
            CallbackTargetTerm.Type.PinSubCategory = UEdGraphSchema_K2::PN_Self;
            EmitTermExpr(&CallbackTargetTerm, Prop);
        }
    }
    else
    {
        // Create a new term for each property, and serialize it out
        FBPTerminal NewTerm;
        if(Schema->ConvertPropertyToPinType(Prop, NewTerm.Type))
        {
    }
```

```

        NewTerm.bIsLiteral = true;
        Prop->ExportText_InContainer(0, NewTerm.Name, StructData,
StructData, NULL, PPF_None);

        EmitTermExpr(&NewTerm, Prop);
    }
    else
    {
        // Do nothing for unsupported/unhandled property types. This will
        leave the value unchanged from its constructed default.
        writer << EX_Nothing;
    }
}

writer << EX_EndStructConst;
}

```

## LatentInfo

- **Function description:** Used in conjunction with Latent to specify which function parameter is the LatentInfo parameter.
- **Use location:** UFUNCTION
- **Metadata type:** string="abc"
- **Related items:** Latent
- **Frequency:** ★★

Latent functions require FLatentActionInfo to operate. FLatentActionInfo contains details such as the ID of the delayed operation and the name of the next function to be executed. In the blueprint virtual machine environment, when a Latent function is executed, the blueprint VM gathers the current function context information (e.g., the next node connected to the Latent function) and then assigns it to the FLatentActionInfo parameter of the Latent function, subsequently registering it with FPendingLatentAction in the FLatentActionManager. Once the time elapses or the trigger condition is met, FLatentActionManager invokes CallbackTarget->ProcessEvent(ExecutionFunction, &(LinkInfo.LinkID)), thereby resuming execution.

If function parameters are not specified using LatentInfo, the assignment operation for LatentInfo is disrupted, and thus the function will not operate correctly. The blueprint diagram can be found on the Latent page.

The value of LatentInfo, much like WorldContext, is automatically populated by the Blueprint VM system. This value population occurs within EmitLatentInfoTerm, where the values of LatentInfoStruct are filled into the function parameters of LatentInfo. The position of the LatentInfo parameter is not significant, and the function parameter pin specified by LatentInfo will be hidden.

```

void EmitFunctionCall(FKismetCompilerContext& CompilerContext,
FKismetFunctionContext& FunctionContext, FBlueprintCompiledStatement& Statement,
UEdGraphNode* SourceNode)
{
    if (bIsUbergraph && FuncParamProperty->GetName() == FunctionToCall-
>GetMetaData(FBlueprintMetadata::MD_LatentInfo))

```

```

    {
        EmitLatentInfoTerm(Term, FuncParamProperty,
Statement.TargetLabel);
    }
}

void EmitLatentInfoTerm(FBPTerminal* Term, FProperty* LatentInfoProperty,
FBlueprintCompiledStatement* TargetLabel)
{
    // Special case of the struct property emitter. Needs to emit a linkage
    // property for fixup
    FStructProperty* StructProperty = CastFieldChecked<FStructProperty>
(LatentInfoProperty);
    check(StructProperty->Struct == LatentInfoStruct);

    int32 StructSize = LatentInfoStruct->GetStructureSize();
    uint8* StructData = (uint8*)FMemory_Alloca(StructSize);
    StructProperty->InitializeValue(StructData);

    // Assume that any errors on the import of the name string have been caught
    // in the function call generation
    StructProperty->ImportText_Direct(*Term->Name, StructData, NULL, 0, GLog);

    Writer << EX_StructConst;
    Writer << LatentInfoStruct;
    Writer << StructSize;

    checksSlow(Schema);
    for (FProperty* Prop = LatentInfoStruct->PropertyLink; Prop; Prop = Prop-
>PropertyLinkNext)
    {
        if (TargetLabel && Prop-
>GetBoolMetaData(FBlueprintMetadata::MD_NeedsLatentFixup))
        {
            // Emit the literal and queue a fixup to correct it once the address
            // is known
            Writer << EX_SkipOffsetConst;
            CodeskipSizeType PatchUpNeededAtOffset =
Writer.EmitPlaceholderskip();
            JumpTargetFixupMap.Add(PatchUpNeededAtOffset,
FCodeSkipInfo(FCodeSkipInfo::Fixup, TargetLabel));
        }
        else if (Prop-
>GetBoolMetaData(FBlueprintMetadata::MD_LatentCallbackTarget))
        {
            FBPTerminal CallbackTargetTerm;
            CallbackTargetTerm.bIsLiteral = true;
            CallbackTargetTerm.Type.PinSubCategory = UEdGraphSchema_K2::PN_Self;
            EmitTermExpr(&CallbackTargetTerm, Prop);
        }
    }
    else
    {
        // Create a new term for each property, and serialize it out
        FBPTerminal NewTerm;
        if(Schema->ConvertPropertyParamsToPinType(Prop, NewTerm.Type))
        {

```

```

        NewTerm.bIsLiteral = true;
        Prop->ExportText_InContainer(0, NewTerm.Name, StructData,
StructData, NULL, PPF_None);

        EmitTermExpr(&NewTerm, Prop);
    }
    else
    {
        // Do nothing for unsupported/unhandled property types. This will
        leave the value unchanged from its constructed default.
        writer << EX_Nothing;
    }
}

writer << EX_EndStructConst;
}

```

The collection of LatentInfo occurs in FKCHandler\_CallFunction::CreateFunctionCallStatement

## LatentCallbackTarget

- **Function Description:** Specifies the target object on which the function should be invoked for the FLatentActionInfo::CallbackTarget property, guiding the Blueprint VM accordingly.
- **Usage Location:** UPROPERTY
- **Metadata Type:** bool
- **Associated Items:** Latent
- **Commonality:** ★

Indicates the target object for function invocation on the FLatentActionInfo::CallbackTarget property, instructing the Blueprint VM on which object to call the function.

```

USTRUCT(BlueprintInternalUseOnly)
struct FLatentActionInfo
{
    GENERATED_USTRUCT_BODY()

    /** Object to execute the function on. */
    UPROPERTY(meta=(LatentCallbackTarget = true))
    TObjectPtr<UObject> CallbackTarget;

    //...
};

```

## Location of Use in Source Code:

```

void EmitLatentInfoTerm(FBPTerminal* Term, FProperty* LatentInfoProperty,
FBlueprintCompiledStatement* TargetLabel)
{
    // Special case of the struct property emitter. Needs to emit a linkage
    property for fixup
}

```

```

FStructProperty* StructProperty = CastFieldChecked<FStructProperty>
(LatentInfoProperty);
check(StructProperty->Struct == LatentInfoStruct);

int32 StructSize = LatentInfoStruct->GetStructureSize();
uint8* StructData = (uint8*)FMemory_Alloca(StructSize);
StructProperty->InitializeValue(StructData);

// Assume that any errors on the import of the name string have been caught
// in the function call generation
StructProperty->ImportText_Direct(*Term->Name, StructData, NULL, 0, GLog);

Writer << EX_StructConst;
Writer << LatentInfoStruct;
Writer << StructSize;

checksSlow(Schema);
for (FProperty* Prop = LatentInfoStruct->PropertyLink; Prop; Prop = Prop-
>PropertyLinkNext)
{
    if (TargetLabel && Prop-
>GetBoolMetaData(FBlueprintMetadata::MD_NeedsLatentFixup))
    {
        // Emit the literal and queue a fixup to correct it once the address
        // is known
        Writer << EX_SkipOffsetConst;
        CodeskipSizeType PatchUpNeededAtOffset =
Writer.EmitPlaceholderskip();
        JumpTargetFixupMap.Add(PatchUpNeededAtOffset,
FCodeSkipInfo(FCodeSkipInfo::Fixup, TargetLabel));
    }
    else if (Prop-
>GetBoolMetaData(FBlueprintMetadata::MD_LatentCallbackTarget))
    {
        FBPTerminal CallbackTargetTerm;
        CallbackTargetTerm.bIsLiteral = true;
        CallbackTargetTerm.Type.PinSubCategory = UEdGraphSchema_K2::PN_Self;
        EmitTermExpr(&CallbackTargetTerm, Prop);
    }
    else
    {
        // Create a new term for each property, and serialize it out
        FBPTerminal NewTerm;
        if(Schema->ConvertPropertyToPinType(Prop, NewTerm.Type))
        {
            NewTerm.bIsLiteral = true;
            Prop->ExportText_InContainer(0, NewTerm.Name, StructData,
StructData, NULL, PPF_None);

            EmitTermExpr(&NewTerm, Prop);
        }
        else
        {
            // Do nothing for unsupported/unhandled property types. This will
            // leave the value unchanged from its constructed default.
            Writer << EX_Nothing;
        }
    }
}

```

```

        }
    }

    writer << EX_EndStructConst;
}

```

## NativeMakeFunc

- **Function description:** Specifies a function to use the MakeStruct icon
- ◦ **Use location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** Boolean
- **Associated Items:** NativeBreakFunc
- **Usage Frequency:** ★

Principle:

The sole instance found in the engine source code is as follows. Thus, there is no actual logical difference between the tags, but there is a difference in their display.

## Test Code:

```

USTRUCT(BlueprintType)
struct FMyStruct_ForNative
{
    GENERATED_BODY()

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 X = 0;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 Y = 0;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 Z = 0;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_NativeMakeBreak :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()

public:
    UFUNCTION(BlueprintPure, meta = (NativeMakeFunc))
    static FMyStruct_ForNative MakeMyStructNative(FString ValueString);

    UFUNCTION(BlueprintPure)
    static FMyStruct_ForNative MakeMyStructNative_NoMeta(FString ValueString);

    UFUNCTION(BlueprintPure, meta = (NativeBreakFunc))

```

```

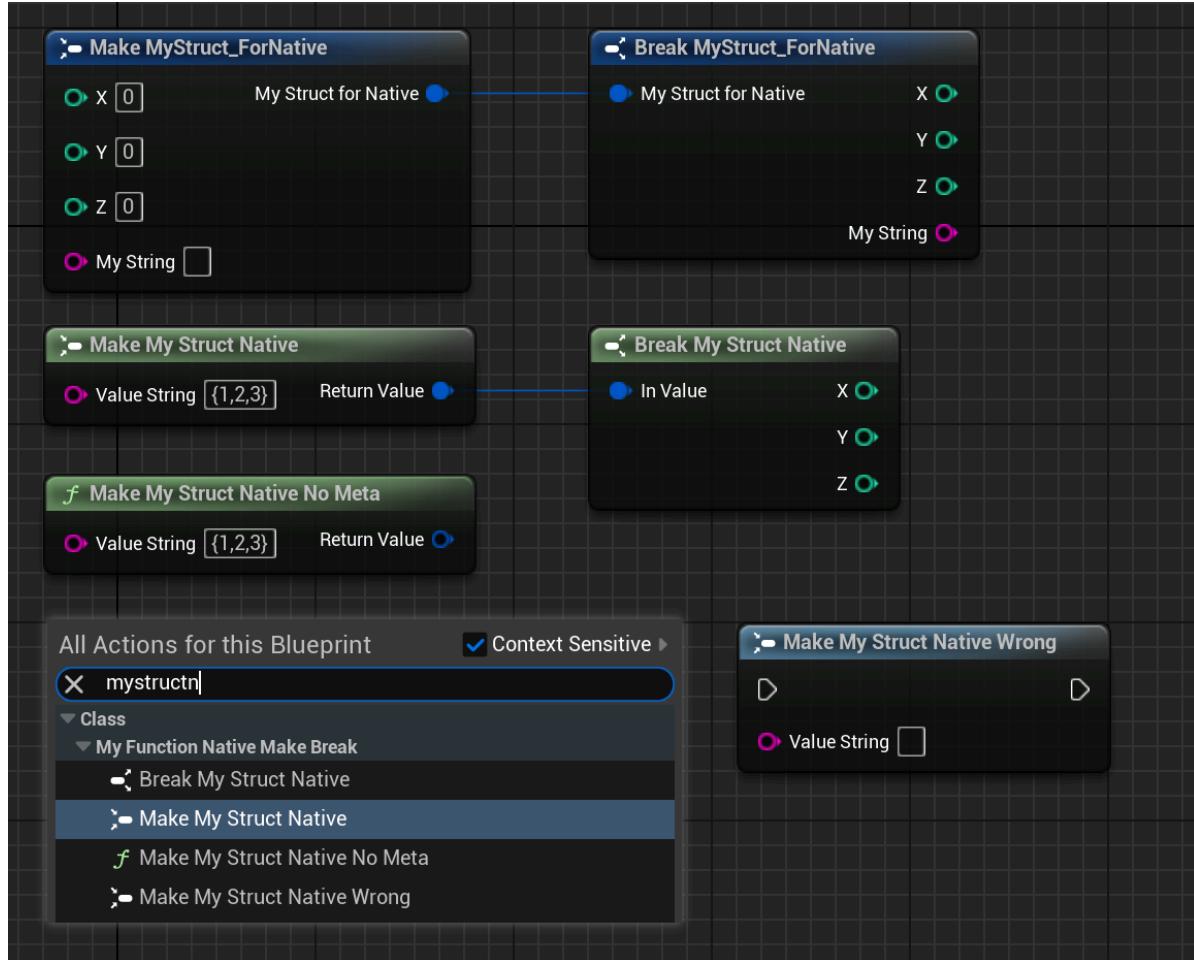
    static void BreakMyStructNative(const FMyStruct_ForNative& InValue, int32& X,
int32& Y, int32& Z);

    UFUNCTION(BlueprintCallable, meta = (NativeMakeFunc))
    static void MakeMyStructNative_Wrong(FString ValueString);
};


```

## Effect within the Blueprint:

Observe that when "NoMeta" is present, the function's icon is the standard 'f' icon; otherwise, a different icon is displayed. Furthermore, note that a Struct can have multiple "Make" and "Break" functions, all of which can be utilized concurrently without issue.



## Principle:

The sole instance of this code was found within the engine's source code. Thus, while the tags do not differ logically, there is a discernible difference in their visual presentation.

Distinct icons are employed for the NativeMakeFunc and NativeBrakeFunc, indicating their unique roles.

```

FSlateIcon UK2Node_CallFunction::GetPaletteIconForFunction(UFunction const*
Function, FLinearColor& OutColor)
{
    static const FName NativeMakeFunc(TEXT("NativeMakeFunc"));
    static const FName NativeBrakeFunc(TEXT("NativeBreakFunc"));

    if (Function && Function->HasMetaData(NativeMakeFunc))

```

```

{
    static FSlateIcon Icon(FAppStyle::GetAppStyleSetName(),
"GraphEditor.MakeStruct_16x");
    return Icon;
}
else if (Function && Function->HasMetaData(NativeBreakFunc))
{
    static FSlateIcon Icon(FAppStyle::GetAppStyleSetName(),
"GraphEditor.BreakStruct_16x");
    return Icon;
}
// Check to see if the function is calling an function that could be an
event, display the event icon instead.
else if (Function && UEdGraphSchema_K2::FunctionCanBePlacedAsEvent(Function))
{
    static FSlateIcon Icon(FAppStyle::GetAppStyleSetName(),
"GraphEditor.Event_16x");
    return Icon;
}
else
{
    OutColor = GetPaletteIconColor(Function);

    static FSlateIcon Icon(FAppStyle::GetAppStyleSetName(),
"Kismet.AllClasses.FunctionIcon");
    return Icon;
}
}
}

```

## NativeBreakFunc

- **Function Description:** Specifies a function that utilizes the icon from BreakStruct.
- **Usage Location:** UFUNCTION
- **Metadata Type:** boolean
- **Associated Items:** NativeMakeFunc
- **Commonality:** ★

The functionality has already been explained in NativeMakeFunc

## UnsafeDuringActorConstruction

- **Function Description:** Indicates that this function should not be invoked within an Actor's constructor
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Commonality:** ★★

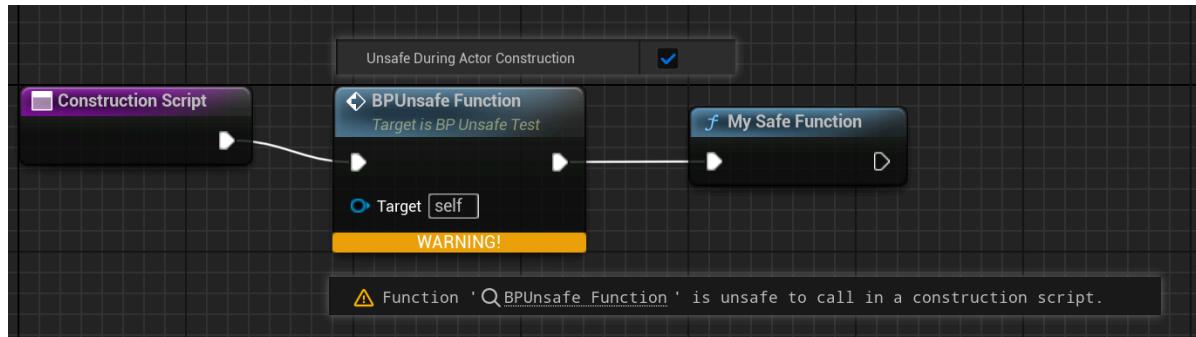
Signifies that this function is not to be called during the construction of an Actor. Typically, functions related to rendering and physics are prohibited from being called during this phase.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Unsafe :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    static void MySafeFunction();
    UFUNCTION(BlueprintCallable, meta=(UnsafeDuringActorConstruction = "true"))
    static void MyUnsafeFunction();
};
```

The Meta: UnsafeDuringActorConstruction attribute can also be set in the function's details panel within a Blueprint, achieving the same effect as setting it in C++.

It is observed that the MyUnsafeFunction cannot be invoked within an Actor's constructor, and custom functions within Blueprints will also produce warnings and compilation errors when marked with the UnsafeDuringActorConstruction attribute.



## Principle:

In Blueprints, there are clear checks in place for this series of conditions, making them easily identifiable.

```
bool UEdGraphSchema_K2::CanFunctionBeUsedInGraph(const UClass* InClass, const
UFunction* InFunction, const UEdGraph* InDestGraph, uint32
InAllowedFunctionTypes, bool bInCalledForEach, FText* OutReason) const
{
    const bool bIsUnsafeForConstruction = InFunction-
>GetBoolMetaData(FBlueprintMetadata::MD_UnsafeForConstructionScripts);
    if (bIsUnsafeForConstruction && bIsConstructionScript)
    {
        if(OutReason != nullptr)
        {
            *OutReason = LOCTEXT("FunctionUnsafeForConstructionScript", "Function
cannot be used in a Construction Script.");
        }
    }
    return false;
}
```

# BlueprintAutocast

- **Function Description:** Instruct the blueprint system that this function facilitates automatic conversion from type A to type B.
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Commonality:** ★

Inform the blueprint system that this function facilitates automatic conversion from type A to type B.

Automatic conversion refers to the process where, when dragging a Pin of type A to a Pin of type B, the blueprint automatically generates a type conversion node.

This conversion function must be marked as BlueprintPure, as it is invoked passively without an active execution node.

## Test Code:

```
USTRUCT(BlueprintType)
struct FAutoCastFrom
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 X = 0;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 Y = 0;
};

USTRUCT(BlueprintType)
struct FAutoCastTo
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 Sum = 0;
};

USTRUCT(BlueprintType)
struct FNoAutoCastTo
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 Sum = 0;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_AutoCast :public UBlueprintFunctionLibrary
{
public:
```

```

GENERATED_BODY()

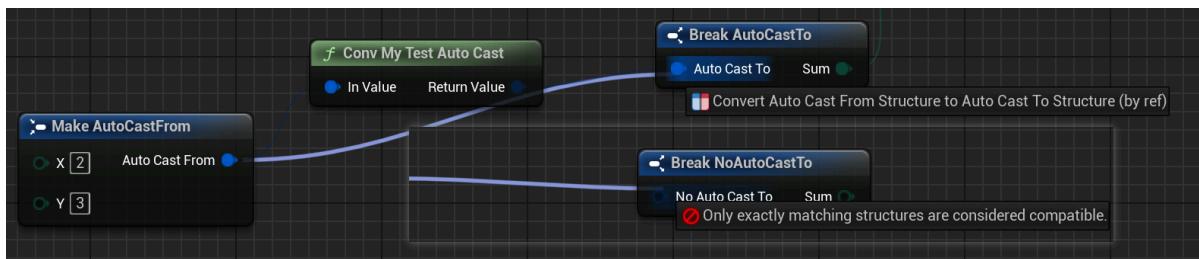
public:
    UFUNCTION(BlueprintPure, meta = (BlueprintAutocast))
        static FAutoCastTo Conv_MyTestAutoCast(const FAutoCastFrom& InValue);
};

//Conversion functions are frequently paired with CompactNodeTitle for use.
UFUNCTION(BlueprintPure, Category="Widget", meta = (CompactNodeTitle = "->", BlueprintAutocast))
static UMG_API FInputEvent GetInputEventFromKeyEvent(const FKeyEvent& Event);

```

## Example Effect:

FAutoCastTo, which supports automatic conversion, automatically generates nodes during the connection drag operation, whereas FNoAutoCastTo, lacking an automatic conversion function, will result in an error.



## Principle Code:

It is evident that the function must be static, a public C++ function, marked with BlueprintPure, have a return value, and include an input parameter. The automatic conversion relationships between types in the engine are maintained by the FAutocastFunctionMap.

```

static bool IsAutocastFunction(const UFunction* Function)
{
    const FName BlueprintAutocast(TEXT("BlueprintAutocast"));
    return Function
        && Function->HasMetaData(BlueprintAutocast)
        && Function->HasAllFunctionFlags(FUNC_Static | FUNC_Native | FUNC_Public
        | FUNC_Blueprint)
        && Function->GetReturnProperty()
        && GetFirstInputProperty(Function);
}

```

## DeterminesOutputType

- Function description:** Specifies a parameter type as a reference type for the function to dynamically adjust the output parameter type
- Use location:** UFUNCTION
- Engine module:** Blueprint
- Metadata type:** string="abc"
- Restricted type:** Class or Object pointer type, or container type
- Related items:** DynamicOutputParam

- **Frequency:** ★★★

Specifies a parameter type as the type of the function's output parameter.

Assuming a function prototype as follows:

```
UFUNCTION(BlueprintCallable, meta = (DeterminesOutputType =
"A", DynamicOutputParam="P1,P2"))
TypeR MyFunc(TypeA A, Type1 P1, Type2 P2, Type3 P3);
```

The value of `DeterminesOutputType` specifies a function parameter name, A. The type of `TypeA` must be a Class or Object, typically `TSubClassOf` or `XXX`, *but it can also be `TArray<XXX>`*, or it may point to a property within a parameter structure, such as `Args_ActorClassType`. `TSoftObjectPtr` is also acceptable, pointing to a subclass Asset object, which can then correspondingly change the base class `Asset*` output.

The term "output parameters" includes both return values and function output parameters. Therefore, `TypeR`, `P1`, and `P2` in the function prototype above are all output parameters. For the types of output parameters to change dynamically, the types of `TypeR`, `Type1`, and `Type2` must also be Class or Object types, and the type actually selected for parameter A on the blueprint node must be a subclass of the output parameter type to allow for automatic conversion.

If there are no `P1` and `P2`, and the return value is considered as `TypeR`, there is no need to specify `DynamicOutputParam`, and the return value can be automatically treated as a dynamic output parameter by default. Otherwise, `DynamicOutputParam` must be manually specified to indicate which function parameters support dynamic typing.

## Test code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyAnimalActor :public AActor
{
public:
    GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyCatActor :public AMyAnimalActor
{
public:
    GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyDogActor :public AMyAnimalActor
{
public:
    GENERATED_BODY()
};

USTRUCT(BlueprintType)
struct FMyOutputTypeArgs
{
    GENERATED_BODY()
};
```

```

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt = 1;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    TSubclassOf<AMyAnimalActor> ActorClassType;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunctionLibrary_OutputTypeTest : public
UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    //class
    UFUNCTION(BlueprintCallable, meta = (DeterminesOutputType =
"ActorClassType"))
    static TArray<AActor*> MyGetAnimals(TSubclassOf<AMyAnimalActor>
ActorClassType);

    //have to add DynamicOutputParam
    UFUNCTION(BlueprintCallable, meta = (DeterminesOutputType = "ActorClassType",
DynamicOutputParam = "OutActors"))
    static void MyGetAnimalsOut(TSubclassOf<AMyAnimalActor> ActorClassType,
TArray<AActor*>& OutActors);

    //have to add DynamicOutputParam
    UFUNCTION(BlueprintCallable, meta = (DeterminesOutputType = "ActorClassType",
DynamicOutputParam = "FirstOutActor,OutActors"))
    static void MyGetAnimalsOut2(TSubclassOf<AMyAnimalActor> ActorClassType,
AActor*& FirstOutActor, TArray<AActor*>& OutActors);

    //object
    UFUNCTION(BlueprintCallable, meta = (DeterminesOutputType = "ExampleActor"))
    static TArray<AActor*> MyGetAnimalsWithActor(AMyAnimalActor* ExampleActor);

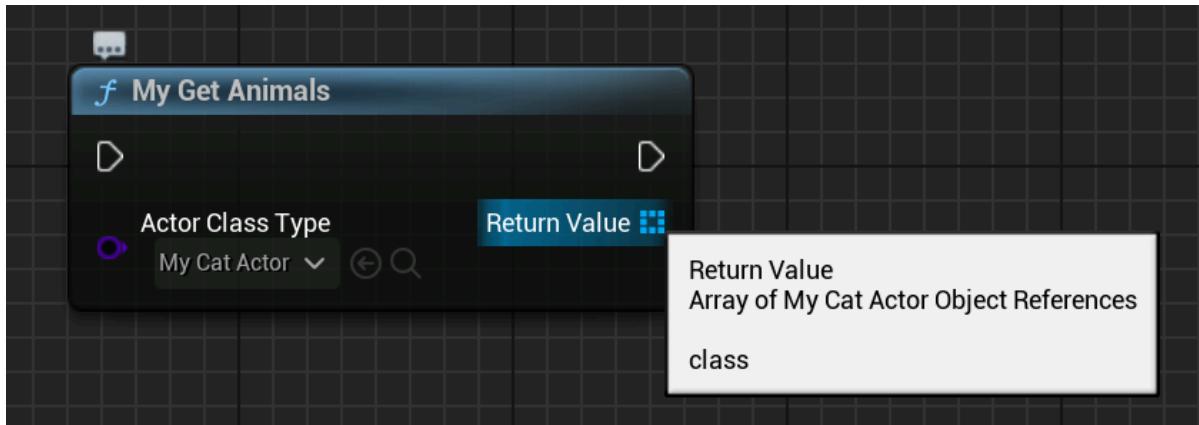
    UFUNCTION(BlueprintCallable, meta = (DeterminesOutputType =
"ExampleActorArray"))
    static TArray<AACTOR*> MyGetAnimalsWithActorArray(TArray<AMyAnimalActor*>
ExampleActorArray);

    //struct property
    UFUNCTION(BlueprintCallable, meta = (DeterminesOutputType =
"Args_ActorClassType"))
    static TArray<AACTOR*> MyGetAnimalsWithStructProperty(const
FMyOutputTypeArgs& Args);
};

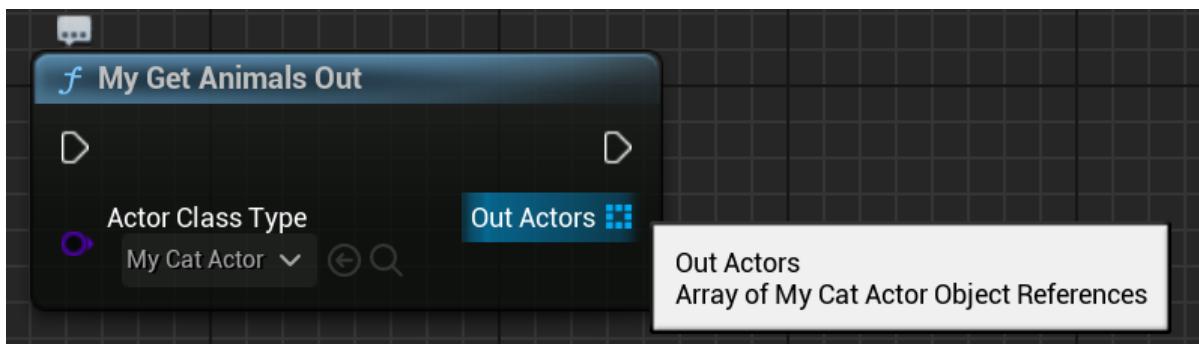
```

## Blueprint effect:

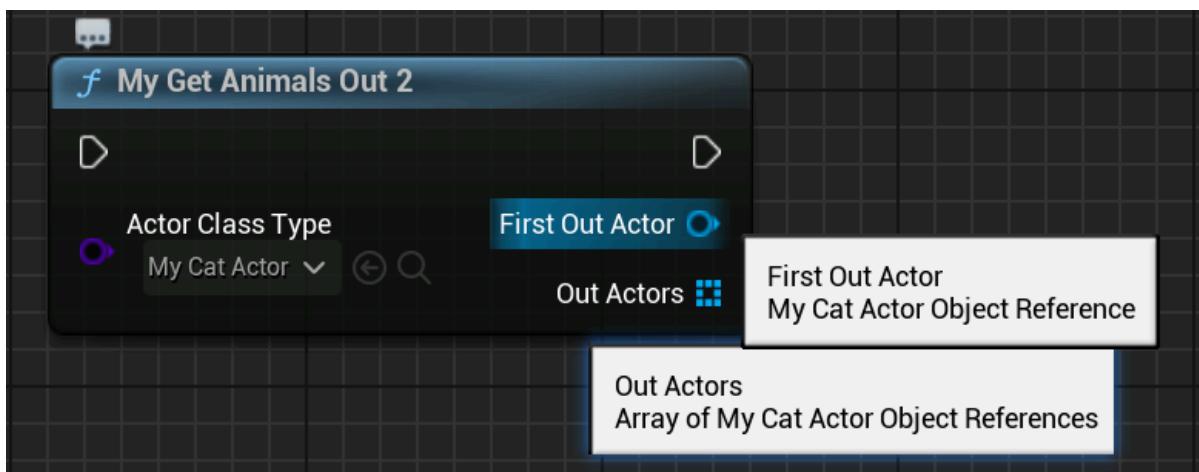
In an example where the return value is used as an output parameter, note that the return value type actually becomes `TArray<AMyCatActor*>`.



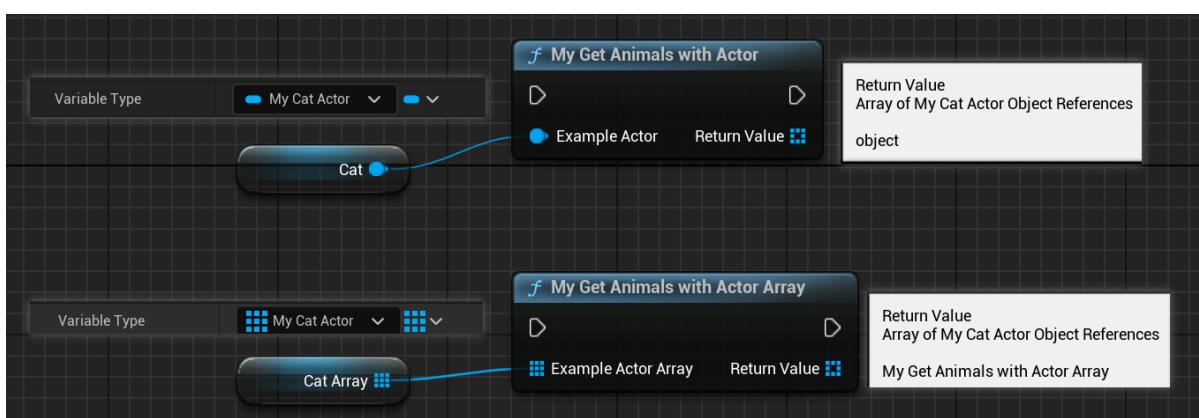
You can also add DynamicOutputParam to specify output parameters as dynamic type parameters:



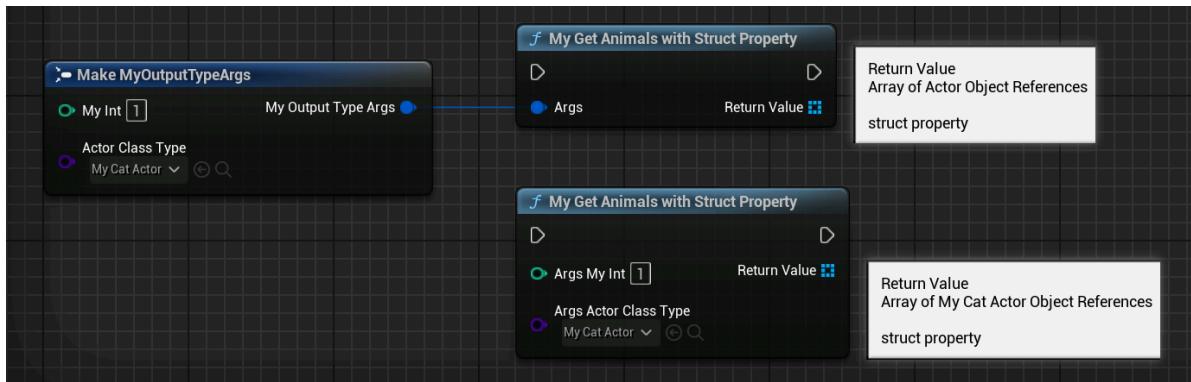
DynamicOutputParam can specify multiple parameters



The parameter type of DeterminesOutputType can also be an Object or a container of Objects



The parameter of `DeterminesOutputType` can even be an attribute within a structure, but this only applies when the structure is `SplitStruct`, because only then do the structure's attributes become function Pins, allowing for name comparison with `DeterminesOutputType`. In such cases, it should be written as "A\_B" rather than "A.B".



## Principle:

The mechanism of `DeterminesOutputType` operates by searching for a Pin on the function blueprint node based on the specified name. This Pin must be of Class or Object type (or a container), as these types support pointer conversion. The Pin is used to specify values through various TypePickers, such as `ClassPicker` or `ObjectPicker`, on the blueprint node. Subsequently, based on the value selected by the TypePicker, the type of the parameter (or return parameter) specified by `DynamicOutputParam` can be dynamically adjusted. The actual line that changes the type is:

```
Pin->PinType.PinSubCategoryObject = PickedClass;
```

```
void FDynamicOutputHelper::ConformOutputType() const
{
    if (IsTypePickerPin(MutatingPin))
    {
        UClass* PickedClass = GetPinClass(MutatingPin);
        UK2Node_CallFunction* FuncNode = GetFunctionNode();

        // See if there is any dynamic output pins
        TArray<UEdGraphPin*> DynamicPins;
        GetDynamicOutPins(FuncNode, DynamicPins);

        // Set the pins class
        for (UEdGraphPin* Pin : DynamicPins)
        {
            if (ensure(Pin != nullptr))
            {
                Pin->PinType.PinSubCategoryObject = PickedClass; // Set the subtype for each dynamic parameter
            }
        }
    }
}
```

# DynamicOutputParam

---

- **Function Description:** In conjunction with DeterminesOutputType, define multiple output parameters that support dynamic types.
- **Usage Location:** UFUNCTION
- **Metadata Type:** strings = "a, b, c"
- **Restricted Types:** Class or Object pointer types, or container types
- **Associated Items:** DeterminesOutputType

Frequently used in tandem with DeterminesOutputType. The number of dynamic parameters can be variable.

# ReturnDisplayName

---

- **Function Description:** Modify the name of the function's return value, with the default being ReturnValue
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Commonliness:** ★★★★☆

The default name for the function's return value pin is ReturnValue. If you wish to provide a more descriptive name, you can use ReturnDisplayName to customize it.

## Test Code:

---

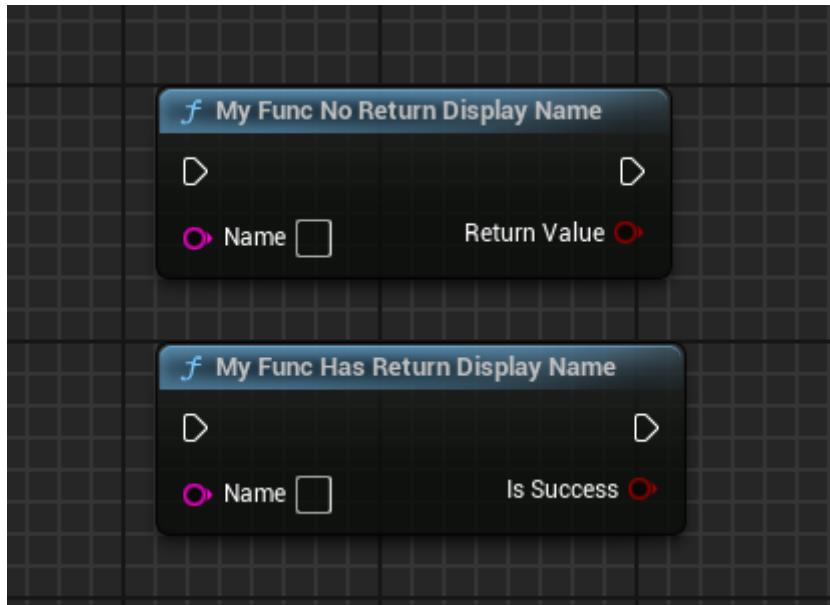
```
UFUNCTION(BlueprintCallable, meta = (ReturnDisplayName = "IsSuccess"))
static bool MyFunc_HasReturnDisplayName(FString Name) { return true; }

UFUNCTION(BlueprintCallable, meta = ())
static bool MyFunc_NoReturnDisplayName(FString Name) { return true; }
```

## Blueprint Effect:

---

The effect can be confirmed by comparing the names of the return values.



## Principle:

The principle is quite straightforward; it involves checking the Meta on the Pin and setting the PinFriendlyName

```
if (Function->GetReturnProperty() == Param && Function-
>HasMetaData(FBlueprintMetadata::MD_ReturnDisplayName))
{
    Pin->PinFriendlyName = Function-
>GetMetaDataText(FBlueprintMetadata::MD_ReturnDisplayName);
}
```

## WorldContext

- **Function Description:** Automatically assigns a parameter of the specified function to receive the WorldContext object to determine the current World of execution
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Related Items:** CallableWithoutWorldContext, ShowWorldContextPin
- **Commonly Used:** ★★★★☆

Automatically assigns a parameter of the specified function to receive the WorldContext object to determine the current World. The function can be either BlueprintCallable or BlueprintPure, and it can be either a static function or a member function. Typically, this is used for static functions in function libraries, such as the numerous static functions in UGameplayStatics.

```

UFUNCTION(BlueprintPure, Category="Game", meta= (WorldContext="WorldContextObject"))
static ENGINE_API class UGameInstance* GetGameInstance(const UObject* WorldContextObject)
{
    Uworld* World = GEngine->GetWorldFromContextObject(WorldContextObject, EGetWorldErrorMode::LogAndReturnNull);
    return World ? World->GetGameInstance() : nullptr;
}

//The general method to obtain the World in Runtime is:
Uworld* World = GEngine->GetWorldFromContextObject(WorldContextObject, EGetWorldErrorMode::ReturnNull);
//The general method to obtain the World in the Editor (e.g., in CallInEditor functions) is:
Uobject* WorldContextObject = EditorEngine->GetEditorWorldContext().World();

```

## Test Code:

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunctionLibrary_WorldContextTest :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    static FString MyFunc_NoWorldContextMeta(const UObject* WorldContextObject, FString name, FString value);

    UFUNCTION(BlueprintCallable, meta = (WorldContext = "WorldContextObject"))
    static FString MyFunc_HasWorldContextMeta(const UObject* WorldContextObject, FString name, FString value);

    UFUNCTION(BlueprintPure)
    static FString MyPure_NoWorldContextMeta(const UObject* WorldContextObject, FString name, FString value);

    UFUNCTION(BlueprintPure, meta = (WorldContext = "WorldContextObject"))
    static FString MyPure_HasWorldContextMeta(const UObject* WorldContextObject, FString name, FString value);
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyObject_NoGetWorld :public UObject
{
    GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyObject_HasGetWorld :public UObject
{
    GENERATED_BODY()
};

```

```

Uworld* WorldPrivate = nullptr;
public:
    UFUNCTION(BlueprintCallable)
    void RegisterwithOuter()
    {
        if (Uobject* outer = Getouter())
        {
            WorldPrivate = outer->Getworld();
        }
    }

    virtual Uworld* Getworld() const override final { return WorldPrivate; }
};

//.cpp

FString UMyFunctionLibrary_WorldContextTest::MyFunc_HasWorldContextMeta(const
Uobject* WorldContextObject, FString name, FString value)
{
    Uworld* World = GEngine->GetworldFromContextObject(WorldContextObject,
EGetworldErrorMode::LogAndReturnNull);
    if (World != nullptr)
    {
        return WorldContextObject->GetName();
    }
    return TEXT("None");
}

FString UMyFunctionLibrary_WorldContextTest::MyFunc_NoWorldContextMeta(const
Uobject* WorldContextObject, FString name, FString value)
{
    return MyFunc_HasWorldContextMeta(WorldContextObject, name, value);
}

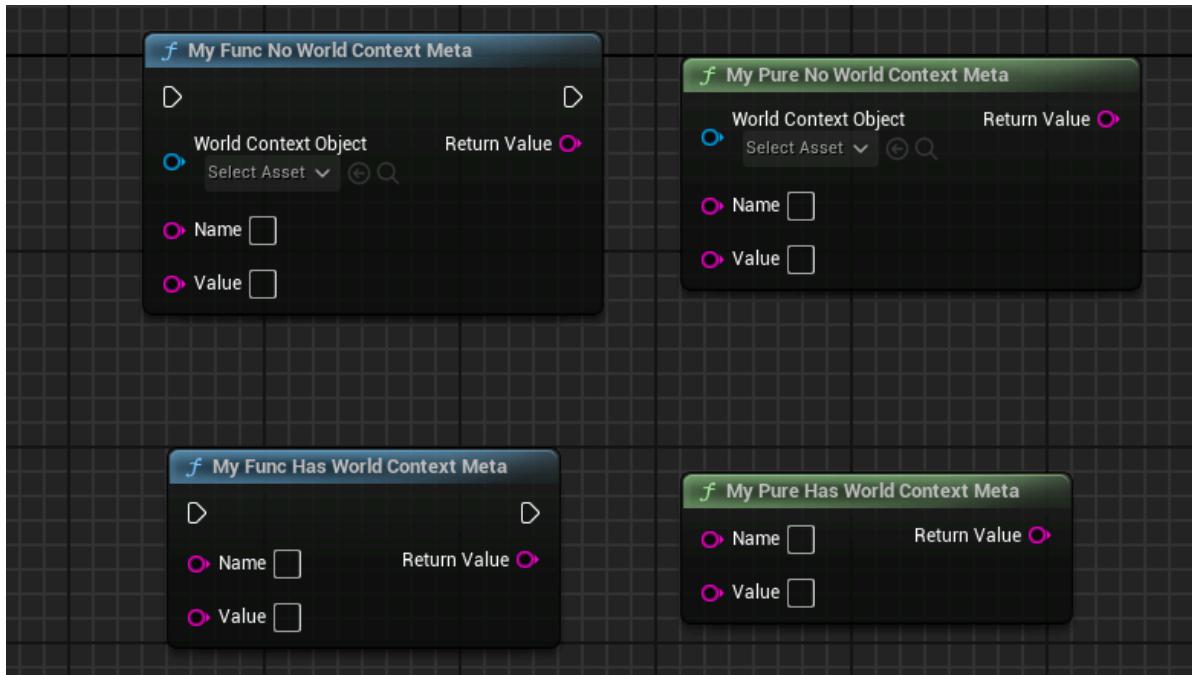
FString UMyFunctionLibrary_WorldContextTest::MyPure_NoWorldContextMeta(const
Uobject* WorldContextObject, FString name, FString value)
{
    return MyFunc_HasWorldContextMeta(WorldContextObject, name, value);
}

FString UMyFunctionLibrary_WorldContextTest::MyPure_HasWorldContextMeta(const
Uobject* WorldContextObject, FString name, FString value)
{
    return MyFunc_HasWorldContextMeta(WorldContextObject, name, value);
}

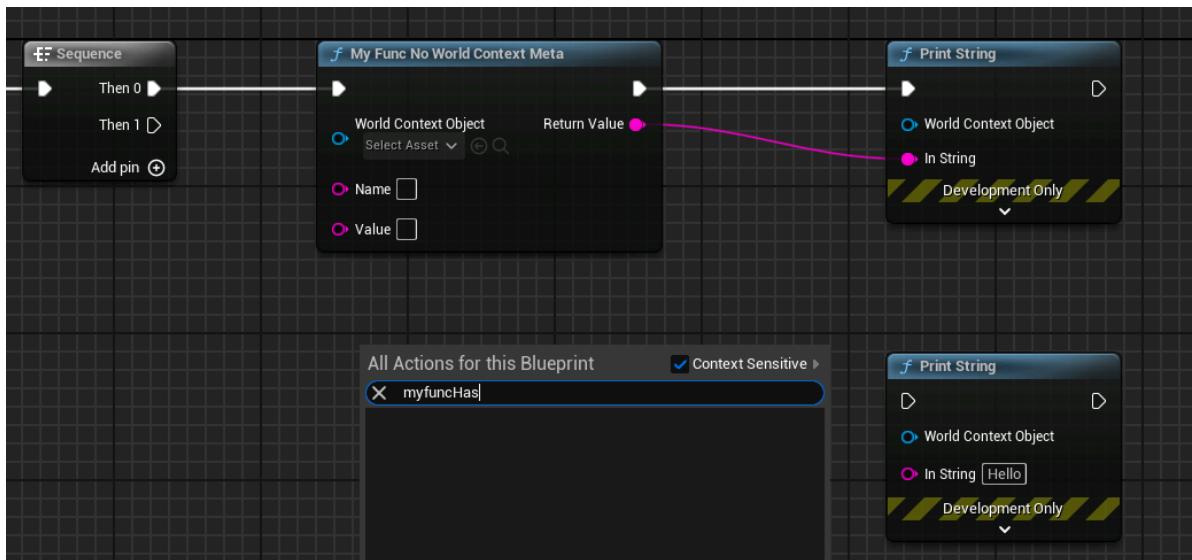
```

## Test Results in Blueprint:

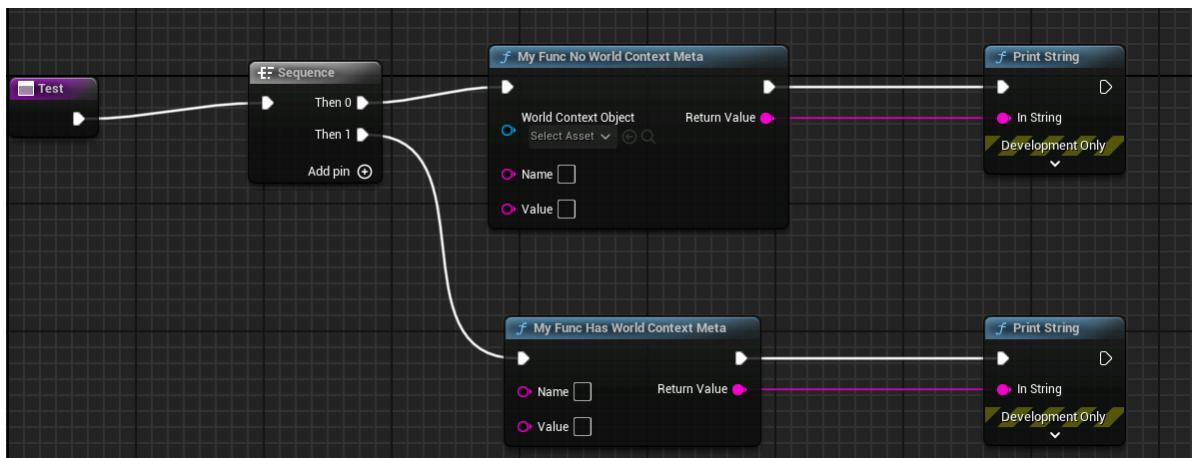
When called within an Actor, you can see that functions without a specified WorldContext will expose the Object parameter, requiring manual specification. Functions with WorldContext, however, hide the WorldContextObject parameter by default, as the WorldContextObject can be automatically assigned in the Actor (its value being the current Actor).



In the subclass of UMyObject\_NoGetWorld, since GetWorld is not implemented, the World cannot be obtained, thus preventing the automatic assignment of the WorldContextObject and the invocation of MyFunc\_HasWorldContextMeta.



In the subclass of UMyObject\_HasGetWorld, because UMyObject\_HasGetWorld implements GetWorld, it allows the calling of MyFunc\_HasWorldContextMeta, with the WorldContextObject value being the UMyObject\_HasGetWorld subclass object, which calls GetWorld() on itself to obtain the previously registered WorldPrivate object.



# Principle:

When a function needs to interact with the World and cannot directly find the World object (usually a static function), an additional parameter must be manually passed from the outside to the function's parameters to trace back to the OuterWorld. This parameter is called `WorldContextObject` and is typically of type `UObject*`, facilitating the passing of various types of objects.

This `WorldContextObject` can be manually passed. It can also be automatically assigned if the Object class implements the virtual `GetWorld()` interface and does not return `nullptr`, allowing for normal acquisition of the World object and interaction with the runtime game world.

In everyday use, most Blueprint objects are already aware of the World they are in, such as Actors, which definitely know which World they belong to.

```
UWorld* AActor::GetWorld() const
{
    // CDO objects do not belong to a world
    // If the actors outer is destroyed or unreachable we are shutting down and
    // the world should be nullptr
    if (!HasAnyFlags(RF_ClassDefaultObject) && ensureMsgf(GetOuter(),
        TEXT("Actor: %s has a null OuterPrivate in AActor::GetWorld()"), *GetFullName())
        && !GetOuter()->HasAnyFlags(RF_BeginDestroyed) && !GetOuter()-
        >IsUnreachable())
    {
        if (ULevel* Level = GetLevel())
        {
            return Level->OwningWorld;
        }
    }
    return nullptr;
}
```

Within an Actor, calling a static function would require manually setting the `WorldContextObject` each time, which is cumbersome and redundant. Therefore, the `WorldContext` meta instructs the Blueprint system to automatically assign the value of our `WorldContextObject`, which is the Actor itself, eliminating the need for manual parameter passing and hiding this functionally irrelevant glue parameter, resulting in a cleaner and more elegant appearance.

And if you call a function inside an ordinary Object object, you don't know which World it belongs to. At this time, the Object class needs to implement `GetWorld()`. Use the `bGetWorldOverridden` variable in the editor to determine whether a `UObject` subclass has overridden `GetWorld`. If the subclass has overriding, during detection, just call `ImplementsGetWorld` on CDO to get the result of `bGetWorldOverridden==true`, which allows the function version of `WorldContextObject` to be automatically specified to be called. To mention a little more, `bGetWorldOverridden` is not a `UObject` member variable. It is only a temporary variable used when `ImplementsGetWorld()` is called, so it does not need to be saved.

```
#if DO_CHECK || WITH_EDITOR
// Used to check to see if a derived class actually implemented GetWorld() or not
thread_local bool bGetWorldOverridden = false;
#endif // #if DO_CHECK || WITH_EDITOR
```

```

class Uworld* Uobject::Getworld() const
{
    if (Uobject* outer = GetOuter())
    {
        return outer->Getworld();
    }

#if DO_CHECK || WITH_EDITOR
    bGetworldoverridden = false;
#endif
    return nullptr;
}

#if WITH_EDITOR

bool Uobject::ImplementsGetworld() const
{
    bGetworldoverridden = true;
    Getworld();
    return bGetworldoverridden;
}

#endif // #if WITH_EDITOR

```

Furthermore, static Blueprint functions (functions in BPTYPE\_FunctionLibrary or marked as FUNC\_Static) have a hidden parameter "**WorldContext**". When **UK2Node\_CallFunction** is expanded, the value on "WorldContext" is assigned to the parameters specified by the DefaultToSelf or WorldContext meta tags, thus automatically assigning Self to DefaultToSelf and WorldContext.

```

void UK2Node_CallFunction::ExpandNode(class FKismetCompilerContext&
CompilerContext, UEdGraph* SourceGraph)

else if (UEdGraphPin* BetterSelfPin = EntryPoints[0]->GetAutoWorldContextPin())
{
    const FString& DefaultToSelfMetaValue = Function-
>GetMetaData(FBlueprintMetadata::MD_DefaultToSelf);
    const FString& WorldContextMetaValue = Function-
>GetMetaData(FBlueprintMetadata::MD_WorldContext);

    struct FStructConnectHelper
    {
        static void Connect(const FString& PinName, UK2Node* Node, UEdGraphPin*
BetterSelf, const UEdGraphSchema_K2* InSchema, FCompilerResultsLog& MessageLog)
        {
            UEdGraphPin* Pin = Node->FindPin(PinName);
            if (!PinName.IsEmpty() && Pin && !Pin->LinkedTo.Num())
            {
                const bool bConnected = InSchema->TryCreateConnection(Pin,
BetterSelf);
                if (!bConnected)
                {
                    MessageLog.Warning(*LOCTEXT("DefaultToSelfNotConnected",
"DefaultToSelf pin @@ from node @@ cannot be connected to @@").ToString(), Pin,
Node, BetterSelf);
                }
            }
        }
    };
}

```

```

        }
    }
}

};

FStructConnectHelper::Connect(DefaultToSelfMetaValue, this, BetterSelfPin,
Schema, CompilerContext.MessageLog);
if (!Function-
>HasMetaData(FBlueprintMetadata::MD_CallablewithoutworldContext))
{
    FStructConnectHelper::Connect(WorldContextMetaValue, this, BetterSelfPin,
Schema, CompilerContext.MessageLog);
}
}

```

## ShowWorldContextPin

- **Function Description:** Placed on UCLASS, this attribute ensures that function calls within this class must display the WorldContext pin, irrespective of its default hidden state
- **Usage Location:** UCLASS
- **Metadata Type:** bool
- **Associated Item:** WorldContext

Placed on UCLASS, this attribute specifies that function calls within this class must display the WorldContext pin, regardless of whether it is normally hidden by default. This is because this Object class cannot be used as a WorldContextObject, even if the GetWorld() method is implemented; it must be treated as if it cannot automatically obtain the WorldContext, thus requiring the user to manually specify a WorldContextObject.

Generally placed on UObject, but it has been found in the source code on AGameplayCueNotify\_Actor and AEditorUtilityActor as well. AEditorUtilityActor does not operate at runtime and therefore lacks a World. AGameplayCueNotify\_Actor might be used and recycled on a CDO, so it cannot be assumed that a WorldContext is always present.

## Test Code:

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunctionLibrary_WorldContextTest : public
UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    static FString MyFunc_NoWorldContextMeta(const UObject* WorldContextObject,
FString name, FString value);

    UFUNCTION(BlueprintCallable, meta = (WorldContext = "WorldContextObject"))
    static FString MyFunc_HasWorldContextMeta(const UObject* WorldContextObject,
FString name, FString value);
};

UCLASS(Blueprintable, BlueprintType, meta = (ShowWorldContextPin = "true"))
class INSIDER_API UMyObject_ShowWorldContextPin : public UObject

```

```

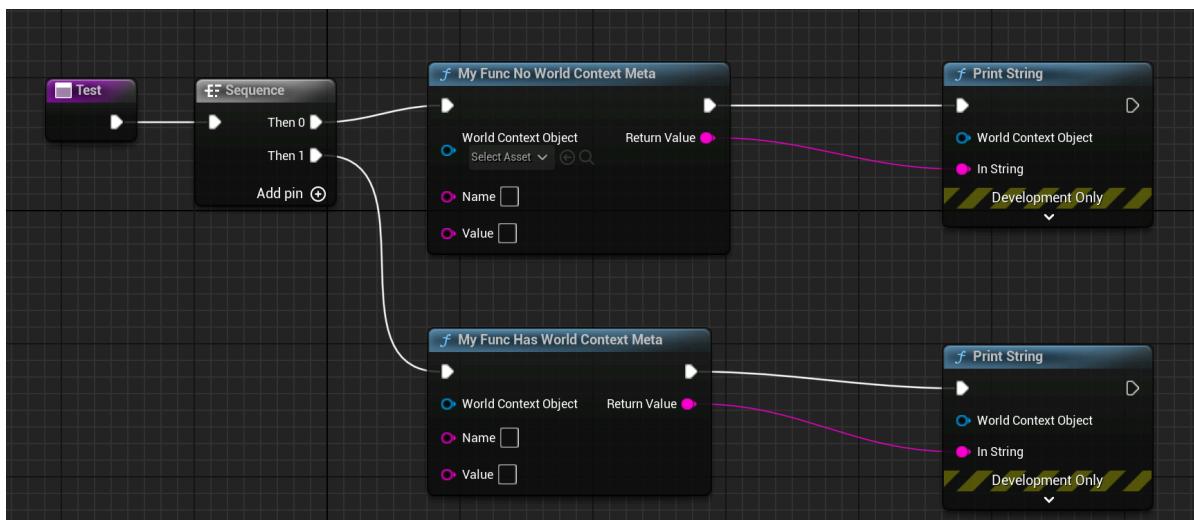
{
    GENERATED_BODY()
    Uworld* WorldPrivate = nullptr;
public:
    UFUNCTION(BlueprintCallable)
    void Registerwithouter()
    {
        if (Uobject* outer = Getouter())
        {
            WorldPrivate = outer->GetWorld();
        }
    }

    virtual Uworld* Getworld() const override final { return WorldPrivate; }
};

```

## Blueprint Test Effect:

It can be observed that although the UMyObject\_ShowWorldContextPin class implements the GetWorld() method, the WorldContextObject, which should normally be automatically assigned and remain hidden in MyFunc\_HasWorldContextMeta, is explicitly displayed in this class. Additionally, note that the WorldContextObject is also output in the PrintString.



## Principle:

In the CallFunction blueprint node, if bShowWorldContextPin is set, the function parameters specified by WorldContextMetaValue or DefaultToSelfMetaValue will not be hidden.

```

bool UK2Node_CallFunction::CreatePinsForFunctionCall(const UFunction* Function)
{
    const bool bShowWorldContextPin = ((PinsToHide.Num() > 0) && BP->ParentClass->HasMetaDataHierarchical(FBlueprintMetadata::MD_ShowWorldContextPin));
    //...
    if (PinsToHide.Contains(Pin->PinName))
    {
        const FString PinNameStr = Pin->PinName.ToString();
        const FString& DefaultToSelfMetaValue = Function->GetMetaData(FBlueprintMetadata::MD_DefaultToSelf);

```

```

        const FString& WorldContextMetavalue = Function-
>GetMetaData(FBlueprintMetadata::MD_WorldContext);
    bool bIsSelfPin = ((PinNameStr == DefaultToSelfMetavalue) || (PinNameStr
== WorldContextMetavalue));

    if (!bShowWorldContextPin || !bIsSelfPin)
    {
        Pin->bHidden = true;
        Pin->bNotConnectable = InternalPins.Contains(Pin->PinName);
    }
}

}

```

## CallableWithoutWorldContext

- **Function Description:** Allows a function to be used without depending on a WorldContextObject
- **Usage Location:** UFUNCTION
- **Metadata Type:** bool
- **Associated Items:** WorldContext

Enables the function to be used without a WorldContextObject.

The CallableWithoutWorldContext attribute is used in conjunction with WorldContext or DefaultToSelf. These options require that a function be provided with an external WorldContext object to be invoked. Consequently, such functions cannot be called in object subclasses that lack the implementation of GetWorld. However, there are instances where certain functions do not necessarily need a WorldContextObject to operate, such as those in PrintString or VisualLogger.

## Test Code:

```

UFUNCTION(BlueprintPure, meta = (WorldContext = "WorldContextObject"))
static FString MyFunc_HasWorldContextMeta(const Uobject* WorldContextObject,
FString name, FString value);

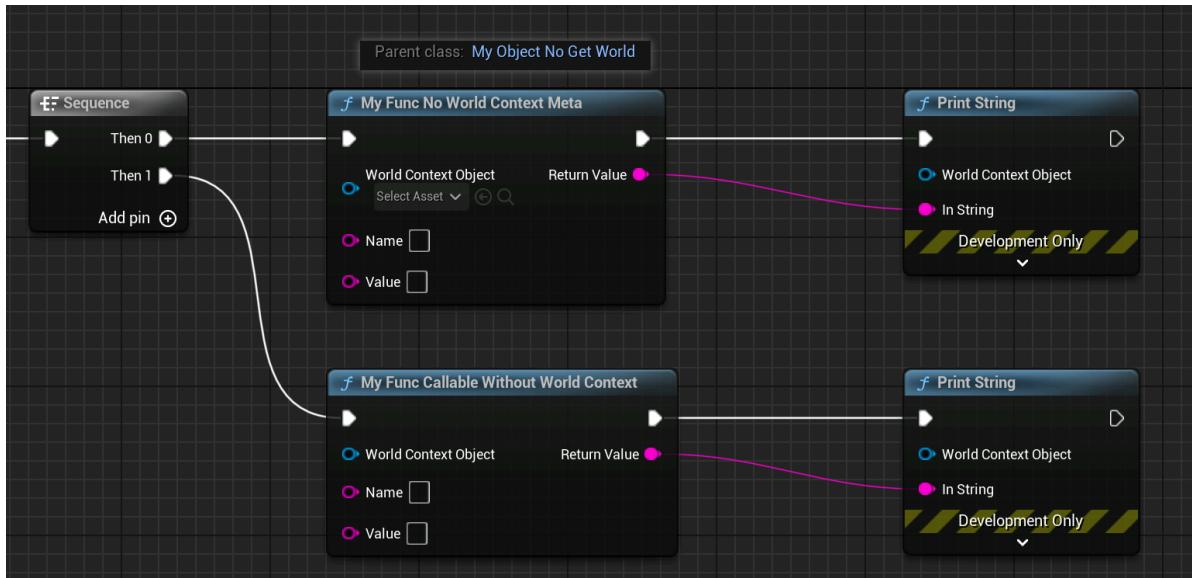
UFUNCTION(BlueprintCallable, meta = (WorldContext =
"WorldContextObject", CallablewithoutWorldContext))
static FString MyFunc_CallablewithoutWorldContext(const Uobject*
WorldContextObject, FString name, FString value);

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyobject_NoGetWorld :public Uobject
{
    GENERATED_BODY()
};

```

## Blueprint Test Effect:

In a subclass of UMyObject\_NoGetWorld, MyFunc\_HasWorldContextMeta cannot be invoked because its enclosing class is required to supply a WorldContextObject. On the other hand, MyFunc\_CallableWithoutWorldContext can be called and does not require the provision of a WorldContextObject.



Typical applications within the source code include:

```
UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject",
CallablewithoutworldContext, Keywords = "log print", AdvancedDisplay = "2",
DevelopmentOnly), Category="Development")
static ENGINE_API void PrintString(const UObject* WorldContextObject, const
FString& InString = FString(TEXT("Hello")), bool bPrintToScreen = true, bool
bPrintToLog = true, FLinearColor TextColor = FLinearColor(0.0f, 0.66f, 1.0f),
float Duration = 2.f, const FName Key = NAME_None);
```

## AutoCreateRefTerm

- **Function Description:** Automatically assigns default values to multiple input reference parameters of a specified function when they are not connected
- **Usage Location:** UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** strings = "a, b, c"
- **Commonly Used:** ★★★★☆

The specified function's multiple input reference parameters will automatically have default values created for them when there is no connection.

Note the two prerequisites: "input" and "reference."

In situations where you want to pass function parameters by reference but do not wish to connect a variable each time, and you want to provide an inline default value when not connected, the Blueprint system offers this convenient feature.

## Test Code:

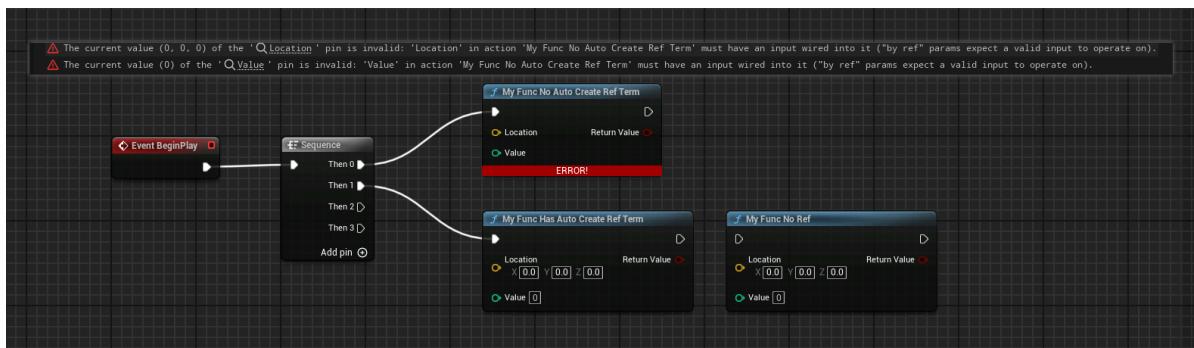
```
UFUNCTION(BlueprintCallable, meta = (AutoCreateRefTerm = "Location,Value"))
static bool MyFunc_HasAutoCreateRefTerm(const FVector& Location, const int32&
value) { return false; }

UFUNCTION(BlueprintCallable)
static bool MyFunc_NoAutoCreateRefTerm(const FVector& Location, const int32&
value) { return false; }

UFUNCTION(BlueprintCallable)
static bool MyFunc_NoRef(FVector Location, int32 value) { return false; }
```

## Blueprint Effect:

You can see that the function MyFunc\_NoAutoCreateRefTerm will result in a compilation error because it has reference parameters that are not connected, leading to missing actual parameters for the reference.



## Principle Code:

```
void UEdGraphSchema_K2::GetAutoEmitTermParameters(const UFunction* Function,
TArray< FString>& AutoEmitParameterNames)
{
    AutoEmitParameterNames.Reset();

    const FString& MetaData = Function->GetMetaData(FBlueprintMetadata::MD_AutoCreateRefTerm);
    if (!MetaData.IsEmpty())
    {
        MetaData.ParseIntoArray(AutoEmitParameterNames, TEXT(","), true);

        for (int32 NameIndex = 0; NameIndex < AutoEmitParameterNames.Num();)
        {
            FString& ParameterName = AutoEmitParameterNames[NameIndex];
            ParameterName.TrimStartAndEndInline();
            if (ParameterName.IsEmpty())
            {
                AutoEmitParameterNames.RemoveAtSwap(NameIndex);
            }
            else
            {
                ++NameIndex;
            }
        }
    }
}
```

```

        }

    }

    // Allow any params that are blueprint defined to be autocreated:
    if (!FBBlueprintEditorUtils::IsNativeSignature(Function))
    {
        for (TFieldIterator<FProperty> ParamIter(Function,
EFieldIterationFlags::Default);
            ParamIter && (ParamIter->PropertyFlags & CPF_Parm);
            ++ParamIter)
        {
            FProperty* Param = *ParamIter;
            if(Param->HasAnyPropertyFlags(CPF_ReferenceParm))
            {
                AutoEmitParameterNames.Add(Param->GetName());
            }
        }
    }
}

```

还有在

```

void UK2Node_CallFunction::ExpandNode(class FKismetCompilerContext&
CompilerContext, UEdGraph* SourceGraph)
{
    if (Function)
    {
        TArray< FString> AutoCreateRefTermPinNames;
        CompilerContext.GetSchema()->GetAutoEmitTermParameters(Function,
AutoCreateRefTermPinNames);
        const bool bHasAutoCreateRefTerms = AutoCreateRefTermPinNames.Num() != 0;

        for (UEdGraphPin* Pin : Pins)
        {
            const bool bIsRefInputParam = Pin && Pin->PinType.bIsReference && (Pin-
>Direction == EGPD_Input) && !CompilerContext.GetSchema()->IsMetaPin(*Pin);
            if (!bIsRefInputParam)
            {
                continue;
            }

            const bool bHasConnections = Pin->LinkedTo.Num() > 0;
            const bool bCreateDefaultValRefTerm = bHasAutoCreateRefTerms &&
                !bHasConnections && AutoCreateRefTermPinNames.Contains(Pin-
>PinName.ToString());

            if (bCreateDefaultValRefTerm)
            {
                const bool bHasDefaultValue = !Pin->DefaultValue.IsEmpty() || Pin-
>DefaultObject || !Pin->DefaultTextValue.IsEmpty();

                // copy defaults as default values can be reset when the pin is
connected
                const FString DefaultValue = Pin->DefaultValue;
                Uobject* DefaultObject = Pin->DefaultObject;
                const FText DefaultTextValue = Pin->DefaultTextValue;
                bool bMatchesDefaults = Pin->DoesDefaultValueMatchAutogenerated();
            }
        }
    }
}

```

```

UEdGraphPin* ValuePin = InnerHandleAutoCreateRef(this, Pin,
CompilerContext, SourceGraph, bHasDefaultValue);
    if (ValuePin)
    {
        if (bMatchesDefaults)
        {
            // Use the latest code to set default value
            Schema->SetPinAutogeneratedDefaultValueBasedOnType(ValuePin);
        }
        else
        {
            ValuePin->DefaultValue = DefaultValue;
            ValuePin->DefaultObject = DefaultObject;
            ValuePin->DefaultTextValue = DefaultTextValue;
        }
    }
}

// since EX_Self does not produce an addressable (referenceable)
FProperty, we need to shim
    // in a "auto-ref" term in its place (this emulates how UHT generates a
local value for
    // native functions; hence the IsNative() check)
    else if (bHasConnections && Pin->LinkedTo[0]->PinType.PinSubCategory ==
UEdGraphSchema_K2::PSC_Self && Pin->PinType.bIsConst && !Function->IsNative())
    {
        InnerHandleAutoCreateRef(this, Pin, CompilerContext, SourceGraph,
/*bForceAssignment =*/true);
    }
}
}
}

```

## ProhibitedInterfaces

- **Function Description:** Lists interfaces incompatible with blueprint classes, thereby preventing their implementation
- **Usage Location:** UCLASS
- **Engine Module:** Blueprint
- **Metadata Type:** strings = "a, b, c"
- **Commonality:** ★★

## Test Code:

```

UINTERFACE(Blueprintable,MinimalAPI)
class UMyInterface_First:public UInterface
{
    GENERATED_UINTERFACE_BODY()
};

class INSIDER_API IMyInterface_First
{

```

```

GENERATED_IINTERFACE_BODY()
public:
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
        void FirstFunc() const;
};

UINTERFACE(Blueprintable,MinimalAPI)
class UMyInterface_Second:public UIInterface
{
    GENERATED_UINTERFACE_BODY()
};

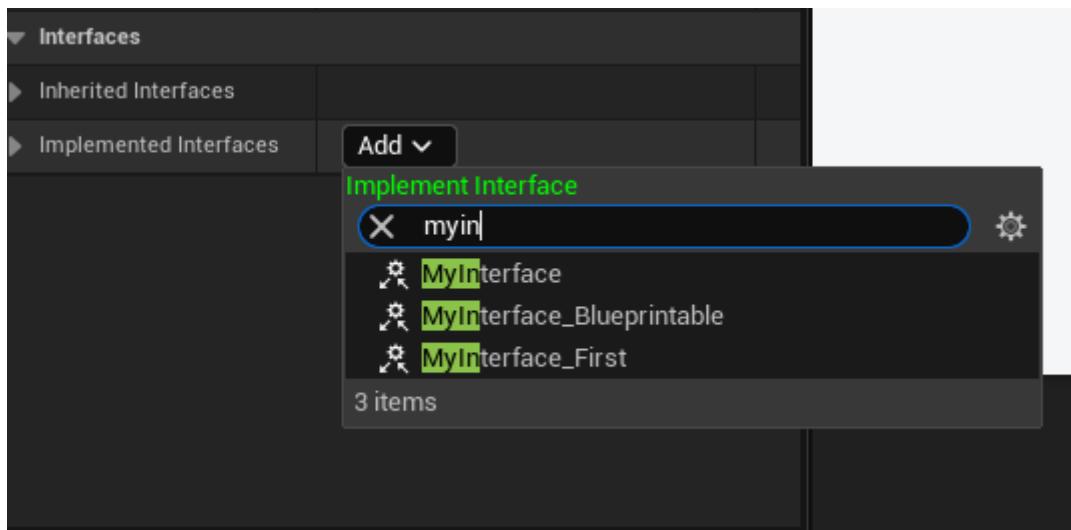
class INSIDER_API IMyInterface_Second
{
    GENERATED_IINTERFACE_BODY()
public:
    UFUNCTION(BlueprintCallable, BlueprintImplementableEvent)
        void SecondFunc() const;
};

UCLASS(Blueprintable,meta=(ProhibitedInterfaces="UMyInterface_Second"))
class INSIDER_API UMyClass_ProhibitedInterfaces :public UObject
{
    GENERATED_BODY()
public:
};

```

## Test Results:

Discovered that UMyInterface\_Second is prevented from being implemented, while UMyInterface\_First remains implementable



## Principle Code:

As seen in the construction of the list, filtering is applied. Additionally, the use of .RightChop(1) is noted; hence, the interface name provided must include the 'U' prefix. If UMyInterface\_Second

```

TSharedRef<SWidget>
FBlueprintEditorUtils::ConstructBlueprintInterfaceClassPicker( const TArray<
    UBlueprint*>& Blueprints, const FOnClassPicked& OnPicked)

```

```

{
//...

    UClass const* const ParentClass = Blueprint->ParentClass;
    // see if the parent class has any prohibited interfaces
    if ((ParentClass != nullptr) && ParentClass-
>HasMetaData(FBlueprintMetadata::MD_ProhibitedInterfaces))
    {
        FString const& ProhibitedList = Blueprint->ParentClass-
>GetMetaData(FBlueprintMetadata::MD_ProhibitedInterfaces);

        TArray< FString> ProhibitedInterfaceNames;
        ProhibitedList.ParseIntoArray(ProhibitedInterfaceNames, TEXT(","),
true);

        // Loop over all the prohibited interfaces
        for (int32 ExclusionIndex = 0; ExclusionIndex <
ProhibitedInterfaceNames.Num(); ++ExclusionIndex)
        {
            ProhibitedInterfaceNames[ExclusionIndex].TrimStartInline();
            FString const& ProhibitedInterfaceName =
ProhibitedInterfaceNames[ExclusionIndex].RightChop(1);
            UClass* ProhibitedInterface = UClass::TryFindTypeSlow<UClass>
(ProhibitedInterfaceName);
            if(ProhibitedInterface)
            {
                Filter->DisallowedClasses.Add(ProhibitedInterface);
                Filter->DisallowedChildrenOfClasses.Add(ProhibitedInterface);
            }
        }
    }

    // Do not allow adding interfaces that are already added to the Blueprint
    TArray< UClass*> InterfaceClasses;
    FindImplementedInterfaces(Blueprint, true, InterfaceClasses);
    for(UClass* InterfaceClass : InterfaceClasses)
    {
        Filter->DisallowedClasses.Add(InterfaceClass);
    }

    // Include a class viewer filter for imported namespaces if the class
    // picker is being hosted in an editor context
    TSharedPtr< IToolkit > AssetEditor =
FToolkitManager::Get().FindEditorForAsset(Blueprint);
    if (AssetEditor.IsValid() && AssetEditor->IsBlueprintEditor())
    {
        TSharedPtr< IBlueprintEditor > BlueprintEditor =
StaticCastSharedPtr< IBlueprintEditor >(AssetEditor);
        TSharedPtr< IClassViewerFilter > ImportedClassViewerFilter =
BlueprintEditor->GetImportedClassViewerFilter();
        if (ImportedClassViewerFilter.IsValid())
        {

Options.ClassFilters.AddUnique(ImportedClassViewerFilter.ToshradRef());
        }
    }
}

```

```

}

// never allow parenting to children of itself
for (UClass* BPClass : BlueprintClasses)
{
    Filter->DisallowedChildrenOfClasses.Add(BPClass);
}

return FModuleManager::LoadModuleChecked<FCClassViewerModule>
("ClassViewer").CreateClassViewer(Options, OnPicked);
}

```

## HiddenNode

---

- **Function Description:** Hides the specified UBTNode so that it does not appear in the context menu when right-clicked.
- **Usage Location:** UClass
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Restriction Type:** UBTNode
- **Commonality:** ★

Hides the specified UBTNode from appearing in the right-click menu.

## Test Code:

---

```

UCLASS(MinimalAPI, meta = ())
class UMyBT_NotHiddenNode : public UBTDecorator
{
    GENERATED_UCLASS_BODY()

    UPROPERTY(Category = Node, EditAnywhere)
    float MyFloat;
};

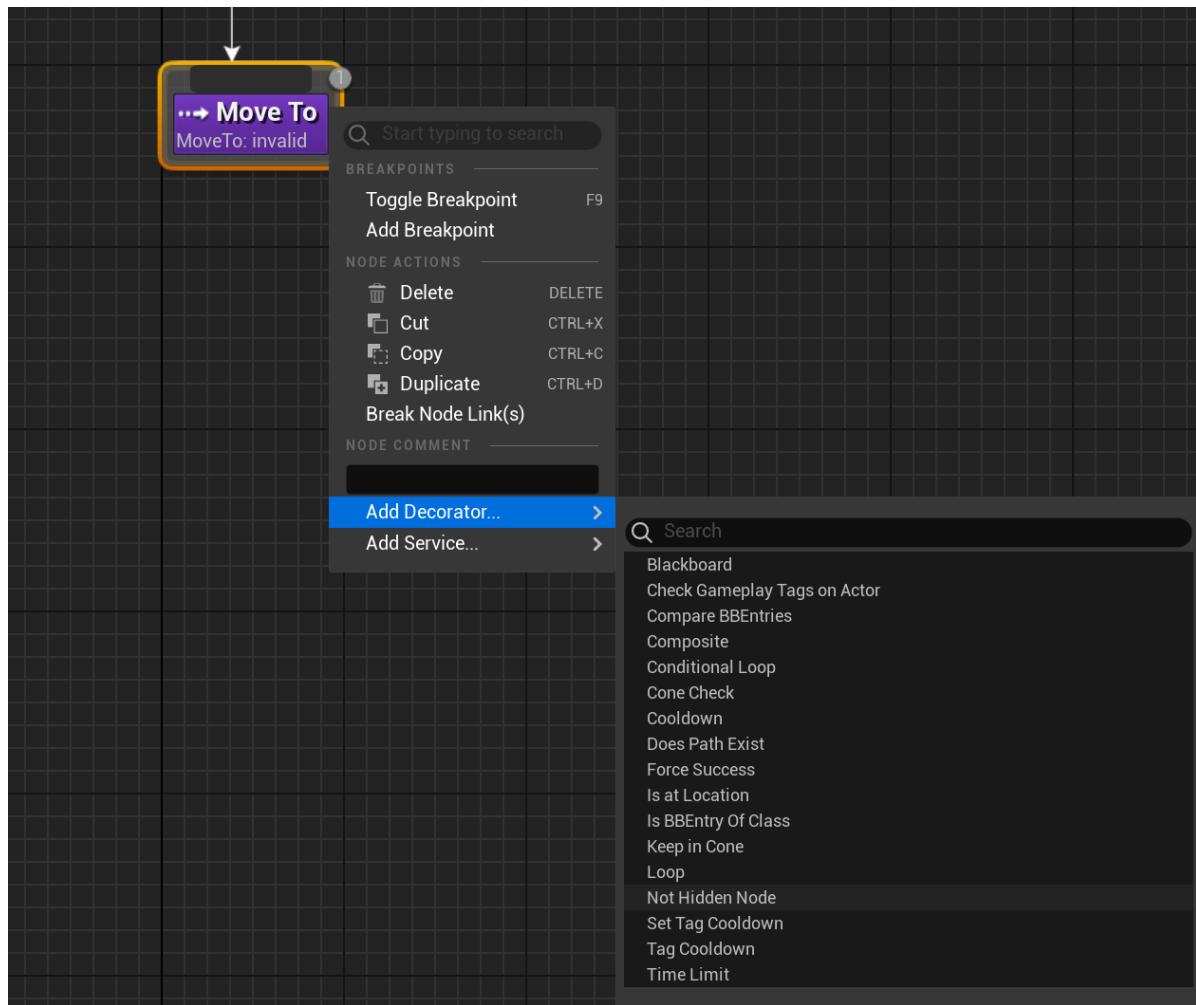
UCLASS(MinimalAPI, meta = (HiddenNode))
class UMyBT_HiddenNode : public UBTDecorator
{
    GENERATED_UCLASS_BODY()

    UPROPERTY(Category = Node, EditAnywhere)
    float MyFloat;
};

```

# Test Results:

Only UMyBT\_NotHiddenNode is visible, while UMyBT\_HiddenNode is hidden from view.



# Principle:

The principle is quite straightforward; it involves marking the metadata and then setting the `bIsHidden` property.

```
bool FGraphNodeClassHelper::IsHidingClass(UClass* Class)
{
    static FName MetaHideInEditor = TEXT("HiddenNode");

    return
        Class &&
        ((Class->HasAnyClassFlags(CLASS_Native) && Class-
>HasMetaData(MetaHideInEditor))
         || ForcedHiddenClasses.Contains(Class));
}

//D:\github\UnrealEngine\Engine\Source\Editor\AIGraph\Private\AIGraphTypes.cpp
void FGraphNodeClassHelper::BuildClassGraph()
{
    for (TObjectIterator<UClass> It; It; ++It)
    {
        UClass* TestClass = *It;
        if (TestClass->HasAnyClassFlags(CLASS_Native) && TestClass-
>IsChildOf(RootNodeClass))
```

```

    {

        NewData.bIsHidden = IsHidingClass(TestClass);

        NewNode->Data = NewData;

        if (TestClass == RootNodeClass)
        {
            RootNode = NewNode;
        }

        NodeList.Add(NewNode);
    }
}

```

## HideFunctions

- **Function Description:** Hides all functions within a specified category from display in the property inspector.
- **Usage Location:** UCLASS
- **Engine Module:** Blueprint
- **Metadata Type:** strings = "a, b, c"
- **Associated Items:**  
UCLASS: HideFunctions, ShowFunctions
- **Commonality:** ★★★

## ExposedAsyncProxy

- **Function Description:** Expose a proxy object of this class within the Async Task node.
- **Usage Location:** UCLASS
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Restriction Type:** Async Blueprint node
- **Associated Items:** HideSpawnParms, HasDedicatedAsyncNode, HideThen
- **Commonly Used:** ★★★

Utilized in UK2Node\_BaseAsyncTask to reveal an asynchronous object pin for the blueprint's async node, enabling further manipulation of this asynchronous behavior.

Within the source code, it is used in two contexts: as a subclass of UBlueprintAsyncActionBase and as a subclass of UGameplayTask, each accompanied by separate UK2Node\_BaseAsyncTask and UK2Node\_LatentGameplayTaskCall to parse the class's declaration and definition, and to wrap and generate the corresponding asynchronous blueprint node.

The base classes are all derived from UBlueprintAsyncActionBase. The ExposedAsyncProxy is used to specify the name of the asynchronous task object. By returning the asynchronous object on the async blueprint node, it allows for the option to cancel the asynchronous operation at a later stage.

## Test Code:

UCancellableAsyncAction is a convenient subclass provided by the engine, inheriting from UBlueprintAsyncActionBase. UMyFunction\_Async defines a blueprint async node called DelayLoop.

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FDelayOutputPin);
UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy = MyAsyncObject))
class INSIDER_API UMyFunction_Async :public UBlueprintAsyncActionBase
{
public:
    GENERATED_BODY()

public:
    UPROPERTY(BlueprintAssignable)
    FDelayOutputPin Loop;

    UPROPERTY(BlueprintAssignable)
    FDelayOutputPin Complete;

    UFUNCTION(BlueprintCallable, meta = (BlueprintInternalUseOnly = "true",
    worldContext = "worldContextObject"), Category = "Flow Control")
        static UMyFunction_Async* DelayLoop(const UObject* WorldContextObject, const
    float DelayInSeconds, const int Iterations);

    virtual void Activate() override;

    UFUNCTION()
    static void Test();

private:
    const UObject* WorldContextObject = nullptr;
    float MyDelay = 0.f;
    int MyIterations = 0;
    bool Active = false;

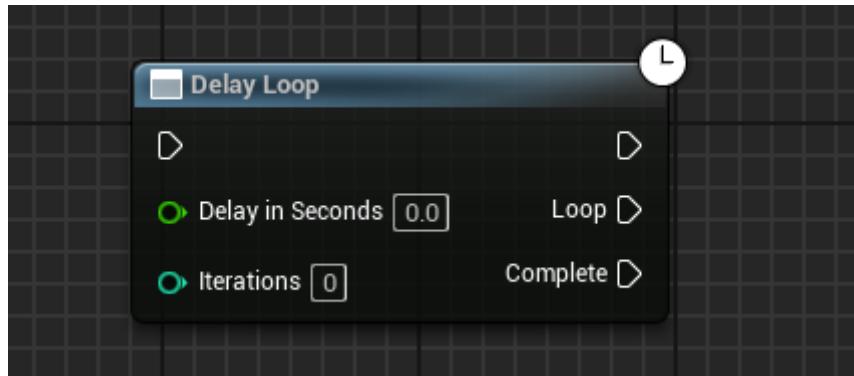
    UFUNCTION()
    void ExecuteLoop();

    UFUNCTION()
    void ExecuteComplete();
};

};
```

## Default blueprint nodes are:

If UMyFunction\_Async inherits directly from UBlueprintAsyncActionBase without setting ExposedAsyncProxy, the resulting blueprint async node will look as shown in the following diagram.

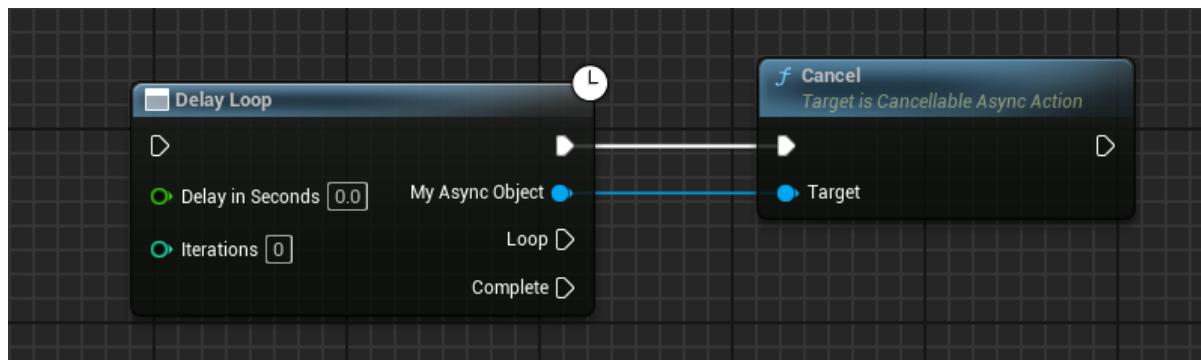


However, if inheriting from UCancellableAsyncAction (which provides a Cancel method) and setting ExposedAsyncProxy to the desired AsyncObject pin name.

```
UCLASS(Abstract, BlueprintType, meta = (ExposedAsyncProxy = AsyncAction),
MinimalAPI)
class UCancellableAsyncAction : public UBlueprintAsyncActionBase
{
    UFUNCTION(BlueprintCallable, Category = "Async Action")
    ENGINE_API virtual void Cancel();
}

DECLARE_DYNAMIC_MULTICAST_DELEGATE(FDelayOutputPin);
UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy = MyAsyncObject))
class INSIDER_API UMyFunction_Async :public UCancellableAsyncAction
{}
```

**The effect after modification is as shown in the following diagram:**



**Location of this Meta in the source code:**

```
void UK2Node_BaseAsyncTask::AllocateDefaultPins()
{
    bool bExposeProxy = false;
    bool bHideThen = false;
    FText ExposeProxyDisplayName;
    for (const UStruct* TestStruct = ProxyClass; TestStruct; TestStruct =
TestStruct->GetSuperStruct())
    {
        bExposeProxy |= TestStruct->HasMetaData(TEXT("ExposedAsyncProxy"));
        bHideThen |= TestStruct->HasMetaData(TEXT("HideThen"));
        if (ExposeProxyDisplayName.IsEmpty())
        {
```

```

        ExposeProxyDisplayName = TestStruct-
>GetMetaDataText(TEXT("ExposedAsyncProxy"));
    }

    if (bExposeProxy)
    {
        UEdGraphPin* ProxyPin = CreatePin(EGPD_Output,
UEdGraphSchema_K2::PC_Object, ProxyClass,
FBaseAsyncTaskHelper::GetAsyncTaskProxyName());
        if (!ExposeProxyDisplayName.IsEmpty())
        {
            ProxyPin->PinFriendlyName = ExposeProxyDisplayName;
        }
    }

}

```

## HasDedicatedAsyncNode

- **Usage Location:** UCLASS
- **Metadata Type:** bool
- **Associated Items:** ExposedAsyncProxy

Hide the blueprint asynchronous node automatically generated by the factory method in the UBlueprintAsyncActionBase subclass, allowing for manual creation of a corresponding UK2Node\_XXX.

```

/**
 * BlueprintCallable factory functions for classes which inherit from
UBlueprintAsyncActionBase will have a special blueprint node created for it:
UK2Node_AsyncAction
* You can stop this node spawning and create a more specific one by adding the
UCLASS metadata "HasDedicatedAsyncNode"
*/

```

```

UCLASS(MinimalAPI)
class UBlueprintAsyncActionBase : public UObject
{}

```

## Test Code:

```

UCLASS(Blueprintable, BlueprintType,meta = (ExposedAsyncProxy =
MyAsyncObject,HasDedicatedAsyncNode))
class INSIDER_API UMyFunction_Async :public UCancellableAsyncAction
{};

//You can customize a K2Node
UCLASS()
class INSIDER_API UK2Node_MyFunctionAsyncAction : public UK2Node_AsyncAction
{
    GENERATED_BODY()
}

```

```

// UK2Node interface
    virtual void GetMenuActions(FBlueprintActionDatabaseRegistrar&
ActionRegistrar) const override;
    virtual void AllocateDefaultPins() override;
// End of UK2Node interface

protected:
    virtual bool HandleDelegates(
        const TArray<FBaseAsyncTaskHelper::FOutputPinAndLocalVariable>&
VariableOutputs, UEdGraphPin* ProxyObjectPin,
        UEdGraphPin*& InOutLastThenPin, UEdGraph* SourceGraph,
FKismetCompilerContext& CompilerContext) override;
};

void
UK2Node_MyFunctionAsyncAction::GetMenuActions(FBlueprintActionDatabaseRegistrar&
ActionRegistrar) const
{
    struct GetMenuActions_Utils
    {
        static void SetNodeFunc(UEdGraphNode* NewNode, bool /*bIsTemplateNode*/,
TweakObjectPtr<UFunction> FunctionPtr)
        {
            UK2Node_MyFunctionAsyncAction* AsyncTaskNode =
CastChecked<UK2Node_MyFunctionAsyncAction>(NewNode);
            if (FunctionPtr.IsValid())
            {
                UFunction* Func = FunctionPtr.Get();
                FObjectProperty* ReturnProp = CastFieldChecked<FObjectProperty>
(Func->GetReturnProperty());

                AsyncTaskNode->ProxyFactoryFunctionName = Func->GetFName();
                AsyncTaskNode->ProxyFactoryClass = Func->GetOuterUClass();
                AsyncTaskNode->ProxyClass = ReturnProp->PropertyClass;
                AsyncTaskNode->NodeComment = TEXT("This is MyCustomK2Node");
            }
        }
    };
}

UClass* NodeClass = UClass();
ActionRegistrar.RegisterClassFactoryActions<UMyFunction_Async>
(FBlueprintActionDatabaseRegistrar::FMakeFuncSpawnerDelegate::CreateLambda([NodeC
lass](const UFunction* FactoryFunc)->UBlueprintNodeSpawner*
{
    UBlueprintNodeSpawner* NodeSpawner =
UBlueprintFunctionNodeSpawner::Create(FactoryFunc);
    check(NodeSpawner != nullptr);
    NodeSpawner->NodeClass = NodeClass;

    TweakObjectPtr<UFunction> FunctionPtr =
MakeWeakObjectPtr(const_cast<UFunction*>(FactoryFunc));
    NodeSpawner->CustomizeNodeDelegate =
UBlueprintNodeSpawner::FCustomizeNodeDelegate::CreateStatic(GetMenuActions_Utils:
:SetNodeFunc, FunctionPtr);

    return NodeSpawner;
}

```

```

        })));
    }

void UK2Node_MyFunctionAsyncAction::AllocateDefaultPins()
{
    Super::AllocateDefaultPins();
}

bool UK2Node_MyFunctionAsyncAction::HandleDelegates(const
TArray<FBaseAsyncTaskHelper::FOutputPinAndLocalVariable>& VariableOutputs,
UEdGraphPin* ProxyObjectPin, UEdGraphPin*& InOutLastThenPin, UEdGraph*
SourceGraph, FKismetCompilerContext& CompilerContext)
{
    bool bIsErrorFree = true;

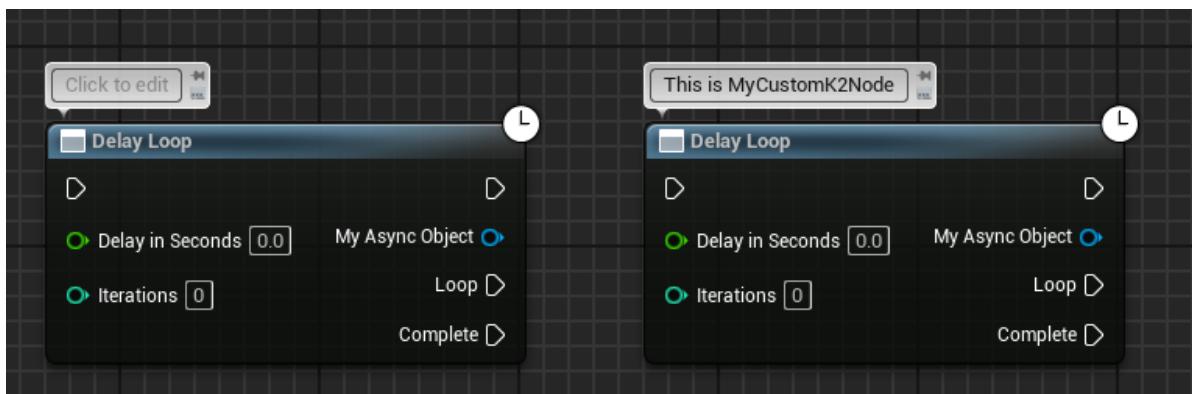
    for (TFieldIterator<FMulticastDelegateProperty> PropertyIt(ProxyClass);
PropertyIt && bIsErrorFree; ++PropertyIt)
    {
        UEdGraphPin* LastActivatedThenPin = nullptr;
        bIsErrorFree &=
FBaseAsyncTaskHelper::HandleDelegateImplementation(*PropertyIt, VariableOutputs,
ProxyObjectPin, InOutLastThenPin, LastActivatedThenPin, this, SourceGraph,
CompilerContext);
    }

    return bIsErrorFree;
}

```

## Blueprint Effect:

On the left is the node generated by the engine's built-in UK2Node\_AsyncAction, while on the right is the custom UK2Node\_MyFunctionAsyncAction blueprint node. Although they have identical functions, the right side includes an additional comment for differentiation. With this foundation, you can further customize by overloading methods within it.



## Currently used in two places in the source code:

```

UCLASS(BlueprintType, meta = (ExposedAsyncProxy = "AsyncTask",
HasDedicatedAsyncNode))
class GAMEPLAYMESSAGES_API UAsyncAction_RegisterGameplayMessageReceiver : public
UBlueprintAsyncActionBase
{

```

```

    UFUNCTION(BlueprintCallable, Category = Messaging, meta=
(worldContext="WorldContextObject", BlueprintInternalUseOnly="true"))
    static UASyncAction_RegisterGameplayMessageReceiver*
RegisterGameplayMessageReceiver(UObject* WorldContextObject, FEventMessageTag
Channel, UScriptStruct* PayloadType, EGameplayMessageMatchType MatchType =
EGameplayMessageMatchType::ExactMatch, AActor* ActorContext = nullptr);

}

//Created by UK2Node_GameplayMessageAsyncAction
void
UK2Node_GameplayMessageAsyncAction::GetMenuActions(FBlueprintActionDatabaseRegistrar& ActionRegistrar) const
{
    //...
    UClass* NodeClass = GetClass();

ActionRegistrar.RegisterClassFactoryActions<UASyncAction_RegisterGameplayMessageReceiver>
(FBlueprintActionDatabaseRegistrar::FMakeFuncSpawnerDelegate::CreateLambda([NodeClass](const UFunction* FactoryFunc) -> UBlueprintNodeSpawner*
{
    UBlueprintNodeSpawner* NodeSpawner =
UBlueprintFunctionNodeSpawner::Create(FactoryFunc);
    check(NodeSpawner != nullptr);
    NodeSpawner->NodeClass = NodeClass;

    TweakObjectPtr<UFunction> FunctionPtr =
MakeWeakObjectPtr(const_cast<UFunction*>(FactoryFunc));
    NodeSpawner->CustomizeNodeDelegate =
UBlueprintNodeSpawner::FCustomizeNodeDelegate::CreateStatic(GetMenuActions_Utils::SetNodeFunc, FunctionPtr);

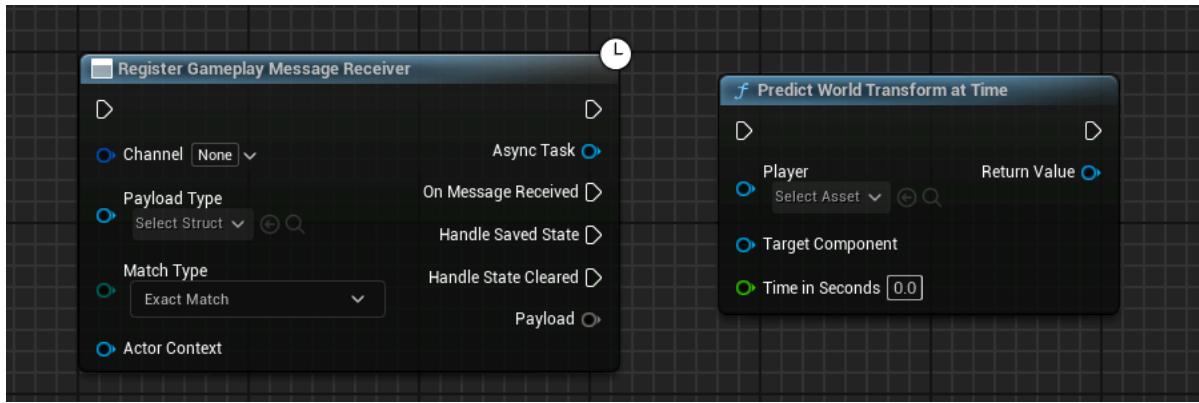
    return NodeSpawner;
}) );
}

UCLASS(BlueprintType, meta=(ExposedAsyncProxy = "AsyncTask",
HasDedicatedAsyncNode))
class UMovieSceneAsyncAction_SequencePrediction : public
UBlueprintAsyncActionBase
{
    UFUNCTION(BlueprintCallable, Category=Cinematics)
    static UMovieSceneAsyncAction_SequencePrediction*
PredictWorldTransformAtTime(UMovieSceneSequencePlayer* Player, USceneComponent* TargetComponent, float TimeInSeconds);
}

```

## Generated Blueprint:

UAsyncAction\_RegisterGameplayMessageReceiver is created by a custom UK2Node\_GameplayMessageAsyncAction, providing a generic Payload output pin. UMovieSceneAsyncAction\_SequencePrediction's factory method PredictWorldTransformAtTime, because the automatically generated version is hidden and no BlueprintInternalUseOnly attribute is added to suppress the UHT-generated version, results in the presentation of a standard static function blueprint node.



## The Mechanism of Action in the Source Code:

It can be observed that if `HasDedicatedAsyncNode` is present in a class, `nullptr` is returned directly, and `NodeSpawner` is not generated, thereby preventing the creation of blueprint nodes.

```

void UK2Node_AsyncAction::GetMenuActions(FBlueprintActionDatabaseRegistrar&
ActionRegistrar) const
{
    ActionRegistrar.RegisterClassFactoryActions<UBlueprintAsyncActionBase>
(FBlueprintActionDatabaseRegistrar::FMakeFuncSpawnerDelegate::CreateLambda([NodeC
lass](const UFunction* FactoryFunc) -> UBlueprintNodeSpawner*
{
    UClass* FactoryClass = FactoryFunc ? FactoryFunc->GetOwnerClass() :
nullptr;
    if (FactoryClass && FactoryClass-
>HasMetaData(TEXT("HasDedicatedAsyncNode")))
    {
        // Wants to use a more specific blueprint node to handle the async
action
        return nullptr;
    }

    UBlueprintNodeSpawner* NodeSpawner =
UBlueprintFunctionNodeSpawner::Create(FactoryFunc);
    check(NodeSpawner != nullptr);
    NodeSpawner->NodeClass = NodeClass;

    TWeakObjectPtr<UFunction> FunctionPtr =
MakeweakObjectPtr(const_cast<UFunction*>(FactoryFunc));
    NodeSpawner->CustomizeNodeDelegate =
UBlueprintNodeSpawner::FCustomizeNodeDelegate::CreateStatic(GetMenuActions_Utils:
: SetNodeFunc, FunctionPtr);

    return NodeSpawner;
}) );
}

```

# HideThen

- **Function Description:** Hide the 'Then' pin of asynchronous blueprint nodes
- **Usage Location:** UCLASS
- **Metadata Type:** boolean
- **Restriction Type:** Blueprint Asynchronous Node
- **Associated Items:** ExposedAsyncProxy

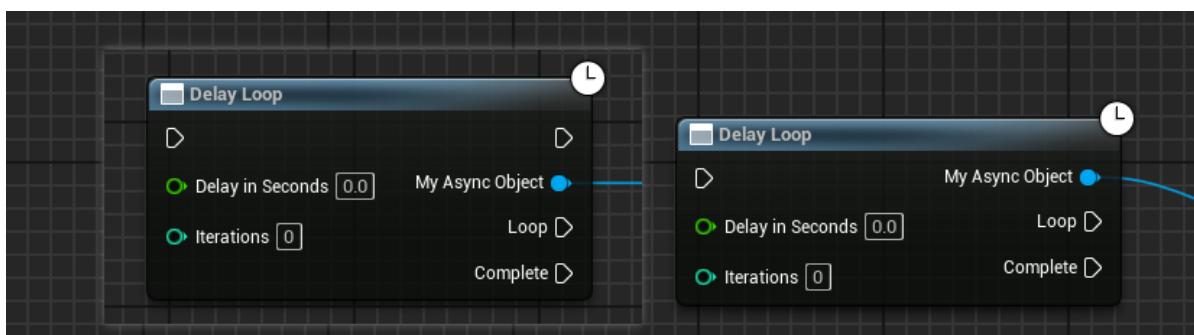
In the source code HideThen is only judged in UK2Node\_BaseAsyncTask , so this label only acts on blueprint asynchronous nodes.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy = MyAsyncObject))
class INSIDER_API UMyFunction_Async :public UCancellableAsyncAction
{};

UCLASS(Blueprintable, BlueprintType, meta = (ExposedAsyncProxy =
MyAsyncObject, HideThen))
class INSIDER_API UMyFunction_Async :public UCancellableAsyncAction
{}
```

## Comparison Before and After Using HideThen:



## Source Code Position:

```
void UK2Node_BaseAsyncTask::AllocateDefaultPins()
{
    bool bExposeProxy = false;
    bool bHideThen = false;
    FText ExposeProxyDisplayName;
    for (const UStruct* TestStruct = ProxyClass; TestStruct; TestStruct =
TestStruct->GetSuperStruct())
    {
        bExposeProxy |= TestStruct->HasMetaData(TEXT("ExposedAsyncProxy"));
        bHideThen |= TestStruct->HasMetaData(TEXT("HideThen"));
        if (ExposeProxyDisplayName.IsEmpty())
        {
            ExposeProxyDisplayName = TestStruct-
>GetMetaDataText(TEXT("ExposedAsyncProxy"));
        }
    }
}
```

```

if (!bHideThen)
{
    CreatePin(EGPD_Output, UEdGraphSchema_K2::PC_Exec,
UEdGraphSchema_K2::PN_Then);
}

}

```

## HideSpawnParms

- **Function Description:** Hides certain properties within the inheritance chain of a UGameplayTask subclass on the blueprint asynchronous node it generates.
- **Usage Location:** UFUNCTION
- **Metadata Type:** strings="a, b, c"
- **Associated Items:** ExposedAsyncProxy

hides certain properties in the inheritance chain of a UGameplayTask subclass on the blueprint asynchronous node generated by the subclass.

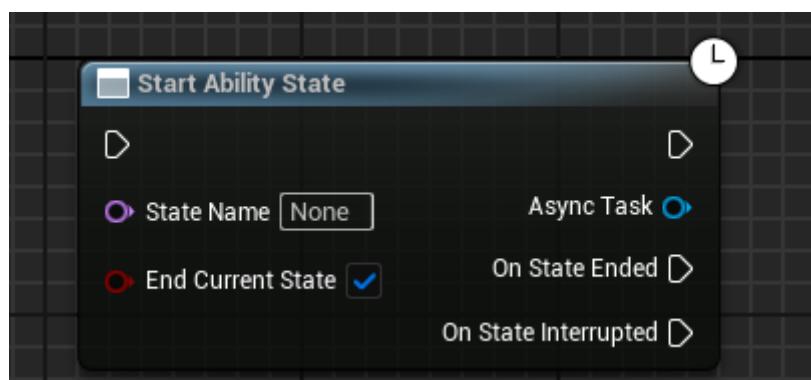
HideSpawnParms is only evaluated in UK2Node\_LatentGameplayTaskCall, thus it only affects subclasses of UGameplayTask. The sole usage found in the source code is HideSpawnParms = "Instigator", but this attribute does not exist in the inheritance chain of the UGameplayTask subclass, rendering it ineffective.

```

UFUNCTION(BlueprintCallable, Meta = (HidePin = "OwningAbility", DefaultToSelf
= "OwningAbility", BlueprintInternalUseOnly = "true", HideSpawnParms =
"Instigator"), Category = "Ability|Tasks")
    static UAbilityTask_StartAbilityState* StartAbilityState(UGameplayAbility*
OwningAbility, FName StateName, bool bEndCurrentState = true);

```

Both the blueprint nodes that retain and remove HideSpawnParms are:



## Location in the source code:

```

void UK2Node_LatentGameplayTaskCall::CreatePinsForClass(UClass* InClass)
{
    // Tasks can hide spawn parameters by doing meta =
    (HideSpawnParms="PropertyA,PropertyB")
    // (For example, hide Instigator in situations where instigator is not
    relevant to your task)
}

```

```

TArray< FString> IgnorePropertyList;
{
    UFunction* ProxyFunction = ProxyFactoryClass->FindFunctionByName(ProxyFactoryFunctionName);

    const FString& IgnorePropertyListStr = ProxyFunction->GetMetaData(FName(TEXT("HideSpawnParms")));
}

if (!IgnorePropertyListStr.IsEmpty())
{
    IgnorePropertyListStr.ParseIntoArray(IgnorePropertyList, TEXT(","),
true);
}
}
}

```

## NotInputConfigurable

- **Function Description:** Prevents certain UInputModifier and UInputTrigger instances from being configurable in ProjectSettings.
- **Usage Location:** UCLASS
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Restricted Types:** Subclasses of UInputModifier and UInputTrigger
- **Commonality:** ★

Prevents certain UInputModifier and UInputTrigger instances from being configurable in ProjectSettings.

### Source Code Example:

```

UCLASS(NotBlueprintable, meta = (DisplayName = "Chorded Action",
NotInputConfigurable = "true"))
class ENHANCEDINPUT_API UInputTriggerChordAction : public UInputTrigger
{};

UCLASS(NotBlueprintable, meta = (DisplayName = "Combo (Beta)",
NotInputConfigurable = "true"))
class ENHANCEDINPUT_API UInputTriggerCombo : public UInputTrigger
{};


```

### Test Code:

```

UCLASS( meta = (NotInputConfigurable = "true"))
class INSIDER_API UMyInputTrigger_NotInputConfigurable :public UInputTrigger
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere)
    float MyFloat = 123;
};


```

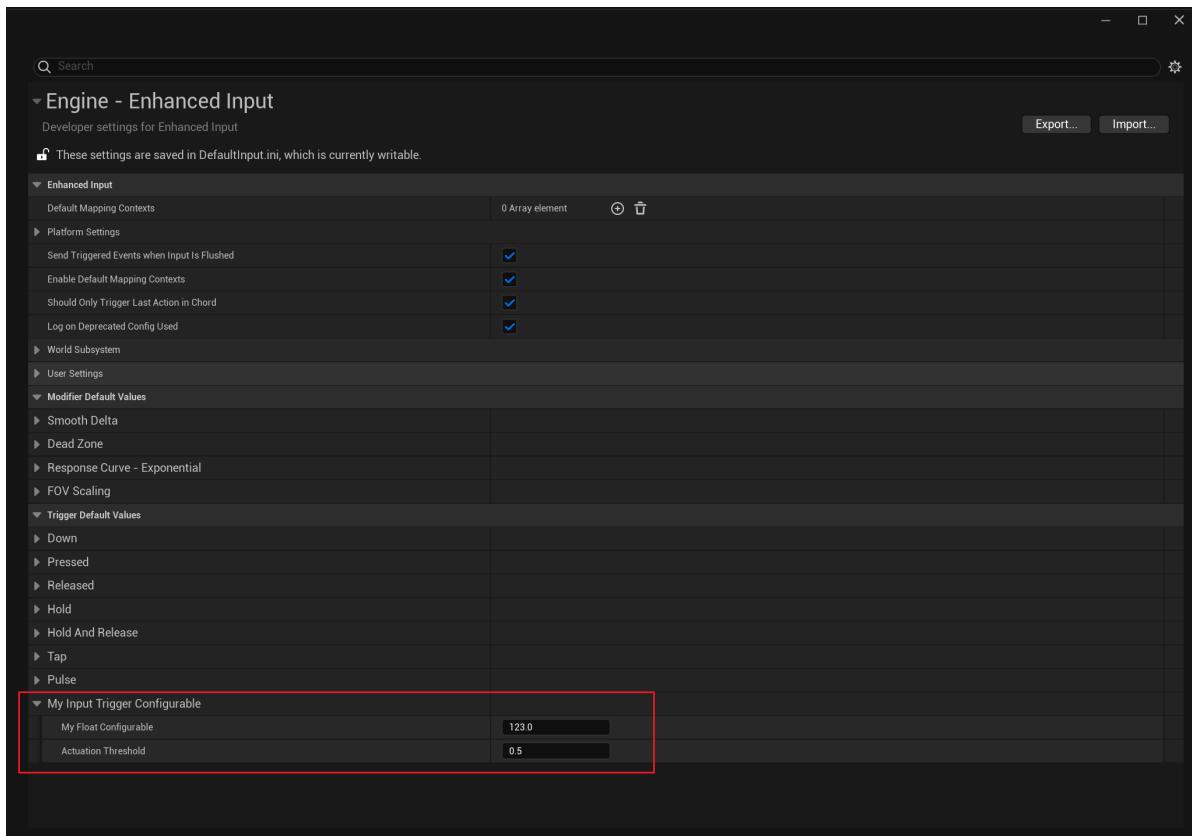
```

UCLASS( meta = () )
class INSIDER_API UMyInputTrigger_Configurable :public UInputTrigger
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere)
    float MyFloatConfigurable = 123;
};

```

## Test Results:

Only UMyInputTrigger\_Configurable is visible for editing the default value.



## Principle:

The UI customization in UEnhancedInputDeveloperSettings gathers the CDO objects of UInputModifier and UInputTrigger, then filters out those that are not configurable based on the NotInputConfigurable metadata.

```

GatherNativeClassDetailsCDOs(UInputModifier::StaticClass(), ModifierCDOs);
GatherNativeClassDetailsCDOs(UInputTrigger::StaticClass(), TriggerCDOs);

void
FEnhancedInputDeveloperSettingsCustomization::GatherNativeClassDetailsCDOs(UClass
* Class, TArray<UObject*>& CDOs)
{
    // Strip objects with no config stored properties
    CDOs.RemoveAll([Class]( UObject* Object) {

```

```

UClass* ObjectClass = Object->GetClass();
if (ObjectClass->GetMetaData(TEXT("NotInputConfigurable")).ToBool())
{
    return true;
}
while (ObjectClass)
{
    for (FProperty* Property : TFieldRange<FProperty>(ObjectClass,
EFieldIteratorFlags::ExcludeSuper, EFieldIteratorFlags::ExcludeDeprecated))
    {
        if (Property->HasAnyPropertyFlags(CPF_Config))
        {
            return false;
        }
    }

    // Stop searching at the base type. We don't care about
configurable properties lower than that.
ObjectClass = ObjectClass != Class ? ObjectClass->GetSuperClass()
: nullptr;
}
return true;
);
}

```

## BlueprintThreadSafe

- **Function Description:** Used on classes or functions to indicate that all functions within the class are thread-safe.  
Thus, they can be invoked in non-game threads, such as animation blueprints.
- **Usage Location:** UCLASS, UFUNCTION
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Restriction Type:** Typically, classes are usually of type BlueprintFunctionLibrary
- **Associated Items:** NotBlueprintThreadSafe
- **Commonly Used:** ★★★

The AimGraph in animation blueprints has thread-safe updates enabled by default. This setting is found in ClassSettings (enabled by default)

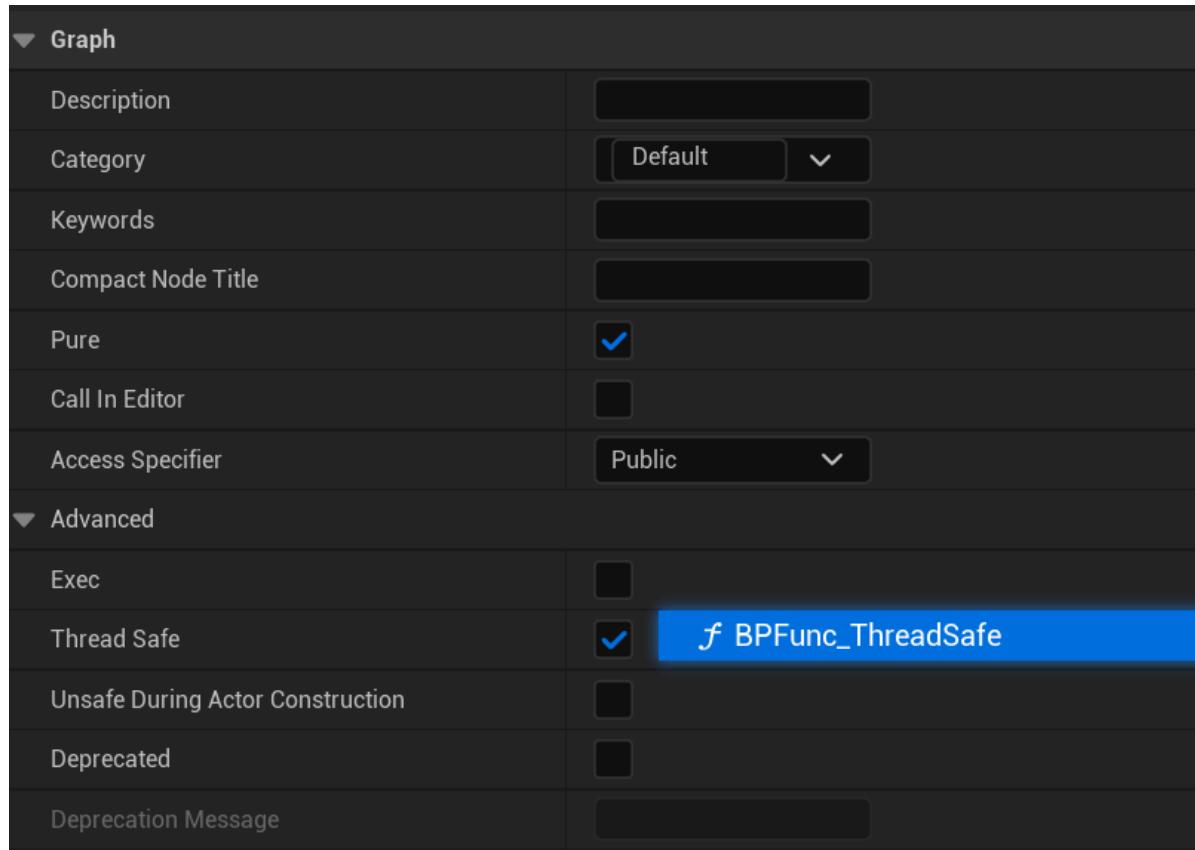


Refer to the official documentation under **CPU Thread Usage and Performance** for more information

### [Graphing in Animation Blueprints](#)

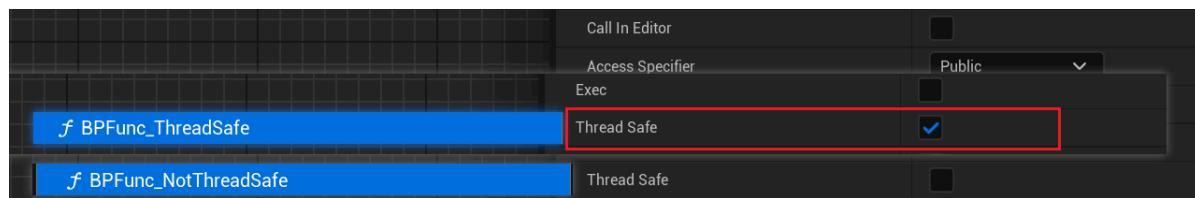
Therefore, all functions within AimGraph must be thread-safe. Your C++ functions or functions within blueprint libraries need to be manually marked as ThreadSafe; those without the ThreadSafe mark are not considered thread-safe by default.

In blueprints, if ThreadSafe is checked in the function panel, the function's object will be set to bThreadSafe=True, which in turn sets (BlueprintThreadSafe = true) on the compiled BlueprintGeneratedClass

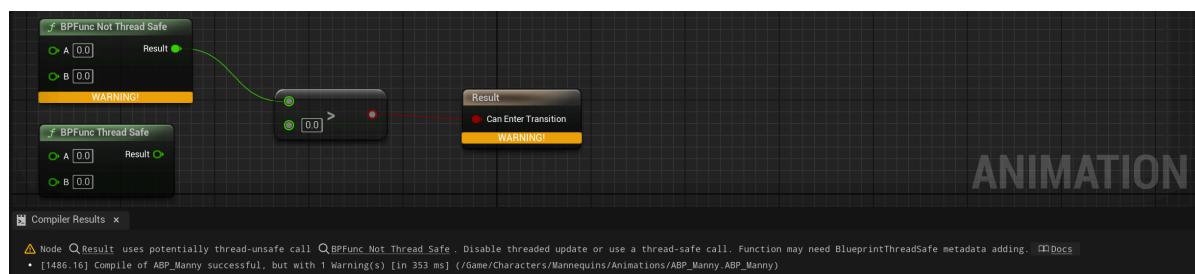


## Test the Blueprint Function Library:

For the same function, one with ThreadSafe enabled and one without, the function without ThreadSafe will trigger a warning during compilation when used in the AnimGraph of an animation blueprint.



Test Results:



## In C++, the C++ test code:

```
//(BlueprintThreadSafe = , IncludePath = Class/Blueprint/MyClass_Threadsafe.h,
ModuleRelativePath = Class/Blueprint/MyClass_ThreadSafe.h)
UCLASS(meta=(BlueprintThreadSafe))
```

```

class INSIDER_API UMyBlueprintFunctionLibrary_ThreadSafe : public
UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintPure)
    static float MyFunc_ClassThreadSafe_Default(float value) {return value+100;}

    //((ModuleRelativePath = Class/Blueprint/MyClass_ThreadSafe.h,
NotBlueprintThreadSafe = )
    UFUNCTION(BlueprintPure,meta=(NotBlueprintThreadSafe))
    static float MyFunc_ClassThreadSafe_FuncNotThreadSafe(float value) {return
value+100;}
};

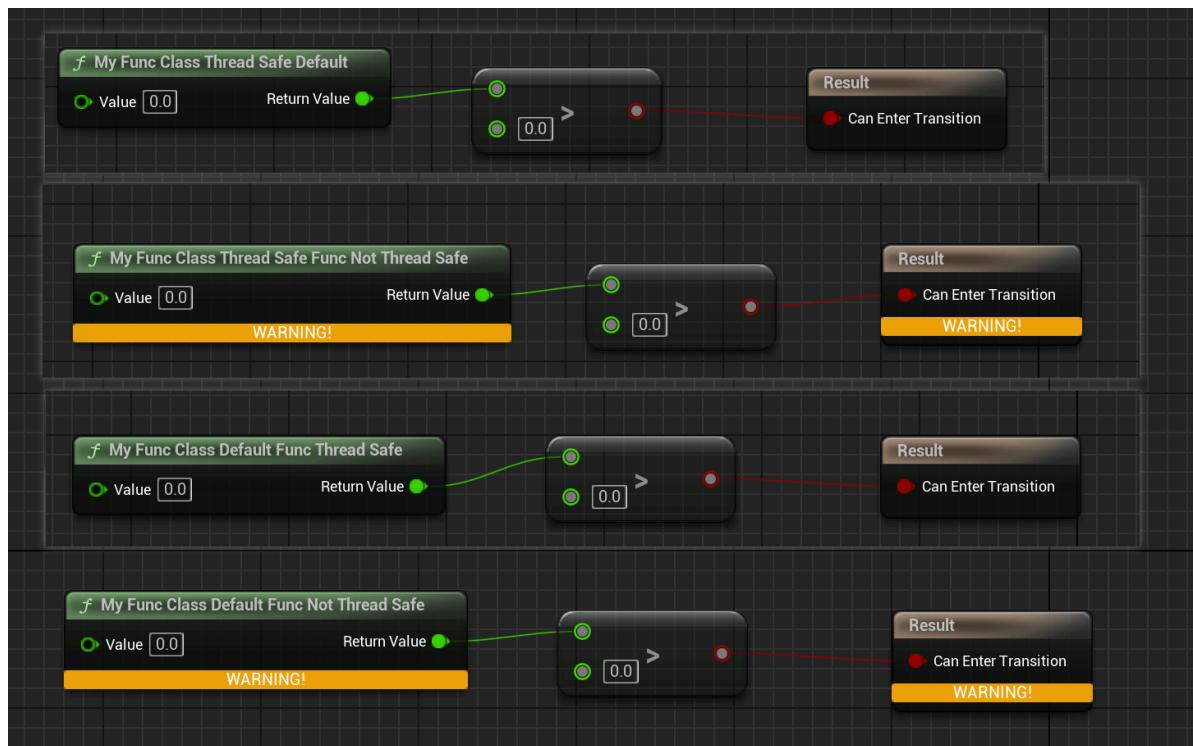
UCLASS()
class INSIDER_API UMyBlueprintFunctionLibrary_NoThreadSafe : public
UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    //((BlueprintThreadSafe = , ModuleRelativePath =
Class/Blueprint/MyClass_ThreadSafe.h)
    UFUNCTION(BlueprintPure,meta=(BlueprintThreadSafe))
    static float MyFunc_ClassDefault_FuncThreadSafe(float value) {return
value+100;}

    //((ModuleRelativePath = Class/Blueprint/MyClass_ThreadSafe.h,
NotBlueprintThreadSafe = )
    UFUNCTION(BlueprintPure,meta=(NotBlueprintThreadSafe))
    static float MyFunc_ClassDefault_FuncNotThreadSafe(float value) {return
value+100;}
};

UCLASS()
class INSIDER_API UMyBlueprintFunctionLibrary_Default : public
UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintPure)
    static float MyFunc_ClassDefault_FuncDefault(float value) {return value+100;}
};

```

# Test Results for the Animation Blueprint:



## Analysis of Principles:

```
bool FBlueprintEditorUtils::HasFunctionBlueprintThreadSafeMetaData(const
UFunction* InFunction)
{
    if(InFunction)
    {
        const bool bHasThreadSafeMetaData = InFunction->
>HasMetaData(FBlueprintMetadata::MD_ThreadSafe);
        const bool bHasNotThreadSafeMetaData = InFunction->
>HasMetaData(FBlueprintMetadata::MD_NotThreadSafe);
        const bool bClassHasThreadSafeMetaData = InFunction->GetOwnerClass() &&
InFunction->GetOwnerClass()->HasMetaData(FBlueprintMetadata::MD_ThreadSafe);

        // Native functions need to just have the correct class/function metadata
        const bool bThreadSafeNative = InFunction->
>HasAnyFunctionFlags(FUNC_Native) && (bHasThreadSafeMetaData ||
(bClassHasThreadSafeMetaData && !bHasNotThreadSafeMetaData));

        // Script functions get their flag propagated from their entry point, and
        // dont pay heed to class metadata
        const bool bThreadSafeScript = !InFunction->
>HasAnyFunctionFlags(FUNC_Native) && bHasThreadSafeMetaData;

        return bThreadSafeNative || bThreadSafeScript;
    }

    return false;
}
```

It can be logically deduced that if a UCLASS is marked with BlueprintThreadSafe, its internal functions are thread-safe by default, unless explicitly marked with NotBlueprintThreadSafe to opt out. If a UCLASS is not marked, each UFUNCTION must be manually marked with BlueprintThreadSafe. Both methods are acceptable.

Note that UCLASS(meta=(NotBlueprintThreadSafe)) is not recognized for this purpose and therefore holds no significance.

## NotBlueprintThreadSafe

---

- **Function Description:** Indicates that the function is not thread-safe when used
- **Usage Location:** UFUNCTION
- **Metadata Type:** boolean
- **Associated Item:** BlueprintThreadSafe
- **Commonality:** ★

## RestrictedToClasses

---

- **Function Description:** Restricts the creation of functions within the Blueprint Function Library to only be possible by right-clicking within blueprints of classes specified by RestrictedToClasses
- **Usage Location:** UCLASS
- **Engine Module:** Blueprint
- **Metadata Type:** strings = "a, b, c"
- **Restriction Type:** BlueprintFunctionLibrary
- **Commonality:** ★★★

Applied to the Blueprint Function Library, this restricts the functions within the library to be used solely in blueprints of classes defined by RestrictedToClasses, and they cannot be created by right-clicking in other blueprint classes.

## Test Code:

---

```
UCLASS(Blueprintable)
class INSIDER_API UMyClass_RestrictedToClasses : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
};

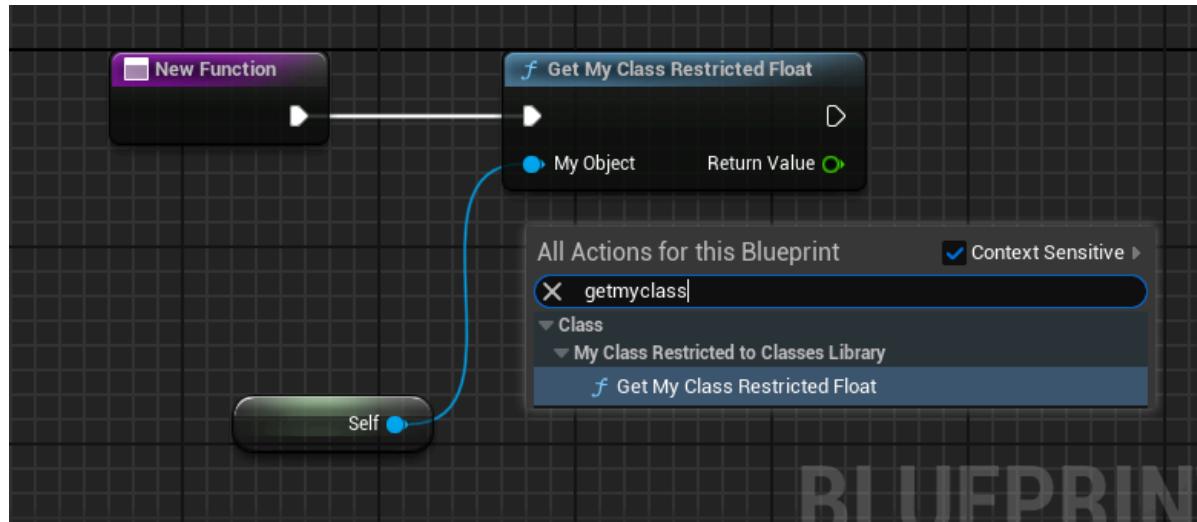
UCLASS(meta=(RestrictedToClasses="MyClass_RestrictedToClasses"))
class INSIDER_API UMyClass_RestrictedToClassesLibrary : public UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
```

```

static float GetMyClassRestrictedFloat(UMyClass_RestrictedToClasses*
myObject) {return myObject->MyFloat;}
};

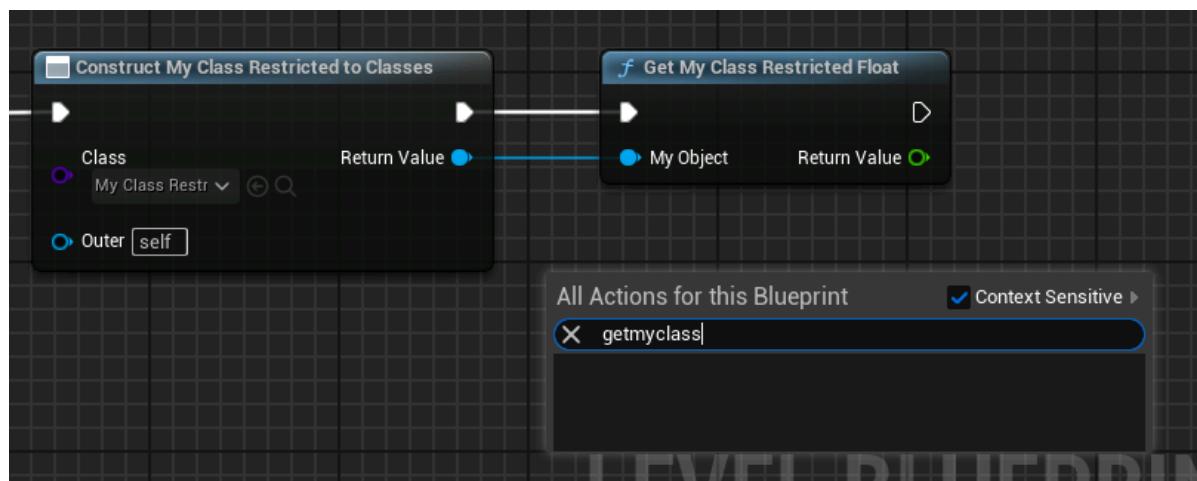
```

Test the code for functionality:



Test the effect in other locations, for example, within a level blueprint:

Thus, it cannot be created through a right-click, but the function can still be invoked by directly pasting the node.



## Assess the impact elsewhere, such as within a level blueprint:

Nodes specified in UBTFunctionLibrary can only be used within BTNode, as it would be pointless elsewhere.

```

UCLASS(meta=(RestrictedToClasses="BTNode"), MinimalAPI)
class UBTFunctionLibrary : public UBlueprintFunctionLibrary
{
    UFUNCTION(BlueprintPure, Category="AI|BehaviorTree", Meta=(HidePin="NodeOwner",
    DefaultToSelf="NodeOwner"))
        static AIMODULE_API UBlackboardComponent* GetOwnersBlackboard(UBTNode*
NodeOwner);
    //....
}

```

## Example in the Source Code:

```
static bool
BlueprintActionFilterImpl::IsRestrictedClassMember(FBlueprintActionFilter const&
Filter, FBlueprintActionInfo& BlueprintAction)
{
    bool bIsFilteredOut = false;
    FBlueprintActionContext const& FilterContext = Filter.Context;

    if (UClass const* ActionClass = BlueprintAction.GetOwnerClass())
    {
        if (ActionClass->HasMetaData(FBlueprintMetadata::MD_RestrictedToClasses))
        {
            FString const& ClassRestrictions = ActionClass-
>GetMetaData(FBlueprintMetadata::MD_RestrictedToClasses);

            // Parse the the metadata into an array that is delimited by ',' and
            trim whitespace
            TArray<FString> ParsedClassRestrictions;
            ClassRestrictions.ParseIntoArray(ParsedClassRestrictions, TEXT(","));
            for (FString& ValidClassName : ParsedClassRestrictions)
            {
                ValidClassName = ValidClassName.TrimStartAndEnd();
            }

            for (UBlueprint const* TargetContext : FilterContext.Blueprints)
            {
                UClass* TargetClass = TargetContext->GeneratedClass;
                if (!TargetClass)
                {
                    // skip possible null classes (e.g. macros, etc)
                    continue;
                }

                bool bIsClassListed = false;

                UClass const* QueryClass = TargetClass;
                // walk the class inheritance chain to see if this class is one
                // of the allowed
                while (!bIsClassListed && (QueryClass != nullptr))
                {
                    FString const ClassName = QueryClass->GetName();
                    // If this class is on the list of valid classes
                    for (const FString& ValidClassName : ParsedClassRestrictions)
                    {
                        bIsClassListed = (ClassName == ValidClassName);
                        if (bIsClassListed)
                        {
                            break;
                        }
                    }

                    QueryClass = QueryClass->GetSuperClass();
                }
            }
        }
    }
}
```

```

        // if the blueprint class wasn't listed as one of the few
        // classes that this can be accessed from, then filter it out
        if (!bIsClassListed)
        {
            bIsFilteredOut = true;
            break;
        }
    }

    return bIsFilteredOut;
}

```

## DontUseGenericSpawnObject

- **Function Description:** Prevent the use of the Generic Create Object node in blueprints from instantiating objects of this class.
- **Usage Location:** UCLASS
- **Engine Module:** Blueprint
- **Metadata Type:** boolean
- **Restriction Type:** When the BlueprintType class is neither an Actor nor an ActorComponent
- **Commonality:** ★★

Used to inhibit the construction of this class using the generic ConstructObject Blueprint node.

Typical examples in the source code include UDragDropOperation and UUserWidget; the former is created by the specialized node UK2Node\_CreateDragDropOperation (which internally calls UWidgetBlueprintLibrary::CreateDragDropOperation), and the latter is created by CreateWidget. Therefore, the typical use case is to create a custom New function to instantiate the object yourself.

## Test Code:

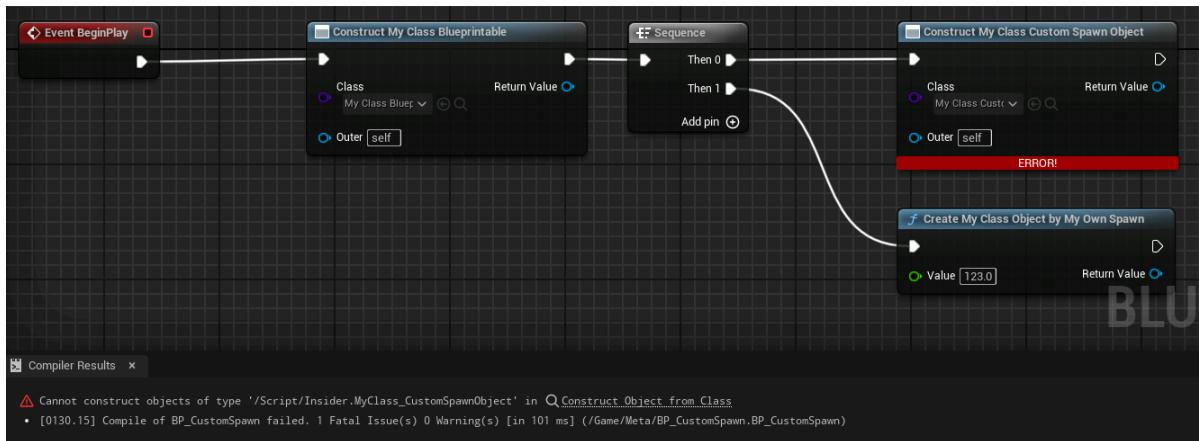
```

UCLASS(Blueprintable,meta=(DontUseGenericSpawnObject="true"))
class INSIDER_API UMyClass_CustomSpawnObject :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite,EditAnywhere)
    float MyFloat;

    UFUNCTION(BlueprintCallable)
    static UMyClass_CustomSpawnObject* Create MyClassObjectByMyOwnSpawn(float
value)
    {
        UMyClass_CustomSpawnObject* obj= NewObject<UMyClass_CustomSpawnObject>();
        obj->MyFloat=value;
        return obj;
    }
};

```

# Test Code:



## 测试效果:

The presence of the DontUseGenericSpawnObject metadata is checked in advance. Since the GetBoolMetaData function is used, the value must be explicitly set to "true"

```
struct FK2Node_GenericCreateObject_Utils
{
    static bool CanSpawnobjectofclass(Tsubclassef<Uobject> ObjectClass, bool bAllowAbstract)
    {
        // Initially include types that meet the basic requirements.
        // Note: CLASS_Deprecated is an inherited class flag, so any subclass of
        // an explicitly-deprecated class also cannot be spawned.
        bool bCanSpawnObject = (nullptr != *ObjectClass)
            && (bAllowAbstract || !ObjectClass->HasAnyClassFlags(CLASS_Abstract))
            && !ObjectClass->HasAnyClassFlags(CLASS_Deprecated) ||
        CLASS_NewerVersionExists);

        // Uobject is a special case where if we are allowing abstract we are
        // going to allow it through even though it doesn't have BlueprintType on it
        if (bCanSpawnObject && (!bAllowAbstract || (*ObjectClass != Uobject::StaticClass())))
        {
            static const FName BlueprintTypeName(TEXT("BlueprintType"));
            static const FName NotBlueprintTypeName(TEXT("NotBlueprintType"));
            static const FName
DontUseGenericSpawnObjectName(TEXT("DontUseGenericSpawnobject"));

            auto IsClassAllowedLambda = [] (const Uclass* Inclass)
            {
                return Inclass != AActor::StaticClass()
                    && Inclass != UActorComponent::StaticClass();
            };

            // Exclude all types in the initial set by default.
            bCanSpawnObject = false;
            const Uclass* CurrentClass = ObjectClass;
```

```

        // Climb up the class hierarchy and look for "BlueprintType." If
        "NotBlueprintType" is seen first, or if the class is not allowed, then stop
        searching.

        while (!bCanSpawnObject && CurrentClass != nullptr && !CurrentClass-
>GetBoolMetaData(NotBlueprintTypeName) && IsClassAllowedLambda(CurrentClass))
        {
            // Include any type that either includes or inherits
            'BlueprintType'

            bCanSpawnObject = CurrentClass-
>GetBoolMetaData(BlueprintTypeName);

            // Stop searching if we encounter 'BlueprintType' with
            'DontUseGenericSpawnObject'
            if (bCanSpawnObject && CurrentClass-
>GetBoolMetaData(DontUseGenericSpawnObjectName))
            {
                bCanSpawnObject = false;
                break;
            }

            CurrentClass = CurrentClass->GetSuperclass();
        }

        // If we validated the given class, continue walking up the hierarchy
        to make sure we exclude it if it's an Actor or ActorComponent derivative.

        while (bCanSpawnObject && CurrentClass != nullptr)
        {
            bCanSpawnObject &= IsClassAllowedLambda(CurrentClass);

            CurrentClass = CurrentClass->GetSuperclass();
        }
    }

    return bCanSpawnObject;
}
};


```

## ObjectType

---

- **Function Description:** Specifies the type of object collection for the statistical page.
- **Usage Location:** UClass
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Commonality:** ★

Specifies the type of object collection for the statistical page.

Belonging to the StatViewer module, it is used exclusively on a few specific internal classes.

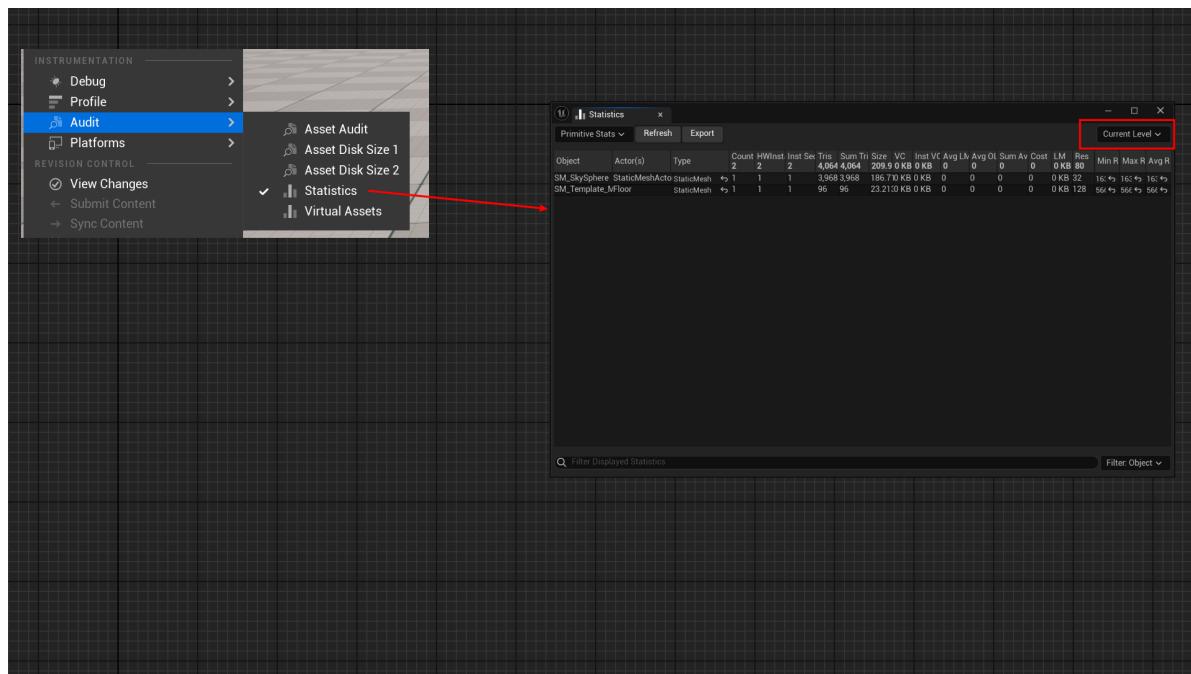
# Source Code Example:

```
/** Enum defining the object sets for this stats object */
UENUM()
enum EPrimitiveObjectSets : int
{
    PrimitiveObjectSets_AllObjects           UMETA( DisplayName = "All objects" ,
    ToolTip = "View primitive statistics for all objects in all levels" ),
    PrimitiveObjectSets_CurrentLevel        UMETA( DisplayName = "Current Level" ,
    , ToolTip = "View primitive statistics for objects in the current level" ),
    PrimitiveObjectSets_SelectedObjects     UMETA( DisplayName = "Selected
Objects" , ToolTip = "View primitive statistics for selected objects" ),
};

/** Statistics page for primitives. */
UCLASS(Transient, MinimalAPI, meta=( DisplayName = "Primitive Stats",
ObjectSetType = "EPrimitiveObjectSets" ) )
class UPrimitiveStats : public UObject
{}
```

## Respective Effect:

On the statistical page, the type is visible in the upper right corner.



## Principles:

```
template <typename Entry>
class FStatsPage : public IStatsPage
{
public:
    FStatsPage()
    {
        FString EnumName = Entry::StaticClass()->GetName();
        EnumName += TEXT(".");
    }
};
```

```

        EnumName += Entry::StaticClass()->GetMetaData( TEXT("ObjectSetType") );
        ObjectSetEnum = FindObject<UEnum>( nullptr, *EnumName );
        bRefresh = false;
        bShow = false;
        ObjectSetIndex = 0;
    }
};


```

## SparseClassDataTypes

- **Usage Location:** UCLASS
- **Engine Module:** Blueprint
- **Metadata Type:** string = "abc"
- **Associated Items:** GetByRef  
UCLASS: Sparse Class Data Type
- **Commonality:** ★★★

## KismetHideOverrides

- **Function description:** List of blueprint events that must not be overridden.
- **Usage location:** UCLASS
- **Engine module:** Blueprint
- **Metadata type:** strings = "a, b, c"

In the source code, many definitions are found on ALevelScriptActor to prevent them from being overridden.

### Example:

```

UCLASS(notplaceable, meta=(ChildCanTick, KismetHideOverrides =
"ReceiveAnyDamage,ReceivePointDamage,ReceiveRadialDamage,ReceiveActorBeginOverlap,
,ReceiveActorEndOverlap,ReceiveHit,ReceiveDestroyed,ReceiveActorBeginCursorOver,R
eceiveActorEndCursorOver,ReceiveActorOnClicked,ReceiveActorOnReleased,ReceiveActo
rOnInputTouchBegin,ReceiveActorOnInputTouchEnd,ReceiveActorOnInputTouchEnter,Rece
iveActorOnInputTouchLeave"), HideCategories=(Collision,Rendering,Transformation),
MinimalAPI)
class ALevelScriptActor : public AActor
{};


```

However, these events can still be overridden in subclasses of LevelScriptActor. Some hidden events are actually concealed using HideCategories. Thus, this Meta is not actually implemented. To achieve the intended effect, one must still use HideFunctions or HideCategories.

OVERRIDE FUNCTION	
ActorBeginCursorOver	Actor
ActorEndCursorOver	Actor
ActorOnClicked	Actor
ActorOnReleased	Actor
AnyDamage	Actor
Async Physics Tick	Actor
BeginInputTouch	Actor
Destroyed	Actor
End Play	Actor
EndInputTouch	Actor
Level Reset	Level Script Actor
OnBecomeViewTarget	Actor
OnEndViewTarget	Actor
OnReset	Actor
PointDamage	Actor
RadialDamage	Actor
TouchEnter	Actor
TouchLeave	Actor
World Origin Location Changed	Level Script Actor

## Principle:

It can be observed that the judgment here does not make use of this Meta

```
void SMyBlueprint::CollectAllActions(FGraphActionListBuilderBase& OutAllActions)
{
    // Cache potentially overridable functions
    UClass* ParentClass = BlueprintObj->SkeletonGeneratedClass ? BlueprintObj-
>SkeletonGeneratedClass->GetSuperClass() : *BlueprintObj->ParentClass;
    for (TFieldIterator FunctionIt(ParentClass,
        EFieldIteratorFlags::IncludeSuper); FunctionIt; ++FunctionIt )
    {
        const* Function = *FunctionIt;
        const FName& FunctionName = Function->GetFName();

        UClass* OuterClass = CastChecked<UClass>(Function->GetOuter());
        // ignore skeleton classes and convert them into their "authoritative" types
        so they
        // can be found in the graph
        if(UBlueprintGeneratedClass* GeneratedOuterClass =
            Cast<UBlueprintGeneratedClass>(OuterClass))
        {
            OuterClass = GeneratedOuterClass->GetAuthoritativeClass();
        }

        if (UEdGraphSchema_K2::CanKismetOverrideFunction(Function)
            && !OverridableFunctionNames.Contains(FunctionName)
            && !ImplementedFunctionCache.Contains(FunctionName)
            && !ObjectEditorUtils::IsFunctionHiddenFromClass(Function, ParentClass)
            && !FBlueprintEditorUtils::FindOverrideForFunction(BlueprintObj,
            OuterClass, Function->GetFName())
    }
}
```

```

        && Blueprint->AllowFunctionOverride(Function)
    )
{
    FText FunctionTooltip =
FText::FromString(UK2Node_CallFunction::GetDefaultTooltipForFunction(Function));
    FText FunctionDesc = K2Schema->GetFriendlySignatureName(Function);
    if (FunctionDesc.IsEmpty())
    {
        FunctionDesc = FText::FromString(Function->GetName());
    }

    if (Function->HasMetaData(FBlueprintMetadata::MD_DeprecatedFunction))
    {
        FunctionDesc =
FBlueprintEditorUtils::GetDeprecatedMemberMenuItemName(FunctionDesc);
    }

    FText FunctionCategory = FObjectEditorUtils::GetCategoryText(Function);

    TSharedPtr<FEdGraphSchemaAction_K2Graph> NewFuncAction =
MakeShareable(new
FEdGraphSchemaAction_K2Graph(EEdGraphSchemaAction_K2Graph::Function,
FunctionCategory, FunctionDesc, FunctionTooltip, 1,
NodeSectionID::FUNCTION_OVERRIDABLE));
    NewFuncAction->FuncName = FunctionName;

    OverridableFunctionActions.Add(NewFuncAction);
    OverridableFunctionNames.Add(ClassName);
}
}
}

```

## BlueprintType

- **Function Description:** Indicates that it can be used as a variable in Blueprints
- **Usage Locations:** UCLASS, UENUM, UINTERFACE, USTRUCT
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Associated Items:**
  - UCLASS: Blueprintable, NotBlueprintable, BlueprintType, NotBlueprintType
  - Meta: BlueprintInternalUseOnly, BlueprintInternalUseOnlyHierarchical
  - UENUM: BlueprintType
  - UFUNCTION: BlueprintInternalUseOnly
  - UINTERFACE: Blueprintable, NotBlueprintable
  - USTRUCT: BlueprintInternalUseOnly, BlueprintType
- **Commonly Used:** ★★★★☆

# IsConversionRoot

---

- **Function Description:** Allows the Actor to transform between itself and its subclasses
- **Usage Location:** UCLASS, UINTERFACE
- **Engine Module:** Blueprint
- **Metadata Type:** boolean
- **Associated Items:**
  - UCLASS: ConversionRoot
  - UINTERFACE: ConversionRoot
- **Commonly Used:** ★★★

# BlueprintInternalUseOnlyHierarchical

---

- **Function Description:** This attribute signifies that the structure and its subclasses are not intended for user definition or use and are exclusively for internal use within the blueprint system
- **Usage Location:** USTRUCT
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Associated Items:**
  - Meta: BlueprintInternalUseOnly, BlueprintType
  - USTRUCT: BlueprintInternalUseOnlyHierarchical
- **Commonality:** ★

Indicates a structure of type BlueprintType and its derived structures that are not disclosed to the end users.

# BlueprintSetter

---

- **Function Description:** Utilizes a custom set function for reading.  
Defaults to setting BlueprintReadWrite.
- **Usage Location:** UFUNCTION, UPROPERTY
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Associated Items:**

## UFILEN : Blueprint Setter

---

## UPROPERTY : Blueprint Setter

---

- **Commonly Used:** ★★★

# DisplayName

---

- **Functional Description:** The name assigned to this node within the Blueprint will be replaced by the value specified here, instead of the name generated by the code.
- **Usage Locations:** UCLASS, UENUM::UMETA, UFUNCTION, UPARAM, UPROPERTY
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Associated Items:**  
UPARAM: DisplayName
- **Commonality:** ★★★★☆

# ExposeOnSpawn

---

- **Function Description:** Make this attribute exposed during object creation using ConstructObject or SpawnActor, etc.
- **Use location:** UPROPERTY
- **Engine module:** Blueprint
- **Metadata type:** bool
- **Frequency:** ★★★★☆

Expose this property during object creation operations like ConstructObject or SpawnActor.

- Specifically, searching the source code reveals its usage in UK2Node.AddComponent, UK2Node\_ConstructObjectFromClass, UK2Node\_SpawnActor, and UK2Node\_LatentGameplayTaskCall.
- The effect of setting this in C++ is analogous to toggling the "ExposeOnSpawn" option in blueprints.
- Applying this meta setting will also simultaneously set CPF\_ExposeOnSpawn in PropertyFlags

## Test Code:

---

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_ExposeOnSpawn :public UObject
{
    GENERATED_BODY()
public:
    // (Category = MyProperty_ExposeOnSpawn, ModuleRelativePath =
    // Property/Blueprint/MyProperty_ExposeOnSpawn.h)
    // PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
    // CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString MyString = TEXT("First");

    // (Category = MyProperty_ExposeOnSpawn, ExposeOnSpawn = ,
    // ModuleRelativePath = Property/Blueprint/MyProperty_ExposeOnSpawn.h)
    // PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
    // CPF_ExposeOnSpawn | CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
```

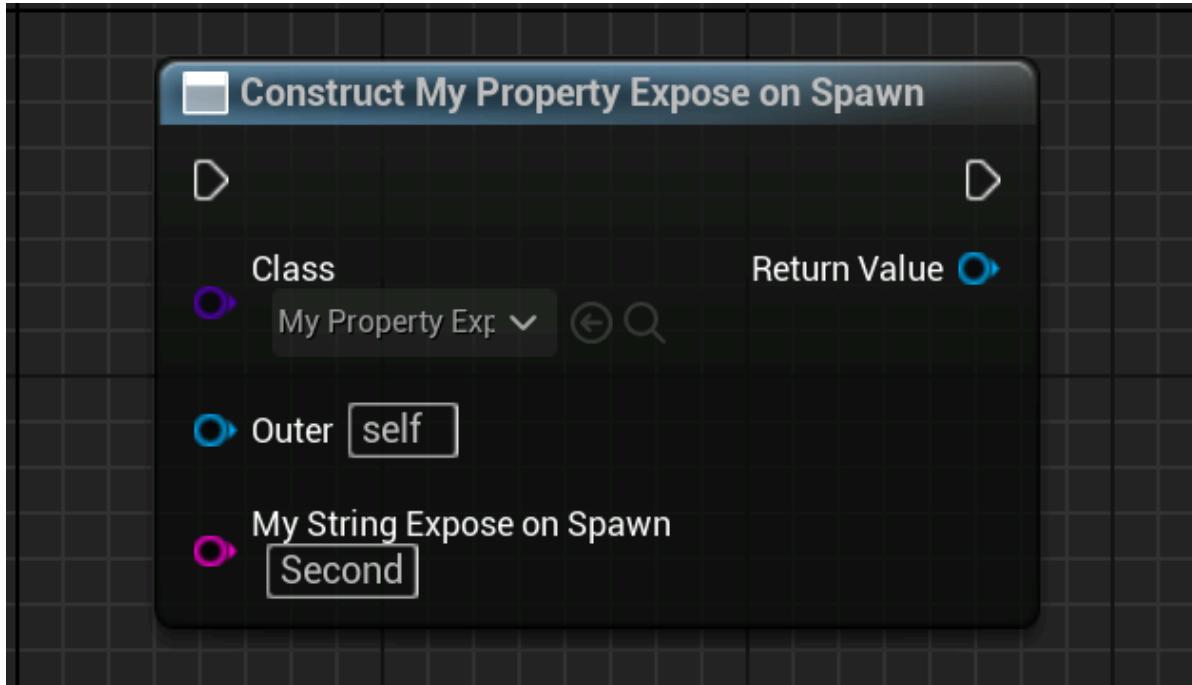
```

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ExposeOnSpawn))
FString MyString_ExposeOnSpawn = TEXT("Second");
};

```

## Test Results:

Notice that MyString\_ExposeOnSpawn is exposed, while MyString is not.



## Principle:

During UHT analysis, if a property contains ExposeOnSpawn, CPF\_ExposeOnSpawn will be set synchronously.

The IsPropertyExposedOnSpawn function determines whether to expose the property. This function is referenced by the aforementioned four function nodes.

UK2Node\_ConstructObjectFromClass's CreatePinsForClass serves as an example in the source code, demonstrating that additional Pins are created for the blueprint node only when bIsExposedToSpawn is true.

```

if (propertySettings.MetaData.ContainsKey(UhtNames.ExposeOnSpawn))
{
    propertySettings.PropertyFlags |= EPropertyFlags.ExposeOnSpawn;
}

bool UEdGraphSchema_K2::IsPropertyExposedOnSpawn(const FPropertyParams*PropertyParams)
{
   PropertyParams = FBlueprintEditorUtils::GetMostUpToDatePropertyParams(PropertyParams);
    if (PropertyParams)
    {
        const bool bMeta =PropertyParams->HasMetaData(FBlueprintMetadata::MD_ExposeOnSpawn);
        const bool bFlag =PropertyParams->HasAllPropertyFlags(CPF_ExposeOnSpawn);
        if (bMeta != bFlag)
        {
            const FCoreTexts& CoreTexts = FCoreTexts::Get();

```

```

        UE_LOG(LogBlueprint, warning
            , TEXT("ExposeOnSpawn ambiguity. Property '%s', MetaData '%s',
Flag '%s'"))
            , *Property->GetFullName()
            , bMeta ? *CoreTexts.True.ToString() :
*CoreTexts.False.ToString()
            , bFlag ? *CoreTexts.True.ToString() :
*CoreTexts.False.ToString());
        }
        return bMeta || bFlag;
    }
    return false;
}

void UK2Node_ConstructObjectFromClass::CreatePinsForClass(UClass* InClass,
TArray<UEdGraphPin*>* OutClassPins)
{
    for (TFieldIterator<FProperty> PropertyIt(InClass,
EFieldIteratorFlags::IncludeSuper); PropertyIt; ++PropertyIt)
    {
        FProperty* Property = *PropertyIt;
        UClass* PropertyClass = CastChecked<UClass>(Property->GetOwner<UObject>());
        const bool bIsDelegate = Property-
>IsA(FMulticastDelegateProperty::StaticClass());
        const bool bIsExposedToSpawn =
UEdGraphSchema_K2::IsPropertyExposedOnSpawn(Property);
        const bool bIsSettableExternally = !Property-
>HasAnyPropertyFlags(CPF_DisableEditOnInstance);

        if( bIsExposedToSpawn &&
            !Property->HasAnyPropertyFlags(CPF_Parm) &&
            bIsSettableExternally &&
            Property->HasAllPropertyFlags(CPF_BlueprintVisible) &&
            !bIsDelegate &&
            (nullptr == FindPin(Property->GetFName()) ) &&
            FBlueprintEditorUtils::PropertyStillExists(Property))
        {
            if (UEdGraphPin* Pin = CreatePin(EGPD_Input, NAME_None, Property-
>GetFName()))
            {
        }
    }
}

```

## NativeConst

---

- **Function description:** Specify the const sign in C++
- **Usage Location:** UPARAM
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Associated Items:**  
UPARAM: Const
- **Commonality:** ★

# CPP\_Default\_XXX

- **Function Description:** XXX=Parameter Name
- **Usage Location:** UPARAM
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Commonliness:** ★★★★☆

Save the default value of the parameter in the UFUNCTION's meta data.

## Test Code:

```
//(CPP_Default_intValue = 123, CPP_Default_intvalue2 = 456, ModuleRelativePath =
Function/Param/MyFunction_TestParam.h)
UFUNCTION(BlueprintCallable)
FString MyFuncTestParam_DefaultInt2(int intValue=123,int intvalue2=456);
```

Default values for attributes can also be directly specified in the Meta, such as Duration.

```
UFUNCTION(BlueprintCallable, Category="Utilities|FlowControl", meta=(Latent,
WorldContext="WorldContextObject", LatentInfo="LatentInfo", Duration="0.2",
Keywords="sleep"))
static ENGINE_API void MyDelay(const UObject* WorldContextObject, float
Duration, struct FLatentActionInfo LatentInfo );
```

## Principle Code:

In UEdGraphSchema\_K2::FindFunctionParameterDefaultValue it will try to find the Meta corresponding to the parameter name. If it cannot be found, it will continue to look for the name CPP\_Default\_ParamName . Then set to Pin->AutogeneratedDefaultValue

```
bool UK2Node_CallFunction::CreatePinsForFunctionCall(const UFunction* Function)
{
    FString ParamValue;
    if (K2Schema->FindFunctionParameterDefaultValue(Function, Param,
ParamValue))
    {
        K2Schema->SetPinAutogeneratedDefaultValue(Pin, ParamValue);
    }
    else
    {
        K2Schema->SetPinAutogeneratedDefaultValueBasedOnType(Pin);
    }
}
```

# BlueprintGetter

---

- **Function Description:** Utilizes a custom get function for reading.  
Should no BlueprintSetter or BlueprintReadWrite be configured, BlueprintReadOnly will be set by default.
- **Usage Locations:** UFUNCTION, UPROPERTY
- **Engine Module:** Blueprint
- **Metadata Type:** string="abc"
- **Associated Items:**
  - UFnGet : Blueprint Getter
  - UPubProperty : Blueprint Getter
- **Commonly Used:** ★★★

# IsBlueprintBase

---

- **Function description:** Indicates whether this class is an acceptable base class for creating blueprints, similar to the UCLASS specifier, Blueprintable or ' NotBlueprintable '.
- **Usage Locations:** UCLASS, UINTERFACE
- **Engine Module:** Blueprint
- **Metadata Type:** bool
- **Associated Items:**
  - UCLASS: Blueprintable, NotBlueprintable
  - UINTERFACE: Blueprintable, NotBlueprintable
- **Commonality:** ★★★★★

# BlueprintInternalUseOnly

---

- **Function description:** Indicates that the element is intended for internal use within the blueprint system and is not exposed for direct definition or use at the user level.
- **Usage locations:** UFUNCTION, USTRUCT
- **Engine module:** Blueprint
- **Metadata type:** bool
- **Associated items:**
  - Meta: BlueprintType, BlueprintInternalUseOnlyHierarchical
  - UFUNCTION: BlueprintInternalUseOnly
  - USTRUCT: BlueprintInternalUseOnly
- **Commonly used:** ★★★

It can also be applied to USTRUCT, signifying that the structure is not to be used for defining new Blueprint variables but can be exposed and passed as member variables of other classes.

When used on UFUNCTION: This function is an internal implementation detail, utilized for the realization of another function or node, and it is never directly exposed within the Blueprint diagram.

# UseComponentPicker

---

- **Function description:** When used on the ComponentReference attribute, it displays the components under the Actor in the selector list for selection.
- **Use location:** UPROPERTY
- **Engine module:** Component Property
- **Metadata type:** bool
- **Limit type:** FComponentReference, FSoftComponentReference
- **Related items:** AllowAnyActor
- **Frequency of use:** ★★

When used on the ComponentReference property, it displays the components under the Actor in the selector list for easy selection.

- By default, the selector list expanded by the Referenced Actor property of FComponentReference allows you to select Actors in the scene, but it does not display all the components under those Actors. The ComponentName attribute under ComponentReference requires manual input from the player. This method is relatively basic and prone to errors.
- Adding UseComponentPicker will display a list of components for selection. However, by default, it is limited to all components under the current Actor, excluding components from other Actors in the scene.
- To further list all components under all Actors in the scene, you need to additionally add AllowAnyActor to expand the filtering scope.
- There are two property types for ComponentReference: FComponentReference and FSoftComponentReference, both of which correspond to FComponentReferenceCustomization. The test code omits FSoftComponentReference for brevity.

## Test Code:

---

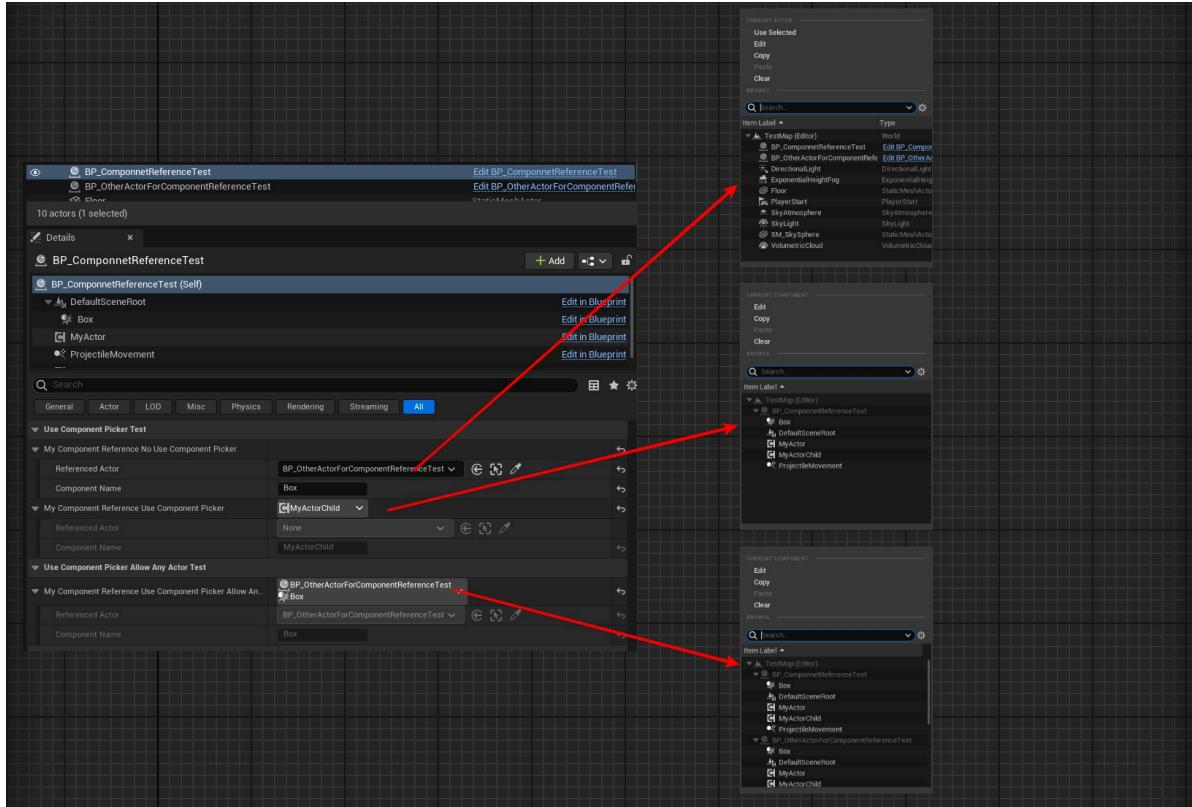
```
UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"UseComponentPickerTest")
FComponentReference MyComponentReference_NoUseComponentPicker;

UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"UseComponentPickerTest", meta = (UseComponentPicker))
FComponentReference MyComponentReference_UsedComponentPicker;

UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"UseComponentPicker_AllowAnyActor_Test", meta =
(UseComponentPicker,AllowAnyActor))
FComponentReference MyComponentReference_UsedComponentPicker_AllowAnyActor;
```

# Test Results:

- The default setting lists all Actors, but ComponentName needs to be manually entered.
- After adding UseComponentPicker, the second setting lists all components under the current Actor, but components from other Actors cannot be selected.
- The third setting, with AllowAnyActor added, lists all components of all Actors.



## Principle:

Both FComponentReference and FSoftComponentReference correspond to FComponentReferenceCustomization. Examining the source code reveals that using bUseComponentPicker creates ClassFilters and ComboBox, employing different type filters and UIs for component selection. Otherwise, the else branch uses a very basic structural attribute expansion and editing approach.

```
void
FComponentReferenceCustomization::CustomizeHeader(TSharedRef<IPropertyHandle>
InPropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&
CustomizationUtils)
{
    PropertyHandle = InPropertyHandle;

    CachedComponent.Reset();
    CachedFirstOuterActor.Reset();
    CachedPropertyAccess = FPropertyAccess::Fail;

    bAllowClear = false;
    bAllowAnyActor = false;
    bUseComponentPicker = PropertyHandle->HasMetaData(NAME_UseComponentPicker);
    bIsSoftReference = false;
```

```

if (bUseComponentPicker)
{
    FProperty* Property = InPropertyParams->GetProperty();
    check(CastField<FStructPropertyParams>(Property) &&
        (FComponentReference::StaticStruct() == CastFieldChecked<const
FPropertyParams>(Property)->Struct || 
        FSoftComponentReference::StaticStruct() == CastFieldChecked<const
FPropertyParams>(Property)->Struct));

    bAllowClear = !(InPropertyParams->GetMetaDataPropertyParams()->PropertyFlags &
CPF_NoClear);
    bAllowAnyActor = InPropertyParams->HasMetaData(NAME_AllowAnyActor);
    bIsSoftReference = FSoftComponentReference::StaticStruct() ==
CastFieldChecked<const FPropertyParams>(Property)->Struct;

    BuildClassFilters();
    BuildComboBox();

    InPropertyParams-
>SetOnPropertyValuechanged(FSimpleDelegate::CreateSP(this,
&FComponentReferenceCustomization::OnPropertyValuechanged));

    // set cached values
    {
        CachedComponent.Reset();
        CachedFirstOuterActor = GetFirstOuterActor();

        FComponentReference TmpComponentReference;
        CachedPropertyAccess = GetValue(TmpComponentReference);
        if (CachedPropertyAccess == FPropertyAccess::Success)
        {
            CachedComponent =
TmpComponentReference.GetComponent(CachedFirstOuterActor.Get());
            if (!IsComponentReferenceValid(TmpComponentReference))
            {
                CachedComponent.Reset();
            }
        }
    }

    HeaderRow.NameContent()
    [
        InPropertyParams->CreatePropertyNameWidget()
    ]
    .valueContent()
    [
        ComponentComboBox.ToSharedRef()
    ]
    .IsEnabled(MakeAttributesSP(this,
&FComponentReferenceCustomization::CanEdit));
}

else
{
    HeaderRow.NameContent()
    [
        InPropertyParams->CreatePropertyNameWidget()
}

```

```

        ]
        .valueContent()
        [
            InPropertyParams->CreatePropertyValueWidget()
        ]
        .IsEnabled(MakeAttributesP(this,
&FComponentReferenceCustomization::CanEdit));
    }
}

```

## AllowAnyActor

- **Function Description:** Used on the ComponentReference attribute, this allows the component selector to include components from other Actors within the scene when UseComponentPicker is enabled.
- **Usage Location:** UPROPERTY
- **Engine Module:** Component Property
- **Metadata Type:** bool
- **Restricted Types:** FComponentReference, FSoftComponentReference
- **Associated Items:** UseComponentPicker
- **Commonality:** ★★

Applied to the ComponentReference attribute, this enables the component selector to include components from other Actors within the scene when UseComponentPicker is in use.

- Note that AllowAnyActor only affects the component selection in the UI. A ComponentReference can still reference another Actor via ReferencedActor, even without AllowAnyActor, and manually input the name of its child components. This allows the correct component object to be retrieved using GetComponent in C++. Thus, AllowAnyActor is unrelated to the logic.

For test code and results, refer to UseComponentPicker.

## Principle:

Primarily involves FComponentReferenceCustomization. As indicated by the source code, bAllowAnyActor is only active when bUseComponentPicker is already enabled and is used to filter the list of Actors.

```

void
FComponentReferenceCustomization::CustomizeHeader(TSharedRef<IPPropertyParams>
InPropertyParams, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&
CustomizationUtils)
{
   PropertyParams = InPropertyParams;

    CachedComponent.Reset();
    CachedFirstOuterActor.Reset();
    CachedPropertyAccess = FPropertyAccess::Fail;

    bAllowClear = false;
    bAllowAnyActor = false;
}

```

```

bUseComponentPicker = PropertyHandle->HasMetaData(NAME_UseComponentPicker);
bIsSoftReference = false;

if (bUseComponentPicker)
{
    FProperty* Property = InPropertyParams->GetProperty();
    check(CastField<FStructPropertyParams>(Property) &&
        (FComponentReference::StaticStruct() == CastFieldChecked<const
FPropertyParams>(Property)->Struct || 
        FSoftComponentReference::StaticStruct() == CastFieldChecked<const
FPropertyParams>(Property)->Struct));

    bAllowClear = !(InPropertyParams->GetMetaDataProperty()->PropertyFlags &
CPF_NoClear);
    bAllowAnyActor = InPropertyParams->HasMetaData(NAME_AllowAnyActor);
    bIsSoftReference = FSoftComponentReference::StaticStruct() ==
CastFieldChecked<const FPropertyParams>(Property)->Struct;

    BuildClassFilters();
    BuildComboBox();
}

InPropertyParams-
>SetOnPropertyValuechanged(FSimpleDelegate::CreateSP(this,
&FComponentReferenceCustomization::OnPropertyValuechanged));

// set cached values
{
    CachedComponent.Reset();
    CachedFirstOuterActor = GetFirstOuterActor();

    FComponentReference TmpComponentReference;
    CachedPropertyParams = GetValue(TmpComponentReference);
    if (CachedPropertyParams == FPropertyParams::Success)
    {
        CachedComponent =
TmpComponentReference.GetComponent(CachedFirstOuterActor.Get());
        if (!IsComponentReferenceValid(TmpComponentReference))
        {
            CachedComponent.Reset();
        }
    }
}

HeaderRow.NameContent()
[
    InPropertyParams->CreatePropertyNameWidget()
]
.valueContent()
[
    ComponentComboBox.ToSharedRef()
]
.IsEnabled(MakeAttributeSP(this,
&FComponentReferenceCustomization::CanEdit));
}
else
{
}

```

```

HeaderRow.NameContent()
[
    InPropertyParams->CreatePropertyNameWidget()
]
.valueContent()
[
    InPropertyParams->CreatePropertyValueWidget()
]
.IsEnabled(MakeAttributesP(this,
&FComponentReferenceCustomization::CanEdit));
}

bool FComponentReferenceCustomization::IsFilteredActor(const AActor* const Actor)
const
{
    return bAllowAnyActor || Actor == CachedFirstOuterActor.Get();
}

```

## BlueprintSpawnableComponent

- **Function description:** Allows this component to appear in the component addition panel within an Actor Blueprint.
- **Usage location:** UCLASS
- **Engine module:** Component Property
- **Metadata type:** boolean
- **Restriction type:** Component class
- **Commonly used:** ★★★★

Enables the component to be displayed in the Add Component panel in the Actor Blueprint.

In the blueprint node, the component can be added whether or not the BlueprintSpawnableComponent is present.

### Test Code:

```

UCLASS(Blueprintable, meta = (BlueprintSpawnableComponent))
class INSIDER_API UMyActorComponent_Spawnable : public UActorComponent
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
};

UCLASS(Blueprintable)
class INSIDER_API UMyActorComponent_NotSpawnable : public UActorComponent
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)

```

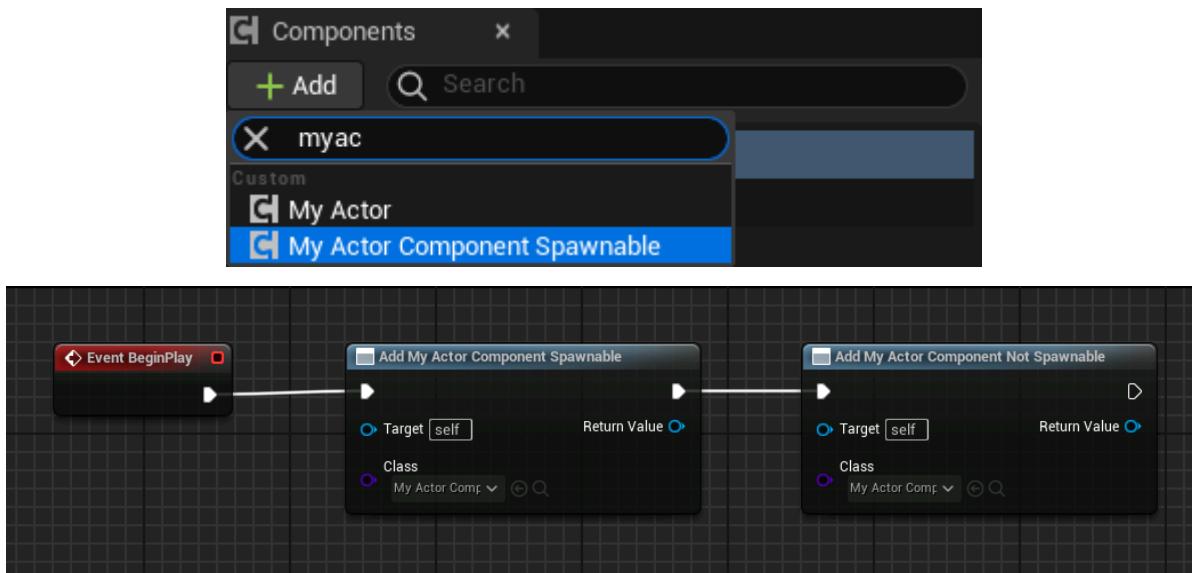
```

    float MyFloat;
};

```

## Effect in Blueprint:

As seen, under the Add button on the left side of the Actor, UMyActorComponent\_Spawnable can be added, but UMyActorComponent\_NotSpawnable is prevented from being added. However, it is also worth noting that both components can be added if using the AddComponent node in the blueprint.



## Principle:

```

bool FKismetEditorUtilities::IsClassABlueprintSpawnableComponent(const UClass* Class)
{
    // @fixme: Cooked packages don't have any metadata (yet; they might become available via the sidecar editor data)
    // However, all uncooked BPs that derive from ActorComponent have the BlueprintSpawnableComponent metadata set on them
    // (see FBlueprintEditorUtils::RecreateClassMetaData), so include any ActorComponent BP that comes from a cooked package
    return (!Class->HasAnyClassFlags(CLASS_Abstract) &&
            Class->IsChildOf<UActorComponent>() &&
            (Class->HasMetaData(FBlueprintMetadata::MD_BlueprintSpawnableComponent) || class->GetPackage()->bIsCookedForEditor));
}

```

## ConsoleVariable

- **Function description:** Synchronize the value of a Config attribute with a console variable of the same name.
- **Use location:** UPROPERTY
- **Engine module:** Config
- **Metadata type:** string="abc"
- **Commonly used:** ★★★★

Synchronize the value of a Config attribute with a console variable of the same name.

- Config values often need to be modified in the console (by pressing ~). This requirement is quite common, hence the creation of this tag. A classic example is the series of control variables starting with "r." in URendererSettings within the source code.
- The value in the Config file will also become the name of the ConsoleVariable (typically in a format like r.XXX.XX), rather than the property name.
- However, simply adding this tag is insufficient; the console variable will not be created automatically. You must manually create it using code and register a console variable with the same name, such as with TAutoConsoleVariable.
- Once the console variables are in place, additional code is required to synchronize their values. Refer to the test code calls for ImportConsoleVariableValues and ExportValuesToConsoleVariables.
- Be particularly aware that ConsoleVariable settings have priority levels. The console has a higher priority than ProjectSettings, so if you attempt to change a value in ProjectSettings after modifying it in the console, an error will occur.

## Test Code:

```
UCLASS(config = InsiderSettings, defaultconfig)
class UMyProperty_InsiderSettings :public UDeveloperSettings
{
    GENERATED_BODY()
public:
    UPROPERTY(Config, EditAnywhere, BlueprintReadWrite, Category = Console, meta
= (ConsoleVariable = "i.Insider.MyStringConsole"))
    FString MyString_Consolevariable;
public:
    virtual void PostInitProperties() override;
#if WITH_EDITOR
    virtual void PostEditChangeProperty(FPropertyChangedEvent&
PropertyChangedEvent) override;
#endif
};

//.cpp
static TAutoConsolevariable<FString> CVarInsiderMyStringConsole(
    TEXT("i.Insider.MyStringConsole"),
    TEXT("Hello"),
    TEXT("Insider test config to set MyString."));

void UMyProperty_InsiderSettings::PostInitProperties()
{
    Super::PostInitProperties();

#if WITH_EDITOR
    if (IsTemplate())
    {
        ImportConsoleVariableValues();
    }
#endif // #if WITH_EDITOR
}
```

```

#if WITH_EDITOR
void UMyProperty_InsiderSettings::PostEditChangeProperty(FPropertyChangedEvent&
PropertyChangedEvent)
{
    Super::PostEditChangeProperty(PropertyChangedEvent);

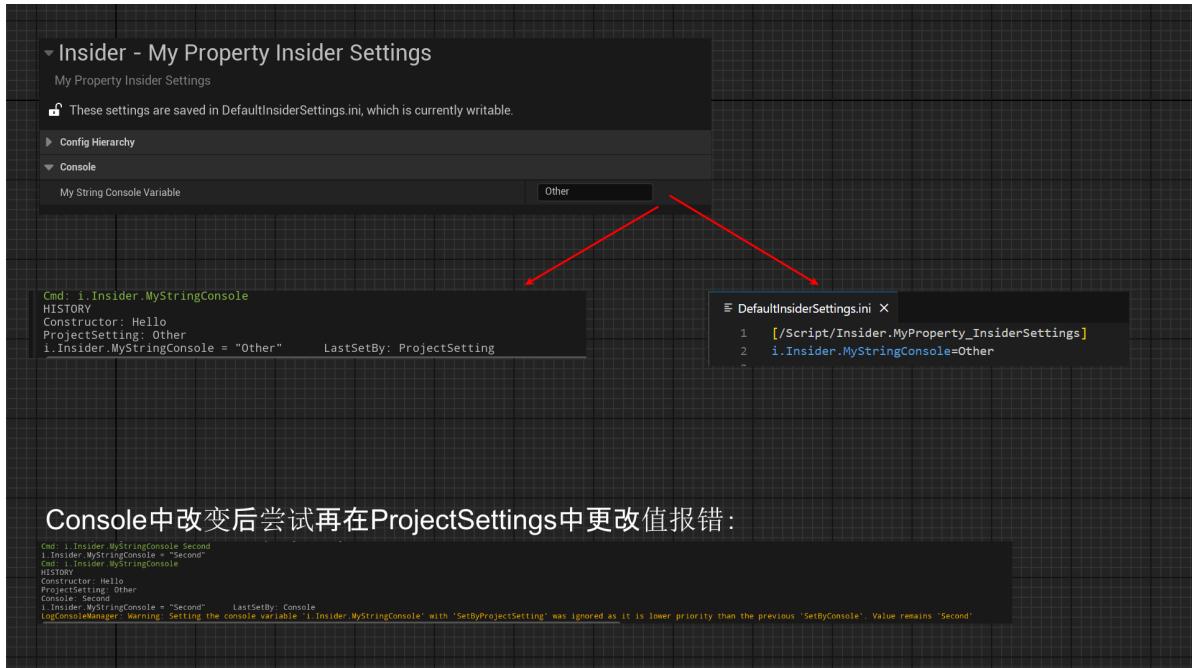
    if (PropertyChangedEvent.Property)
    {
        ExportValuesToConsoleVariables(PropertyChangedEvent.Property);
    }
}
#endif // #if WITH_EDITOR

```

## Test Results:

It is evident that initially, the values in the console and the configuration files are synchronized with those in ProjectSettings.

If a value is changed in the console and then an attempt is made to modify it in ProjectSettings, an error will occur.



## Principle:

The specific logic for value synchronization can be understood by examining the following two functions, which essentially involve finding the corresponding ConsoleVariable by name and then performing get/set operations on its value.

```

void UDeveloperSettings::ImportConsoleVariablevalues()
{}

void UDeveloperSettings::ExportValuesToConsoleVariables(FPropertyChangedEvent*
PropertyChanged)
{}

```

# EditorConfig

---

- **Function Description:** Preserve the configuration settings for the editor
- **Usage Location:** UCLASS
- **Engine Module:** Config
- **Metadata Type:** string="abc"
- **Associated Items:**
  - UCLASS: EditorConfig
- **Commonality:** ★★★

# ConfigHierarchyEditable

---

- **Function description:** Allows an attribute to be configured at various levels within the Config system.
- **Use location:** UPROPERTY
- **Engine module:** Config
- **Metadata type:** bool
- **Commonly used:** ★★★

Enables a property to be configured at different levels of the Config system.

- What is meant by the "levels of Config" refers to the hierarchy where Base, ProjectDefault, EnginePlatform, and ProjectPlatform are progressively overridden by higher-priority settings. For more information on this topic, you can refer to other detailed articles about config settings available online.
- Usually, properties with Config have their configuration values specified only in the config file indicated by the config specifier on UCLASS. However, if a property is marked with ConfigHierarchyEditable, it permits different configurations at each level. This type of property typically requires different values based on platform-specific needs, such as platform-related performance parameters.

## Test Example:

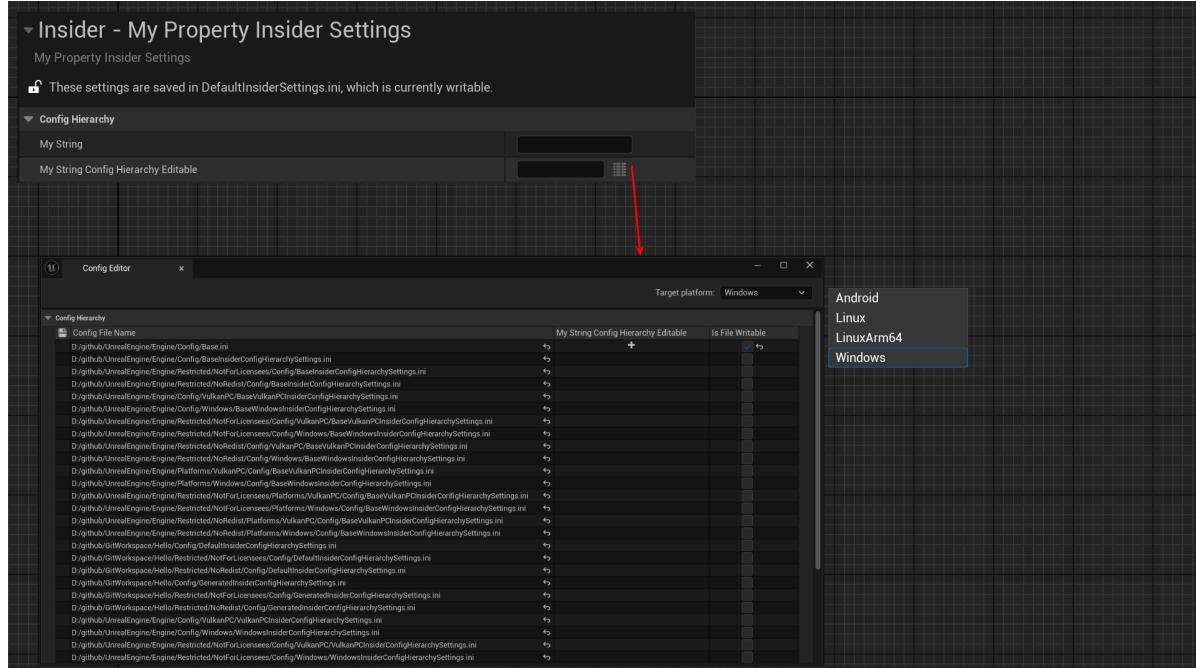
---

```
UCLASS(config = InsiderSettings, defaultconfig)
class UMyProperty_InsiderSettings :public UDeveloperSettings
{
    GENERATED_BODY()
public:
    UPROPERTY(Config, EditAnywhere, BlueprintReadWrite, Category =
ConfigHierarchy)
    FString MyString;

    UPROPERTY(Config, EditAnywhere, BlueprintReadWrite, Category =
ConfigHierarchy, meta = (ConfigHierarchyEditable))
    FString MyString_ConfigHierarchyEditable;
};
```

# Test Results:

You can observe that a hierarchy button has appeared on the right side of the MyString\_ConfigHierarchyEditable input field, which can be used to open a dedicated ConfigEditor, facilitating the configuration of different values across various platforms and levels.



## Source Code Example:

```
UCLASS(config = Game, defaultconfig)
class COMMONUI_API UCommonUISettings : public uobject
{
    /** The set of traits defined per-platform (e.g., the default input mode,
    whether or not you can exit the application, etc...) */
    UPROPERTY(config, EditAnywhere, Category = "Visibility", meta=
    Categories="Platform.Trait", ConfigHierarchyEditable)
        TArray<FGameplayTag> PlatformTraits;
}
```

## Principle:

The logic is straightforward: when generating the ValueWidget in the details panel, an additional button for hierarchical configuration is created based on the ConfigHierarchyEditable setting.

```
void FDetailPropertyRow::MakeValueWidget( FDetailWidgetRow& Row, const
TSharedPtr< FDetailWidgetRow> InCustomRow, bool bAddWidgetDecoration ) const
{
    // Don't add config hierarchy to container children, can't edit child
    properties at the hierarchy's per file level
    TSharedPtr< IPropertyHandle> ParentHandle = PropertyHandle->GetPropertyHandle();
    bool bIsChildProperty = ParentHandle && (ParentHandle->AsArray() ||
    ParentHandle->AsMap() || ParentHandle->Asset());
    if (!bIsChildProperty && PropertyHandle-
    >HasMetaData(TEXT("ConfigHierarchyEditable")))
}
```

```

{
    valueWidget->Addslot()
    .Autowidth()
    .VAlign(VAlign_Center)
    .HAlign(HAlign_Left)
    .Padding(4.0f, 0.0f, 4.0f, 0.0f)
    [
        PropertyCustomizationHelpers::MakeEditConfigHierarchyButton(FSimpleDelegate::CreateSP(PropertyEditor.ToSharedRef(), &FPropertyEditor::EditConfigHierarchy))
    ];
}

```

#ConfigRestartRequired

- **Function Description:** Prompts a dialog to restart the editor after properties are modified in the settings.
- **Usage Location:** UPROPERTY
- **Engine Module:** Config
- **Metadata Type:** bool
- **Commonality:** ★★★

Triggers a restart dialog for the editor after properties are changed in the settings.

Naturally, it is typically used for settings that necessitate a restart of the editor.

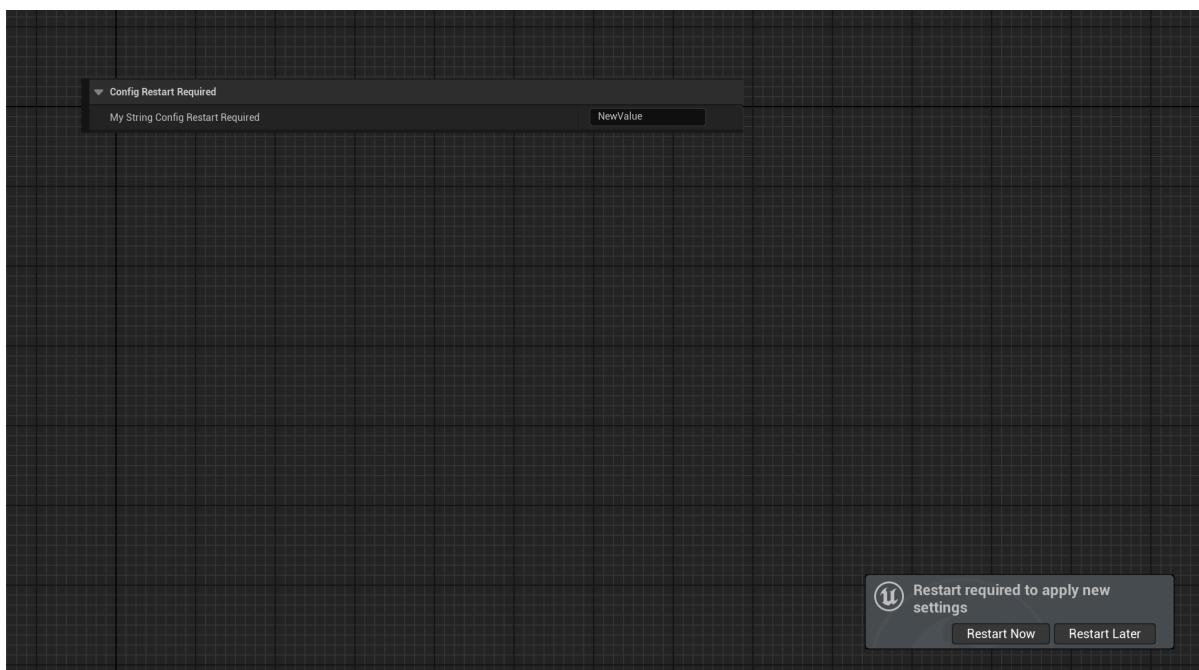
## Test Code:

```

public:
    UPROPERTY(Config, EditAnywhere, BlueprintReadWrite, Category =
ConfigRestartRequired, meta = (ConfigRestartRequired="true"))
    FString MyString_ConfigRestartRequired;

```

## Test Effects:



## Principle:

The effect is active in SSettingsEditor, indicating changes occur within the UI window, followed by a dialog prompt.

```
void SSettingsEditor::NotifyPostChange( const FPropertyChangedEvent&
PropertyChangeEvent, class FEditPropertyChain* PropertyThatChanged )
{
    static const FName ConfigRestartRequiredKey = "ConfigRestartRequired";
    if (PropertyChangeEvent.Property->GetBoolMetaData(ConfigRestartRequiredKey) ||
PropertyChangeEvent.MemberProperty->GetBoolMetaData(ConfigRestartRequiredKey))
    {
        OnApplicationRestartRequiredDelegate.ExecuteIfBound();
    }
}
```

## ReadOnlyKeys

- **Function Description:** Prevents the Key of a TMap attribute from being edited.
- **Usage Location:** UPROPERTY
- **Engine Module:** Container Property
- **Metadata Type:** bool
- **Restriction Type:** TMap attribute
- **Commonality:** ★★

Prevents the Key of the TMap attribute from being edited.

Means that the elements within this TMap are set prior to this (e.g., initialized in the constructor), but we only want users to modify the content of the values, not the names of the Keys. This is particularly useful in scenarios such as using Platform as the Key, where the list of Platforms is fixed and should not be altered by users.

## Test Code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ReadonlyKeysTest)
TMap<int32, FString> MyIntMap_NoReadOnlyKeys;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ReadonlyKeysTest, meta
= (ReadOnlyKeys))
TMap<int32, FString> MyIntMap_ReadOnlyKeys;
```

## Test Results:

It is evident that the Key of MyIntMap\_ReadOnlyKeys is grayed out and cannot be edited.

▼ Read Only Keys Test				
My Int Map No Read Only Keys		3 Map elements	⊕	⊖
0	Hello	▼	↑	↑
1	Hello	▼	↑	↑
2	Hello	▼	↑	↑
▼ My Int Map Read Only Keys		3 Map elements	⊕	⊖
0	Hello	▼	↑	↑
1	Hello	▼	↑	↑
2	Hello	▼	↑	↑

## Found in the Source Code:

```
void FDetailPropertyRow::MakeNameOrKeyWidget( FDetailWidgetRow& Row, const
TSharedPtr< FDetailWidgetRow> InCustomRow ) const
{
    if (PropertyHandle->HasMetaData(TEXT("ReadOnlyKeys")))
    {
        PropertyKeyEditor->GetPropertyNode()->SetNodeFlags(EPropertyNodeFlags::IsReadonly, true);
    }
}
```

## ArraySizeEnum

- **Function Description:** Provides an enumeration for a fixed array, allowing array elements to be indexed and displayed according to the enumeration values.
- **Usage Location:** UPROPERTY
- **Engine Module:** Container Property
- **Metadata Type:** string="abc"
- **Restriction Type:** T Array[Size]
- **Commonality:** ★★★

Provides an enumeration for a fixed array, allowing array elements to be indexed and displayed according to the enumeration values.

- The term "fixed array" refers to an array that is distinct from a TArray, which can dynamically change in size. Instead, it is a simple array defined directly with [size]. This type of fixed array (static array), which does not undergo additions or deletions, is sometimes suitable for using all the values in the enumeration as indices, thereby offering greater convenience.
- Within enumerations, the last enumeration item (typically named Max, Size, count, etc.) is generally used as the value indicating the data size.

- Enumeration values that should not be displayed can be hidden using Hidden. However, because the array index corresponds to the index of the enumeration item (i.e., the position of the enumeration value) rather than the value of the enumeration item itself, it may be noticed that the actual number of items displayed in the array is less than the defined Size.

## Test Code:

```

UENUM(BlueprintType)
enum class EMyArrayEnumNormal : uint8
{
    First,
    Second,
    Third,
    Max,
};

UENUM(BlueprintType)
enum class EMyArrayEnumHidden : uint8
{
    First,
    Second,
    Cat = 5 UMETA(Hidden),
    Third = 2,
    Max = 3,
};

UPROPERTY(EditAnywhere, Category = ArraySizeEnumTest)
int32 MyIntArray_NoArraySizeEnum[3];

UPROPERTY(EditAnywhere, Category = ArraySizeEnumTest, meta = (ArraySizeEnum =
"MyArrayEnumNormal"))
int32 MyIntArray_Normal_HasArraySizeEnum[(int)EMyArrayEnumNormal::Max];

UPROPERTY(EditAnywhere, Category = ArraySizeEnumTest, meta = (ArraySizeEnum =
"MyArrayEnumHidden"))
int32 MyIntArray_Hidden_HasArraySizeEnum[(int)EMyArrayEnumHidden::Max];

```

## Test Effects:

Both are fixed arrays with a size of 3.

- MyIntArray\_NoArraySizeEnum is the most common array format.
- MyIntArray\_Normal\_HasArraySizeEnum is a standard example of using enumeration items as array indices. Notice that the index names are not 012 but the names of the enumeration items.
- MyIntArray\_Hidden\_HasArraySizeEnum uses an enumeration with a hidden item called Cat, but its index is 2 (due to the order of definition), so the third item in the array is hidden.

▼ Array Size Enum Test		
My Int Array No Array Size Enum	3 Array elements	
Index [ 0 ]	0	
Index [ 1 ]	0	
Index [ 2 ]	0	
My Int Array Normal Has Array Size Enum	3 Array elements	
First	0	
Second	0	
Third	0	
My Int Array Hidden Has Array Size Enum	2 Array elements	
First	0	
Second	0	

## Principle:

---

It can be observed that initially, it is determined whether it is a fixed array (ArrayDim>1 actually signifies a fixed array), then the enumeration is located by name, and for each item in the array, the corresponding enumeration item is found to generate sub-rows in the details panel.

```
void FItemPropertyNode::InitChildNodes()
{
    if( MyProperty->ArrayDim > 1 && ArrayIndex == -1 )
    {
        // Do not add array children which are defined by an enum but the
        // enum at the array index is hidden
        // This only applies to static arrays
        static const FName NAME_ArraySizeEnum("ArraySizeEnum");
        UEnum* ArraySizeEnum = NULL;
        if (MyProperty->HasMetaData(NAME_ArraySizeEnum))
        {
            ArraySizeEnum = FindObject<UEnum>(NULL, *MyProperty-
>GetMetaData(NAME_ArraySizeEnum));
        }

        // Expand array.
        for( int32 Index = 0 ; Index < MyProperty->ArrayDim ; Index++ )
        {
            bool bShouldBeHidden = false;
            if( ArraySizeEnum )
            {
                // The enum at this array index is hidden
                bShouldBeHidden = ArraySizeEnum->HasMetaData(TEXT("Hidden"),
Index );
            }

            if( !bShouldBeHidden )
            {
                TSharedPtr<FItemPropertyNode> NewItemNode( new
FItemPropertyNode );

```

```

FPropertyNodeInitParams InitParams;
InitParams.ParentNode = SharedThis(this);
InitParams.Property = MyProperty;
InitParams.ArrayOffset = Index*MyProperty->ElementSize;
InitParams.ArrayIndex = Index;
InitParams.bAllowChildren = true;
InitParams.bForceHiddenPropertyVisibility =
bShouldShowHiddenProperties;
InitParams.bCreateDisableEditOnInstanceNodes =
bShouldShowDisableEditOnInstance;

NewItemNode->InitNode( InitParams );
AddChildNode(NewItemNode);
}
}
}
}

```

## TitleProperty

---

- **Function description:** Specifies the content of a structure member attribute within a structure array to serve as the display title for the array's elements.
- **Use location:** UPROPERTY
- **Engine module:** Container Property
- **Metadata type:** string = "abc"
- **Restriction type:** TArray
- **Commonly used:** ★★

Specifies the content of a structure member attribute within a structure array to serve as the display title for the array's elements.

## The Point Is:

---

- The target is a structure array TArray , other TSet , TMap are not supported.
- TitleProperty, as the name suggests, is the property used as a title. To be more specific, the title refers to the title displayed for each element in the structure array within the details panel. "Property" refers to the properties within the individual structures contained in the array.
- Then the next step is how to write the format of TitleProperty. According to the engine source code, TitleProperty finally uses FTitleMetadataFormatter to parse the calculation content. By looking at its internal code, we can see that its TitleProperty format can be:
  - If TitleProperty contains "{", the string inside will be treated as an FTextFormat (a format string organized with "{ArgName}..."). The final format is to use a string organized by "{PropertyName}..." to find multiple corresponding properties.
  - Otherwise, the entire TitleProperty will be treated as PropertyName, and the corresponding property name will be found directly.

## Test Code:

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyArrayTitleStruct
{
    GENERATED_BODY()
public:
    FMyArrayTitleStruct() = default;
    FMyArrayTitleStruct(int32 id) : MyInt(id) {}
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt = 123;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString=TEXT("Hello");
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat=456.f;
};

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = TitlePropertyTest)
TArray<FMyArrayTitleStruct> MyStructArray_NoTitleProperty;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = TitlePropertyTest, meta =
(TitleProperty="{MyString}[{MyInt}]"))
TArray<FMyArrayTitleStruct> MyStructArray_HasTitleProperty;
```

## Test Results:

It can be observed that the title of the array elements has changed to "Hello[x]" instead of the default "3 members".

Title Property Test			
▼ My Struct Array No Title Property	3 Array elements	⊕	↶
▶ Index [ 0 ]	3 members	⌄	↶
▶ Index [ 1 ]	3 members	⌄	↶
▶ Index [ 2 ]	3 members	⌄	↶
▼ My Struct Array Has Title Prop...	3 Array elements	⊕	↶
...▶ Index [ 0 ]	Hello[0]	⌄	↶
▶ Index [ 1 ]	Hello[1]	⌄	↶
▶ Index [ 2 ]	Hello[2]	⌄	↶

## Principle:

If a TitleProperty is found in the attribute node in the attribute editor, a TitlePropertyFormatter of FTitleMetadataFormatter will be generated to parse the string format and output the result content. In fact, what is actually used internally is FTextFormat, so that the contents of multiple attributes can be spliced into a target string.

Of course, SPropertyEditorTitle also noticed that if there is a TitleProperty, it may change in real time, so a function is bound to perform Tick update.

```

//Bind a function to retrieve the name every tick
void SPropertyEditorTitle::Construct( const FArguments& InArgs, const
TSharedRef<FPropertyEditor>& InPropertyEditor )
{
    // If our property has title support we want to fetch the value every tick,
otherwise we can just use a static value
    static const FName NAME_TitleProperty = FName(TEXT("TitleProperty"));
    const bool bHasTitleProperty = InPropertyEditor->GetProperty() &&
InPropertyEditor->GetProperty()->HasMetaData(NAME_TitleProperty);
    if (bHasTitleProperty)
    {
        NameTextBlock =
            SNew(STextBlock)
            .Text(InPropertyEditor, &FPropertyEditor::GetDisplayName)
            .Font(NameFont);
    }
    else
    {
        NameTextBlock =
            SNew(STextBlock)
            .Text(InPropertyEditor->GetDisplayName())
            .Font(NameFont);
    }
}

FText FPropertyEditor::GetDisplayName() const
{
    FItemPropertyParams* ItemPropertyParams =PropertyParams->AsPropertyParams();

    if (PropertyParams != NULL)
    {
        returnPropertyParams->GetDisplayName();
    }

    if (const FComplexPropertyParams* ComplexPropertyParams =PropertyParams-
>AsComplexPropertyParams())
    {
        const FText DisplayName = ComplexPropertyParams->GetDisplayName();

        // Does this property define its own name?
        if (!DisplayName.IsEmpty())
        {
            return DisplayName;
        }
    }

    FString DisplayName;
   PropertyParams->GetQualifiedName( DisplayName, true );
    return FText::FromString(DisplayName);
}

//Generate TitleFormatter to parse the content in TitleProperty and finally get
the text. It was found that Map and Set are not supported, so only array is
supported. The signature also has a branch that determines ArrayIndex()==1 and
goes into the ordinary attributes
FText FPropertyParams::GetDisplayName() const

```

```

{
    if (CastField<F SetProperty>(ParentProperty) == nullptr &&
        CastField<F MapProperty>(ParentProperty) == nullptr)
    {
        // Check if this property has Title Property Meta
        static const FName NAME_TitleProperty = FName(TEXT("TitleProperty"));
        FString TitleProperty = PropertyPtr->GetMetaData(NAME_TitleProperty);
        if (!TitleProperty.IsEmpty())
        {
            // Find the property and get the right property handle
            if (PropertyStruct != nullptr)
            {
                const TSharedPtr<IPROPERTYHandle> ThisAsHandle =
                    PropertyEditorHelpers::GetPropertyHandle(NonConstThis->AsShared(), nullptr,
                    nullptr);
                TSharedPtr<FTitleMetadataFormatter> TitleFormatter =
                    FTitleMetadataFormatter::TryParse(ThisAsHandle, TitleProperty);
                if (TitleFormatter)
                {
                    TitleFormatter->GetDisplayText(FinalDisplayName);
                }
            }
        }
    }

    //Generate a TitlePropertyFormatter
    void SPropertyEditorArrayItem::Construct( const FArguments& InArgs, const
    TSharedRef< class FPropertyEditor>& InPropertyEditor )
    {
        static const FName TitlePropertyFName = FName(TEXT("TitleProperty"));

        // if this is a struct property, try to find a representative element to use
        // as our stand in
        if (PropertyEditor->PropertyIsA( FStructProperty::StaticClass() ))
        {
            const FProperty* MainProperty = PropertyEditor->GetProperty();
            const FProperty* ArrayProperty = MainProperty ? MainProperty-
                >GetOwner<const FProperty>() : nullptr;
            if (ArrayProperty) // should always be true
            {
                TitlePropertyFormatter =
                    FTitleMetadataFormatter::TryParse(PropertyEditor->GetPropertyHandle(),
                    ArrayProperty->GetMetaData(TitlePropertyFName));
            }
        }
    }
}

```

Example in source code:

Readers can also find application examples in UPropertyEditorTestObject. You can open it with the testprops command line.

```

UPROPERTY(EditAnywhere, Category=ArraysOfProperties, meta=
(TitleProperty=IntPropertyInsideAStruct))
TArray<FPropertyEditorTestBasicStruct> StructPropertyArrayWithTitle;

UPROPERTY(EditAnywhere, Category=ArraysOfProperties, meta=(TitleProperty=""
{IntPropertyInsideAStruct} + {FloatPropertyInsideAStruct}"))
TArray<FPropertyEditorTestBasicStruct> StructPropertyArrayWithFormattedTitle;

UPROPERTY(EditAnywhere, Category=ArraysOfProperties, meta=
(TitleProperty=ErrorProperty))
TArray<FPropertyEditorTestBasicStruct> StructPropertyArrayWithTitleError;

UPROPERTY(EditAnywhere, Category=ArraysOfProperties, meta=(TitleProperty=""
{ErrorProperty}"))
TArray<FPropertyEditorTestBasicStruct>
StructPropertyArrayWithFormattedTitleError;

```

## EditFixedOrder

- **Function Description:** Prevents array elements from being reordered via dragging.
- **Usage Location:** UPROPERTY
- **Engine Module:** Container Property
- **Metadata Type:** bool
- **Restriction Type:** TArray
- **Commonliness:** ★★

## Test Code:

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest)
TArray<int32> MyIntArray_NoEditFixedorder;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest,
meta = (EditFixedorder))
TArray<int32> MyIntArray_EditFixedOrder;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest)
TSet<int32> MyIntSet_NoEditFixedOrder;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest,
meta = (EditFixedOrder))
TSet<int32> MyIntSet_EditFixedOrder;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest)
TMap<int32,FString> MyIntMap_NoEditFixedOrder;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = EditFixedOrderTest,
meta = (EditFixedOrder))
TMap<int32,FString> MyIntMap_EditFixedOrder;

```

## Test Effects:

- It can be seen that only the first MyIntArray\_NoEditFixedOrder appears on the array element as a draggable mark, and then the order can be changed.
- With the addition of EditFixedOrder's TArray, the order cannot be changed.
- Other types like TSet and TMap do not support this meta, as their internal order is inherently irrelevant.

Edit Fixed Order Test				
▼ My Int Array No Edit Fixed Order	3 Array elements	<input type="button" value="⊕"/>	<input type="button" value="⊖"/>	<input type="button" value="↶"/>
Index [ 0 ]	0	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
Index [ 1 ]	1	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
Index [ 2 ]	2	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
▼ My Int Array Edit Fixed Order	3 Array elements	<input type="button" value="⊕"/>	<input type="button" value="⊖"/>	<input type="button" value="↶"/>
Index [ 0 ]	0	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
Index [ 1 ]	1	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
Index [ 2 ]	2	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
▼ My Int Set No Edit Fixed Order	3 Set elements	<input type="button" value="⊕"/>	<input type="button" value="⊖"/>	<input type="button" value="↶"/>
Index [ 0 ]	0	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
Index [ 1 ]	1	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
Index [ 2 ]	2	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
▼ My Int Set Edit Fixed Order	3 Set elements	<input type="button" value="⊕"/>	<input type="button" value="⊖"/>	<input type="button" value="↶"/>
Index [ 0 ]	0	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
Index [ 1 ]	1	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
Index [ 2 ]	2	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
▼ My Int Map No Edit Fixed Order	3 Map elements	<input type="button" value="⊕"/>	<input type="button" value="⊖"/>	<input type="button" value="↶"/>
0	Hello	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
1	Hello	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
2	Hello	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
▼ My Int Map Edit Fixed Order	3 Map elements	<input type="button" value="⊕"/>	<input type="button" value="⊖"/>	<input type="button" value="↶"/>
0	Hello	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
1	Hello	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>
2	Hello	<input type="button" value="▼"/>	<input type="button" value="↶"/>	<input type="button" value="↷"/>

## Principle:

As you can see, the judgment of whether an attribute row in the details panel is reorderable is that the outer part is an array, and there is no EditFixedOrder and ArraySizeEnum (fixed array). Of course, this attribute itself must also be in an editable state (for example, if it is disabled and grayed out, it will obviously not be editable)

```

bool FPropertyNode::IsReorderable()
{
    FProperty* NodeProperty = GetProperty();
    if (NodeProperty == nullptr)
    {
        return false;
    }
    // It is reorderable if the parent is an array and metadata doesn't prohibit
    it
    const FArrayProperty* OuterArrayProp = NodeProperty->GetOwner<FArrayProperty>
());
}

static const FName Name_DisableReordering("EditFixedOrder");
static const FName NAME_ArraySizeEnum("ArraySizeEnum");
return OuterArrayProp != nullptr
    && !OuterArrayProp->HasMetaData(Name_DisableReordering)
    && !IsEditConst()
    && !OuterArrayProp->HasMetaData(NAME_ArraySizeEnum)
    && !FApp::IsGame();
}

```

## NoElementDuplicate

---

- **Function Description:** Removes the Duplicate menu item button for elements within a TArray attribute.
- **Usage Location:** UPROPERTY
- **Engine Module:** Container Property
- **Metadata Type:** bool
- **Restriction Type:** TArray
- **Commonliness:** ★

Removes the Duplicate menu item button for elements within a TArray attribute.

Applied to TArray attributes, where the value can be of any type, including numbers, structures, or Object\*.

## Test Code:

---

```

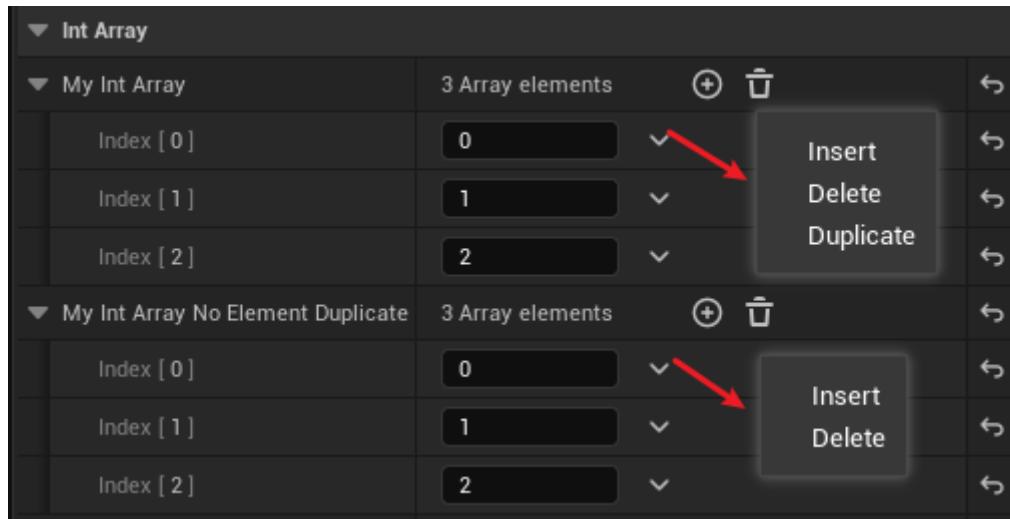
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = IntArray)
    TArray<int32> MyIntArray;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = IntArray, meta =
    (NoElementDuplicate))
    TArray<int32> MyIntArray_NoElementDuplicate;

```

## Effect:

You can observe that for arrays with NoElementDuplicate, the dropdown menu to the right of the value only contains two items.



## Principle:

If the condition of "NoElementDuplicate" is met, the system will generate only the "Insert\_Delete" menu; otherwise, the default "Insert\_Delete\_Duplicate" menu will be used. It is imperative that the current attribute in question is an array attribute and not of the "EditFixedSize" fixed-size type.

```
void GetRequiredPropertyButtons( TSharedRef<FPropertyNode> PropertyNode,
TArray<EPropertyButton::Type>& OutRequiredButtons, bool bUsingAssetPicker )
{
    const FArrayProperty* OuterArrayProp = NodeProperty-
>GetOwner<FArrayProperty>();

    if( OuterArrayProp )
    {
        if( PropertyNode->HasNodeFlags(EPropertyNodeFlags::singleSelectOnly)
&& !(OuterArrayProp->PropertyFlags & CPF_EditFixedSize) )
        {
            if (OuterArrayProp->HasMetaData(TEXT("NoElementDuplicate")))
            {
                OutRequiredButtons.Add( EPropertyButton::Insert_Delete );
            }
            else
            {
                OutRequiredButtons.Add(
EPropertyButton::Insert_Delete_Duplicate );
            }
        }
    }
}
```

# DebugTreeLeaf

- **Function Description:** Prevent the BlueprintDebugger from expanding the properties of this class to enhance the debugger's performance in the editor
- **Usage Location:** UCLASS
- **Engine Module:** Debug
- **Metadata Type:** bool
- **Commonliness:** ★

Prevents the BlueprintDebugger from expanding the properties of this class to enhance the debugger's performance in the editor. When a class possesses an excessive number of properties (or has too many properties with recursive nesting), the BlueprintDebugger consumes significant performance resources when displaying the class's property data, leading to editor lag. Therefore, for such classes, we can manually add this flag to halt the further expansion of the property tree, stopping at this point. As the name implies, this class becomes the leaf of the property tree during debugging.

In the source code, only UAnimDataModel utilizes this tag, but it can also be added to our custom classes when they have a large number of properties and we do not wish to debug their data.

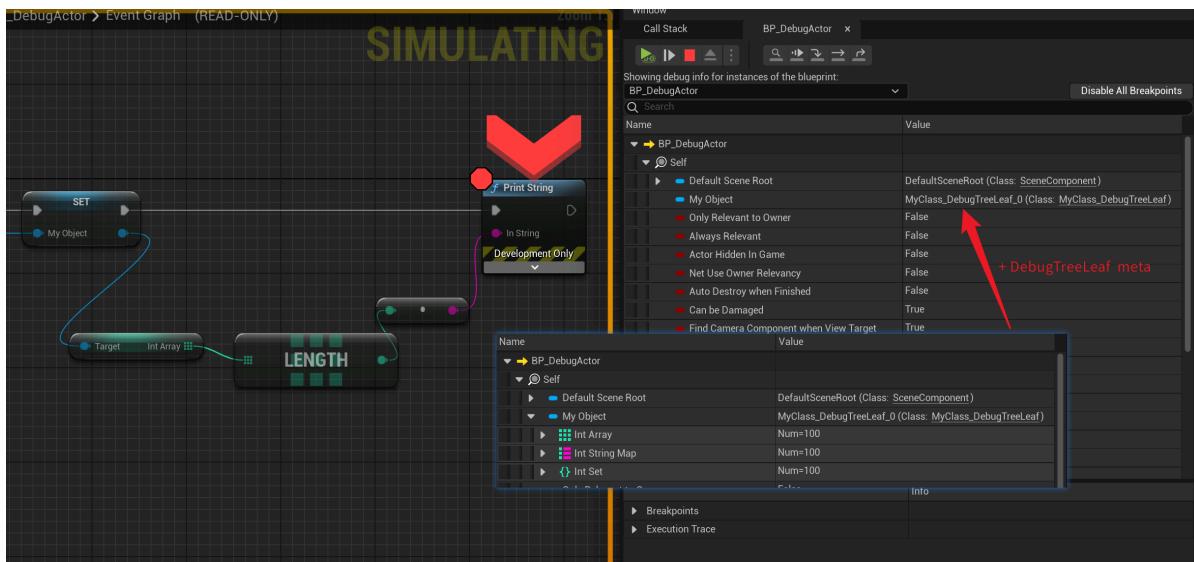
## Test Code:

```
UCLASS(BlueprintType, meta = (DebugTreeLeaf))
class INSIDER_API UMyClass_DebugTreeLeaf :public UObject
{
    GENERATED_BODY()
    UMyClass_DebugTreeLeaf();

public:
    UPROPERTY(BlueprintReadWrite)
    TArray<int32> IntArray;
    UPROPERTY(BlueprintReadWrite)
    TMap<int32, FString> IntStringMap;
    UPROPERTY(BlueprintReadWrite)
    TSet<int32> IntSet;
};
```

## Effect in Blueprint:

UMyClass\_DebugTreeLeaf object, when used as a class member variable (or otherwise), will behave differently during debugging in the blueprint. Without the DebugTreeLeaf flag, all internal properties are expanded by default when viewing variable attributes with BlueprintDebugger. However, with the DebugTreeLeaf flag added, the recursion is stopped, preventing the expansion of attribute variables.



## HideInDetailPanel

- Function Description:** Hides the dynamic multicast delegate attribute within the Actor's event panel.
- Usage Location:** UPROPERTY
- Engine Module:** DetailsPanel
- Metadata Type:** bool
- Restriction Type:** Dynamic multicast delegate within an Actor
- Commonality:** ★★

Hides the dynamic multicast delegate property in the Actor's event panel.

## Test Code:

```
UCLASS(BlueprintType, Blueprintable)
class INSIDER_API AMyProperty_HideInDetailPanel : public AActor
{
    GENERATED_BODY()
public:
    DECLARE_DYNAMIC_MULTICAST_DELEGATE(FOnMyHideTestEvent);

    UPROPERTY(BlueprintAssignable, Category = "Event")
    FOnMyHideTestEvent MyEvent;

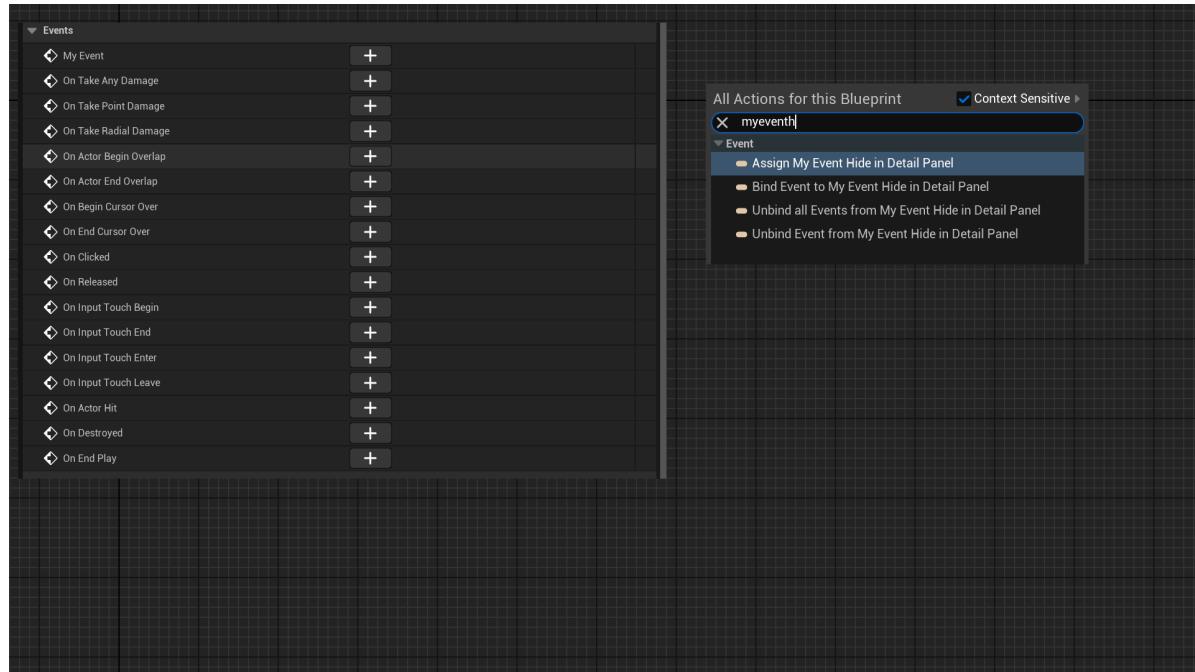
    UPROPERTY(BlueprintAssignable, Category = "Event", meta =
(HideInDetailPanel))
    FOnMyHideTestEvent MyEvent_HideInDetailPanel;
};
```

## Test Effects:

The test procedure involves creating a subclass of AMyProperty\_HideInDetailPanel in the Blueprint and observing the display of the Event.

Visible MyEvent is displayed in the Events section of Class Defaults, whereas MyEvent\_HideInDetailPanel is not shown.

However, MyEvent\_HideInDetailPanel can still be bound within the Blueprint; it's just not displayed in the UI by default.



## Principle:

The system first checks for the absence of this marker before creating the corresponding UI controls.

```
void FActorDetails::AddEventsCategory(IDetailLayoutBuilder& DetailBuilder)
{
    IDetailCategoryBuilder& EventsCategory =
DetailBuilder.EditCategory("Events", FText::GetEmpty(),
ECategoryPriority::Uncommon);
    static const FName HideInDetailPanelName("HideInDetailPanel");

    // Find all the Multicast delegate properties and give a binding button
    // for them
    for (TFieldIterator<FMulticastDelegateProperty> PropertyIt(Actor-
>GetClass(), EFieldIteratorFlags::IncludeSuper); PropertyIt; ++PropertyIt)
    {
        FMulticastDelegateProperty* Property = *PropertyIt;

        // Only show BP assiangular, non-hidden delegates
        if (!Property->HasAnyPropertyFlags(CPF_Parm) && Property-
>HasAllPropertyFlags(CPF_BlueprintAssignable) && !Property-
>HasMetaData(HideInDetailPanelName))
            {}
    }
}

void FBlueprintDetails::AddEventsCategory(IDetailLayoutBuilder& DetailBuilder,
FName PropertyName, UClass*> PropertyClass)
{
    static const FName HideInDetailPanelName("HideInDetailPanel");
```

```
// Check for multicast delegates that we can safely assign
if ( !Property->HasAnyPropertyFlags(CPF_Parm) && Property-
>HasAllPropertyFlags(CPF_BlueprintAssignable) &&
    !Property->HasMetaData(HideInDetailPanelName) )
}
```

## DisplayAfter

- **Function description:** Ensures this attribute is displayed after a specified attribute.
- **Use location:** UPROPERTY
- **Engine module:** DetailsPanel
- **Metadata type:** string="abc"
- **Commonly used:** ★★★

Ensures this attribute is displayed after the specified attribute.

- By default, the order of properties in the details panel follows the sequence defined in the header file. However, if we wish to customize this order, this attribute can be used.
- The constraint is that both attributes must belong to the same Category. This is logical, as the Category's organization takes precedence.

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Priority :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = AfterTest)
    int32 MyInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = AfterTest)
    FString MyString;

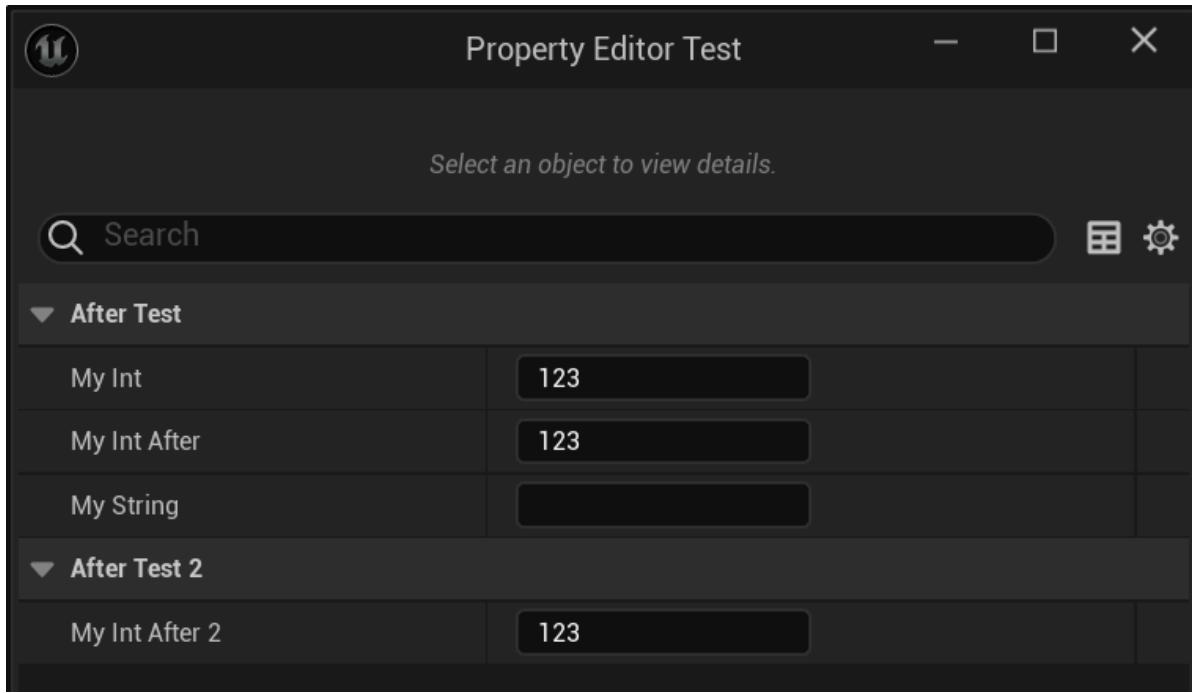
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = AfterTest, meta =
(DisplayAfter = "MyInt"))
    int32 MyInt_After = 123;
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = AfterTest2, meta =
(DisplayAfter = "MyInt"))
    int32 MyInt_After2 = 123;

};
```

## Test Results:

Visible MyInt\_After is displayed directly after Int.

But MyInt\_After2, being in a different Category, remains unchanged.



## Principle:

Checks if the property has a DisplayAfter attribute and, if so, inserts it after the specified property.

```
void PropertyEditorHelpers::OrderPropertiesFromMetadata(TArray<FProperty*>& Properties)
{
    const FString& DisplayAfterPropertyName = Prop->GetMetaData(NAME_DisplayAfter);
    if (DisplayAfterPropertyName.IsEmpty())
    {
        InsertProperty(OrderedProperties);
    }
    else
    {
        TArray<TPair<FProperty*, int32>>& DisplayAfterProperties =
        DisplayAfterPropertyMap.FindOrAdd(FName(*DisplayAfterPropertyName));
        InsertProperty(DisplayAfterProperties);
    }
}
```

## EditCondition

- **Function Description:** Specify another attribute or an expression as a condition for the editability of a given attribute.
- **Usage Location:** UPROPERTY
- **Engine Module:** DetailsPanel
- **Metadata Type:** string="abc"
- **Associated Items:** EditConditionHides, InlineEditConditionToggle, HideEditConditionToggle
- **Commonality:** ★★★★☆

An attribute's editability is determined by specifying another attribute or an expression as its condition.

- Properties referenced in the expression must be within the same class or structure scope.

## Test Code:

```

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_EditCondition_Test :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property)
    bool MyBool;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property)
    int32 MyInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property, meta =
    (EditCondition = "MyBool"))
    int32 MyInt_EditCondition = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property, meta =
    (EditCondition = "!MyBool"))
    int32 MyInt_EditCondition_Not = 123;

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression)
    int32 MyFirstInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression)
    int32 MySecondInt = 123;

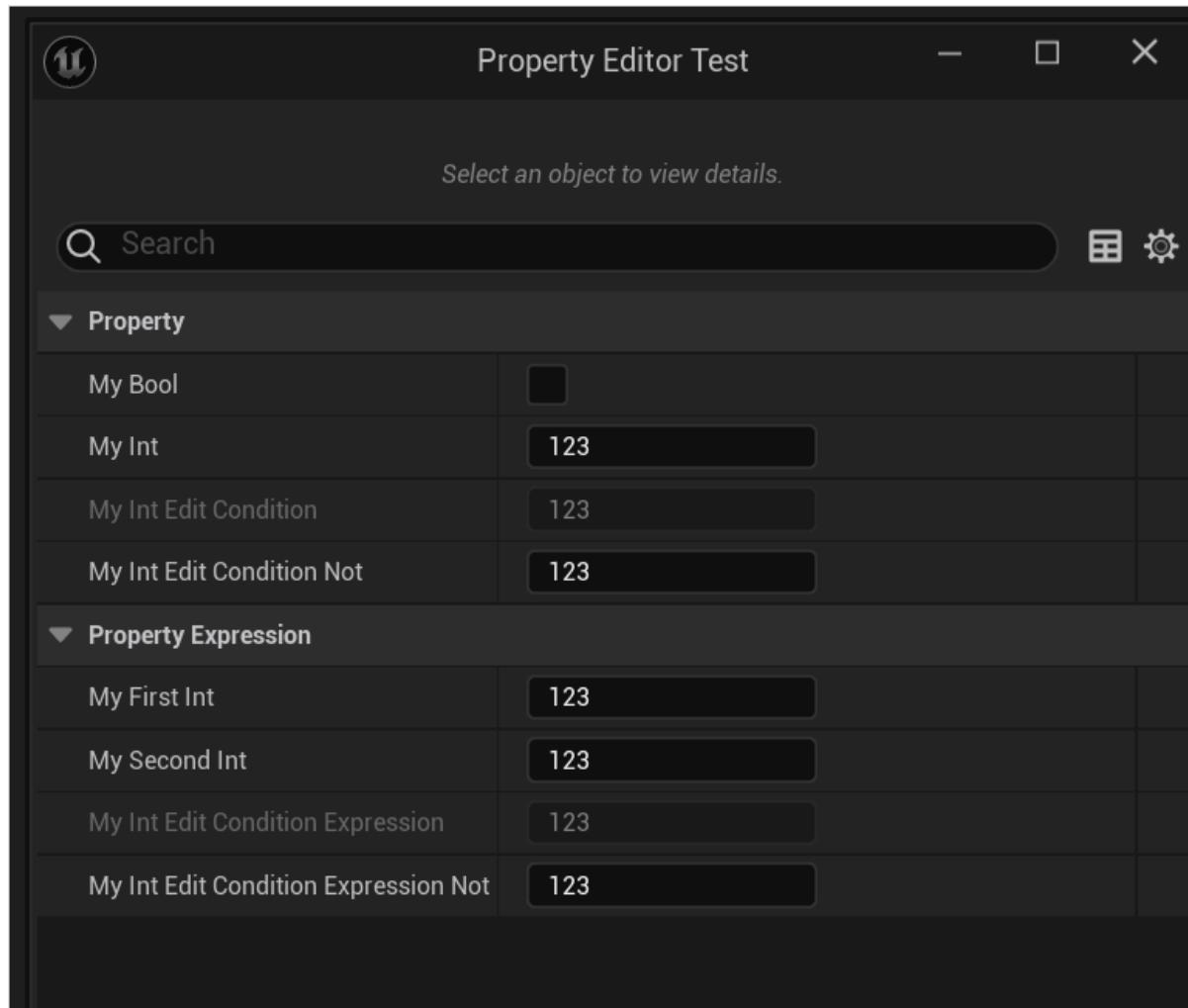
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression,
    meta = (EditCondition = "(MyFirstInt+MySecondInt)==500"))
    int32 MyInt_EditConditionExpression = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression,
    meta = (EditCondition = "!(MyFirstInt+MySecondInt)==500"))
    int32 MyInt_EditConditionExpression_Not = 123;
};

```

## Test Results:

- Editability of other properties can be controlled by a single boolean property
- Complex decision mechanisms for editability can be introduced through an expression.



## Principle:

During the initialization of properties in the Details Panel, the `EditCondition` setting for the property is evaluated. If a value is present, a `FEditConditionParser` is created to parse the expression and then its value is computed.

```
void FPropertyNode::InitNode(const FPropertyParams& InitParams)
{
    const FString& EditConditionString = MyProperty-
>GetMetaData(TEXT("EditCondition"));

    // see if the property supports some kind of edit condition and this isn't
    // the "parent" property of a static array
    const bool bIsStaticArrayParent = MyProperty->ArrayDim > 1 && GetArrayIndex()
!= -1;
    if (!EditConditionString.IsEmpty() && !bIsStaticArrayParent)
    {
        EditConditionExpression = EditconditionParser.Parse(EditConditionString);
        if (EditConditionExpression.IsValid())
        {
            EditConditionContext = MakeShareable(new
FEditConditionContext(*this));
        }
    }
}
```

# EditConditionHides

- **Function Description:** When an EditCondition already exists, this attribute specifies that it should be hidden if the EditCondition is not met.
- **Usage Location:** UPROPERTY
- **Metadata Type:** bool
- **Associated Item:** EditCondition
- **Commonality:** ★★★★☆

With an existing EditCondition, this attribute is specified to be hidden if the EditCondition is not satisfied.

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_EditCondition_Test :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property)
    bool MyBool;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Property, meta =
    (EditConditionHides, EditCondition = "MyBool"))
    int32 MyInt_EditCondition_Hides = 123;

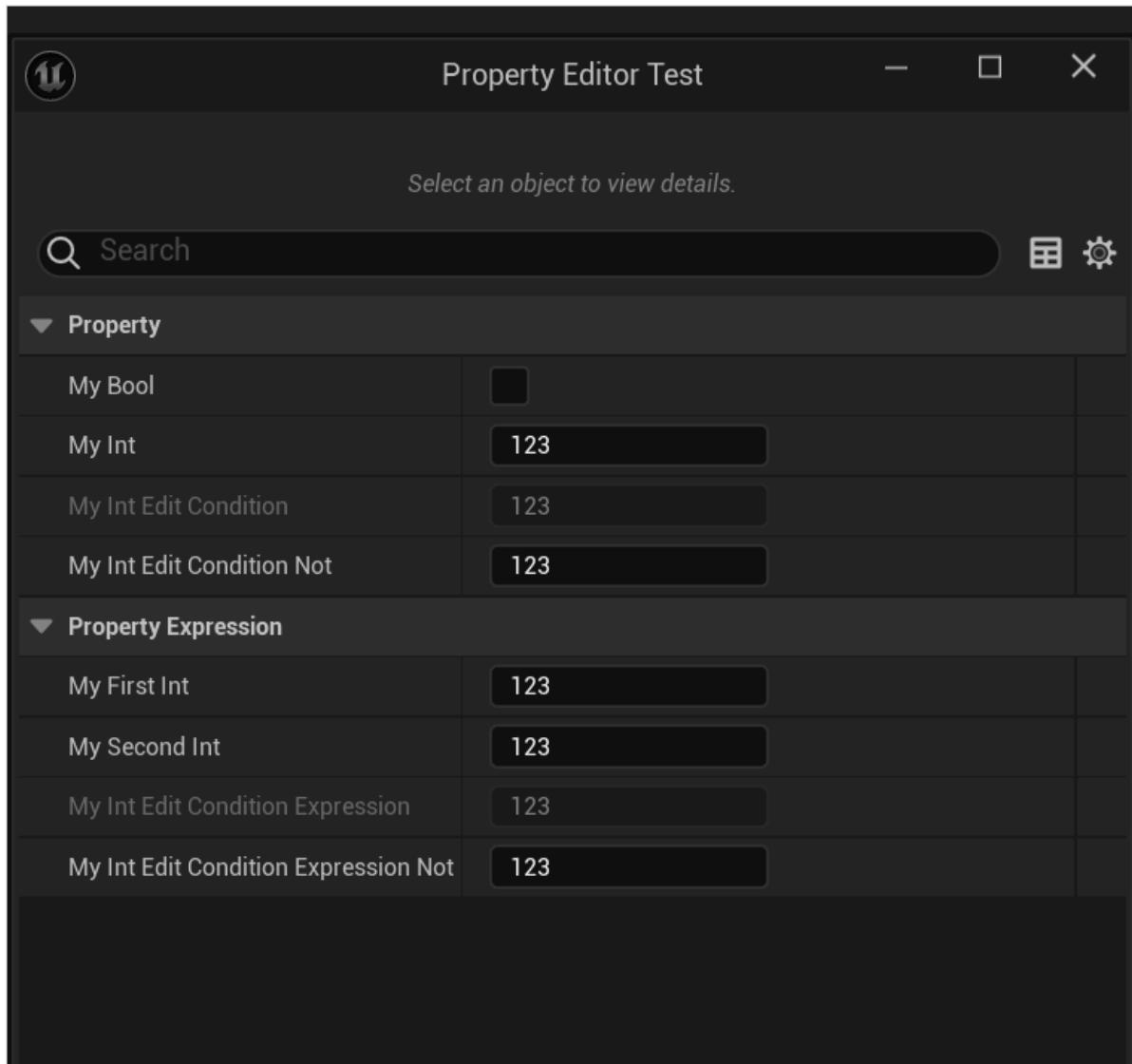
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression)
    int32 MyFirstInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression)
    int32 MySecondInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PropertyExpression,
    meta = (EditConditionHides, EditCondition = "(MyFirstInt+MySecondInt)==500"))
    int32 MyInt_EditConditionExpression_Hides = 123;
};
```

## Test Effects:

The figure below clearly shows that two attributes are displayed as the conditions are met.



## Principle:

Effectively, it adds a conditional check for visibility.

```
bool FPropertyNode::IsOnlyVisibleWhenEditConditionMet() const
{
    static const FName Name_EditConditionHides("EditConditionHides");
    if (Property.IsValid() && Property->HasMetaData(Name_EditConditionHides))
    {
        return HasEditCondition();
    }

    return false;
}
```

## InlineEditConditionToggle

- **Function description:** This feature allows the bool attribute to be integrated into the attribute field of another entity as a radio button when used as an EditCondition, rather than appearing as a separate edit field.
- **Usage Location:** UPROPERTY
- **Metadata Type:** bool

- **Restriction Type:** bool
- **Associated Items:** EditCondition
- **Commonality:** ★★★★☆

When used as an EditCondition, this bool attribute should be inlined within the attribute row of the other entity, presented as a radio button rather than forming its own edit row.

While the EditCondition is capable of handling other types of attributes or expressions, the InlineEditConditionToggle is specifically designed to work with bool attributes only.

## Test Code:

```
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    InlineEditConditionToggle, meta = (InlineEditConditionToggle))
    bool MyBool_Inline;

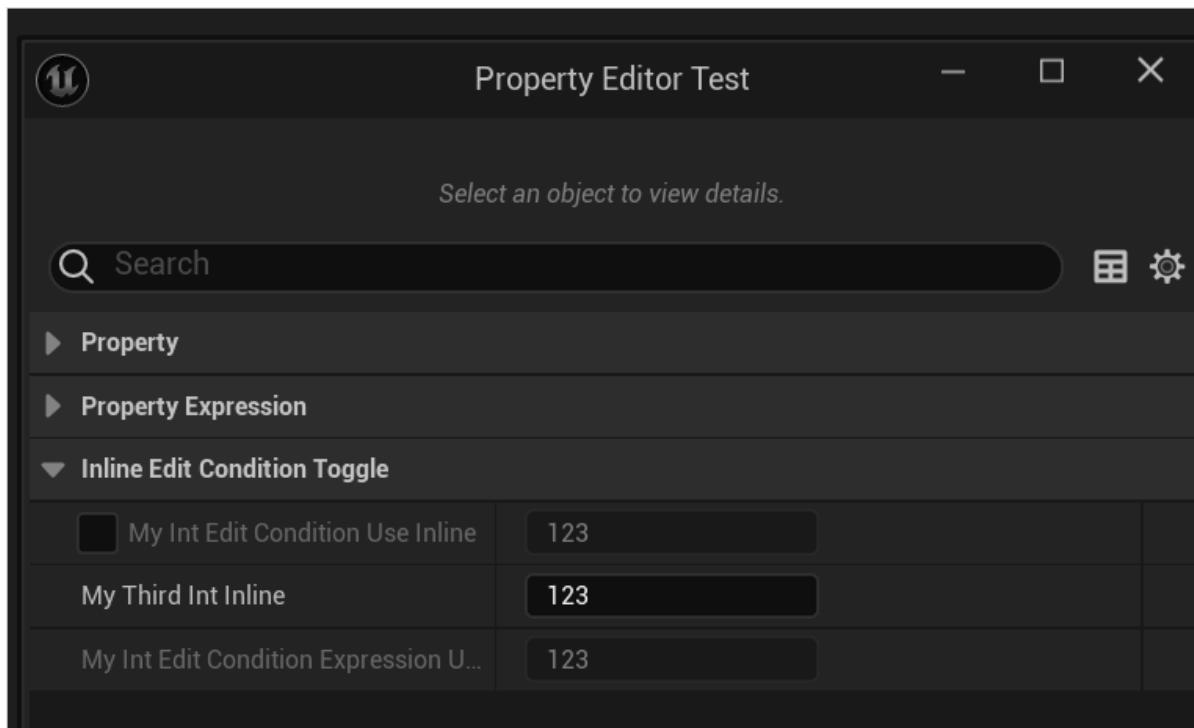
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    InlineEditConditionToggle, meta = (EditCondition = "MyBool_Inline"))
    int32 MyInt_EditCondition_UseInline = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    InlineEditConditionToggle)
    int32 MyThirdInt_Inline = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
    InlineEditConditionToggle, meta = (EditCondition = "MyThirdInt_Inline>200"))
    int32 MyInt_EditConditionExpression_UseInline = 123;
```

## Test Results:

As a result, MyBool\_Inline is now presented as a radio button. In contrast, MyThirdInt\_Inline remains visible and is not concealed.



# Principle:

It can be observed that this is used to determine whether a radio button is supported for display.

```
bool FPropertyNode::SupportsEditConditionToggle() const
{
    if (!Property.IsValid())
    {
        return false;
    }

    FProperty* MyProperty = Property.Get();

    static const FName Name_HideEditConditionToggle("HideEditConditionToggle");
    if (EditConditionExpression.IsValid() && !Property->HasMetaData(Name_HideEditConditionToggle))
    {
        const FBoolProperty* ConditionalProperty = EditConditionContext->GetSingleBoolProperty(EditConditionExpression);
        if (ConditionalProperty != nullptr)
        {
            // There are 2 valid states for inline edit conditions:
            // 1. The property is marked as editable and has
            InLineEditConditionToggle set.
            // 2. The property is not marked as editable and does not have
            InLineEditConditionToggle set.
            // In both cases, the original property will be hidden and only show
            up as a toggle.

            static const FName
Name_InlineEditConditionToggle("InlineEditConditionToggle");
            const bool bIsInlineEditCondition = ConditionalProperty->HasMetaData(Name_InlineEditConditionToggle);
            const bool bIsEditable = ConditionalProperty->HasAllPropertyFlags(CPF_Edit);

            if (bIsInlineEditCondition == bIsEditable)
            {
                return true;
            }

            if (bIsInlineEditCondition && !bIsEditable)
            {
                UE_LOG(LogPropertyName, Warning, TEXT("Property being used as
inline edit condition is not editable, but has redundant
InlineEditConditionToggle flag. Field \"%s\" in class \"%s\"."),
*ConditionalProperty->GetNameCPP(), *Property->GetOwnerStruct()->GetName());
                return true;
            }

            // The property is already shown, and not marked as inline edit
            condition.
            if (!bIsInlineEditCondition && bIsEditable)
            {

```

```

        return false;
    }
}

return false;
}

```

## HideEditConditionToggle

- **Function Description:** Used on attributes utilizing EditCondition, indicating that the attribute does not wish the properties involved in its EditCondition to be hidden.
- **Usage Location:** UPROPERTY
- **Engine Module:** DetailsPanel
- **Metadata Type:** bool
- **Restriction Type:** bool
- **Associated Items:** EditCondition
- **Commonality:** ★★★★☆

Applied to properties that use EditCondition, signifying that the property does not want the properties referenced by its EditCondition to be concealed. This has an opposite effect to InlineEditConditionToggle.

### Test Code:

```

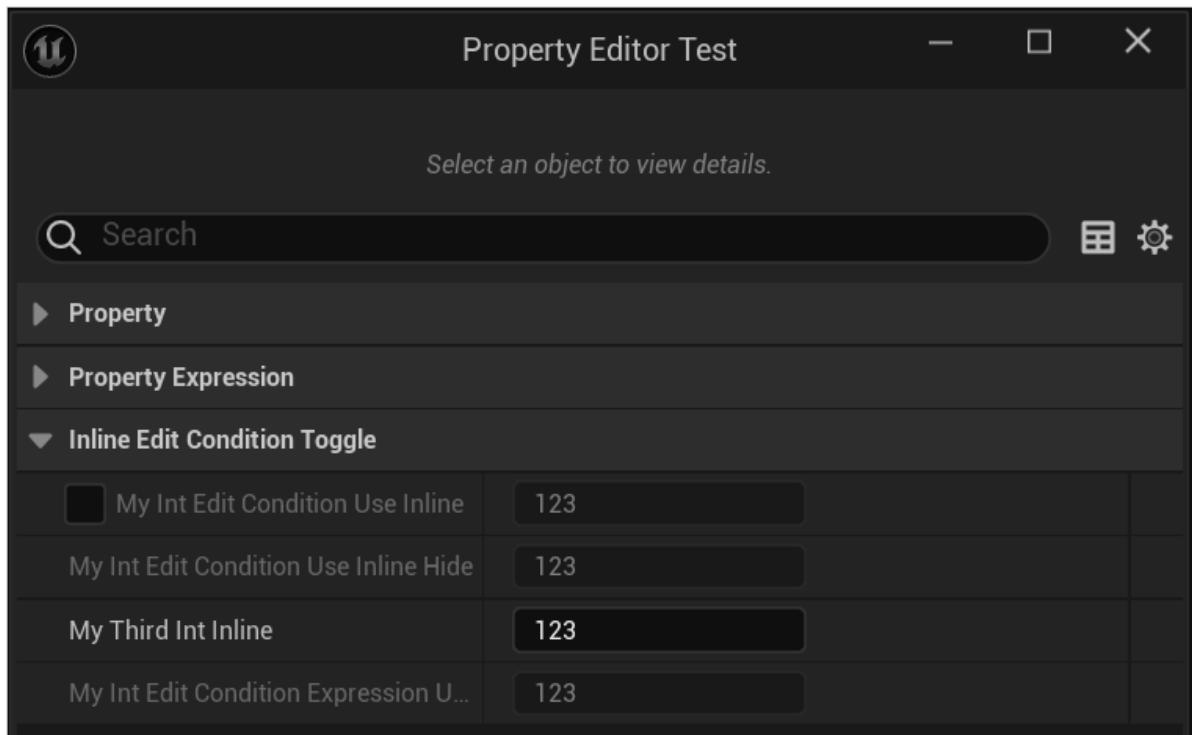
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
InlineEditConditionToggle, meta = (InlineEditConditionToggle))
    bool MyBool_Inline;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
InlineEditConditionToggle, meta = (EditCondition = "MyBool_Inline"))
    int32 MyInt_EditCondition_UseInline = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
InlineEditConditionToggle, meta = (HideEditConditionToggle, EditCondition =
"MyBool_Inline"))
    int32 MyInt_EditCondition_UseInline_Hide = 123;
};

```

## Testing Code:



## 测试效果:

Decide whether the current row supports a button with a radio button if a `HideEditConditionToggle` is present.

```
bool FPropertyNode::SupportsEditConditionToggle() const
{
    if (!Property.IsValid())
    {
        return false;
    }

    FProperty* MyProperty = Property.Get();

    static const FName Name_HideEditConditionToggle("HideEditConditionToggle");
    if (EditConditionExpression.IsValid() && !Property->HasMetaData(Name_HideEditConditionToggle))
    {
        const FBoolProperty* ConditionalProperty = EditConditionContext->GetSingleBoolProperty(EditConditionExpression);
        if (ConditionalProperty != nullptr)
        {
            // There are 2 valid states for inline edit conditions:
            // 1. The property is marked as editable and has
            //     InLineEditConditionToggle set.
            // 2. The property is not marked as editable and does not have
            //     InLineEditConditionToggle set.
            // In both cases, the original property will be hidden and only show
            // up as a toggle.
        }
    }
}
```

```

        static const FName
Name_InlineEditConditionToggle("InlineEditConditionToggle");
        const bool bIsInlineEditCondition = ConditionalProperty-
>HasMetaData(Name_InlineEditConditionToggle);
        const bool bIsEditable = ConditionalProperty-
>HasAllPropertyFlags(CPF_Edit);

        if (bIsInlineEditCondition == bIsEditable)
{
    return true;
}

        if (bIsInlineEditCondition && !bIsEditable)
{
    UE_LOG(LogPropertyName, Warning, TEXT("Property being used as
inline edit condition is not editable, but has redundant
InlineEditConditionToggle flag. Field \\\"%s\\\" in class \\\"%s\\\"."),
*ConditionalProperty->GetNameCPP(), *Property->GetOwnerStruct()->GetName());
    return true;
}

// The property is already shown, and not marked as inline edit
condition.
        if (!bIsInlineEditCondition && bIsEditable)
{
    return false;
}
}

return false;
}

```

## DisplayPriority

---

- **Function Description:** Specifies the display order priority of this attribute within the details panel; lower values indicate higher priority.
- **Usage Location:** UPROPERTY
- **Engine Module:** DetailsPanel
- **Metadata Type:** int32
- **Commonly Used:** ★★★

Specifies the display order priority of this attribute within the details panel; lower values indicate higher priority.

- If there is a setting for DisplayAfter, it takes precedence over the priority value.
- The same restriction applies within the same Category.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PriorityTest, meta =  
(DisplayPriority = 3))  
        int32 MyInt_P3 = 123;  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PriorityTest, meta =  
(DisplayPriority = 1))  
        int32 MyInt_P1 = 123;  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PriorityTest, meta =  
(DisplayPriority = 2))  
        int32 MyInt_P2 = 123;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PriorityTest, meta =  
(DisplayPriority = 4, DisplayAfter="MyInt_P1"))  
        int32 MyInt_P4 = 123;
```

## Test Results:

Even with a lower priority, P4 is still placed after P1 due to the DisplayAfter setting.

▼ Priority Test	
My Int P1	123
My Int P4	123
My Int P2	123
My Int P3	123

## Principle:

The sorting logic is contained within this function; a simple insertion sort algorithm is employed.

```
void PropertyEditorHelpers::OrderPropertiesFromMetadata(TArray<FProperty*>&  
Properties)  
{}
```

## AdvancedClassDisplay

- **Function Description:** Specifies that variables of this type should be displayed in an advanced display context
- **Usage Location:** UCLASS
- **Engine Module:** DetailsPanel
- **Metadata Type:** bool
- **Associated Items:**
  - UCLASS: AdvancedClassDisplay
- **Commonality:** ★★★

# bShowOnlyWhenTrue

- **Function Description:** Determines the display of the current attribute based on the field value within the editor's configuration file.
- **Usage Location:** UPROPERTY
- **Engine Module:** DetailsPanel
- **Metadata Type:** string="abc"
- **Commonly Used:** ★

Based on the field value in the editor's configuration file, the display of the current attribute is decided.

- This editor's configuration file refers to GEditorPerProjectIni, typically located at Config\DefaultEditorPerProjectUserSettings.ini
- The section name is "UnrealEd.PropertyFilters"
- The value for the Key can then be defined.

No examples were found in the source code, but this feature still functions correctly.

## Test Code:

```
D:\github\Gitworkspace\Hello\Config\DefaultEditorPerProjectUserSettings.ini
[UnrealEd.PropertyFilters]
ShowMyInt=true
ShowMyString=false

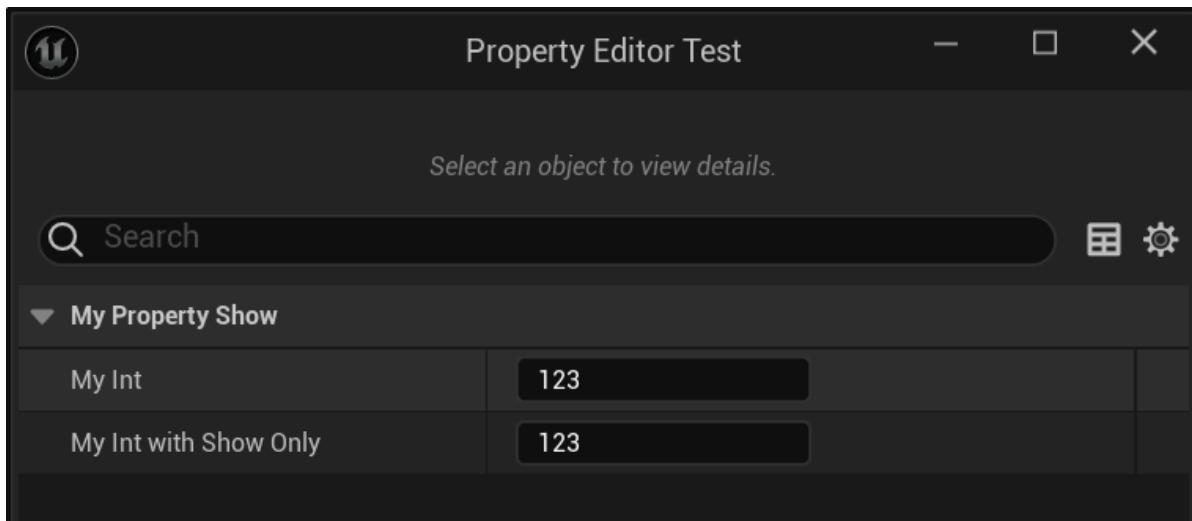
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Show :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyInt = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (bShowOnlyWhenTrue =
"ShowMyInt"))
    int32 MyInt_WithShowOnly = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (bShowOnlyWhenTrue =
"ShowMyString"))
    FString MyString_WithShowOnly;
};
```

## Test Results:

The property MyString\_WithShowOnly is not displayed because we set ShowMyString=false in DefaultEditorPerProjectUserSettings.



## Principle:

It involves retrieving the value from the configuration to decide whether the property box should be displayed.

```
void FObjectPropertyNode::GetCategoryProperties(const TSet<UClass*>& ClassesToConsider, const FProperty* CurrentProperty, bool bShouldShowDisableEditOnInstance, bool bShouldShowHiddenProperties, const TSet<FName>& CategoriesFromBlueprints, TSet<FName>& CategoriesFromProperties, TArray<FName>& SortedCategories)
{
    bool bMetaDataAllowVisible = true;
    const FString& ShowOnlyWhenTrueString = CurrentProperty->GetMetaData(Name_bShowOnlyWhenTrue);
    if (ShowOnlyWhenTrueString.Len())
    {
        //ensure that the metadata visibility string is actually set to true in
        //order to show this property
        GConfig->GetBool(TEXT("UnrealEd.PropertyFilters"),
*ShowOnlyWhenTrueString, bMetaDataAllowVisible, GEditorPerProjectIni);
    }

    if (bMetaDataAllowVisible)
    {
        if (PropertyEditorHelpers::ShouldBeVisible(*this, CurrentProperty) &&
!HiddenCategories.Contains(CategoryName))
        {
            if (!CategoriesFromBlueprints.Contains(CategoryName) &&
!CategoriesFromProperties.Contains(CategoryName))
            {
                SortedCategories.AddUnique(CategoryName);
            }
            CategoriesFromProperties.Add(CategoryName);
        }
    }
}

void FCategoryPropertyNode::InitChildNodes()
{
```

```

        bool bMetaDataAllowVisible = true;
        if (!bShowHiddenProperties)
        {
            static const FName
Name_bShowOnlyWhenTrue("bShowOnlyWhenTrue");
            const FString& MetaDataVisibilityCheckString = IT-
>GetMetaData(Name_bShowOnlyWhenTrue);
            if (MetaDataVisibilityCheckString.Len())
            {
                //ensure that the metadata visibility string is
                //actually set to true in order to show this property
                // @todo Remove this
                GConfig->GetBool(TEXT("UnrealEd.PropertyFilters"),
*MetaDataVisibilityCheckString, bMetaDataAllowVisible, GEditorPerProjectIni);
            }
        }
    }
}

```

## PrioritizeCategories

---

- **Function Description:** Prioritize the display of the specified attribute directory
- **Usage Location:** UCLASS
- **Engine Module:** DetailsPanel
- **Metadata Type:** strings = "a, b, c"
- **Associated Items:**  
UCLASS: PrioritizeCategories
- **Commonality:** ★★★

## AutoExpandCategories

---

- **Function Description:** Specifies that the attribute directory within a class should automatically expand
- **Usage Location:** UCLASS
- **Engine Module:** DetailsPanel
- **Metadata Type:** strings = "a, b, c"
- **Associated Items:**  
UCLASS: AutoExpandCategories, AutoCollapseCategories
- **Commonly Used:** ★★★

## AutoCollapseCategories

---

- **Function Description:** Automatically collapses the attribute categories within a specified class
- **Usage Location:** UCLASS
- **Engine Module:** DetailsPanel
- **Metadata Type:** strings = "a, b, c"

- **Associated Items:**

UCLASS: AutoCollapseCategories, DontAutoCollapseCategories, AutoExpandCategories

- **Commonality:** ★★★

## ClassGroupNames

---

- **Function Description:** Specifies the names for ClassGroups

- **Usage Location:** UCLASS

- **Engine Module:** DetailsPanel

- **Metadata Type:** strings = "a, b, c"

- **Restriction Type:** TArray

- **Associated Items:**

UCLASS: ClassGroup

- **Commonly Used:** ★★★

## MaxPropertyDepth

---

- **Function Description:** Specifies the depth of levels to expand an object or structure within the Details Panel.

- **Usage Location:** UPROPERTY

- **Engine Module:** DetailsPanel

- **Metadata Type:** int32

- **Restriction Type:** Object or structure property

- **Commonliness:** ★

Specifies the depth of levels at which the object or structure is expanded in the Details Panel.

- By default, there is no limit, allowing for expansion recursively down to the deepest nested fields.
- If an object has sub-objects that also have sub-objects, this recursive expansion can go many levels deep, and we may wish to limit the depth to prevent excessive expansion. Thus, we can specify a depth limit.
- A value of -1 indicates no limit, 0 means no expansion at all, and >0 denotes the number of layers to be restricted.
- No examples were found in the source code, but this feature is functional.

## Test Code:

---

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyStructDepth1
{
    GENERATED_BODY()

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt1 = 123;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
```

```
FString MyString1;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStructDepth2
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
        FMyStructDepth1 MyStruct1;
};

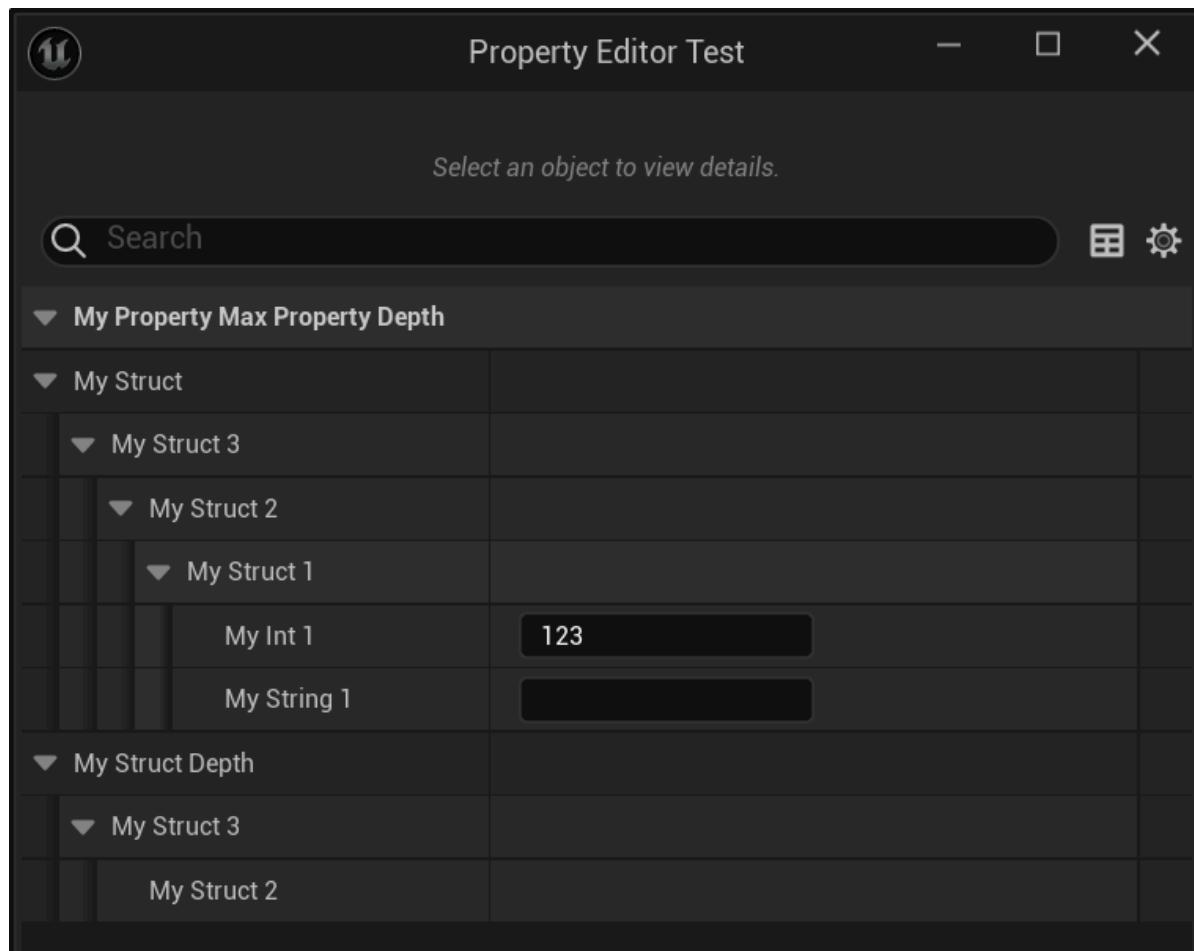
USTRUCT(BlueprintType)
struct INSIDER_API FMyStructDepth3
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
        FMyStructDepth2 MyStruct2;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStructDepth4
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
        FMyStructDepth3 MyStruct3;
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_MaxPropertyDepth :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        FMyStructDepth4 MyStruct;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta=(MaxPropertyDepth=2))
        FMyStructDepth4 MyStruct_Depth;
};
```

## Test Results:



## Principle:

When constructing child nodes for each FPropertyNode, the current MaxChildDepthAllowed is checked. If the limit is exceeded, construction does not proceed further.

```
/** Safety value representing Depth in the property tree used to stop diabolical
 * topology cases
 * -1 = No limit on children
 * 0 = No more children are allowed. Do not process child nodes
 * >0 = A limit has been set by the property and will tick down for successive
 * children
 */
int32 MaxChildDepthAllowed;

void FPropertyNode::InitNode(const FPropertyParams& InitParams)
{

    //Get the property max child depth
    static const FName Name_MaxPropertyDepth("MaxPropertyDepth");
    if (Property->HasMetaData(Name_MaxPropertyDepth))
    {
        int32 NewMaxChildDepthAllowed = Property-
>GetIntMetaData(Name_MaxPropertyDepth);
        //Ensure new depth is valid. Otherwise just let the parent specified
        //value stand
        if (NewMaxChildDepthAllowed > 0)
    {
```

```

        //if there is already a limit on the depth allowed, take the
minimum of the allowable depths
        if (MaxChildDepthAllowed >= 0)
        {
            MaxChildDepthAllowed = FMath::Min(MaxChildDepthAllowed,
NewMaxChildDepthAllowed);
        }
        else
        {
            //no current limit, go ahead and take the new limit
            MaxChildDepthAllowed = NewMaxChildDepthAllowed;
        }
    }
}

void FPropertyNode::RebuildChildren()
{
    if (MaxChildDepthAllowed != 0)
    {
        //the case where we don't want init child nodes is when an Item has children
        //that we don't want to display
        //the other option would be to make each node "Read only" under that item.
        //The example is a material assigned to a static mesh.
        if (HasNodeFlags(EPropertyNodeFlags::CanBeExpanded) && (childNodes.Num() ==
0))
        {
            InitChildNodes();
            if (ExpandedPropertyItemSet.size() > 0)
            {
                FPropertyNodeUtils::SetExpandedItems(ThisAsSharedRef,
ExpandedPropertyItemSet);
            }
        }
    }
}

```

## DeprecatedNode

- **Function Description:** Used for BehaviorTreeNode or EnvQueryNode, indicating that the class is deprecated, with a red error display in the editor and an error tooltip for notification
- **Usage Location:** UCLASS
- **Engine Module:** DetailsPanel
- **Metadata Type:** bool
- **Restriction Type:** BehaviorTreeNode, EnvQueryNode
- **Commonality:** ★★

Applied to nodes in the AI behavior tree or EQS to mark them as deprecated.

## Example in Source Code:

```
UCLASS(meta = (DeprecatedNode, DeprecationMessage = "Please use IsAtLocation
decorator instead."), MinimalAPI)
class UBTDecorator_ReachedMoveGoal : public UBTDecorator
{
    GENERATED_UCLASS_BODY()
};

UCLASS(MinimalAPI, meta=(DeprecatedNode, DeprecationMessage = "This class is now
deprecated, please use RunMode supporting random results instead."))
class UEnvQueryTest_Random : public UEnvQueryTest
{
    GENERATED_UCLASS_BODY()
};
```

## C++ Test Code:

```
UCLASS(meta = (DeprecatedNode, DeprecationMessage = "This BT node is deprecated.
Don't use this anymore."), MinimalAPI)
class UBTTTask_MyDeprecatedNode : public UBTTTaskNode
{
    GENERATED_UCLASS_BODY()
};
```

In the behavior tree, if the DeprecatedNode attribute is added, the node will be highlighted in red and an error message will be displayed.



## Code Tested Within the Source Code:

```
FString FGraphNodeClassHelper::GetDeprecationMessage(const uclass* Class)
{
    static FName MetaDeprecated = TEXT("DeprecatedNode");
    static FName MetaDeprecatedMessage = TEXT("DeprecationMessage");
    FString DefDeprecatedMessage("Please remove it!");
    FString DeprecatedPrefix("DEPRECATED");
    FString DeprecatedMessage;
```

```

    if (Class && Class->HasAnyClassFlags(CLASS_Native) && Class-
>HasMetaData(MetaDeprecated))
{
    DeprecatedMessage = DeprecatedPrefix + TEXT(": ");
    DeprecatedMessage += Class->HasMetaData(MetaDeprecatedMessage) ? Class-
>GetMetaData(MetaDeprecatedMessage) : DefDeprecatedMessage;
}

return DeprecatedMessage;
}

```

## UsesHierarchy

- **Function Description:** The class utilizes hierarchical data, enabling the instantiation of the hierarchical editing feature within the Details panel.
- **Usage Location:** UCLASS
- **Engine Module:** DetailsPanel
- **Metadata Type:** bool
- **Commonality:** 0

## IgnoreCategoryKeywordsInSubclasses

- **Function Description:** Allows the first subclass of a class to ignore all inherited ShowCategories and HideCategories specifiers.
- **Usage Location:** UCLASS
- **Engine Module:** DetailsPanel
- **Metadata Type:** boolean  
Related To UCLASS: ComponentWrapperClass  
(../../Specifier/UCLASS/ComponentWrapperClass.md)
- **Commonality:** ★

Associated with ComponentWrapperClass

## NoResetToDefault

- **Function Description:** Disables and hides the "Reset" feature for properties in the details panel.
- **Usage Location:** UPROPERTY
- **Engine Module:** DetailsPanel
- **Metadata Type:** bool
- **Commonly Used:** ★★★

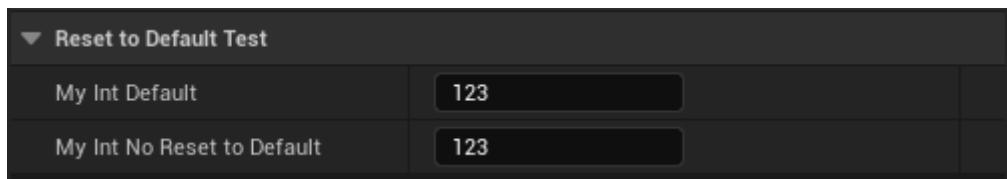
Disables and hides the "Reset" feature for properties in the details panel.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=ResetToDefaultTest)  
    int32 MyInt_Default = 123;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=ResetToDefaultTest, meta  
    = (NoResetToDefault))  
    int32 MyInt_NoResetToDefault = 123;
```

## Test Effects:

You can observe that by default, after changing the value of a property, a reset button appears on the right, allowing the property to be reset to its default value. The purpose of NoResetToDefault is to eliminate this functionality.



## Principle:

The editor checks for this metadata; if it is not present, it creates an SResetToDefaultPropertyEditor.

```
bool SSingleProperty::GeneratePropertyCustomization()  
{  
    if (!PropertyEditor->GetPropertyHandle()->  
        HasMetaData(TEXT("NoResetToDefault")) && !bShouldHideResetToDefault)  
    {  
        HorizontalBox->AddSlot()  
            .Padding( 2.0f )  
            .Autowidth()  
            .VAlign( VAlign_Center )  
        [  
            SNew( SResetToDefaultPropertyEditor, PropertyEditor->  
                GetPropertyHandle() )  
        ];  
    }  
}
```

## ReapplyCondition

- **Function Description:** // Properties with a ReapplyCondition should be disabled and hidden behind the specified property when in reapply mode
- **Usage Location:** UPROPERTY
- **Engine Module:** DetailsPanel
- **Metadata Type:** string = "abc"
- **Commonly Used:** ★

## Code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Placement, meta=(UIMin =  
0, ClampMin = 0, UIMax = 359, ClampMax = 359,  
ReapplyCondition="ReapplyRandomPitchAngle"))  
float RandomPitchAngle;
```

Also used exclusively within Foliage.

## HideBehind

- **Function Description:** This attribute is only displayed when the specified attribute is true or not null
- **Usage Location:** UPROPERTY
- **Engine Module:** DetailsPanel
- **Metadata Type:** string="abc"
- **Restriction Type:** Within the Foliage module
- **Commonality:** ★

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=Placement, meta=(UIMin =  
0, ClampMin = 0, UIMax = 359, ClampMax = 359, HideBehind="AlignToNormal"))  
float AlignMaxAngle;
```

Exclusive to the Foliage module, the same effect can actually be achieved using EditCondition.

## Category

- **Function Description:** Specifies the category for an attribute within the details panel
- **Usage Locations:** UFUNCTION, UPROPERTY
- **Engine Module:** DetailsPanel
- **Metadata Type:** string="A" | B | C"
- **Associated Items:**
  - Unify Function: Category
  - Universal Property Attribute: Category
- **Commonality:** ★★★★☆

## HideCategories

- **Function Description:** Hide certain categories
- **Usage Location:** UCLASS
- **Engine Module:** DetailsPanel
- **Metadata Type:** strings = "a, b, c"  
Related To UCLASS: ShowCategories (../../Specifier/UCLASS>ShowCategories.md)
- **Associated Items:** ShowCategories (ShowCategories.md)
- **Commonality:** ★★★

# ShowCategories

- **Function Description:** Display categories
- **Usage Location:** UCLASS
- **Metadata Type:** strings = "a, b, c"
- **Associated Items:** HideCategories

ShowCategories marked on the class will not be saved to the meta file; it is solely used to override the settings of the base class HideCategories. Therefore, ShowCategories within the meta file is not utilized.

```
//(BlueprintType = true, IncludePath = Class/Display/MyClass_HideCategories.h,
IsBlueprintBase = true, ModuleRelativePath =
Class/Display/MyClass_HideCategories.h)
UCLASS(Blueprintable, ShowCategories = MyGroup1)
class INSIDER_API UMyClass_ShowCategories :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        int Property_NotInGroup;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MyGroup1)
        int Property_Group1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyGroup2 | 
MyGroup22")
        int Property_Group22;
};
```

# EditInline

- **Function Description:** Instantiate an object for the property and treat it as a child object.
  - **Usage Location:** UPROPERTY
  - **Engine Module:** DetailsPanel
  - **Metadata Type:** bool
  - **Associated Items:** NoEditInline, AllowEditInlineCustomization, ForceInlineRow
- 1    @UProperty:Instanced
- **Commonality:** ★★★

Create an instance for the object property and treat it as a child object.

- 2    It can also be manually configured. If the UClass has EditInlineNew but the attribute lacks Instanced, one can manually set EditInline and then manually assign the object reference attribute to enable direct editing of the object.
- 3    Is it equivalent to ShowInnerProperties? EditInline can be applied to object containers (Array, Map, Set), whereas ShowInnerProperties is effective only on individual object properties.
- 4    Can it be set on a Struct? The source code indicates that this setting is available, but it actually has no effect.

## Example Code:

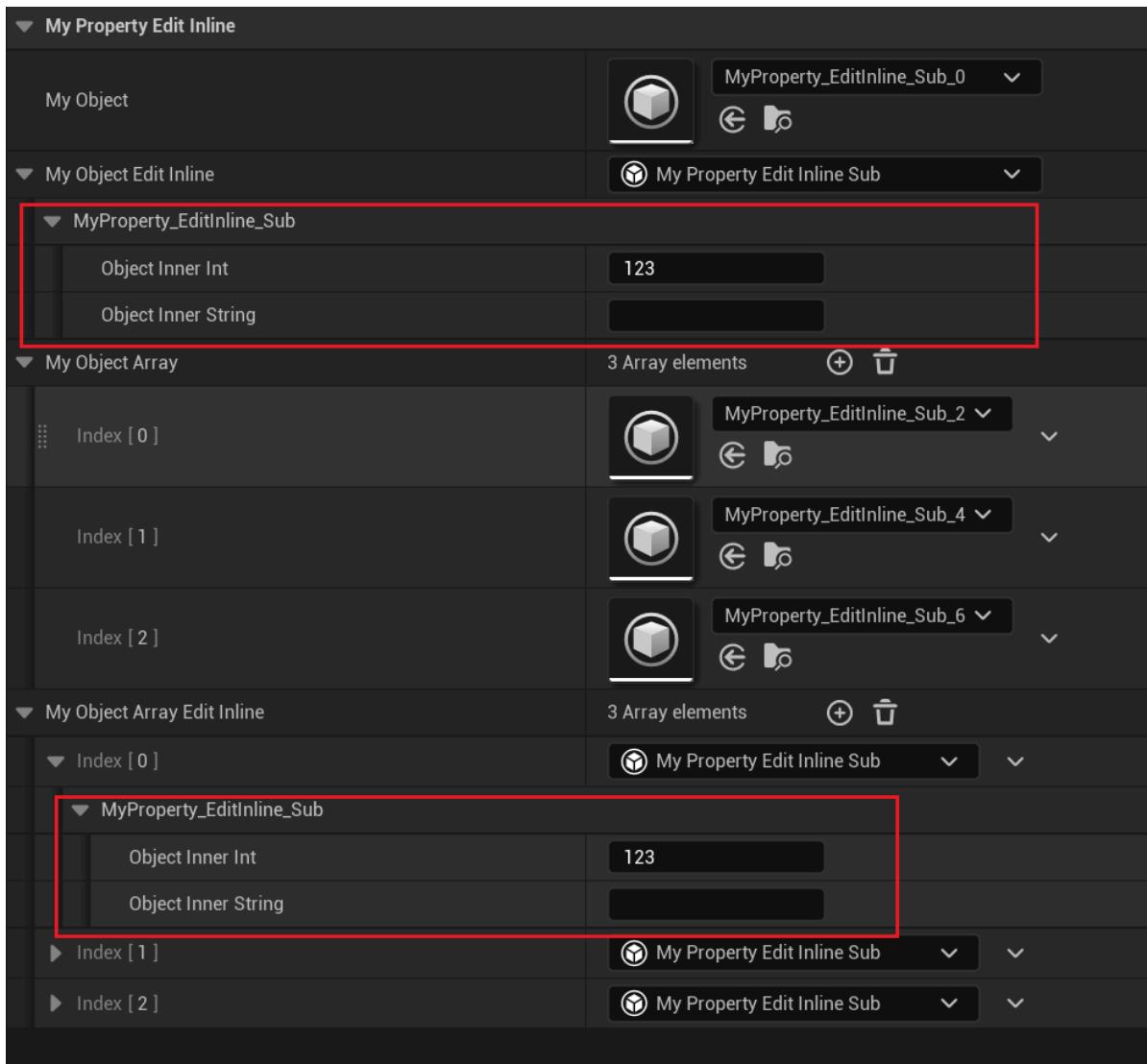
```
UPROPERTY(EditAnywhere, BlueprintReadWrite)
UMyProperty_EditInline_Sub* MyObject;
UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditInline))
UMyProperty_EditInline_Sub* MyObject_EditInline;
UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (NoEditInline))
UMyProperty_EditInline_Sub* MyObject_NoEditInline;

UPROPERTY(EditAnywhere, BlueprintReadWrite)
TArray<UMyProperty_EditInline_Sub*> MyObjectArray;

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditInline))
TArray<UMyProperty_EditInline_Sub*> MyObjectArray_EditInline;

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (NoEditInline))
TArray<UMyProperty_EditInline_Sub*> MyObjectArray_NoEditInline;
```

## Blueprint Effect:



# Principle:

5 It will correspondingly set EPropertyNodeFlags::EditInlineNew.

```
void FPropertyNode::InitNode(const FPropertyParams& InitParams)
{
    // we are EditInlineNew if this property has the flag, or if inside a
    // container that has the flag.
    bIsEditInlineNew = GotReadAddresses && bIsObjectOrInterface &&
    !MyProperty->HasMetaData(Name_NoEditInline) &&
    (MyProperty->HasMetaData(Name_EditInline) || (bIsInsideContainer &&
    OwnerProperty->HasMetaData(Name_EditInline)));
    bShowInnerObjectProperties = bIsObjectOrInterface && MyProperty-
    >HasMetaData(Name_ShowInnerProperties);

    if (bIsEditInlineNew)
    {
        SetNodeFlags(EPropertyNodeFlags::EditInlineNew, true);
    }
    else if (bShowInnerObjectProperties)
    {
        SetNodeFlags(EPropertyNodeFlags::ShowInnerObjectProperties, true);
    }
}

bool SPropertyEditorEditInline::Supports( const FPropertyNode* InTreeNode, int32
InArrayIdx )
{
    return InTreeNode
        && InTreeNode->HasNodeFlags(EPropertyNodeFlags::EditInlineNew)
        && InTreeNode->FindObjectItemParent()
        && !InTreeNode->IsPropertyConst();
}

void FItemPropertyNode::InitExpansionFlags(void)
{
    FProperty* MyProperty = GetProperty();

    if (TSharedPtr<FPropertyNode>& valueNode = GetOrCreateOptionalValueNode())
    {
        // This is a set optional, so check its SetValue instead.
        MyProperty = valueNode->GetProperty();
    }

    bool bExpandableType = CastField<FStructProperty>(MyProperty)
        || (CastField<FArrayProperty>(MyProperty) || CastField<FSetProperty>
    (MyProperty) || CastField<FMapProperty>(MyProperty));

    if (bExpandableType
        || HasNodeFlags(EPropertyNodeFlags::EditInlineNew)
        || HasNodeFlags(EPropertyNodeFlags::ShowInnerObjectProperties)
        || (MyProperty->ArrayDim > 1 && ArrayIndex == -1))
    {
        SetNodeFlags(EPropertyNodeFlags::CanBeExpanded, true);
    }
}
```

```
}
```

## NoEditInline

- **Function description:** Object properties pointing to an UObject instance whos class is marked editinline will not show their properties inline in property windows . Useful for getting actor components to appear in the component tree but not inline in the root actor details panel .
- **Use Location:** UPROPERTY
- **Metadata Type:** bool
- **Restriction Type:** UObject\*
- **Related Items:** EditInline (EditInline.md)

Object references cannot be EditInline by default, so there is no need to add this additionally. Unless after Instanced?

The structure attribute can be EditInline by default, and adding this has no effect, so there is no need to add this.

Found only in the source code:

```
UPROPERTY(VisibleAnywhere, Category = "Connection Point", meta =
(NoEditInline))
FLinearColor Color = FLinearColor::Black;
```

## AllowEditInlineCustomization

- **Function Description:** Allows the customization of the property details panel for EditInline object properties to edit the data within the object.
- **Usage Location:** UPROPERTY
- **Metadata Type:** string="abc"
- **Associated Items:** EditInline
- **Commonality:** ★

Enables the customization of the property details panel for EditInline object properties to edit the data within the object.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyCommonObject :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt = 123;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};
```

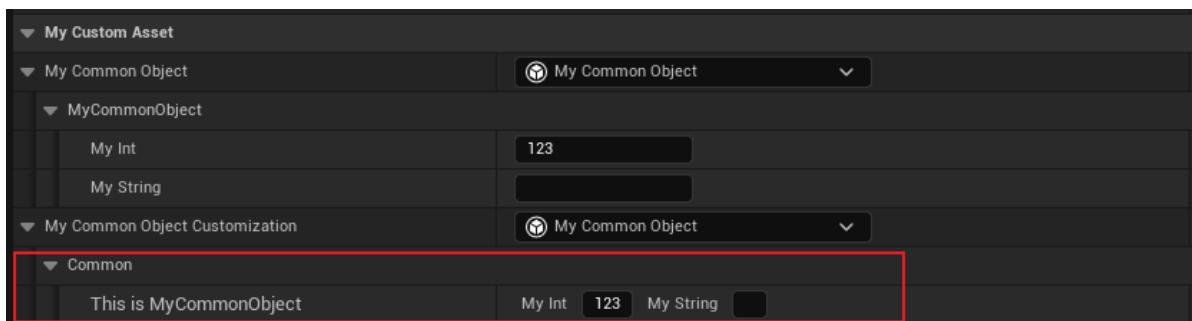
```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyCustomAsset :public UObject
{
    GENERATED_BODY()
public:
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditInline))
    UMyCommonObject* MyCommonObject;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta =
    (EditInline,AllowEditInlineCustomization))
    UMyCommonObject* MyCommonObject_Customization;
};


```

## Effect:



To achieve the custom `EditInline` effect, using a custom `IPropertyTypeCustomization` and `RegisterCustomPropertyParamsLayout` can also be done. The difference is, as in the code example with `UMyCustomAsset`, which contains two `UMyCommonObject*` objects of the same type, using the `IPropertyTypeCustomization` approach would result in both variables being converted to a custom UI mode. However, with `AllowEditInlineCustomization`, you can customize the desired object while leaving the others unchanged.

In usage, `AllowEditInlineCustomization` must be used in conjunction with a custom `FAssetEditorToolkit` to define a `DetailView` (not just customizing the display of a specific type within the engine's unified `DetailView`), then customize `IDetailCustomization` to provide specific Widgets, and finally, use `RegisterInstancedCustomPropertyParamsLayout` to establish the association.

```

DetailsView-
>RegisterInstancedCustomPropertyParamsLayout(UMyCommonObject::StaticClass(), FOnGetDetailCustomizationInstance::CreateStatic(&FMyCommonObjectDetailsCustomization::MakeInstance));

```

(Refer to the relevant implementation in `MyCustomAsset` for this code section)

## Source Code:

```

FDetailPropertyRow::FDetailPropertyRow(TSharedPtr<FPropertyName> InPropertyName,
TSharedRef<FDetailCategoryImpl> InParentCategory,
TSharedPtr<FComplexPropertyNode> InExternalRootNode)
{
    static FName InlineCustomizationKeyMeta("AllowEditInlineCustomization");
}

```

```

    if (PropertyNode->AsComplexNode() && ExternalRootNode.IsValid()) // AsComplexNode works both for objects and structs
    {
        // We are showing an entirely different object inline. Generate a layout for it now.
        if (IDetailsViewPrivate* DetailsView = InParentCategory->GetDetailsView())
        {
            ExternalObjectLayout = MakeShared<FDetailLayoutData>();
            DetailsView->UpdateSinglePropertyMap(ExternalRootNode,
*ExternalObjectLayout, true);
        }
    }
    else if (PropertyNode->HasNodeFlags(EPropertyNodeFlags::EditInlineNew) && PropertyNode->GetProperty()->HasMetaData(InlineCustomizationKeyMeta))
    {
        // Allow customization of 'edit inline new' objects if the metadata key has been specified.
        // The child of this node, if set, will be an object node that we will want to treat as an 'external object layout'
        TSharedPtr<FPropertyNode> ChildNode = PropertyNode->GetNumChildNodes() > 0 ? PropertyNode->GetChildNode(0) : nullptr;
        TSharedPtr<FComplexPropertyNode> ComplexChildNode =
        staticCastSharedPtr<FComplexPropertyNode>(ChildNode);
        if (ComplexChildNode.IsValid())
        {
            // We are showing an entirely different object inline. Generate a layout for it now.
            if (IDetailsViewPrivate* DetailsView = InParentCategory->GetDetailsView())
            {
                ExternalObjectLayout = MakeShared<FDetailLayoutData>();
                DetailsView->UpdateSinglePropertyMap(ComplexChildNode,
*ExternalObjectLayout, true);
            }
        }
    }
}

```

The principle of operation is that when creating an FDetailPropertyRow, which is a row for a property in the details panel, if AllowEditInlineCustomization is present, an ExternalObjectLayout is created. Subsequently, when creating children for FDetailPropertyRow, it checks for the presence of an ExternalObjectLayout. If present, our previous Customization is applied; if not, the default settings are used. The following is the code using ExternalObjectLayout:

```

void FDetailPropertyRow::GenerateChildrenForPropertyNode(
TSharedPtr<FPropertyNode>& RootPropertyName, FDetailNodeList& OutChildren )
{
    // Children should be disabled if we are disabled
    TAttribute<bool> ParentEnabledState = TAttribute<bool>::CreateSP(this,
& FDetailPropertyRow::GetEnabledState);

    if( PropertyTypeLayoutBuilder.IsValid() && bShowCustomPropertyParams )
    {

```

```

        const TArray< FDetailLayoutCustomization *>& ChildRows =
PropertyTypeLayoutBuilder->GetChildCustomizations();

        for( int32 ChildIndex = 0; ChildIndex < ChildRows.Num(); ++ChildIndex )
{
    TSharedRef< FDetailItemNode> ChildNodeItem =
MakeShared< FDetailItemNode>( ChildRows[ChildIndex],
ParentCategory.Pin().ToSharedRef(), ParentEnabledState );
    ChildNodeItem->Initialize();
    OutChildren.Add( ChildNodeItem );
}
}

else if (ExternalObjectLayout.IsValid() && ExternalObjectLayout-
>DetailLayout->HasDetails())
{
    OutChildren.Append(ExternalObjectLayout->DetailLayout-
>GetAllRootTreeNodes());
    //Custom Panel
}
else if ((bShowCustomPropertyChildren || !CustomPropertyWidget.IsValid()) &&
RootPropertyName->GetNumChildNodes() > 0)
{
    //Normal Default child creation
}

```

An example used in the source code is the details panel for the Binding Property on the Bind Actor in LevelSequence.

If we remove some code

```

USTRUCT()
struct FMovieSceneBindingPropertyInfo
{
    GENERATED_BODY()

    // Locator for the entry
    UPROPERTY(EditAnywhere, Category = "Default", meta=(AllowedLocators="Actor",
    DisplayName="Actor"))
    FUniversalObjectLocator Locator;

    // Flags for how to resolve the locator
    UPROPERTY()
    ELocatorResolveFlags ResolveFlags = ELocatorResolveFlags::None;

    UPROPERTY(Instanced, VisibleAnywhere, Category = "Default", meta=(EditInline,
    AllowEditInlineCustomization, DisplayName="Custom Binding Type"))
    UMovieSceneCustomBinding* CustomBinding = nullptr;
};

//Hack the code yourself
UObject* obj =
UInsiderLibrary::FindObjectByNameSmart(TEXT("MovieSceneBindingPropertyInfo"));
UScriptStruct* ss = Cast<UScriptStruct>(obj);
FProperty* prop = ss->FindPropertyByName(TEXT("CustomBinding"));
prop->RemoveMetaData(TEXT("AllowEditInlineCustomization"));

```

The effect will change from left to right:



Registration methods also differ:

```
void ULevelSequenceEditorSubsystem::AddBindingDetailCustomizations(TSharedRef<IDetailsView> DetailsView, TSharedPtr<ISequencer> ActiveSequencer, FGuid BindingGuid)
{
    // TODO: Do we want to create a generalized way for folks to add instanced
    // property layouts for other custom binding types so they can have access to
    // sequencer context?
    if (ActiveSequencer.IsValid())
    {
        UMovieSceneSequence* Sequence = ActiveSequencer-
            >GetFocusedMovieSceneSequence();
        UMovieScene* MovieScene = Sequence ? Sequence->GetMovieScene() : nullptr;
        if (MovieScene)
        {
            FPropertyEditorModule& PropertyEditor =
                FModuleManager::Get().LoadModuleChecked<FPropertyEditorModule>
                (TEXT("PropertyEditor"));
            DetailsView-
                >RegisterInstancedCustomPropertyParamsLayout(FMovieSceneBindingPropertyParams::Static
                    Struct() ->GetFName(), FOnGetPropertyTypeCustomizationInstance::CreateLambda([][]
                    (TweakPtr<ISequencer> InSequencer, UMovieScene* InMovieScene, FGuid
                    InBindingGuid, ULevelSequenceEditorSubsystem* LevelSequenceEditorSubsystem)
                {
                    return
                        MakeShared<FMovieSceneBindingPropertyParamsDetailCustomization>(InSequencer,
                        InMovieScene, InBindingGuid, LevelSequenceEditorSubsystem);
                }, ActiveSequencer.TweakPtr(), MovieScene, BindingGuid, this));
        }
        DetailsView-
            >RegisterInstancedCustomPropertyParamsLayout(UMovieSceneSpawnableActorBinding::StaticCl
                ass(),
            FOnGetDetailCustomizationInstance::CreateStatic(&UMovieSceneSpawnableActorBinding
            BaseCustomization::MakeInstance, ActiveSequencer.TweakPtr(), MovieScene,
            BindingGuid));
    }
}
```

## ForceInlineRow

- **Function description:** Forces the key and other values of a TMap attribute's structure to be merged into the same line for display
- **Usage location:** UPROPERTY
- **Metadata type:** bool
- **Associated items:** EditInline
- **Commonality:** ★

Forces the key and other values in the TMap attribute to be merged into the same line for display.  
Points to consider include:

- The attribute must be a TMap attribute, which contains a Key. TArray or TSet are not applicable.
- FStruct is used as the Key, enabling the source code mechanism to function, as it is the Key Property that is evaluated
- The FStruct must have registered related IPropertyTypeCustomization to allow customization of the structure's display UI
- The ShouldInlineKey of the IPropertyTypeCustomization returns false (the default is), otherwise if it is true, it will be merged into one row regardless of whether it is marked ForceInlineRow or not

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite)  
    TMap<FMyCommonStruct, int32> MyStructMap;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ForceInlineRow))  
    TMap<FMyCommonStruct, int32> MyStructMap_ForceInlineRow;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite)  
    TMap<int32, FMyCommonStruct> MyStructMap2;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ForceInlineRow))  
    TMap<int32, FMyCommonStruct> MyStructMap_ForceInlineRow2;  
  
  
  
void FMyCommonStructCustomization::CustomizeHeader(TSharedRef<IPropertyHandle>  
PropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&  
CustomizationUtils)  
{  
    HeaderRow.NameContent() [SNew(STextBlock).Text(INVTEXT("This is  
MyCommonStruct"))];  
  
    TSharedPtr<IPropertyHandle> IntPropertyHandle = PropertyHandle->  
GetChildHandle(GET_MEMBER_NAME_CHECKED(FMyCommonStruct, MyInt));  
    TSharedPtr<IPropertyHandle> StringPropertyHandle = PropertyHandle->  
GetChildHandle(GET_MEMBER_NAME_CHECKED(FMyCommonStruct, MyString));  
  
    HeaderRow.ValueContent()  
    [  
        SNew(SHorizontalBox)  
        + SHorizontalBox::Slot()  
        .Padding(5.0f, 0.0f).Autowidth()  
        [  
            IntPropertyHandle->CreatePropertyNameWidget()  
        ]  
        + SHorizontalBox::Slot()  
        .Padding(5.0f, 0.0f).Autowidth()  
        [  
    ]
```

```

        IntPropertyHandle->CreatePropertyValueWidget()
    ]
    + SHorizontalBox::slot()
    .Padding(5.0f, 0.0f).Autowidth()
    [
        StringPropertyHandle->CreatePropertyNameWidget()
    ]
    + SHorizontalBox::slot()
    .Padding(5.0f, 0.0f).Autowidth()
    [
        StringPropertyHandle->CreatePropertyValueWidget()
    ]
]
;

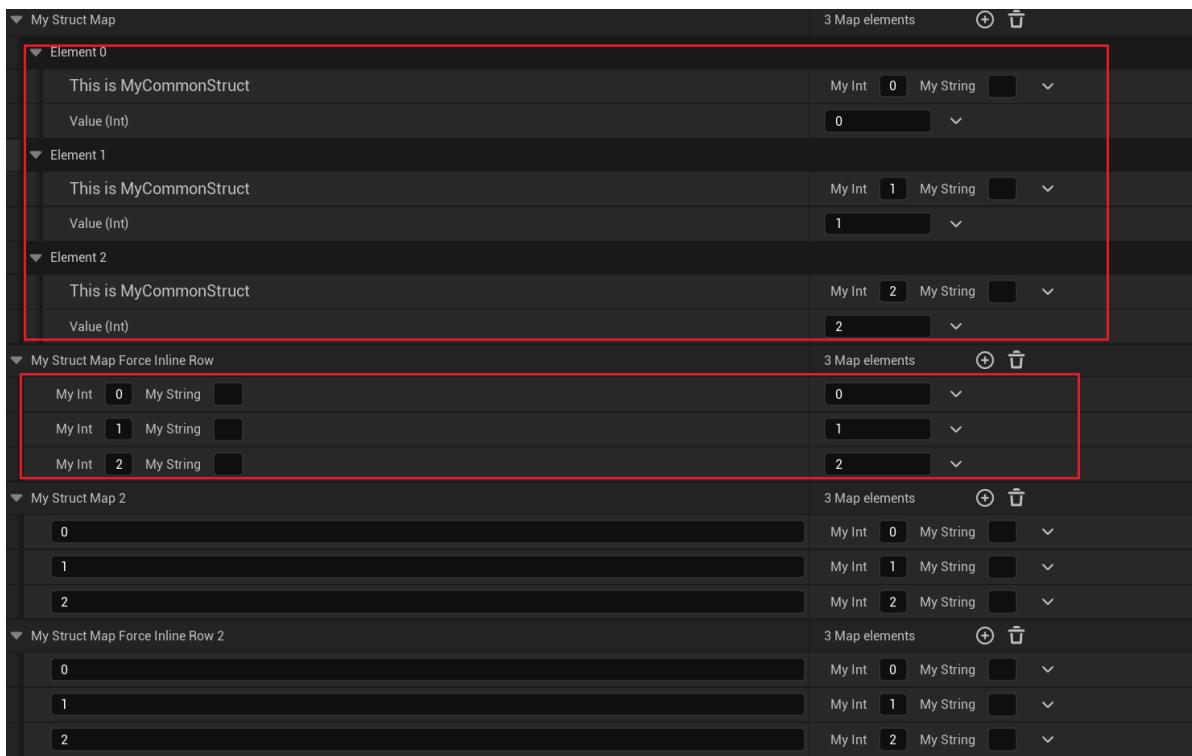
}

```

## Test Results:

You can see that the data items of MyStructMap are displayed across two lines. With ForceInlineRow, the UI for data items is merged into one line, appearing more concise.

It is also specifically noted that if FStruct is used as the Value, there is no such distinction.



If the corresponding IPropertyTypeCustomization for FMyCommonStruct is not registered, the structure's property UI will be displayed in the default manner, split into two lines.

My Struct Map		3 Map elements		
▼ Element 0				
▼ Key (My Common Struct)		2 members		
My Int	0			
My String				
Value (Int)	0			
▼ Element 1				
► Key (My Common Struct)		2 members		
Value (Int)	1			
▼ Element 2				
► Key (My Common Struct)		2 members		
Value (Int)	2			
▼ My Struct Map Force Inline Row		3 Map elements		
▼ Element 0				
► Key (My Common Struct)		2 members		
Value (Int)	0			
▼ Element 1				
► Key (My Common Struct)		2 members		
Value (Int)	1			
▼ Element 2				
► Key (My Common Struct)		2 members		
Value (Int)	2			
▼ My Struct Map 2		3 Map elements		
▼ 0		2 members		
My Int	0			
My String				
► 1		2 members		
► 2		2 members		
▼ My Struct Map Force Inline Row 2		3 Map elements		
► 0		2 members		
► 1		2 members		
► 2		2 members		

If the ShouldInlineKey of FMyCommonStruct's IPropertyTypeCustomization returns true, it will cause the properties with this structure as the Key to be merged into one line even without ForceInlineRow, thereby negating the effect and distinction of ForceInlineRow.

My Struct Map				3 Map elements	
My Int	0	My String		0	
My Int	1	My String		1	
My Int	2	My String		2	
My Struct Map Force Inline Row				3 Map elements	
My Int	0	My String		0	
My Int	1	My String		1	
My Int	2	My String		2	
My Struct Map 2				3 Map elements	
0				My Int	0 My String
1				My Int	1 My String
2				My Int	2 My String
My Struct Map Force Inline Row 2				3 Map elements	
0				My Int	0 My String
1				My Int	1 My String
2				My Int	2 My String

## Principle:

This logic is also part of the construction process of the FDetailPropertyRow constructor, determining whether there is a GetPropertyKeyNode, which essentially requests a TMap attribute.

Then, as the Key type, if it is UObject\*, because NeedsKeyNode always returns false, it will always enter the MakePropertyEditor branch.

Therefore, the type being tested is actually Struct, which must rely on the cooperation of bInlineRow and FoundPropertyCustomisation. At this point, IPropertyTypeCustomization must be present to enter the branch, and if IPropertyTypeCustomization::ShouldInlineKey() returns true, it will enter the branch regardless of the ForceInlineRow attribute. Otherwise, it depends on the ForceInlineRow attribute, which is when this Meta attribute comes into effect.

```

FDetailPropertyRow::FDetailPropertyRow(TSharedPtr<FPropertyNode> InPropertyName,
TSharedRef<FDetailCategoryImpl> InParentCategory,
TSharedPtr<FComplexPropertyNode> InExternalRootNode)
{
    if (PropertyName->GetPropertyKeyNode().IsValid())
    {
        TSharedPtr<IPropertyTypeCustomization> FoundPropertyCustomisation =
GetPropertyCustomization(PropertyName->GetPropertyKeyNode().ToSharedRef(),
ParentCategory.Pin().ToSharedRef());

        bool bInlineRow = FoundPropertyCustomisation != nullptr ?
FoundPropertyCustomisation->ShouldInlineKey() : false;

        static FName InlineKeyMeta("ForceInlineRow");
        bInlineRow |= InPropertyName->GetParentNode()->GetProperty()-
>HasMetaData(InlineKeyMeta);

        // Only create the property editor if it's not a struct or if it requires
        // to be inlined (and has customization)
        if (!NeedsKeyNode(PropertyNameRef, InParentCategory) || (bInlineRow &&
FoundPropertyCustomisation != nullptr))
        {
            CachedKeyCustomTypeInterface = FoundPropertyCustomisation;
        }
    }
}

```

```

        MakePropertyEditor(PropertyNode->GetPropertyKeyNode() .ToSharedRef() ,
Utilities, PropertyKeyEditor);
    }
}

bool FDetailPropertyRow::NeedsKeyNode(TSharedRef<FPropertyNode> InPropertyName,
TSharedRef<FDetailCategoryImpl> InParentCategory)
{
    FStructProperty* KeyStructProp = CastField<FStructProperty>(InPropertyName-
>GetPropertyKeyNode()->GetProperty());
    return KeyStructProp != nullptr;
}

```

Examples used in the source code:

In the source code search, this example was found, but in reality, ForceInlineRow on HLODSetups does not take effect here.

```

USTRUCT()
struct FRuntimePartitionDesc
{
    GENERATED_USTRUCT_BODY()

#if WITH_EDITORONLY_DATA
    /** Partition class */
    UPROPERTY(EditAnywhere, Category = RuntimeSettings)
    TSubclassOf<URuntimePartition> Class;

    /** Name for this partition, used to map actors to it through the
Actor.RuntimeGrid property */
    UPROPERTY(EditAnywhere, Category = RuntimeSettings, Meta = (EditCondition =
"Class != nullptr", HideEditConditionToggle))
    FName Name;

    /** Main partition object */
    UPROPERTY(VisibleAnywhere, Category = RuntimeSettings, Instanced, Meta =
(EditCondition = "Class != nullptr", HideEditConditionToggle, NoResetToDefault,
TitleProperty = "Name"))
    TObjectPtr<URuntimePartition> MainLayer;

    /** HLOD setups used by this partition, one for each layers in the hierarchy
*/
    UPROPERTY(EditAnywhere, Category = RuntimeSettings, Meta = (EditCondition =
"Class != nullptr", HideEditConditionToggle, ForceInlineRow))
    TArray<FRuntimePartitionHLODSetup> HLODSetups;
#endif

#if WITH_EDITOR
    void UpdateHLODPartitionLayers();
#endif
};

```

# DeprecatedProperty

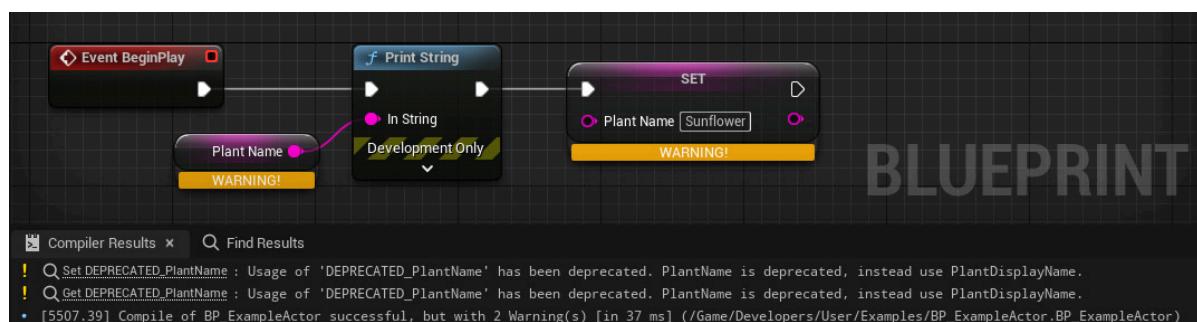
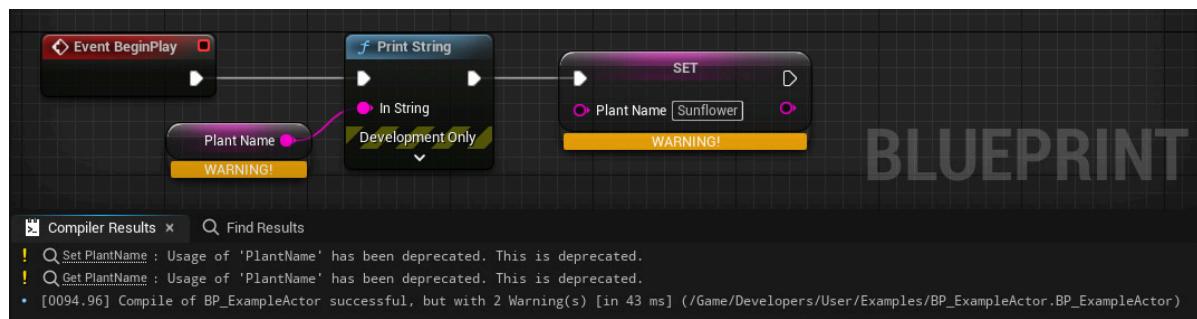
- **Function Description:** Indicates that the property is deprecated; referencing this property in blueprints will generate a warning
- **Usage Location:** UPROPERTY
- **Engine Module:** Development
- **Metadata Type:** bool
- **Associated Items:**
  - UCLASS: Deprecated
- **Commonality:** ★

Indicates that the property is deprecated; referencing this property in blueprints will generate a warning

## Sample Code:

```
// Simple
UPROPERTY(BlueprintReadWrite, meta=(DeprecatedProperty, DeprecationMessage="This
is deprecated"))
FString PlantName;

// Better
UPROPERTY(BlueprintReadWrite, meta=(DisplayName="PlantName", DeprecatedProperty,
DeprecationMessage="PlantName is deprecated, instead use PlantDisplayName."))
FString DEPRECATED_PlantName;
```



## Deprecated

- **Function Description:** Specifies the engine version number at which this element is to be deprecated.
- **Usage Location:** Any

- **Engine Module:** Development
- **Metadata Type:** string="abc"
- **Commonality:** ★

Specifies the engine version number at which this element is scheduled for deprecation.

This value merely serves as a record within the C++ code, indicating information without actually rendering an element deprecated. It is not used or displayed in any UI components elsewhere.

To discard an element, you still need to use other tags, such as **DeprecatedProperty**, **DeprecatedFunction**, etc.

## DevelopmentOnly

---

- **Function Description:** Marks a function as DevelopmentOnly, indicating that it will only execute in Development mode. This is useful for debugging purposes such as outputting debug information, but it will be skipped in the final release version.
- **Usage Location:** UFUNCTION
- **Engine Module:** Development
- **Metadata Type:** bool
- **Commonality:** ★

Marks a function as DevelopmentOnly, meaning it will only run in Development mode. This is suitable for debugging features like outputting debug information, but it will be omitted from the final release.

The most typical example in the source code is PrintString.

## Test Code:

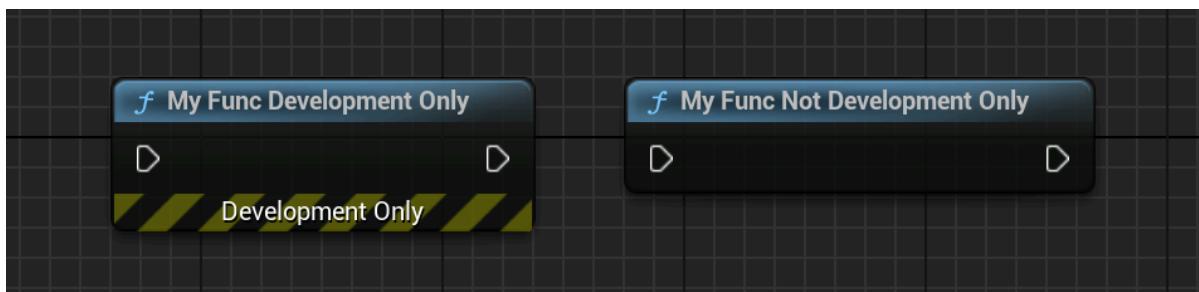
---

```
UFUNCTION(BlueprintCallable,meta=(Developmentonly))
static void MyFunc_DevelopmentOnly(){}

UFUNCTION(BlueprintCallable,meta=())
static void MyFunc_NotDevelopmentOnly{}
```

## Blueprint Effect:

---



## Principle:

It alters the state of the function's blueprint node to DevelopmentOnly, which consequently results in the node being passed through in shipping mode.

```
void UK2Node_CallFunction::Serialize(FArchive& Ar)
{
    if (const UFunction* Function = GetTargetFunction())
    {
        // Enable as development-only if specified in metadata. This way existing functions that have the metadata added to them will get their enabled state fixed up on load.
        if (GetDesiredEnabledState() == ENodeEnabledState::Enabled &&
Function->HasMetaData(FBlueprintMetadata::MD_Developmentonly))
        {
            SetEnabledState(ENodeEnabledState::DevelopmentOnly,
/*bUserAction=*/ false);
        }
        // Ensure that if the metadata is removed, we also fix up the enabled state to avoid leaving it set as development-only in that case.
        else if (GetDesiredEnabledState() ==
ENodeEnabledState::DevelopmentOnly && !Function-
>HasMetaData(FBlueprintMetadata::MD_Developmentonly))
        {
            SetEnabledState(ENodeEnabledState::Enabled,
/*bUserAction=*/ false);
        }
    }
}
```

## Deprecation Message

- **Function Description:** Defines deprecated messages
- **Usage Locations:** UCLASS, UFUNCTION, UPROPERTY
- **Engine Module:** Development
- **Metadata Type:** string="abc"
- **Associated Items:**
  - UCLASS: Deprecated
- **Commonality:** ★

## Example:

```
UFUNCTION(meta=(DeprecatedFunction, DeprecationMessage="This function is
deprecated, please use OtherFunctionName instead."))
ReturnType FunctionName([Parameter, Parameter, ...])

UPROPERTY(BlueprintReadWrite, meta=(DeprecatedProperty, DeprecationMessage="This
is deprecated"))
FString PlantName;
```

# DeprecatedFunction

---

- **Function Description:** Marks a function as deprecated
- **Usage Location:** UFUNCTION
- **Engine Module:** Development
- **Metadata Type:** boolean
- **Commonality:** ★

*Any Blueprint references to this function will cause compilation warnings telling the user that the function is deprecated. You can add to the deprecation warning message (for example, to provide instructions on replacing the deprecated function) using the DeprecationMessage metadata specifier.*

## Comment

---

- **Function description:** Used to record comment content
- **Placement:** Anywhere
- **Engine module:** Development
- **Metadata type:** string="abc"
- **Commonly used:** ★★★

Comment differs from ToolTip; the latter is a hover prompt for users, while the former is simply a record of comments within the code. However, comments written in the code are typically added automatically to the ToolTip, which is why we often see prompts in the UI interface.

But if ToolTips are not desired and only Comments are wanted, they can be added manually in the meta section.

## Test Code:

---

```
//(BlueprintType = true, Comment = //This is a comment on class, IncludePath =
Property/Development/MyProperty_Development.h, ModuleRelativePath =
Property/Development/MyProperty_Development.h, ToolTip = This is a comment on
class)

//This is a comment on class
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Development :public UObject
{
    GENERATED_BODY()
public:
    //Comment = //This is a comment on function, ModuleRelativePath =
Property/Development/MyProperty_Development.h, ToolTip = This is a comment on
function)

    //This is a comment on function
    UFUNCTION(BlueprintCallable)
    int32 MyFunc(FString str){return 0;}

    // (Category = MyProperty_Development, Comment = //This is a comment on
property, ModuleRelativePath = Property/Development/MyProperty_Development.h,
ToolTip = This is a comment on property)
```

```

//This is a comment on property
UPROPERTY(EditAnywhere, BlueprintReadWrite)
int32 MyProperty = 123;

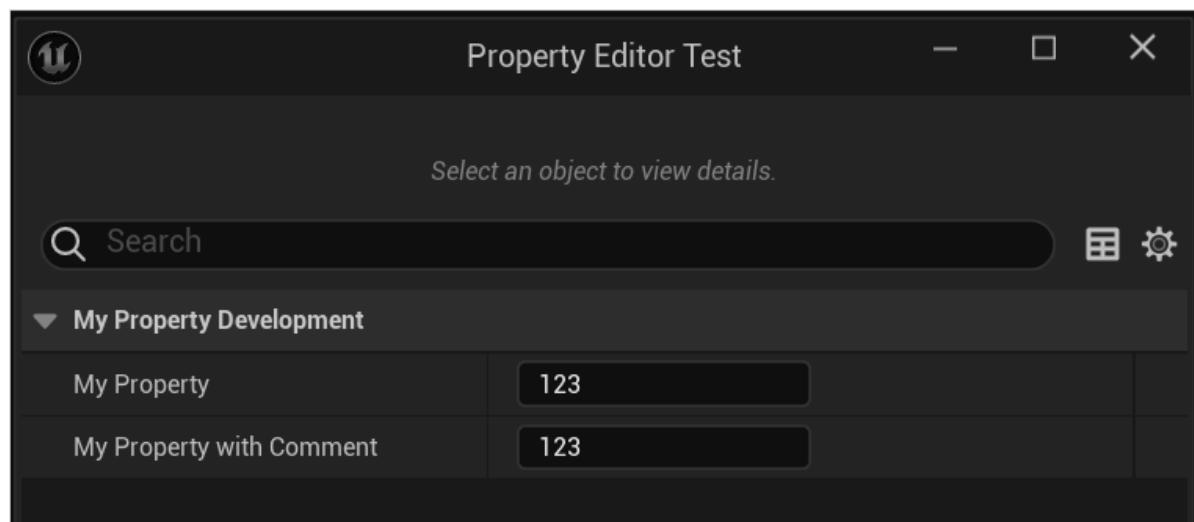
//(Category = MyProperty_Development, Comment = This is my other property.,
ModuleRelativePath = Property/Development/MyProperty_Development.h)

UPROPERTY(EditAnywhere, BlueprintReadWrite,meta=(Comment="This is my other
property."))
int32 MyProperty_WithComment = 123;
};

```

## Test Results:

MyProperty\_WithComment has only the Comment added, and thus lacks the mouse hover effect.



## FriendlyName

- Function Description:** Is it the same as DisplayName?
- Usage Location:** Anywhere
- Engine Module:** Development
- Metadata Type:** string = "abc"

## DevelopmentStatus

- Function Description:** Indicates the development status
- Usage Location:** UCLASS
- Engine Module:** Development
- Metadata Type:** string="abc"
- Associated Items:**
  - UCLASS: Experimental, Early Access Preview
- Commonality:** ★

DevelopmentStatus=Experimental  
DevelopmentStatus=EarlyAccess

# Tooltip

- **Function Description:** Provides a tooltip text in Meta, which overrides the text in the code comments
- **Usage Location:** Anywhere
- **Engine Module:** Development
- **Metadata Type:** string="abc"
- **Associated Items:** ShortTooltip
- **Commonly Used:** ★★★

## Test Code:

```
// This is a ToolTip out of Class. There're so so so so so so so many words I want to say, but here's too narrow.  
UCLASS(BlueprintType, Blueprintable, meta = (ToolTip = "This is a ToolTip within Class. There're so so so so so so so many words I want to say, but here's too narrow."))  
class INSIDER_API UMyClass_ToolTip :public UObject  
{  
    GENERATED_BODY()  
public:  
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (ToolTip = "This is a ToolTip within Property. There're so so so so so so so many words I want to say, but here's too narrow."))  
        float MyFloat_WithToolTip;  
  
    UPROPERTY(BlueprintReadWrite, EditAnywhere)  
        FString MyString;  
  
    UFUNCTION(BlueprintCallable, meta = (ToolTip = "This is a ToolTip within Function. There're so so so so so so so many words I want to say, but here's too narrow."))  
        void MyFunc_WithToolTip();  
  
    UFUNCTION(BlueprintCallable)  
        void MyFunc();  
};  
  
// This is a ToolTip out of Class. There're so so so so so so so many words I want to say, but here's too narrow.  
UCLASS(BlueprintType, Blueprintable, meta = (ToolTip = "This is a ToolTip within Class. There're so so so so so so so many words I want to say, but here's too narrow.", shortToolTip = "This is a ShortToolTip within Class."))  
class INSIDER_API UMyClass_WithAllToolTip :public UObject  
{  
    GENERATED_BODY()  
public:  
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (ToolTip = "This is a ToolTip within Property."))
```

```

float MyFloat_WithToolTip;

UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (ToolTip = "This is a ToolTip
within Property. There're so so so so so so so many words I want to say, but
here's too narrow.\nThis is a new line.",ShortToolTip = "This is a ShortToolTip
within Property."))

float MyFloat_WithAllToolTip;

UPROPERTY(BlueprintReadWrite, EditAnywhere)
FString MyString;

UFUNCTION(BlueprintCallable, meta = (ToolTip = "This is a ToolTip within
Function. There're so so so so so so so many words I want to say, but here's too
narrow.",ShortToolTip = "This is a ShortToolTip within Function."))

void MyFunc_WithAllToolTip() {}

UFUNCTION(BlueprintCallable, meta = (ToolTip = "This is a ToolTip within
Function."))
void MyFunc_WithToolTip() {}

// This is a ToolTip out of Class. There're so so so so so so so many words I want
// to say, but here's too narrow.

UCLASS(BlueprintType, Blueprintable)
class INSIDER_API UMyClass_ToolTip_TypeA :public UObject
{
    GENERATED_BODY()
};

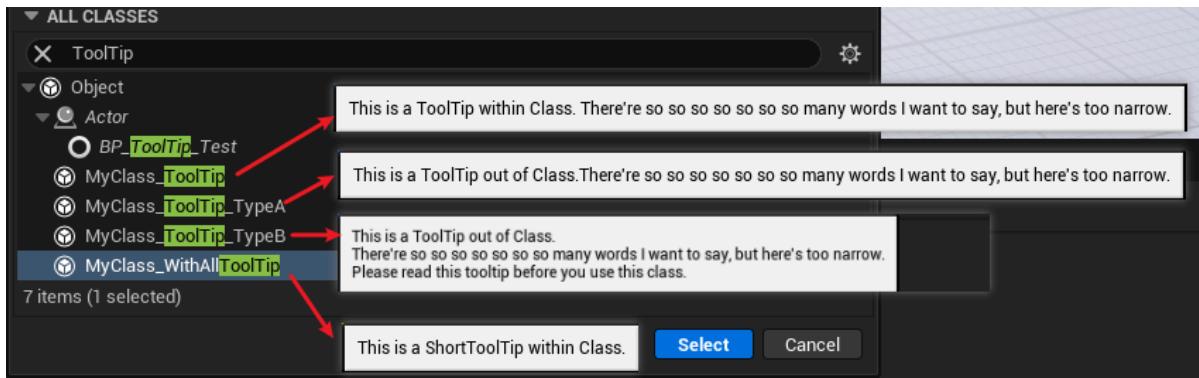
/***
 *  This is a ToolTip out of Class.
 *  There're so so so so so so so many words I want to say, but here's too
narrow.
 *  Please read this tooltip before you use this class.
 */
UCLASS(BlueprintType, Blueprintable)
class INSIDER_API UMyClass_ToolTip_TypeB :public UObject
{
    GENERATED_BODY()
};

```

## Test Effects:

Tooltip when selecting a parent class:

It can be observed that when a ToolTip is provided, it overrides the comments within the code annotations. Additionally, as illustrated in the following image, the prompt text is excessively long, extending beyond the bounds of the option box. In such cases, if a ShortToolTip is supplied, the text from the ShortToolTip will be displayed within the parent class selector, offering a more concise presentation. This principle is also applied in numerous other contexts, such as providing hints for variables (of the specified type) or during the selection of a variable type.



## Principle Code:

In the source code, there are FField and UField. Regular attributes are FField, while UClass inherits from UField. Therefore, it is important to note that FField::GetToolTipText's bShortTooltip is always false, whereas UField::GetToolTipText will pass true.

```

FText FField::GetToolTipText(bool bShortTooltip) const
{
    bool bFoundShortTooltip = false;
    static const FName NAME_Tooltip(TEXT("Tooltip"));
    static const FName NAME_ShortTooltip(TEXT("ShortTooltip"));
    FText LocalizedToolTip;
    FString NativeToolTip;

    if (bShortTooltip)
    {
        NativeToolTip = GetMetaData(NAME_ShortTooltip);
        if (NativeToolTip.IsEmpty())
        {
            NativeToolTip = GetMetaData(NAME_Tooltip);
        }
        else
        {
            bFoundShortTooltip = true;
        }
    }
    else
    {
        NativeToolTip = GetMetaData(NAME_Tooltip);
    }

    const FString Namespace = bFoundShortTooltip ? TEXT("uobjectShortTooltips") :
TEXT("uobjectToolTips");
    const FString Key = GetFullGroupName(false);
    if (!FText::FindText(Namespace, Key, /*OUT*/&LocalizedToolTip,
&NativeToolTip))
    {
        if (!NativeToolTip.IsEmpty())
        {
            static const FString DoxygenSee(TEXT("@see"));
            static const FString TooltipSee(TEXT("See:"));
            if (NativeToolTip.ReplaceInline(*DoxygenSee, *TooltipSee) > 0)
            {
                NativeToolTip.TrimEndInline();
            }
        }
    }
}
```

```

        }

    }

    LocalizedToolTip = FText::FromString(NativeToolTip);

}

return LocalizedToolTip;
}

FText UField::GetToolTipText(bool bShortTooltip) const
{
    bool bFoundShortTooltip = false;
    static const FName NAME_Tooltip(TEXT("Tooltip"));
    static const FName NAME_ShortTooltip(TEXT("ShortTooltip"));
    FText LocalizedToolTip;
    FString NativeToolTip;

    if (bShortTooltip)
    {
        NativeToolTip = GetMetaData(NAME_ShortTooltip);
        if (NativeToolTip.IsEmpty())
        {
            NativeToolTip = GetMetaData(NAME_Tooltip);
        }
        else
        {
            bFoundShortTooltip = true;
        }
    }
    else
    {
        NativeToolTip = GetMetaData(NAME_Tooltip);
    }

    const FString Namespace = bFoundShortTooltip ? TEXT("UObjectShortTooltips") :
TEXT("UObjectToolTips");
    const FString Key = GetFullName(false);
    if ( !FText::FindText( Namespace, Key, /*OUT*/&LocalizedToolTip,
&NativeToolTip ) )
    {
        if (NativeToolTip.IsEmpty())
        {
            NativeToolTip =
FName::NameToDisplayString(FDisplayNameHelper::Get(*this), false);
        }
        else if (!bShortTooltip && IsNative())
        {
            FormatNativeToolTip(NativeToolTip, true);
        }
        LocalizedToolTip = FText::FromString(NativeToolTip);
    }

    return LocalizedToolTip;
}

//In the type selector, ShortTooltip is given priority
FText FClassPickerDefaults::GetDescription() const

```

```

{
    FText Result = LOCTEXT("NullClass", "(null class)");

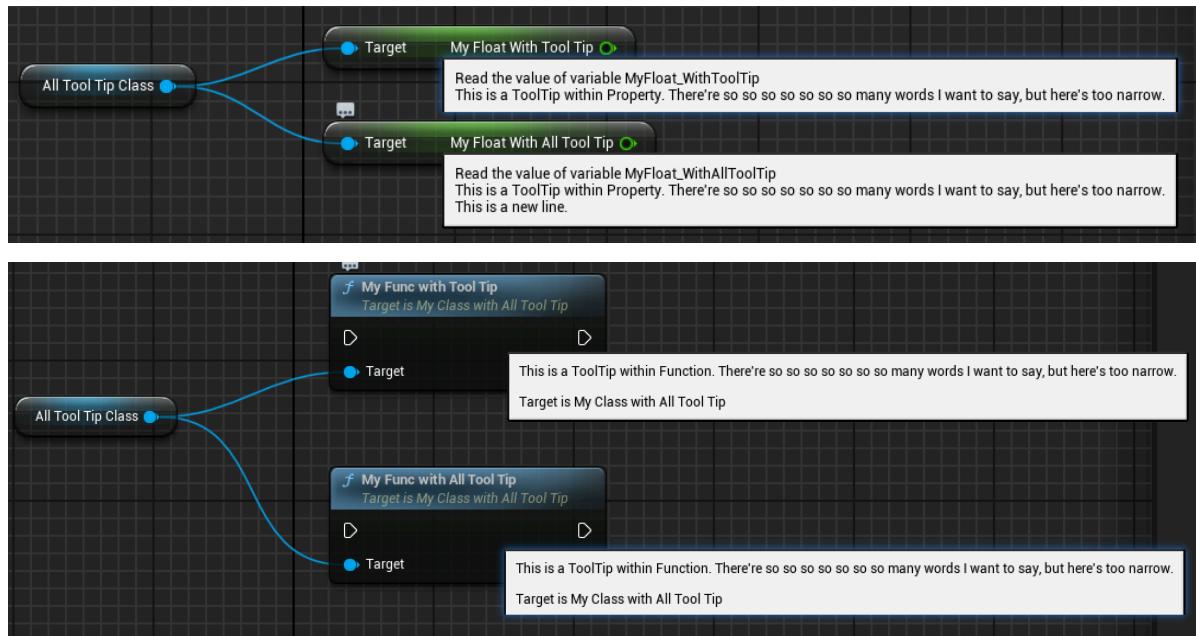
    if (UClass* ItemClass = LoadClass<UObject>(NULL, *ClassName, NULL, LOAD_None,
NULL))
    {
        Result = ItemClass->GetToolTipText(/*bShortTooltip=*/ true);
    }

    return Result;
}

```

However, for Properties and Functions, only the Tooltip will be displayed when shown, and ShortTooltip will not be applied

Variables and Functions:



Another point to note is that the text within comments in the code is also considered as a Tooltip. Both // and /\*\*/ formats are supported. If you want to insert a line break in the Tooltip, you can simply add /n.

```

/*
(BlueprintType = true, Comment = // This is a Tooltip out of Class. There're so so
so so so so many words I want to say, but here's too narrow.
, IncludePath = Any/ToolTip_Test.h, IsBlueprintBase = true, ModuleRelativePath =
Any/ToolTip_Test.h, ToolTip = This is a Tooltip out of Class. There're so so so so
so so so many words I want to say, but here's too narrow.)
*/

// This is a Tooltip out of Class. There're so so so so so so so many words I want
to say, but here's too narrow.

UCLASS(BlueprintType, Blueprintable)
class INSIDER_API UMyClass_ToolTip_TypeA :public UObject
{
    GENERATED_BODY()
};

```

```

// [MyClass_ToolTip_TypeB     Class->Struct->Field->Object
/script/Insider.MyClass_ToolTip_TypeB]
//(BlueprintType = true, Comment = /**
/* This is a ToolTip out of class.
/* There're so so so so so so so many words I want to say, but here's too
narrow.
/* Please read this tooltip before you use this class.
//*, IncludePath = Any/ToolTip_Test.h, IsBlueprintBase = true,
ModuleRelativePath = Any/ToolTip_Test.h, ToolTip = This is a ToolTip out of
class.
//There're so so so so so so so many words I want to say, but here's too narrow.
//Please read this tooltip before you use this class.)

/**
*   This is a ToolTip out of class.
*   There're so so so so so so so many words I want to say, but here's too
narrow.
*   Please read this tooltip before you use this class.
*/
UCLASS(BlueprintType, Blueprintable)
class INSIDER_API UMyClass_ToolTip_TypeB :public UObject
{
    GENERATED_BODY()
};

UCLASS(BlueprintType, Blueprintable, meta = (ToolTip = "This is a ToolTip within
class. There're so so so so so so so many words I want to say, but here's too
narrow.\nThis is a new line.", ShortToolTip = "This is a ShortToolTip within
class."))

```

## ShortTooltip

---

- **Function Description:** Offers a more succinct version of the tooltip text, for instance, when displayed in a type selector
- **Usage Location:** Any
- **Metadata Type:** string="abc"
- **Related Items:** ToolTip

## EnumDisplayNameFn

---

- **Function Description:** Provides a callback function for custom display names of enum fields during Runtime
- **Usage Location:** UENUM
- **Engine Module:** Enum Property
- **Metadata Type:** string="abc"
- **Commonality:** ★★

Only effective during Runtime; it does not function in the Editor.

## Test Code:

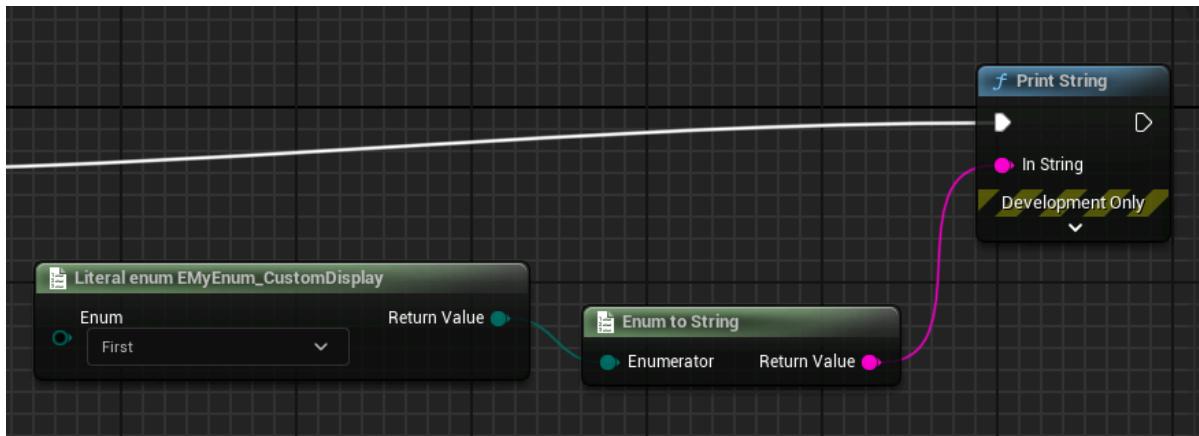
```
//[EMyEnum_CustomDisplay    Enum->Field->Object
/Script/Insider.EMyEnum_CustomDisplay]
//(BlueprintType = true, EnumDisplayNameFn = GetMyEnumCustomDisplayName,
First.Name = EMyEnum_CustomDisplay::First, IsBlueprintBase = true,
ModuleRelativePath = Enum/MyEnum_Test.h, Second.Name =
EMyEnum_CustomDisplay::Second, Third.Name = EMyEnum_CustomDisplay::Third)
// ObjectFlags: RF_Public | RF_Transient
// Outer: Package /Script/Insider
// EnumFlags: EEnumFlags::None
// EnumDisplayNameFn: 6adb4804
// CppType: EMyEnum_CustomDisplay
// CppForm: EnumClass
//{
// First = 0,
// Second = 1,
// Third = 2,
// EMyEnum_MAX = 3
//};

UENUM(Blueprintable, meta = (EnumDisplayNameFn = "GetMyEnumCustomDisplayName"))
enum class EMyEnum_CustomDisplay : uint8
{
    First,
    Second,
    Third,
};

extern FText GetMyEnumCustomDisplayName(int32 val);

FText GetMyEnumCustomDisplayName(int32 val)
{
    EMyEnum_CustomDisplay enumValue = (EMyEnum_CustomDisplay)val;
    switch (enumValue)
    {
        case EMyEnum_CustomDisplay::First:
            return FText::FromString(TEXT("My_First"));
        case EMyEnum_CustomDisplay::Second:
            return FText::FromString(TEXT("My_Second"));
        case EMyEnum_CustomDisplay::Third:
            return FText::FromString(TEXT("My_Third"));
        default:
            return FText::FromString(TEXT("Invalid MyEnum"));
    }
}
```

## Test Code:



The function setup for `EnumDisplayNameFn` is completed in `gen.cpp`, hence there is no need for it to be a UFUNCTION.

```
const UECodeGen_Private::FEnumParams
Z_Construct_UEnum_Insider_EMyEnum_CustomDisplay_Statics::EnumParams = {
    (uobject*)(*)() Z_Construct_UPackage_Script_Insider,
    GetMyEnumCustomDisplayName, //测试蓝图:
    "EMyEnum_CustomDisplay",
    "EMyEnum_CustomDisplay",
    Z_Construct_UEnum_Insider_EMyEnum_CustomDisplay_Statics::Enumerators,
    RF_Public|RF_Transient|RF_MarkAsNative,

    UE_ARRAY_COUNT(Z_Construct_UEnum_Insider_EMyEnum_CustomDisplay_Statics::Enumerators),
    EEnumFlags::None,
    (uint8)UEnum::ECppForm::EnumClass,

    METADATA_PARAMS(UE_ARRAY_COUNT(Z_Construct_UEnum_Insider_EMyEnum_CustomDisplay_Statics::Enum_MetaDataParams),
    Z_Construct_UEnum_Insider_EMyEnum_CustomDisplay_Statics::Enum_MetaDataParams)
};
```

## Test Blueprint:

```
/FText UEnum::GetDisplayNameTextByIndex(int32 NameIndex) const
{
    FString RawName = GetNameStringByIndex(NameIndex);

    if (RawName.IsEmpty())
    {
        return FText::GetEmpty();
    }

#if WITH_EDITOR
    FText LocalizedDisplayName;
    // In the editor, use metadata and localization to look up names
    static const FString Namespace = TEXT("UObjectDisplayNames");
    const FString Key = GetFullGroupName(false) + TEXT(".") + RawName;

```

```

FString NativeDisplayName;
if (HasMetaData(TEXT("DisplayName"), NameIndex))
{
    NativeDisplayName = GetMetaData(TEXT("DisplayName"), NameIndex);
}
else
{
    NativeDisplayName = FName::NameToDisplayString(RawName, false);
}

if (!FText::FindText(Namespace, Key, /*OUT*/&LocalizedDisplayName,
&NativeDisplayName))
{
    LocalizedDisplayName = FText::FromString(NativeDisplayName);
}

if (!LocalizedDisplayName.IsEmpty())
{
    return LocalizedDisplayName;
}
#endif
//这里！！！
if (EnumDisplayNameFn)
{
    return (*EnumDisplayNameFn)(NameIndex);
}

return FText::FromString(GetNameStringByIndex(NameIndex));
}

```

## Bitflags

---

- **Function Description:** Allows an enumeration to be set with bit flags for assignment, enabling its recognition as a BitMask within blueprints
- **Usage Location:** UENUM
- **Engine Module:** Enum Property
- **Metadata Type:** bool
- **Associated Items:** UseEnumValuesAsMaskValuesInEditor
- **Commonality:** ★★★★★

Frequently used in conjunction with bitmask on UPROPERTY.

Please pay attention to the distinction between this and UENUM(flags), which influences how string output functions work in C++.

This indicates that the enumeration supports bit flags, allowing it to be selected in blueprints.

```

UENUM(BlueprintType,Flags)
enum class EMyEnum_Flags:uint8
{
    First,
    Second,
    Third,
}

```

```

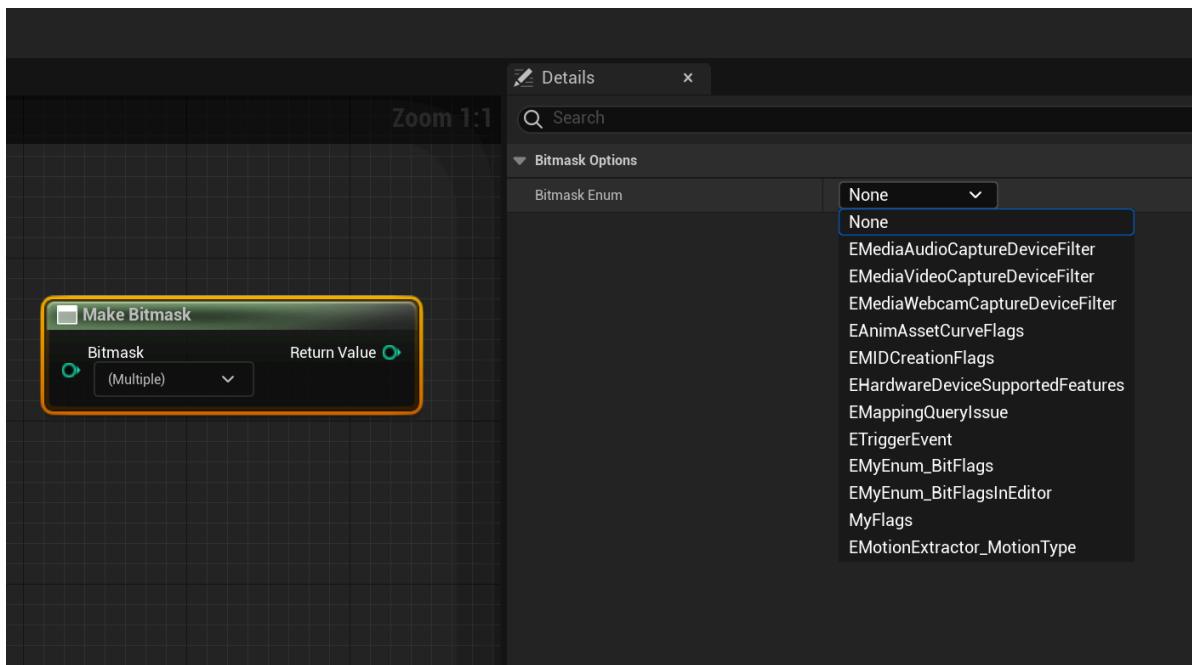
};

UENUM(BlueprintType, Meta = (Bitflags))
enum class EMyEnum_BitFlags:uint8
{
    First,
    Second,
    Third,
};

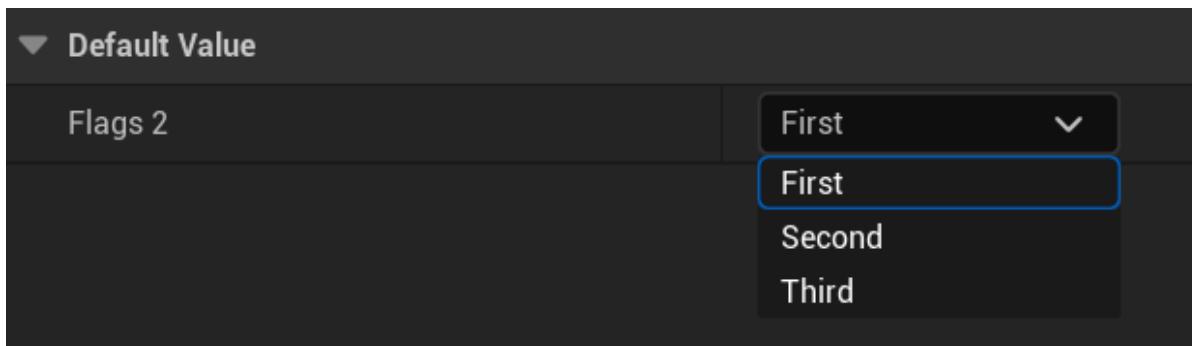
//Example in Source Code:
UENUM(Meta = (Bitflags))
enum class EColorBits
{
    ECB_Red,
    ECB_Green,
    ECB_Blue
};
UPROPERTY(EditAnywhere, Meta = (Bitmask, BitmaskEnum = "EColorBits"))
int32 ColorFlags;

```

As shown in the figure below: EMyEnum\_Flags will not be listed in the options. EMyEnum\_BitFlags, however, can be included.



If UPROPERTY(bitmask) is not used in conjunction, only single-item selection will be available in blueprints



# UseEnumValuesAsMaskValuesInEditor

- **Function description:** Indicates that the enumeration values are already shifted values, rather than the index subscripts of bit flags.
- **Use location:** UENUM
- **Metadata type:** bool
- **Related items:** Bitflags
- **Frequency of use:** ★★

Specifies that the enumeration values used in the editor are the final bit flag values, not the bits themselves. However, note that the definition in C++ retains the same format:

```
UENUM(BlueprintType, meta = (Bitflags, UseEnumValuesAsMaskValuesInEditor =
"true"))
enum class EMotionExtractor_MotionType : uint8
{
    None          = 0 UMETA(Hidden),
    Translation   = 1 << 0,
    Rotation      = 1 << 1,
    Scale         = 1 << 2,
    TranslationSpeed = 1 << 3,
    RotationSpeed   = 1 << 4,
};

UENUM(meta = (Bitflags))
enum class EPCGChangeType : uint8
{
    None = 0,
    Cosmetic = 1 << 0,
    Settings = 1 << 1,
    Input = 1 << 2,
    Edge = 1 << 3,
    Node = 1 << 4,
    Structural = 1 << 5
};
```

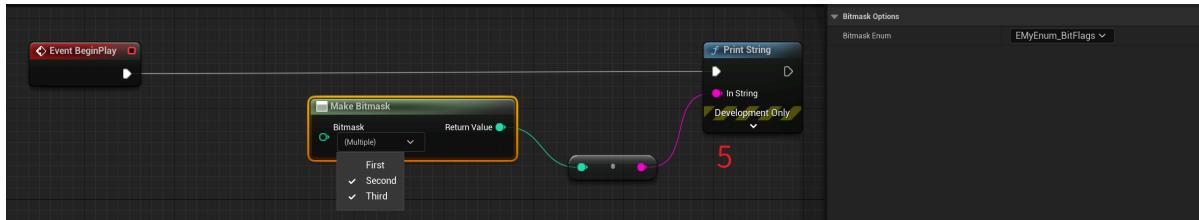
## Testing Code:

```
UENUM(BlueprintType,Meta = (Bitflags))
enum class EMyEnum_BitFlags:uint8
{
    First,//0
    Second,//1
    Third,//2
};

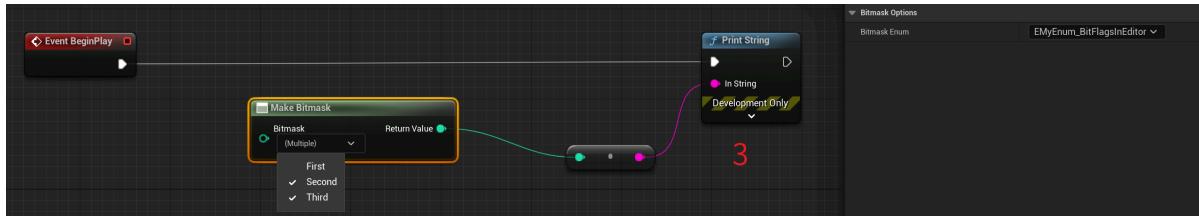
UENUM(BlueprintType, meta = (Bitflags, UseEnumValuesAsMaskValuesInEditor =
"true"))
enum class EMyEnum_BitFlagsInEditor:uint8
{
    First,//0
    Second,//1
```

```
Third, //2
};
```

## Test Blueprint 1:



## Test Blueprint 2:



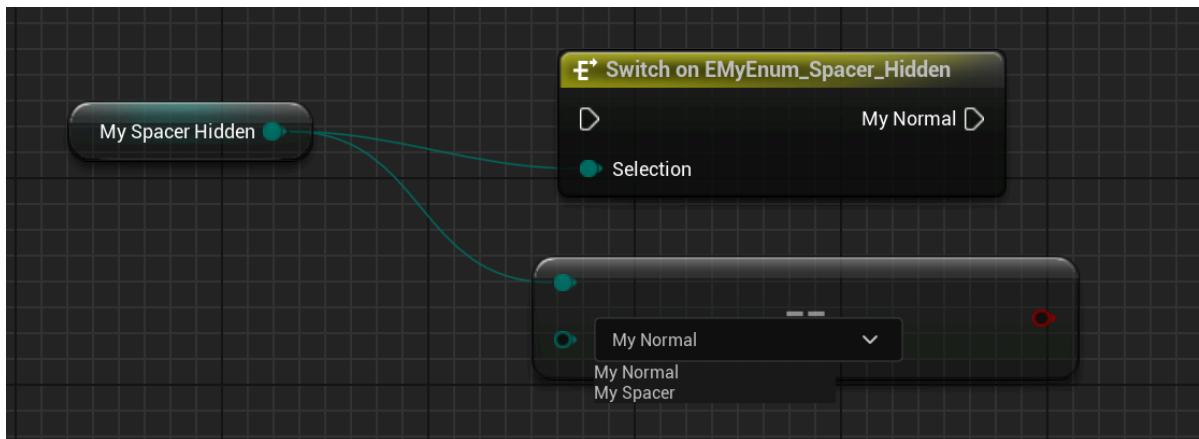
Thus, it can be observed that the former is  $1 \ll 2 + 1 \ll 2$ , whereas the latter is simply  $1 | 2$ , indicating that the latter treats the enumeration value as the already shifted value

## Spacer

- Function Description:** Hide a specific value within UENUM
- Usage Location:** UENUM::UMETA
- Engine Module:** Enum Property
- Metadata Type:** boolean
- Restriction Type:** UENUM
- Commonality:** ★★★★☆

Spacer and Hidden share similar functionalities. The sole distinction is that Spacer remains visible when compared using '==' in blueprints.

Thus, it is still advisable to use Hidden whenever the goal is to conceal enumeration values.



Refer to the Hidden section for additional sample code examples

# ValidEnumValues

- **Function description:** Specifies the optional enumeration values for an enumeration attribute
- **Use location:** UPROPERTY
- **Engine module:** Enum Property
- **Metadata type:** strings = "a, b, c"
- **Restriction type:** Enumeration attribute value
- **Related items:** InvalidEnumValues, GetRestrictedEnumValues, EnumValueDisplayNameOverrides, Enum
- **Frequency:** ★★☆

ValidEnumValues specifies the optional enumeration value options on enumeration attribute values. By default, enumeration properties in the details panel can be set to all enumeration values, but we can restrict the display to only these values using ValidEnumValues.

There are three ways to write enumeration attributes, namely enum class, TEnumAsByte and FString superimposed on enum meta. These three ways of writing will be regarded as an enumeration attribute and then try to generate a combo list to allow users to select attribute values.

## Sample Code:

```
UENUM(BlueprintType)
enum class EMyPropertyTestEnum : uint8
{
    First,
    Second,
    Third,
    Forth,
    Fifth,
};

UENUM(BlueprintType)
namespace EMyPropertyTestEnum2
{
    enum Type : int
    {
        First,
        Second,
        Third,
        Forth,
        Fifth,
    };
}

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Enum : public UObject
{
    GENERATED_BODY()

public:
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite)
EMyPropertyTestEnum MyEnum;

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ValidEnumValues =
"First,Second,Third"))
EMyPropertyTestEnum MyEnumWithValid; // Type 1

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ValidEnumValues =
"First,Second,Third"))
TEnumAsByte<EMyPropertyTestEnum2::Type> MyAnotherEnumWithValid; //Type 2

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (enum =
"EMyPropertyTestEnum"))
FString MyStringWithEnum; //Type 3

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (InvalidEnumValues =
"First,Second,Third"))
EMyPropertyTestEnum MyEnumWithInvalid = EMyPropertyTestEnum::Forth;

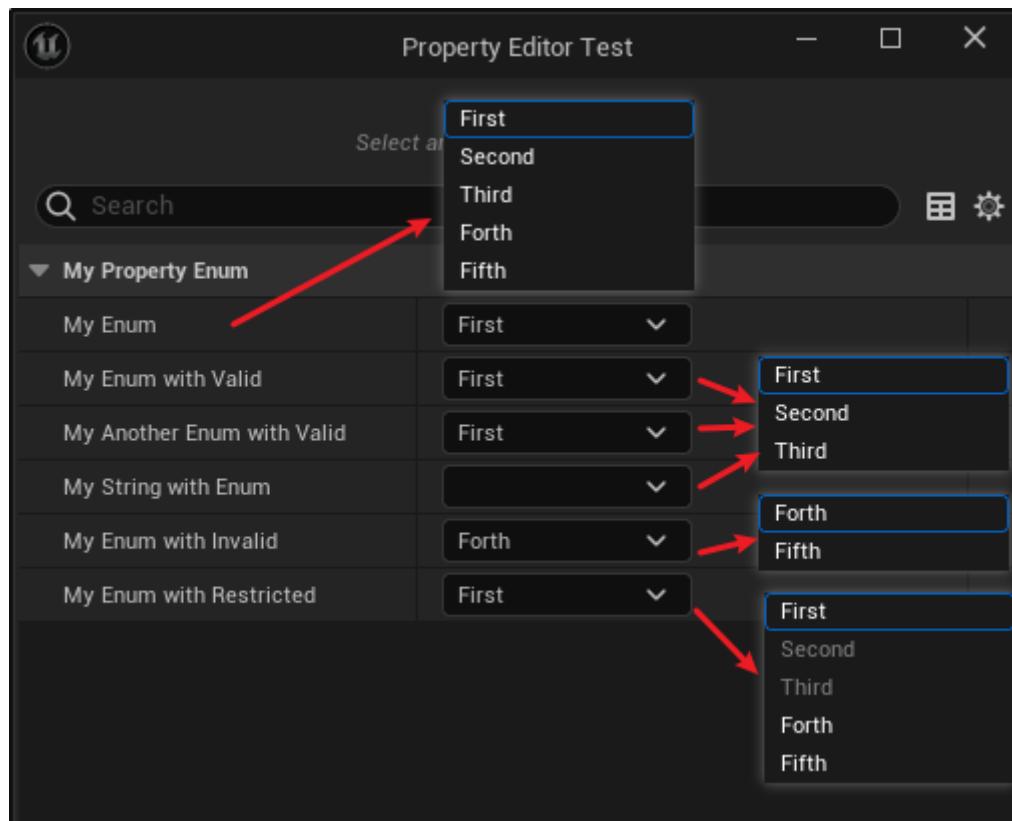
UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (GetRestrictedEnumValues =
"MyGetRestrictedEnumValues"))
EMyPropertyTestEnum MyEnumWithRestricted;

public:
UFUNCTION(BlueprintInternalUseOnly)
TArray<FString> MyGetRestrictedEnumValues() const{ return TArray<FString>
{"Second","Third"}; }
};

```

## Blueprint Effect:

It can be observed that, by default, the enumeration attribute displays all five enumeration values, but the optional lists for the other three enumeration attributes are restricted to three values.



# Principle:

The following code snippet,

```
bool FPropertyHandleBase::GeneratePossibleValues(TArray< TSharedPtr< FString>> &OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems)
{
    TArray< UObject*> OuterObjects;
    GetOuterObjects(OuterObjects);

    const TArray< FName > ValidEnumValues =
    PropertyEditorHelpers::GetValidEnumsFromPropertyOverride(Property, Enum);
    const TArray< FName > InvalidEnumValues =
    PropertyEditorHelpers::GetInvalidEnumsFromPropertyOverride(Property, Enum);
    const TArray< FName > RestrictedEnumValues =
    PropertyEditorHelpers::GetRestrictedEnumsFromPropertyOverride(OuterObjects,
    Property, Enum);

    const TMap< FName, FText > EnumValueDisplayNameOverrides =
    PropertyEditorHelpers::GetEnumValueDisplayNamesFromPropertyOverride(Property,
    Enum);

    //NumEnums() - 1, because the last item in an enum is the _MAX item
    for( int32 EnumIndex = 0; EnumIndex < Enum->NumEnums() - 1; ++EnumIndex )
    {
        // Ignore hidden enums
        bool bShouldBeHidden = Enum->HasMetaData(TEXT("Hidden"), EnumIndex) ||
        Enum->HasMetaData(TEXT("Spacer"), EnumIndex );
        if (!bShouldBeHidden)
        {
            if(ValidEnumValues.Num() > 0)
            {
                bShouldBeHidden = !ValidEnumValues.Contains(Enum-
                >GetNameByIndex(EnumIndex));
            }
            // If both are specified, InvalidEnumValues takes precedence
            else if(InvalidEnumValues.Num() > 0)
            {
                bShouldBeHidden = InvalidEnumValues.Contains(Enum-
                >GetNameByIndex(EnumIndex));
            }
        }

        if (!bShouldBeHidden)
        {
            bShouldBeHidden = ISHidden(Enum->GetNameStringByIndex(EnumIndex));
        }

        if( !bShouldBeHidden )
        {
            // See if we specified an alternate name for this value using
            metadata
            FString EnumName = Enum->GetNameStringByIndex(EnumIndex);
            FString EnumDisplayName = EnumValueDisplayNameOverrides.FindRef(Enum-
            >GetNameByIndex(EnumIndex)).ToString();
        }
    }
}
```

```

        if (EnumDisplayName.IsEmpty())
        {
            EnumDisplayName = Enum-
>GetDisplayNameTextByIndex(EnumIndex).ToString();
        }

        FText RestrictionTooltip;
        const bool bIsRestricted = GenerateRestrictionToolTip(EnumName,
RestrictionTooltip) || RestrictedEnumValues.Contains(Enum-
>GetNameByIndex(EnumIndex));
        OutRestrictedItems.Add(bIsRestricted);

        if (EnumDisplayName.Len() == 0)
        {
            EnumDisplayName = MoveTemp(EnumName);
        }
        else
        {
            bUsesAlternateDisplayValues = true;
        }

        TSharedPtr< FString > EnumStr(new FString(EnumDisplayName));
        OutOptionStrings.Add(EnumStr);

        FText EnumValueToolTip = bIsRestricted ? RestrictionTooltip : Enum-
>GetToolTipTextByIndex(EnumIndex);
        OutToolTips.Add(MoveTemp(EnumValueToolTip));
    }
    else
    {
        OutToolTips.Add(FText());
    }
}
}

```

## InvalidEnumValues

---

- **Function Description:** Specifies the disallowed enumeration values for an enumeration attribute, used to exclude certain options
- **Usage Location:** UPROPERTY
- **Engine Module:** Enum Property
- **Metadata Type:** strings = "a, b, c"
- **Restriction Type:** Enumeration Attribute Value
- **Associated Items:** ValidEnumValues
- **Commonly Used:** ★★★

If InvalidEnumValues and ValidEnumValues are specified at the same time, and the values therein overlap, InvalidEnumValues shall prevail: this enumeration value is illegal.

# GetRestrictedEnumValues

- **Function Description:** Specifies a function to determine which enum options are disabled for an enum attribute value
- **Usage Location:** UPROPERTY
- **Engine Module:** Enum Property
- **Metadata Type:** string="abc"
- **Restriction Type:** TArray FuncName() const;
- **Associated Items:** ValidEnumValues
- **Commonality:** ★★★

The distinction between "Restricted" and "Invalid" lies in:

'Invalid' will hide the option value

'Restricted' will still display the option value, but it will appear grayed out and cannot be selected.

The specified function name must be a UFUNCTION, allowing the function to be located by name.

# EnumValueDisplayNameOverrides

- **Function Description:** Modify the display names of enumeration property values
- **Usage Location:** UPROPERTY
- **Engine Module:** Enum Property
- **Metadata Type:** string="abc"
- **Associated Items:** ValidEnumValues
- **Commonality:** ★★

Rename the enumeration values on the enumeration attributes. You can change multiple names. Simply list them in the format of "A=B; C=D". The collected information is in the TMap<FName, FText> mapping format, so both the original enumeration value names and their corresponding new display names must be provided.

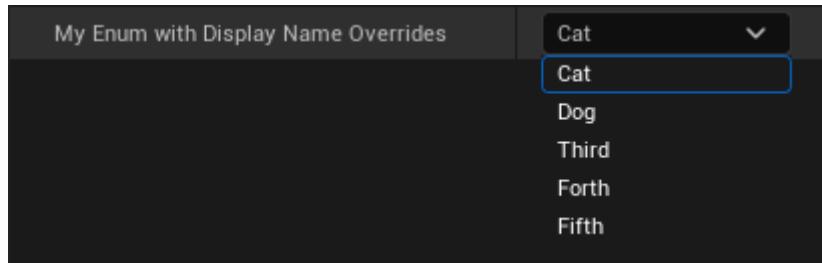
## Test Code:

```
UENUM(BlueprintType)
enum class EMyPropertyTestEnum : uint8
{
    First,
    Second,
    Third,
    Forth,
    Fifth,
};

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EnumValueDisplayNameOverrides
= "First=Cat;Second=Dog"))
EMyPropertyTestEnum MyEnumWithDisplayNameOverrides;
```

## Blueprint Effect:

It is evident that the display names for "First" and "Second" have been altered.



For the code implementation of the principle, please refer to the code on ValidEnumValues

## Enum

- **Function Description:** Assigns an enumeration's value name as an option for a specified String
- **Usage Location:** UPROPERTY
- **Engine Module:** Enum Property
- **Metadata Type:** string="abc"
- **Restriction Type:** FString
- **Associated Items:** ValidEnumValues
- **Commonly Used:** ★★★

## DisplayName

- **Function Description:** Modify the display name of an enumeration value
- **Usage Location:** UENUM::UMETA
- **Engine Module:** Enum Property
- **Metadata Type:** string="abc"
- **Commonly Used:** ★★★★☆

Modify the display name of enumeration values

## Sample Code:

```
/*
[enum 602d0d4e680 EMyEnum_HasDisplayName      Enum->Field->Object
/Script/Insider.EMyEnum_HasDisplayName]
(BlueprintType = true, First.DisplayName = Dog, First.Name =
EMyEnum_HasDisplayName::First, IsBlueprintBase = true, ModuleRelativePath =
Enum/MyEnum_Test.h, Second.DisplayName = Cat, Second.Name =
EMyEnum_HasDisplayName::Second, Third.DisplayName = Pig, Third.Name =
EMyEnum_HasDisplayName::Third)
ObjectFlags:    RF_Public | RF_Transient
Outer:  Package /Script/Insider
EnumFlags:  None
EnumDisplayNameFn:  0
CppType:    EMyEnum_HasDisplayName
```

```

CppForm:    EnumClass
{
    First = 0,
    Second = 1,
    Third = 2,
    EMyEnum_MAX = 3
};

*/
UENUM(Blueprintable)
enum class EMyEnum_HasDisplayName : uint8
{
    First UMETA(DisplayName="Dog"),
    Second UMETA(DisplayName="Cat"),
    Third UMETA(DisplayName="Pig"),
};

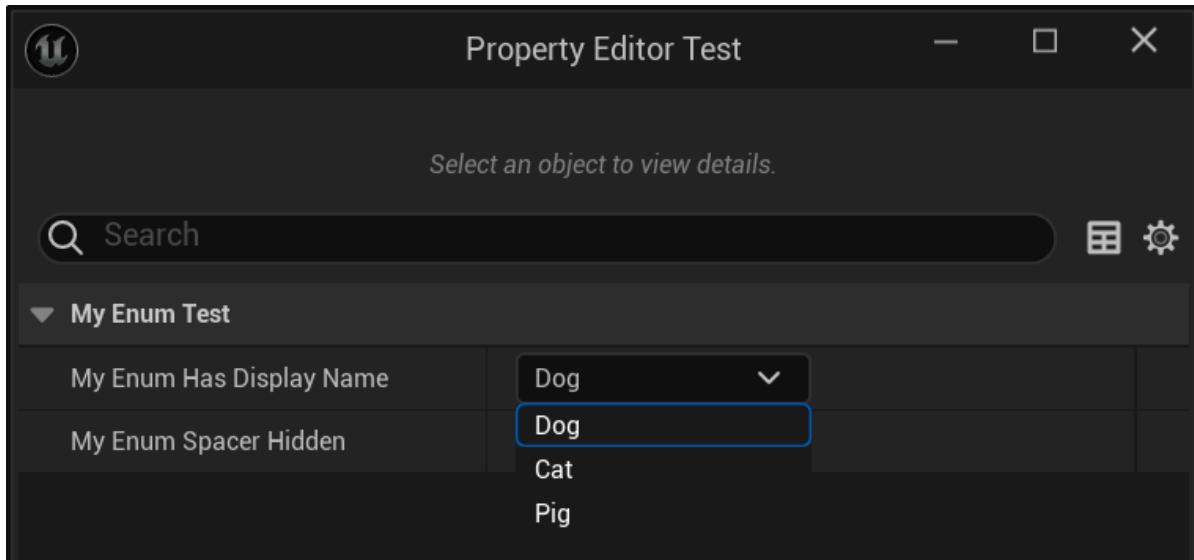
UCLASS(BlueprintType)
class INSIDER_API UMyEnum_Test : public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    EMyEnum_HasDisplayName MyEnum_HasDisplayName;
}

```

## Example Effect:

It is evident that the name has been changed.



## Hidden

- **Function Description:** Hide a specific value within a UENUM
- **Usage Location:** UENUM::UMETA
- **Engine Module:** Enum Property
- **Metadata Type:** bool
- **Restriction Type:** UENUM Value
- **Commonliness:** ★★★★☆

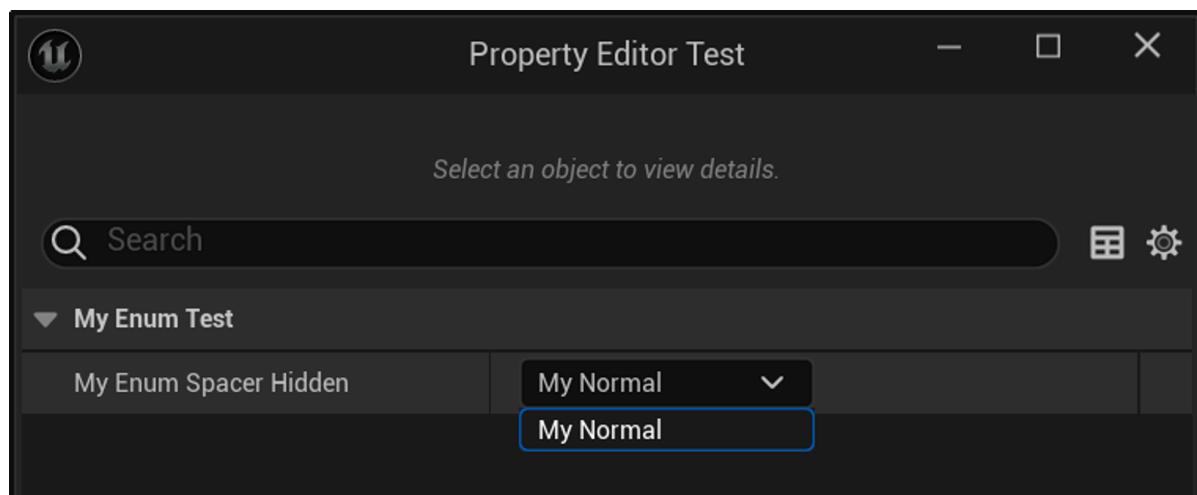
Hide a specific value in UENUM.

## Test Code:

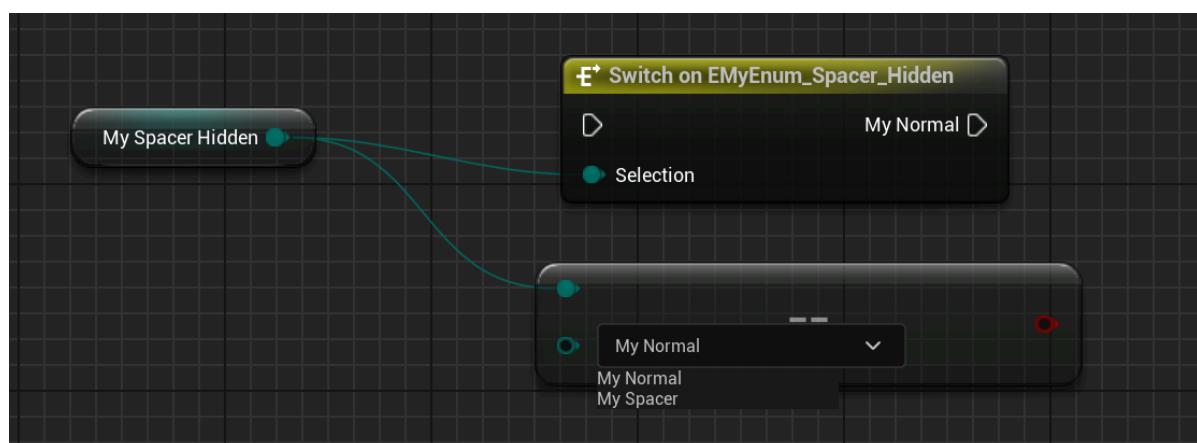
```
UENUM(Blueprintable, BlueprintType)
enum class EMyEnum_Spacer_Hidden : uint8
{
    MyNormal,
    MySpacer UMETA(Spacer),
    MyHidden UMETA(Hidden),
};

UCLASS(BlueprintType)
class INSIDER_API UMyEnum_Test : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    EMyEnum_Spacer_Hidden MyEnum_Spacer_Hidden;
};
```

## Test Effects:



However, when accessed in Blueprint:



## Principle:

In the attribute details panel, when generating the possible values of an enumeration, the Hidden and Spacer options are evaluated to determine what to hide.

Yet, in elements like SEnumComboBox and SGraphPinEnum that are displayed within Blueprints, only the Hidden option is considered, and Spacer is not (perhaps overlooked?) evaluated, thus MySpacer will still be visible.

```
bool FPropertyHandleBase::GeneratePossibleValues(TArray< FString>& OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems, TArray< FText *> OutDisplayNames)
{
    // Ignore hidden enums
    bool bShouldBeHidden = Enum->HasMetaData(TEXT("Hidden"), EnumIndex) || Enum->HasMetaData(TEXT("Spacer"), EnumIndex);
    if (!bShouldBeHidden)
    {
        if (ValidEnumValues.Num() > 0)
        {
            bShouldBeHidden = !ValidEnumValues.Contains(Enum->GetNameByIndex(EnumIndex));
        }
        // If both are specified, InvalidEnumValues takes precedence
        else if (InvalidEnumValues.Num() > 0)
        {
            bShouldBeHidden = InvalidEnumValues.Contains(Enum->GetNameByIndex(EnumIndex));
        }
    }
}

void SEnumComboBox::Construct(const FArguments& InArgs, const UEnum* InEnum)
{
    if (Enum->HasMetaData(TEXT("Hidden"), Index) == false)
    {
        VisibleEnums.Emplace(Index, Enum->GetValueByIndex(Index), Enum->GetDisplayNameTextByIndex(Index), Enum->GetToolTipTextByIndex(Index));
    }
}

void SGraphPinEnum::GenerateComboBoxIndexes( TArray< TSharedPtr< int32 > >& OutComboBoxIndexes )
{
    // Ignore hidden enum values
    if( !EnumPtr->HasMetaData(TEXT("Hidden"), EnumIndex) )
    {
        TSharedPtr< int32 > EnumIdxPtr(new int32(EnumIndex));
        OutComboBoxIndexes.Add(EnumIdxPtr);
    }
}
```

# DisplayValue

- **Function Description:** Enum /Script/Engine.AnimPhysCollisionType
- **Usage Location:** UENUM::UMETA
- **Engine Module:** Enum Property
- **Commonality Level:** 0

## Source Code Example:

```
UENUM()
enum class AnimPhysCollisionType : uint8
{
    CoM UMETA(DisplayName="CoM", DisplayValue="CoM", ToolTip="Only limit the
center of mass from crossing planes."),
    CustomSphere UMETA(ToolTip="Use the specified sphere radius to collide with
planes."),
    InnerSphere UMETA(ToolTip="Use the largest sphere that fits entirely within
the body extents to collide with planes."),
    OuterSphere UMETA(ToolTip="Use the smallest sphere that wholly contains the
body extents to collide with planes.")
};
```

# Grouping

- **Function Description:** Enum /Script/Engine.EAlphaBlendOption
- **Usage Location:** UENUM::UMETA
- **Engine Module:** Enum Property
- **Metadata Type:** boolean
- **Commonality:** Rarely Used

Appears to be utilized within the Sequencer, exclusively in SEasingFunctionGridWidget.

## Source Code Example:

```
UENUM()
enum class EAlphaBlendOption : uint8
{
    Linear = 0 UMETA(Grouping = Linear, DisplayName = "Linear", ToolTip = "Linear
interpolation"),
    Cubic UMETA(Grouping = Cubic, DisplayName = "Cubic In", ToolTip = "Cubic-in
interpolation"),
    HermiteCubic UMETA(Grouping = Cubic, DisplayName = "Hermite-Cubic InOut",
ToolTip = "Hermite-Cubic"),
    Sinusoidal UMETA(Grouping = Sinusoidal, DisplayName = "Sinusoidal", ToolTip =
"Sinusoidal interpolation"),
    QuadraticInOut UMETA(Grouping = Quadratic, DisplayName = "Quadratic InOut",
ToolTip = "Quadratic in-out interpolation"),
    CubicInOut UMETA(Grouping = Cubic, DisplayName = "Cubic InOut", ToolTip =
"Cubic in-out interpolation"),
```

```

        QuarticInOut UMETA(Grouping = Quartic, DisplayName = "Quartic InOut", ToolTip
= "Quartic in-out interpolation"),
        QuinticInOut UMETA(Grouping = Quintic, DisplayName = "Quintic InOut", ToolTip
= "Quintic in-out interpolation"),
        CircularIn UMETA(Grouping = Circular, DisplayName = "Circular In", ToolTip =
"Circular-in interpolation"),
        CircularOut UMETA(Grouping = Circular, DisplayName = "Circular Out", ToolTip =
"Circular-out interpolation"),
        CircularInOut UMETA(Grouping = Circular, DisplayName = "Circular InOut",
ToolTip = "Circular in-out interpolation"),
        ExpIn UMETA(Grouping = Exponential, DisplayName = "Exponential In", ToolTip =
"Exponential-in interpolation"),
        ExpOut UMETA(Grouping = Exponential, DisplayName = "Exponential Out", ToolTip =
"Exponential-Out interpolation"),
        ExpInOut UMETA(Grouping = Exponential, DisplayName = "Exponential InOut",
ToolTip = "Exponential in-out interpolation"),
        Custom UMETA(Grouping = Custom, DisplayName = "Custom", ToolTip = "Custom
interpolation, will use custom curve inside an FAlphaBlend or linear if none has
been set"),
    };

UENUM()
enum class EMovieSceneBuiltInEasing : uint8
{
    // Linear easing
    Linear UMETA(Grouping=Linear,DisplayName="Linear"),
    // Sinusoidal easing
    SinIn UMETA(Grouping=Sinusoidal,DisplayName="Sinusoidal In"), SinOut
UMETA(Grouping=Sinusoidal,DisplayName="Sinusoidal Out"), SinInOut
UMETA(Grouping=Sinusoidal,DisplayName="Sinusoidal InOut"),
    // Quadratic easing
    QuadIn UMETA(Grouping=Quadratic,DisplayName="Quadratic In"), QuadOut
UMETA(Grouping=Quadratic,DisplayName="Quadratic Out"), QuadInOut
UMETA(Grouping=Quadratic,DisplayName="Quadratic InOut"),
    // Cubic easing
    Cubic UMETA(Grouping = Cubic, DisplayName = "Cubic"), CubicIn
UMETA(Grouping=Cubic,DisplayName="Cubic In"), CubicOut
UMETA(Grouping=Cubic,DisplayName="Cubic Out"), CubicInOut
UMETA(Grouping=Cubic,DisplayName="Cubic InOut"), HermiteCubicInOut UMETA(Grouping
= Cubic, DisplayName = "Hermite-Cubic InOut"),
    // Quartic easing
    QuartIn UMETA(Grouping=Quartic,DisplayName="Quartic In"), QuartOut
UMETA(Grouping=Quartic,DisplayName="Quartic Out"), QuartInOut
UMETA(Grouping=Quartic,DisplayName="Quartic InOut"),
    // Quintic easing
    QuintIn UMETA(Grouping=Quintic,DisplayName="Quintic In"), QuintOut
UMETA(Grouping=Quintic,DisplayName="Quintic Out"), QuintInOut
UMETA(Grouping=Quintic,DisplayName="Quintic InOut"),
    // Exponential easing
    ExpoIn UMETA(Grouping=Exponential,DisplayName="Exponential In"), ExpoOut
UMETA(Grouping=Exponential,DisplayName="Exponential Out"), ExpoInOut
UMETA(Grouping=Exponential,DisplayName="Exponential InOut"),
    // Circular easing
    CircIn UMETA(Grouping=Circular,DisplayName="Circular In"), CircOut
UMETA(Grouping=Circular,DisplayName="Circular Out"), CircInOut
UMETA(Grouping=Circular,DisplayName="Circular InOut"),

```

```

    // Custom
    Custom UMETA(Grouping = Custom, DisplayName = "Custom"),
};


```

## Principle:

```

TArray<SEasingFunctionGridWidget::FGroup>
SEasingFunctionGridWidget::ConstructGroups(const TSet<EMovieSceneBuiltInEasing>&
FilterExclude)
{
    const UEnum* EasingEnum = StaticEnum<EMovieSceneBuiltInEasing>();
    check(EasingEnum)

    TArray<FGroup> Groups;

    for (int32 NameIndex = 0; NameIndex < EasingEnum->NumEnums() - 1;
++NameIndex)
    {
        const FString& Grouping = EasingEnum->GetMetaData(TEXT("Grouping"),
NameIndex);
        EMovieSceneBuiltInEasing Value = (EMovieSceneBuiltInEasing)EasingEnum-
>GetValueByIndex(NameIndex);

        if (FilterExclude.IsEmpty() || FilterExclude.Find(Value) == nullptr)
        {
            FindOrAddGroup(Groups, Grouping).Values.Add(Value);
        }
    }

    return Groups;
}

```

## TraceQuery

- **Function Description:** Enum /Script/Engine.ECollisionChannel
- **Usage Location:** UENUM::UMETA
- **Engine Module:** Enum Property
- **Metadata Type:** boolean
- **Commonality:** 0

Used exclusively with ECollisionChannel to indicate which channels are utilized for tracing.

## Source Code Example:

```

UENUM(BlueprintType)
enum ECollisionChannel : int
{
    ECC_WorldStatic UMETA(DisplayName="WorldStatic"),
    ECC_WorldDynamic UMETA(DisplayName="WorldDynamic"),
    ECC_Pawn UMETA(DisplayName="Pawn"),
}

```

```

ECC_Visibility UMETA(DisplayName="Visibility" , TraceQuery="1"),
ECC_Camera UMETA(DisplayName="Camera" , TraceQuery="1"),
ECC_PhysicsBody UMETA(DisplayName="PhysicsBody"),
ECC_Vehicle UMETA(DisplayName="Vehicle"),
ECC_Destructible UMETA(DisplayName="Destructible"),

/** Reserved for gizmo collision */
ECC_EngineTraceChannel1 UMETA(Hidden),

ECC_EngineTraceChannel2 UMETA(Hidden),
ECC_EngineTraceChannel3 UMETA(Hidden),
ECC_EngineTraceChannel4 UMETA(Hidden),
ECC_EngineTraceChannel5 UMETA(Hidden),
ECC_EngineTraceChannel6 UMETA(Hidden),

ECC_GameTraceChannel1 UMETA(Hidden),
ECC_GameTraceChannel2 UMETA(Hidden),
ECC_GameTraceChannel3 UMETA(Hidden),
ECC_GameTraceChannel4 UMETA(Hidden),
ECC_GameTraceChannel5 UMETA(Hidden),
ECC_GameTraceChannel6 UMETA(Hidden),
ECC_GameTraceChannel7 UMETA(Hidden),
ECC_GameTraceChannel8 UMETA(Hidden),
ECC_GameTraceChannel9 UMETA(Hidden),
ECC_GameTraceChannel10 UMETA(Hidden),
ECC_GameTraceChannel11 UMETA(Hidden),
ECC_GameTraceChannel12 UMETA(Hidden),
ECC_GameTraceChannel13 UMETA(Hidden),
ECC_GameTraceChannel14 UMETA(Hidden),
ECC_GameTraceChannel15 UMETA(Hidden),
ECC_GameTraceChannel16 UMETA(Hidden),
ECC_GameTraceChannel17 UMETA(Hidden),
ECC_GameTraceChannel18 UMETA(Hidden),

/** Add new serializeable channels above here (i.e. entries that exist in
FCollisionResponseContainer) */

/** Add only nonserialized/transient flags below */

// NOTE!!!! THESE ARE BEING DEPRECATED BUT STILL THERE FOR BLUEPRINT. PLEASE
DO NOT USE THEM IN CODE

ECC_OverlapAll_Deprecated UMETA(Hidden),
ECC_MAX,
};


```

## Principle: The Fundamental Concept

```

void UCollisionProfile::LoadProfileConfig(bool bForceInit)
{
    static const FString TraceType = TEXT("TraceQuery");
}

```

# Bitmask

- **Function Description:** Assign an attribute using Bitmask
- **Usage Location:** UPROPERTY
- **Engine Module:** Enum Property
- **Metadata Type:** bool
- **Restricted Type:** int32 used to represent enumeration values
- **Associated Items:** BitmaskEnum
- **Commonality:** ★★★★☆

This attribute does not have a fixed relationship with the definition of an enum, and therefore can be defined independently.

```
UENUM(BlueprintType)
enum class EMyEnum_Normal:uint8
{
    First,
    Second,
    Third,
};

UENUM(BlueprintType,Flags)
enum class EMyEnum_Flags:uint8
{
    First,
    Second,
    Third,
};

UENUM(BlueprintType,Meta = (Bitflags))
enum class EMyEnum_BitFlags:uint8
{
    First,
    Second,
    Third,
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_EnumBitFlags_Test:public AActor
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, Meta = (Bitmask, BitmaskEnum = "EMyEnum_Normal"))
    int32 MyNormal;

    UPROPERTY(EditAnywhere, Meta = (Bitmask, BitmaskEnum = "EMyEnum_Flags"))
    int32 MyFlags;

    UPROPERTY(EditAnywhere, Meta = (Bitmask, BitmaskEnum = "EMyEnum_BitFlags"))
    int32 MyBitFlags;
};
```

It can be defined in blueprints using this attribute

▼ My Actor Enum Bit Flags Test		
My Normal	First   Second	▼
My Flags	First   Second	▼
My Bit Flags	First   Second	▼

BitmaskEnum can be used to further specify enumeration values

## BitmaskEnum

- **Function description:** The name of the enumeration used after employing bit flags
- **Usage location:** UPROPERTY
- **Metadata type:** string="abc"
- **Restricted type:** An int32 used to represent enum values
- **Associated items:** Bitmask
- **Commonly used:** ★★★★☆

If the BitmaskEnum is not tagged, the tagged name value cannot be provided

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_EnumBitFlags_Test:public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, Meta = (Bitmask))
    int32 MyNormalWithoutEnum;
};
```

If the BitmaskEnum is not tagged, the tagged name value cannot be provided

Start with Tick Enabled	
Tick Interval (secs)	Flag 1
Allow Tick Before Begin Play	Flag 2
► Advanced	Flag 3
▼ Rendering	Flag 4
Actor Hidden In Game	Flag 5
Editor Billboard Scale	Flag 6
▼ Actor	Flag 7
Can be Damaged	Flag 8
Initial Life Span	Flag 9
Spawn Collision Handling Method	Flag 10
► Advanced	Flag 11
▼ Collision	Flag 12
Generate Overlap Events During Level Stream...	Flag 13
Update Overlaps Method During Level Stream...	Flag 14
Default Update Overlaps Method During Level...	Flag 15
► Advanced	Flag 16
▼ HLOD	Flag 17
Include Actor in HLOD	Flag 18
HLOD Layer	Flag 19
▼ Input	Flag 20
Block Input	Flag 21
Auto Receive Input	Flag 22
Input Priority	Flag 23
▼ My Actor Enum Bit Flags Test	Flag 24
My Normal Without Enum	(No Flags Set) ▾

# FieldNotifyInterfaceParam

- **Function description:** Specifies that a certain parameter of the function provides information for the FieldNotify ViewModel.
- **Use location:** UFUNCTION
- **Engine module:** FieldNotify
- **Metadata type:** string="abc"
- **Restricted type:** The function contains other FFieldNotificationId parameters
- **Frequency of use:** ★★★

Specifies a parameter within the function to provide information for FieldNotify's ViewModel.

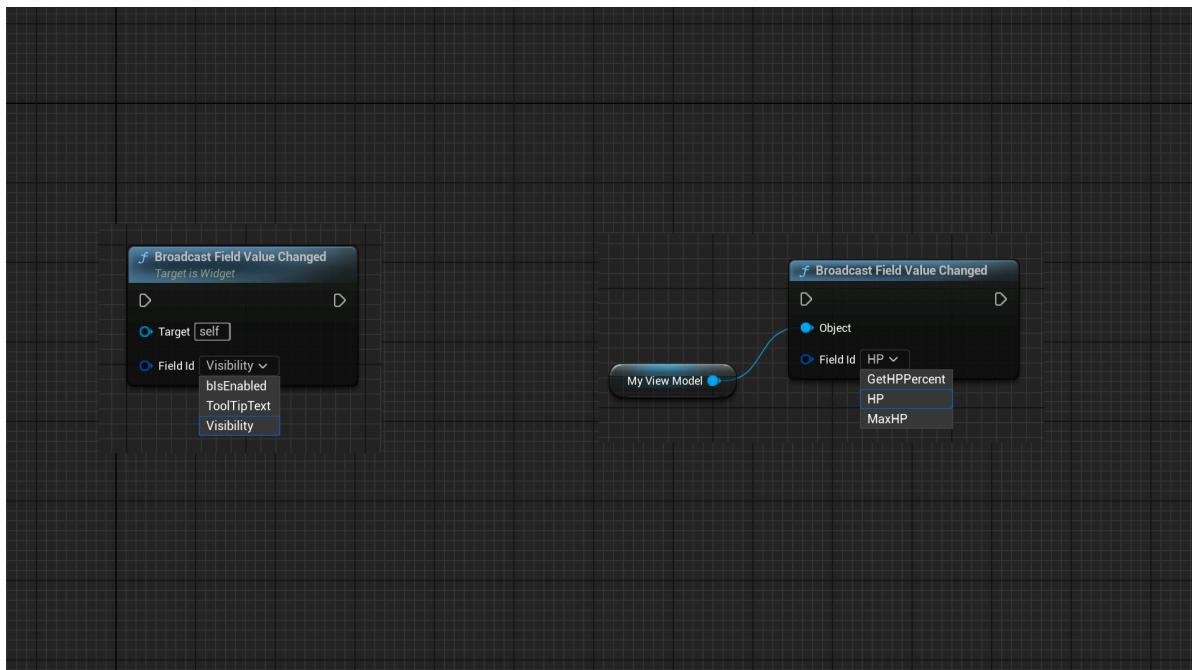
This parameter serves as context for subsequent FFieldNotificationId parameters, allowing the FieldId selection box to determine which values are available for selection.

## Source Code Example:

```
/** Broadcast that the Field value changed. */
UFUNCTION(BlueprintCallable, Category = "FieldNotification", meta =
(FieldNotifyInterfaceParam="Object", DisplayName = "Broadcast Field Value
Changed"))
    static void BroadcastFieldValueChanged(UObject* Object, FFieldNotificationId
FieldId);
```

## Blueprint Visualization:

When tested in a UserWidget, it is observed that if the Target parameter is not connected, it defaults to the current UserWidget, with the FieldId having three possible values. However, after connecting to a custom ViewModel, the FieldId changes to the values defined below.



# Working Principle:

```
TSharedRef<SWidget> SFieldNotificationGraphPin::GetDefaultValueWidget()
{
    UEdGraphPin* SelfPin = GraphPinObj->GetOwningNode()-
>FindPin(UEdGraphSchema_K2::PSC_Self);
    if (UK2Node_CallFunction* CallFunction = Cast<UK2Node_CallFunction>
(GraphPinObj->GetOwningNode()))
    {
        if (UFunction* Function = CallFunction->GetTargetFunction())
        {
            const FString& PinName = Function-
>GetMetaData("FieldNotifyInterfaceParam");
            if (PinName.Len() != 0)
            {
                SelfPin = GraphPinObj->GetOwningNode()->FindPin(*PinName);
            }
        }
    }

    return SNew(SFieldNotificationPicker)
        .value(this, &SFieldNotificationGraphPin::GetValue)
        .onValueChanged(this, &SFieldNotificationGraphPin::SetValue)
        .FromClass_Static/Private::GetPinClass, SelfPin)
        .visibility(this, &SGraphPin::GetDefaultValueVisibility);
}
```

## HideInDetailView

- **Function Description:** Hide the properties of the UAttributeSet subclass within the FGameplayAttribute option list.
- **Usage Locations:** UCLASS, UPROPERTY
- **Engine Module:** GAS
- **Metadata Type:** bool
- **Restriction Type:** UAttributeSet
- **Related Items:** HideFromModifiers, SystemGameplayAttribute
- **Commonliness:** ★★★

Hide the properties of the UAttributeSet subclass within the FGameplayAttribute option list.

Can be applied to UCLASS to hide all properties within an entire class, or to a specific property to hide only that property.

An example used in the source code is UAbilitySystemTestAttributeSet, as it is specifically designed for testing purposes and is intended not to interfere with the standard option list.

## Test Code:

```
UCLASS()
class UMyAttributeSet : public UAttributeSet
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core")
    float HP = 100.f;

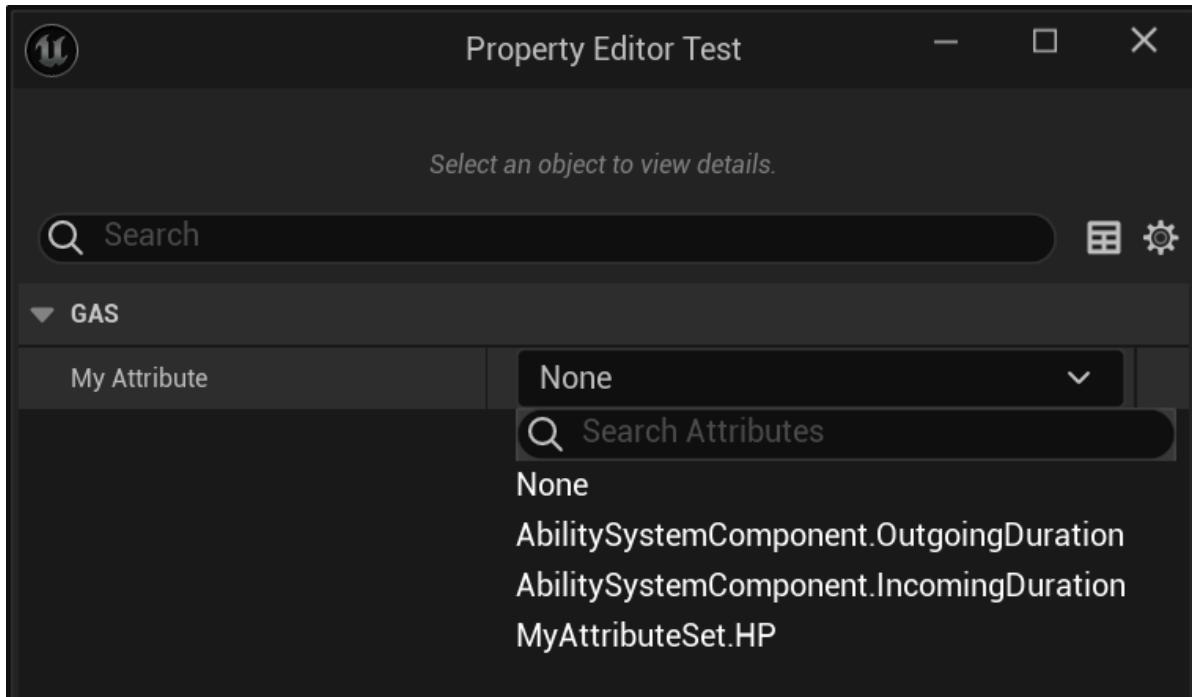
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core", meta =
(HideInDetailsView))
    float HP_HideInDetailsView = 100.f;
};

UCLASS(meta = (HideInDetailsView))
class UMyAttributeSet_Hide : public UAttributeSet
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core")
    float HP = 100.f;
};

UCLASS()
class UMyAttributeSetTest : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
    FGameplayAttribute MyAttribute;
};
```

## Test Effects:

Only the UMyAttributeSet.HP property is selectable, with options in UMyAttributeSet\_Hide being inaccessible.



## Principle:

```
PropertyModule.RegisterCustomPropertyTypeLayout( "GameplayAttribute",
FOnGetPropertyTypeCustomizationInstance::CreateStatic(
&FAttributePropertyParams::MakeInstance ) );

void FGameplayAttribute::GetAllAttributeProperties(TArray<FProperty*>&
OutProperties, FString FilterMetaStr, bool UseEditorOnlyData)
{
    for (TObjectIterator<UClass> ClassIt; ClassIt; ++ClassIt)
    {
        if (UseEditorOnlyData)
        {
            #if WITH_EDITOR
            // Allow entire classes to be filtered globally
            if (Class->HasMetaData(TEXT("HideInDetailsView")))
            {
                continue;
            }
            #endif
        }

        for (TFieldIterator<FProperty> PropertyIt(Class,
EFieldIteratorFlags::ExcludeSuper); PropertyIt; ++PropertyIt)
        {
            FProperty* Property = *PropertyIt;

            if (UseEditorOnlyData)
            {
                // Allow properties to be filtered globally (never show up)
                if (Property->HasMetaData(TEXT("HideInDetailsView")))
                {
                    continue;
                }
            }
        }
    }
}
```

```

        }

        OutProperties.Add(Property);
    }
}

```

# SystemGameplayAttribute

- **Function Description:** Exposes attributes from UAbilitySystemComponent subclasses to the FGameplayAttribute dropdown menu.
- **Usage Location:** UPROPERTY
- **Engine Module:** GAS
- **Metadata Type:** bool
- **Restriction Type:** Attributes within UAbilitySystemComponent subclasses
- **Associated Items:** HideInDetailsView
- **Commonality:** ★★★

Expose attributes from UAbilitySystemComponent subclasses to the FGameplayAttribute dropdown menu.

## Source Code Example:

```

UCLASS(ClassGroup=AbilitySystem, hidecategories=
(Object,LOD,Lighting,Transform,Sockets,TextureStreaming), editinlinenew, meta=
(BlueprintSpawnableComponent))
class GAMEPLAYABILITIES_API UAbilitySystemComponent : public
UGameplayTasksComponent, public IGameplayTagAssetInterface, public
IAbilitySystemReplicationProxyInterface
{
    /** Internal Attribute that modifies the duration of gameplay effects created
    by this component */
    UPROPERTY(meta=(SystemGameplayAttribute="true"))
    float OutgoingDuration;

    /** Internal Attribute that modifies the duration of gameplay effects applied
    to this component */
    UPROPERTY(meta = (SystemGameplayAttribute = "true"))
    float IncomingDuration;
}

```

## Test Code:

```

UCLASS(Blueprintable, BlueprintType)
class UMyAbilitySystemComponent : public UAbilitySystemComponent
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
    float MyFloat;
}

```

```

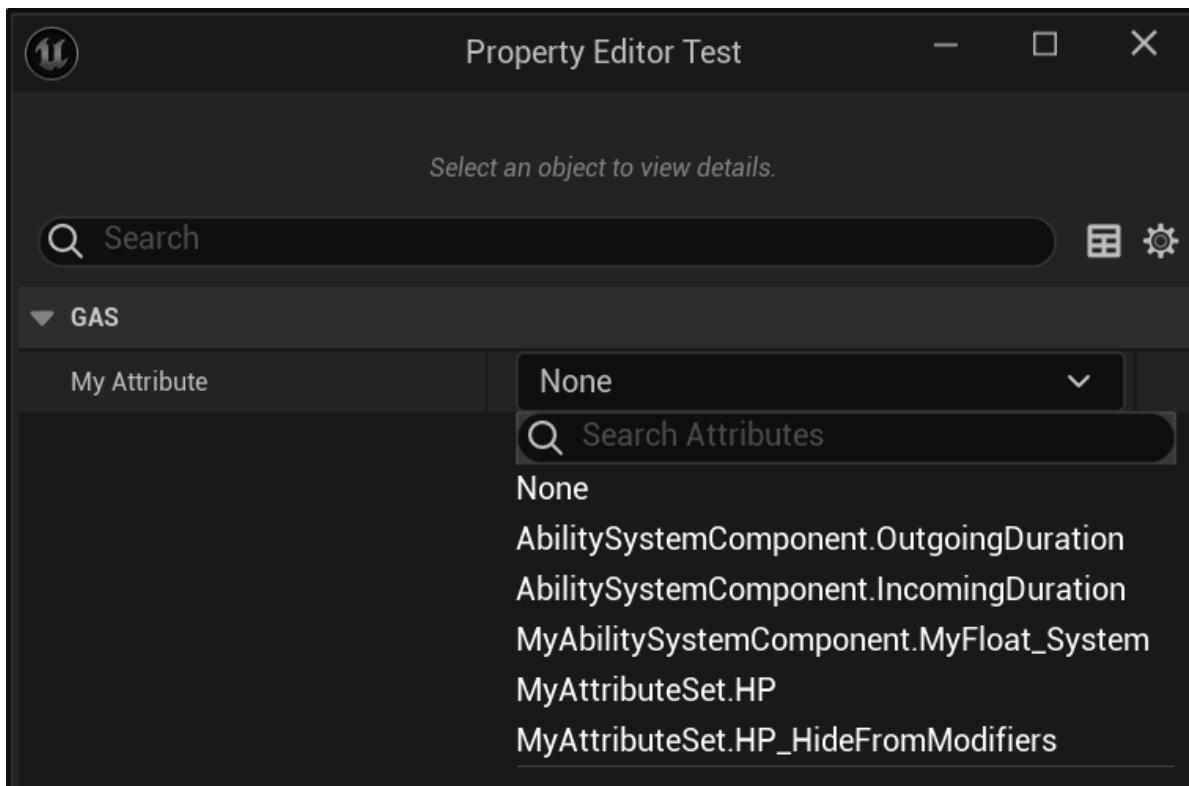
UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS", meta =
(SystemGameplayAttribute))
float MyFloat_System;
};

UCLASS()
class UMyAttributeSetTest : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
    FGameplayAttribute MyAttribute;
};

```

## Test Results:

"MyFloat\_System" is visible and can be exposed in the options list.



## Principle:

The principle is based on checking the SystemGameplayAttribute tag on attributes within the GetAllAttributeProperties function.

```

void FGameplayAttribute::GetAllAttributeProperties(TArray<FProperty*>&
OutProperties, FString FilterMetaStr, bool UseEditorOnlyData)
{
    // UAbilitySystemComponent can add 'system' attributes
    if (Class->IsChildOf(UAbilitySystemComponent::StaticClass()) && !Class-
>ClassGeneratedBy)
    {
        for (TFieldIterator<FProperty> PropertyIt(Class,
EFieldIteratorFlags::ExcludesSuper); PropertyIt; ++PropertyIt)

```

```

    {
        FProperty* Property = *PropertyIt;

        // SystemAttributes have to be explicitly tagged
        if (Property->HasMetaData(TEXT("SystemGameplayAttribute")) == false)
        {
            continue;
        }
        OutProperties.Add(Property);
    }
}

```

## HideFromModifiers

- **Function Description:** Specifies that a particular attribute within an AttributeSet should not be included in the Attribute selection for Modifiers within a GameplayEffect.
- **Usage Location:** UPROPERTY
- **Engine Module:** GAS
- **Metadata Type:** bool
- **Restriction Type:** Attributes within UAttributeSet
- **Related Items:** HideInDetailsView
- **Commonality:** ★★★

Specifies that a particular attribute within an AttributeSet should not be included in the Attribute selection for Modifiers within a GameplayEffect.

## Test Code:

```

UCLASS()
class UMyAttributeSet : public UAttributeSet
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core")
    float HP = 100.f;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core", meta =
(HideInDetailsview))
    float HP_HideInDetailsview = 100.f;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Core", meta =
(HideFromModifiers))
    float HP_HideFromModifiers = 100.f;
};

UCLASS()
class UMyAttributeSetTest : public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")

```

```

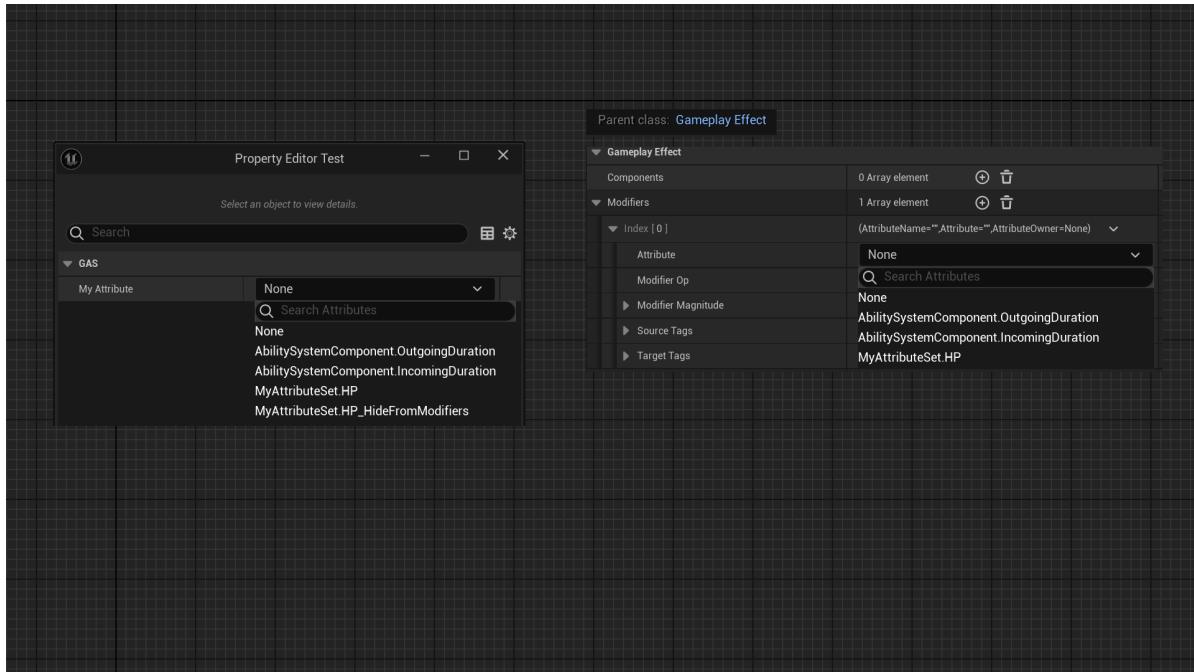
FGameplayAttribute MyAttribute;
};

```

## Test Results:

Create a GameplayEffect in the Blueprint and observe the Attribute selection within its Modifiers.

Notice that HP\_HideFromModifiers can appear in the standard FGameplayAttribute tab, but it will not appear in the Attribute selection tab under Modifiers. This is the intended effect here.



## Principle:

In the FGameplayModifierInfo class, there is a FilterMetaTag metadata on the Attribute property. The value within this metadata is retrieved and ultimately passed to GetAllAttributeProperties as FilterMetaStr for filtering purposes. Hence, an attribute marked with HideFromModifiers will not be displayed.

```

USTRUCT(BlueprintType)
struct GAMEPLAYABILITIES_API FGameplayModifierInfo
{
    GENERATED_USTRUCT_BODY()

    /** The Attribute we modify or the GE we modify modifies. */
    UPROPERTY(EditDefaultsOnly, Category=GameplayModifier, meta=
    (FilterMetaTag="HideFromModifiers"))
        FGameplayAttribute Attribute;
};

void FAttributePropertyDetails::CustomizeHeader( TSharedRef<IPROPERTYHandle>
StructPropertyHandle, class FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils )
{
    const FString& FilterMetaStr = StructPropertyHandle->GetProperty()-
>GetMetaData(TEXT("FilterMetaTag"));
    SNew(SGameplayAttributeWidget)
        .OnAttributeChanged(this, &FAttributePropertyDetails::OnAttributeChanged)
}

```

```

        .DefaultProperty(PropertyName)
        .FilterMetaData(FilterMetaStr)
    }
void FGameplayAttribute::GetAllAttributeProperties(TArray<FProperty*>&
OutProperties, FString FilterMetaStr, bool UseEditorOnlyData)
{}
```

# MaterialParameterCollectionFunction

- **Function Description:** Specifies that this function is designed to operate on UMaterialParameterCollection, thereby enabling the extraction and validation of ParameterName
- **Usage Location:** UFUNCTION
- **Engine Module:** Material
- **Metadata Type:** bool
- **Restriction Type:** Functions with a UMaterialParameterCollection parameter
- **Commonality:** ★★★

Indicates that this function is intended for operating on UMaterialParameterCollection, facilitating the extraction and validation of ParameterName.

## Test Code:

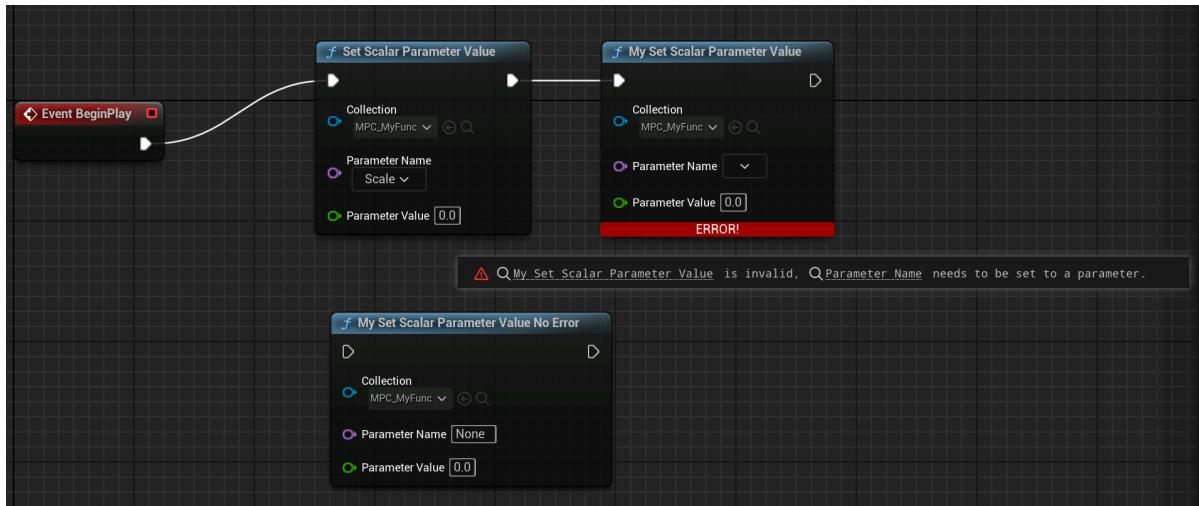
```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyFunction_Material :public UBlueprintFunctionLibrary
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject",
MaterialParameterCollectionFunction))
        static void MySetScalarParameterValue(UObject* WorldContextObject,
UMaterialParameterCollection* Collection, FName ParameterName, float
ParameterValue);

    UFUNCTION(BlueprintCallable, meta=(WorldContext="WorldContextObject"))
        static void MySetScalarParameterValue_NoError(UObject* WorldContextObject,
UMaterialParameterCollection* Collection, FName ParameterName, float
ParameterValue);
};
```

## Blueprint Effect:

The engine's built-in UKismetMaterialLibrary::SetScalarParameterValue and our custom MySetScalarParameterValue functions trigger the verification check for the material parameter collection in blueprints. If ParameterName is not specified, a compilation error will be generated. The MySetScalarParameterValue\_NoError function, which lacks the MaterialParameterCollectionFunction tag, is treated as a regular function; it does not automatically select from the MPC's Parameters collection, nor does it perform error checking for an empty ParameterName.



## Principle:

```

UBlueprintFunctionNodeSpawner* UBlueprintFunctionNodeSpawner::Create(UFunction
const* const Function, UObject* Outer/* = nullptr*/)
{
    bool const bIsMaterialParamCollectionFunc = Function-
>HasMetaData(FBlueprintMetadata::MD_MaterialParameterCollectionFunction);
    if (bIsMaterialParamCollectionFunc)
    {
        NodeClass =
UK2Node_CallMaterialParameterCollectionFunction::StaticClass();
    }
    else
    {
        NodeClass = UK2Node_CallFunction::StaticClass();
    }
}

TSharedPtr<SGraphNode> FNodeFactory::CreateNodeWidget(UEdGraphNode* InNode)
{
    if (UK2Node_CallMaterialParameterCollectionFunction* CallFunctionNode =
Cast<UK2Node_CallMaterialParameterCollectionFunction>(InNode))
    {
        return SNew(SGraphNodeCallParameterCollectionFunction,
CallFunctionNode);
    }
}
TSharedPtr<SGraphPin>
SGraphNodeCallParameterCollectionFunction::CreatePinWidget(UEdGraphPin* Pin)
const
{
    //Operations such as extracting the parameter list from MPC
    if (Collection)
    {
        // Populate the ParameterName pin combobox with valid options from
        // the Collection
        const bool bVectorParameters = CallFunctionNode-
>FunctionReference.GetMemberName().ToString().Contains(TEXT("Vector"));
        Collection->GetParameterNames(NameList, bVectorParameters);
    }
}

```

```
}
```

The MaterialParameterCollectionFunction tag utilizes UK2Node\_CallMaterialParameterCollectionFunction to validate the correct writing of material functions. Additionally, the UK2Node\_CallMaterialParameterCollectionFunction blueprint node is recognized within the engine to further customize the ParameterName pin nodes.

Within the engine source code, the only functions marked with MaterialParameterCollectionFunction are those found in UKismetMaterialLibrary.

## MaterialNewHLSLGenerator

- **Function Description:** Indicates that this UMaterialExpression is a node utilizing the new HLSL generator, which is currently concealed in the right-click menu of the material blueprint.
- **Usage Location:** UCLASS
- **Engine Module:** Material
- **Metadata Type:** boolean
- **Restriction Type:** Subclass of UMaterialExpression
- **Commonality:** ★

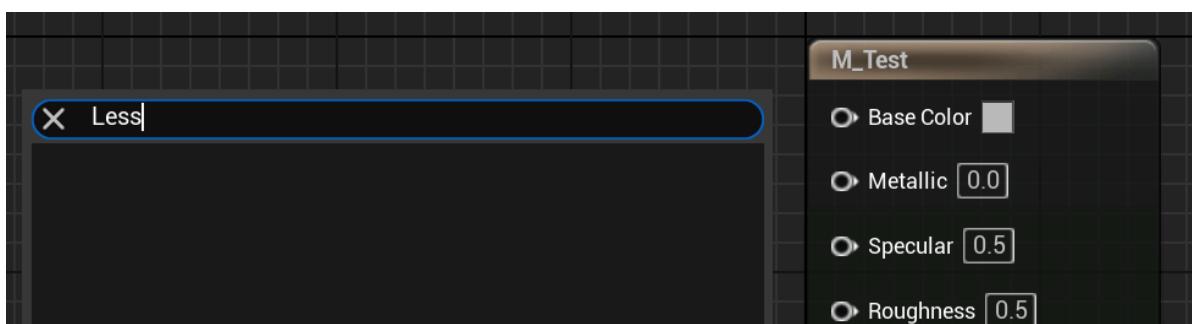
Indicates that this UMaterialExpression is a node utilizing the new HLSL generator, currently hidden in the material blueprint's context menu.

### Source Code Example:

```
UCLASS(MinimalAPI, meta = (MaterialNewHLSLGenerator))
class UMATERIALEXPRESSIONLESS : public UMaterialExpressionBinaryOp
{
    GENERATED_UCLASS_BODY()
#if WITH_EDITOR
    virtual UE::HLSLTree::EOperation GetBinaryOp() const override { return
UE::HLSLTree::EOperation::Less; }
#endif // WITH_EDITOR
};
```

### Test Results:

Less cannot be invoked within the material blueprint.



## Principle:

When iterating through all available FMaterialExpressions, any with MaterialNewHSLGenerator are skipped. Currently, in the engine, r.MaterialEnableNewHSLGenerator is read-only and its implementation is not yet complete.

```
static TAutoConsoleVariable<int32> CVarMaterialEnableNewHSLGenerator(
    TEXT("r.MaterialEnableNewHSLGenerator"),
    0,
    TEXT("Enables the new (WIP) material HLSL generator.\n")
    TEXT("0 - Don't allow\n")
    TEXT("1 - Allow if enabled by material\n")
    TEXT("2 - Force all materials to use new generator\n"),
    ECVF_RenderThreadSafe | ECVF_ReadOnly);
```

```
void MaterialExpressionClasses::InitMaterialExpressionClasses()
{
    static const auto CVarMaterialEnableNewHSLGenerator =
    IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.MaterialEnableNewHSLGenerator"));
    const bool bEnableControlFlow = AllowMaterialControlFlow();
    const bool bEnableNewHSLGenerator = CVarMaterialEnableNewHSLGenerator->GetValueOnAnyThread() != 0;

    for( TObjectIterator<UClass> It ; It ; ++It )
    {
        if( class->IsChildOf(UMaterialExpression::StaticClass()) )
        {
            // Hide node types related to control flow, unless it's
enabled
            if (!bEnableControlFlow && class->HasMetaData("MaterialControlFlow"))
            {
                continue;
            }

            if (!bEnableNewHSLGenerator && class->HasMetaData("MaterialNewHSLGenerator"))
            {
                continue;
            }

            // Hide node types that are tagged private
            if(class->HasMetaData(TEXT("Private")))
            {
                continue;
            }
            AllExpressionClasses.Add(MaterialExpression);
        }
    }
}
```

# ShowAsInputPin

---

- **Function Description:** Enables basic type attributes within UMaterialExpression to be represented as pins on the material node.
- **Usage Location:** UPROPERTY
- **Engine Module:** Material
- **Metadata Type:** bool
- **Restricted Type:** Attributes within UMaterialExpression
- **Commonality:** ★★★

Enables some basic type attributes within UMaterialExpression to be represented as pins on the material node.

- These basic types refer to commonly used types such as float and FVector.
- By default, these basic type properties are not displayed as pins.
- The ShowAsInputPin value has two options: 'Primary' can be displayed directly, while 'Advanced' requires expanding the arrow to be shown.

## Test Code:

---

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinTest)  
    float MyFloat;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinTest, meta =  
(ShowAsInputPin = "Primary"))  
    float MyFloat_Primary;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinTest, meta =  
(ShowAsInputPin = "Advanced"))  
    float MyFloat_Advanced;
```

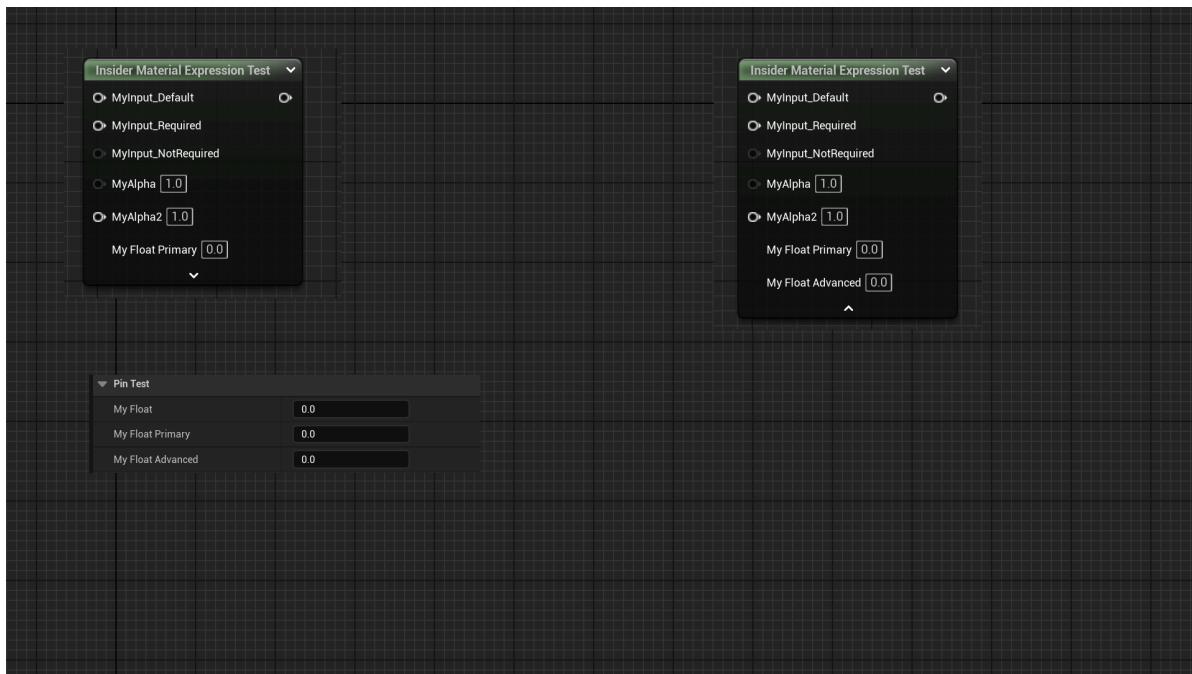
## Test Results:

---

It is evident that MyFloat is not displayed on the node.

MyFloat\_Primary is displayed on the node.

MyFloat\_Advanced needs to have the arrow expanded before it is displayed.



## Principle:

Iterate through the properties within UMaterialExpression and generate pins based on the types that contain ShowAsInputPin.

```

TArray<FProperty*> UMaterialExpression::GetPropertyInputs() const
{
    TArray<FProperty*> PropertyInputs;

    static FName OverridingInputPropertyMetaData(TEXT("OverridingInputProperty"));
    static FName ShowAsInputPinMetaData(TEXT("ShowAsInputPin"));
    for (TFieldIterator<FProperty> InputIt(GetClass(),
EFieldIteratorFlags::IncludeSuper, EFieldIteratorFlags::ExcludeDeprecated);
InputIt; ++InputIt)
    {
        bool bCreateInput = false;
        FProperty* Property = *InputIt;
        // Don't create an expression input if the property is already associated
        // with one explicitly declared
        bool bOverridingInputProperty = Property-
>HasMetaData(OverridingInputPropertyMetaData);
        // It needs to have the 'EditAnywhere' specifier
        const bool bEditAnywhere = Property->HasAnyPropertyParams(CPF_Edit);
        // It needs to be marked with a valid pin category meta data
        const FString ShowAsInputPin = Property-
>GetMetaData(ShowAsInputPinMetaData);
        const bool bShowAsInputPin = ShowAsInputPin == TEXT("Primary") ||
ShowAsInputPin == TEXT("Advanced");

        if (!bOverridingInputProperty && bEditAnywhere && bShowAsInputPin)
        {
            // Check if the property type fits within the allowed widget types
            FFieldClass* PropertyClass = Property->GetClass();
            if (PropertyClass == FFloatPropertyParams::StaticClass()
                || PropertyClass == FDoublePropertyParams::StaticClass())

```

```

        || PropertyClass == FIntProperty::StaticClass()
        || PropertyClass == FUInt32Property::StaticClass()
        || PropertyClass == FByteProperty::StaticClass()
        || PropertyClass == FBoolProperty::StaticClass()
    )
}

{
    bCreateInput = true;
}
else if (PropertyClass == FStructProperty::StaticClass())
{
    FStructProperty* StructProperty = CastField<FStructProperty>
(Property);

    UScriptStruct* Struct = StructProperty->Struct;
    if (Struct == TBaseStructure<FLinearColor>::Get()
        || Struct == TBaseStructure< FVector4>::Get()
        || Struct == TVariantStructure< FVector4d>::Get()
        || Struct == TBaseStructure< FVector>::Get()
        || Struct == TVariantStructure< FVector3f>::Get()
        || Struct == TBaseStructure< FVector2D>::Get()
    )
    {
        bCreateInput = true;
    }
}
if (bCreateInput)
{
    PropertyInputs.Add(Property);
}
}

return PropertyInputs;
}

```

## MaterialControlFlow

- **Function Description:** Identifies this UMaterialExpression as a control flow node, currently concealed in the material blueprint's right-click menu.
- **Usage Location:** UClass
- **Engine Module:** Material
- **Metadata Type:** bool
- **Restriction Type:** Subclass of UMaterialExpression
- **Commonality:** ★

Identifies this UMaterialExpression as a control flow node, which is currently hidden in the material blueprint's right-click menu.

Usually found in nodes like IfThenElse and ForLoop; however, the current engine implementation is incomplete, rendering this feature disabled.

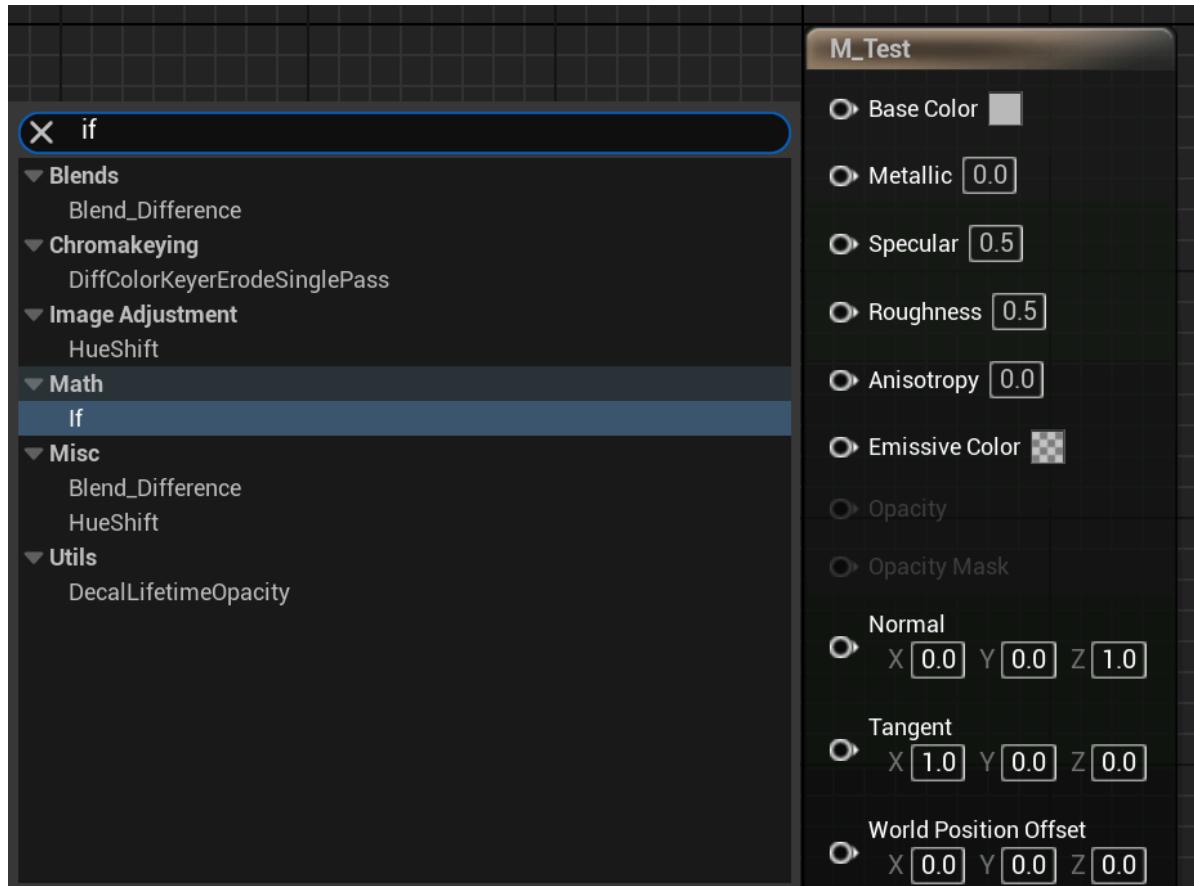
# Source Code Example:

```
UCLASS(collapsecategories, hidecategories=Object, MinimalAPI)
class UMaterialExpressionIf : public UMaterialExpression
{};

UCLASS(collapsecategories, hidecategories = Object, MinimalAPI, meta=
(MaterialControlFlow))
class UMaterialExpressionIfThenElse : public UMaterialExpression
{}
```

## Test Results:

The If node can be located, but calling the IfThenElse node is not possible.



## Principle:

During the traversal of all available FMaterialExpression objects, any with MaterialControlFlow are skipped. Currently, under the engine, AllowMaterialControlFlow is always false and remains unimplemented.

```
// r.MaterialEnableControlFlow is removed and the feature is forced disabled as
// how control flow should be
// implemented in the material editor is still under discussion.
inline bool AllowMaterialControlFlow()
{
    return false;
}
```

```

void MaterialExpressionClasses::InitMaterialExpressionClasses()
{
    static const auto CVarMaterialEnableNewHLSLGenerator =
IConsoleManager::Get().FindConsoleVariableDataInt(TEXT("r.MaterialEnableNewHLSLGenerator"));
    const bool bEnableControlFlow = AllowMaterialControlFlow();
    const bool bEnableNewHLSLGenerator = CVarMaterialEnableNewHLSLGenerator->GetValueOnAnyThread() != 0;

    for( TObjectIterator<UClass> It ; It ; ++It )
    {
        if( Class->IsChildOf(UMaterialExpression::StaticClass()) )
        {
            // Hide node types related to control flow, unless it's
            enabled
            if (!bEnableControlFlow && Class->HasMetaData("MaterialControlFlow"))
            {
                continue;
            }

            if (!bEnableNewHLSLGenerator && Class->HasMetaData("MaterialNewHLSLGenerator"))
            {
                continue;
            }

            // Hide node types that are tagged private
            if (Class->HasMetaData(TEXT("Private")))
            {
                continue;
            }
            AllExpressionClasses.Add(MaterialExpression);
        }
    }
}

```

## OverridingInputProperty

- **Function description:** Specifies the other FExpressionInput properties that this float in UMaterialExpression is intended to override.
- **Use location:** UPROPERTY
- **Engine module:** Material
- **Metadata type:** bool
- **Restriction type:** UMaterialExpression::float
- **Related items:** RequiredInput
- **Frequency:** ★★☆

Within UMaterialExpression, this float specifies the other FExpressionInput properties it is to override.

- In material expressions, there are properties for which we want to provide a default value when the user has not made a connection. In such cases, this float attribute's value can serve as the default.
- However, when the user does connect, this pin should function as a standard input, which necessitates another FExpressionInput property. Hence, the OverridingInputProperty is used to specify this alternate property.
- The FExpressionInput property designated by OverridingInputProperty typically has RequiredInput set to "false", as the usual logic dictates that if a default value is provided, the user is not required to input a value. Nonetheless, setting RequiredInput to "true" is also possible, reminding the user that an input is preferred, but a default value is available if none is provided.
- Many pins like BaseColor on the output node both require an input and offer a default value.

## Test Code:

---

In the Compile function, we mimic the source code to demonstrate how to properly handle this attribute to obtain a value. You may also refer to other examples in the source code for guidance.

```

UCLASS()
class UMyProperty_MyMaterialExpression : public UMaterialExpression
{
    GENERATED_UCLASS_BODY()
public:
    UPROPERTY()
    FExpressionInput MyInput_Default;

    UPROPERTY(meta = (RequiredInput = "true"))
    FExpressionInput MyInput_Required;

    UPROPERTY(meta = (RequiredInput = "false"))
    FExpressionInput MyInput_NotRequired;
public:
    UPROPERTY(EditAnywhere, Category = OverridingInputProperty, meta =
(RequiredInput = "false"))
    FExpressionInput MyAlpha;

    /** only used if MyAlpha is not hooked up */
    UPROPERTY(EditAnywhere, Category = OverridingInputProperty, meta =
(OverridingInputProperty = "MyAlpha"))
    float ConstAlpha = 1.f;

    UPROPERTY(EditAnywhere, Category = OverridingInputProperty, meta =
(RequiredInput = "true"))
    FExpressionInput MyAlpha2;

    /** only used if MyAlpha is not hooked up */
    UPROPERTY(EditAnywhere, Category = OverridingInputProperty, meta =
(OverridingInputProperty = "MyAlpha2"))
    float ConstAlpha2 = 1.f;
public:

    //~ Begin UMaterialExpression Interface
#if WITH_EDITOR

```

```

    virtual int32 Compile(class FMaterialCompiler* Compiler, int32 OutputIndex)
override
{
    int32 IndexAlpha = MyAlpha.GetTracedInput().Expression ?
    MyAlpha.Compile(Compiler) : Compiler->Constant(ConstAlpha);
    return 0;
}
virtual void GetCaption(TArray<FString>& OutCaptions) const override;

virtual bool GenerateHLSLExpression(FMaterialHSLGenerator& Generator,
UE::HLSLTree::FScope& Scope, int32 OutputIndex, UE::HLSLTree::FExpression const*&
OutExpression) const override;
#endif
//~ End UMATERIALExpression Interface
};

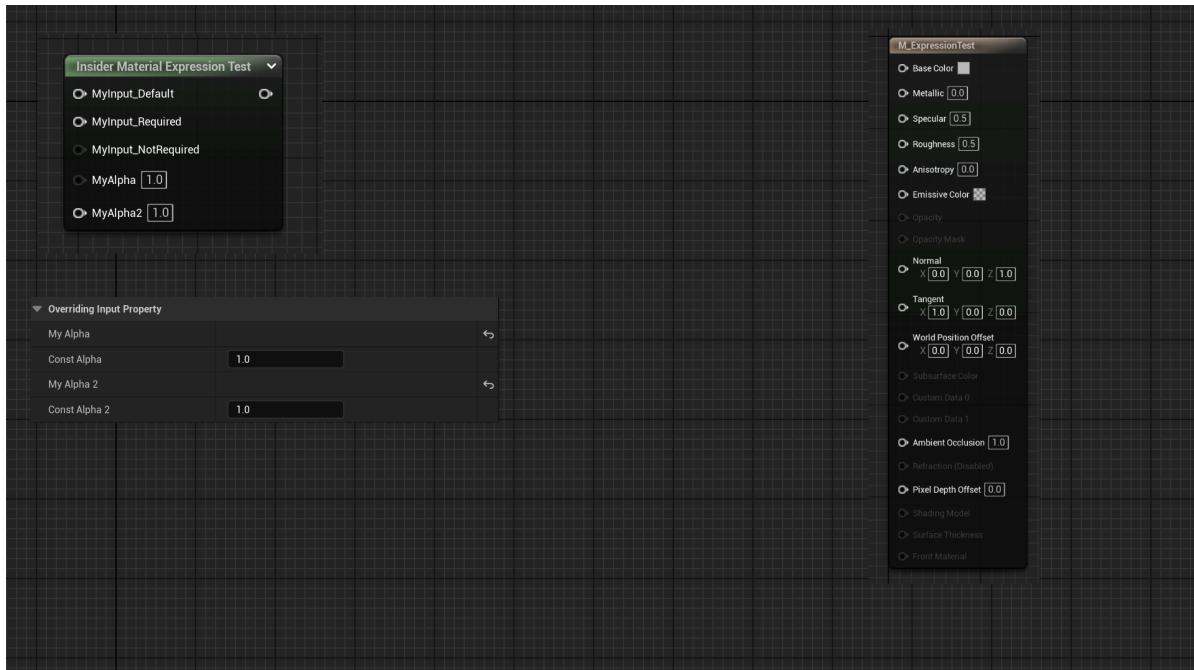
```

## Test Results:

It is evident that both MyAlpha and MyAlpha2 have default values on their right sides, with one appearing gray and the other white due to their differing RequiredInput settings.

The other three MyInput pins are provided for comparison.

The various pins on the final output expression of the material on the right present different scenarios for your reference.



## Principle:

Actually, the principle involves altering the editability of the pin and the input field based on the Meta tag.

```

bool UMATERIALExpression::CanEditChange(const FProperty* InProperty) const
{
    bool bIsEditable = Super::CanEditChange(InProperty);
    if (bIsEditable && InProperty != nullptr)

```

```

{
    // Automatically set property as non-editable if it has
    OverridingInputProperty metadata
        // pointing to an FExpressionInput property which is hooked up as an
        input.
        //
        // e.g. in the below snippet, meta=(OverridingInputProperty = "A")
    indicates that ConstA will
        // be overridden by an FExpressionInput property named 'A' if one is
        connected, and will thereby
        // be set as non-editable.
        //
        // UPROPERTY(meta = (RequiredInput = "false", ToolTip = "Defaults to
    'ConstA' if not specified"))
        // FExpressionInput A;
        //
        // UPROPERTY(EditAnywhere, Category = MaterialExpressionAdd, meta =
    (OverridingInputProperty = "A"))
        // float ConstA;
        //
}

static FName
OverridingInputPropertyMetaData(TEXT("OverridingInputProperty"));

if (InPropertyParams->HasMetaData(OverridingInputPropertyMetaData))
{
    const FString& OverridingPropertyName = InPropertyParams-
>GetMetaData(OverridingInputPropertyMetaData);

    FStructPropertyParams* StructProp = FindPropertyParams<FStructPropertyParams>
(GetClass(), *OverridingPropertyName);
    if (ensure(StructProp != nullptr))
    {
        static FName RequiredInputMetaData(TEXT("RequiredInput"));

        // Must be a single FExpressionInput member, not an array, and
        must be tagged with metadata RequiredInput="false"
        if (ensure( StructProp->Struct->GetFName() ==
NAME_ExpressionInput &&
                StructProp->ArrayDim == 1 &&
                StructProp->HasMetaData(RequiredInputMetaData) &&
                !StructProp->GetBoolMetaData(RequiredInputMetaData)))
        {
            const FExpressionPropertyParams* Input = StructProp-
>ContainerPtrToValuePtr<FExpressionPropertyParams>(this);

            if (Input->Expression != nullptr && Input-
>GetTracedInput().Expression != nullptr)
            {
                bIsEditable = false;
            }
        }
    }
}

if (bIsEditable)

```

```

    {
        // If the property has EditCondition metadata, then whether it's
        // editable depends on the other EditCondition property
        const FString EditConditionPropertyName = InProperty-
>GetMetaData(TEXT("EditCondition"));
        if (!EditConditionPropertyName.IsEmpty())
        {
            FBoolProperty* EditConditionProperty =
FindFProperty<FBoolProperty>(GetClass(), *EditConditionPropertyName);
            {
                bIsEditable = *EditConditionProperty-
>ContainerPtrToValuePtr<bool>(this);
            }
        }
    }

    return bIsEditable;
}

```

#RequiredInput

- **Function Description:** Determines whether the FExpressionInput attribute in UMaterialExpression necessitates an input, with the corresponding pin displayed in white or gray.
- **Usage Location:** UPROPERTY
- **Engine Module:** Material
- **Metadata Type:** Boolean
- **Restriction Type:** UMaterialExpression::FExpressionInput
- **Related Items:** OverridingInputProperty

In UMaterialExpression, it specifies whether the FExpressionInput property requires an input, and the pin will be shown in white or gray.

It is typically used in tandem with the OverridingInputProperty.

For code examples and effects, refer to OverridingInputProperty

## Principle:

---

```

bool UMaterialExpression::CanEditChange(const FProperty* InProperty) const
{
    bool bIsEditable = Super::CanEditChange(InProperty);
    if (bIsEditable && InProperty != nullptr)
    {
        // Automatically set property as non-editable if it has
        OverridingInputProperty metadata
        // pointing to an FExpressionInput property which is hooked up as an
        input.
        //
        // e.g. in the below snippet, meta=(OverridingInputProperty = "A")
        indicates that ConstA will

```

```

        // be overridden by an FExpressionInput property named 'A' if one is
        connected, and will thereby
        // be set as non-editable.
        //
        // UPROPERTY(meta = (RequiredInput = "false", ToolTip = "Defaults to
        'ConstA' if not specified"))
        // FExpressionInput A;
        //
        // UPROPERTY(EditAnywhere, Category = MaterialExpressionAdd, meta =
        (OverridingInputProperty = "A"))
        // float ConstA;
        //

        static FName
OverridingInputPropertyMetaData(TEXT("OverridingInputProperty"));

    if (InPropertyParams->HasMetaData(OverridingInputPropertyMetaData))
    {
        const FString& OverridingPropertyName = InPropertyParams-
>GetMetaData(OverridingInputPropertyMetaData);

        FStructProperty* StructProp = FindFProperty<FStructProperty>
(GetClass(), *OverridingPropertyName);
        if (ensure(StructProp != nullptr))
        {
            static FName RequiredInputMetaData(TEXT("RequiredInput"));

            // Must be a single FExpressionInput member, not an array, and
            must be tagged with metadata RequiredInput="false"
            if (ensure(StructProp->Struct->GetFName() ==
NAME_ExpressionInput &&
                StructProp->ArrayDim == 1 &&
                StructProp->HasMetaData(RequiredInputMetaData) &&
                !StructProp->GetBoolMetaData(RequiredInputMetaData)))
            {
                const FExpressionInput* Input = StructProp-
>ContainerPtrToValuePtr<FExpressionInput>(this);

                if (Input->Expression != nullptr && Input-
>GetTracedInput().Expression != nullptr)
                {
                    bIsEditable = false;
                }
            }
        }
    }

    if (bIsEditable)
    {
        // If the property has EditCondition metadata, then whether it's
        editable depends on the other EditCondition property
        const FString EditConditionPropertyName = InPropertyParams-
>GetMetaData(TEXT("EditCondition"));
        if (!EditConditionPropertyName.IsEmpty())
        {

```

```

        FBoolProperty* EditConditionProperty =
FindFProperty<FBoolProperty>(GetClass(), *EditConditionPropertyName);
{
    bIsEditable = *EditConditionProperty-
>ContainerPtrToValuePtr<bool>(this);
}
}

return bIsEditable;
}

```

# Private

- **Function Description:** Indicates that this UMaterialExpression is a private node, currently hidden in the right-click menu of the material blueprint.
- **Usage Location:** UClass
- **Engine Module:** Material
- **Metadata Type:** bool
- **Restriction Type:** Subclass of UMaterialExpression
- **Commonly Used:** ★

Identifies this UMaterialExpression as a private node, currently concealed in the right-click menu of the material blueprint.

Used within the MaterialX module, currently temporarily hiding all the Expressions inside.

## Source Code Example:

```

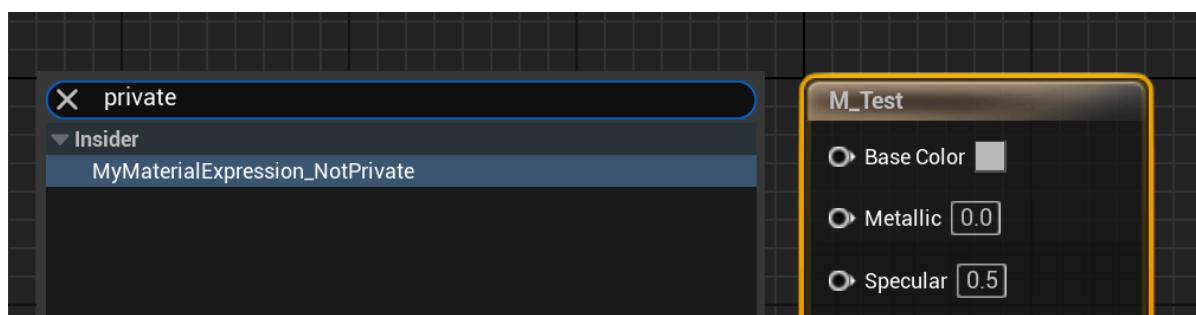
UCLASS()
class UMyMaterialExpression_NotPrivate : public UMaterialExpression
{};

UCLASS(meta=(Private))
class UMyMaterialExpression_Private : public UMaterialExpression
{};

```

## Test Results:

Only UMyMaterialExpression\_NotPrivate can be invoked within the material blueprint.



## Principle:

When iterating through all available FMaterialExpressions, any that are marked as Private will be omitted.

```
static TAutoConsoleVariable<int32> CVarMaterialEnableNewHLSLGenerator(
    TEXT("r.MaterialEnableNewHLSLGenerator"),
    0,
    TEXT("Enables the new (WIP) material HLSL generator.\n")
    TEXT("0 - Don't allow\n")
    TEXT("1 - Allow if enabled by material\n")
    TEXT("2 - Force all materials to use new generator\n"),
    ECVF_RenderThreadSafe | ECVF_ReadOnly);
```

```
void MaterialExpressionClasses::InitMaterialExpressionClasses()
{
    static const auto CVarMaterialEnableNewHLSLGenerator =
IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.MaterialEnableNewHLSLGenerator"));
    const bool bEnableControlFlow = AllowMaterialControlFlow();
    const bool bEnableNewHLSLGenerator = CVarMaterialEnableNewHLSLGenerator->GetValueOnAnyThread() != 0;

    for( TObjectIterator<UClass> It ; It ; ++It )
    {
        if( class->IsChildOf(UMaterialExpression::StaticClass()) )
        {
            // Hide node types related to control flow, unless it's
enabled
            if (!bEnableControlFlow && class->HasMetaData("MaterialControlFlow"))
            {
                continue;
            }

            if (!bEnableNewHLSLGenerator && class->HasMetaData("MaterialNewHLSLGenerator"))
            {
                continue;
            }

            // Hide node types that are tagged private
            if(class->HasMetaData(TEXT("Private")))
            {
                continue;
            }
            AllExpressionClasses.Add(MaterialExpression);
        }
    }
}
```

# NiagaraClearEachFrame

- **Function Description:** ScriptStruct /Script/Niagara.NiagaraSpawnInfo
- **Usage Location:** USTRUCT
- **Engine Module:** Niagara
- **Metadata Type:** bool
- **Commonality:** Rarely Used

Specifies that the data of a certain structure should not be read in each subsequent frame of Niagara, serving only as initial data.

Currently applied exclusively to FNiagaraSpawnInfo, for internal use only.

## Source Code Example:

```
/** Data controlling the spawning of particles */
USTRUCT(BlueprintType, meta = (DisplayName = "Spawn Info", NiagaraClearEachFrame
= "true"))
struct FNiagaraSpawnInfo
{
```

## Principle: The Fundamental Concept

```
// If the NiagaraClearEachFrame value is set on the data set, we don't bother
// reading it in each frame as we know that it is invalid. However,
// this is only used for the base data set. Other reads are potentially from
// events and are therefore perfectly valid.
if (DataSetIndex == 0 && Var.GetType().GetScriptStruct() != nullptr &&
Var.GetType().GetScriptStruct()->GetMetaData(TEXT("NiagaraClearEachFrame")).Equals(TEXT("true"),
ESearchCase::IgnoreCase))
{
    Fmt = VariableName + TEXT("{0} = {4};\n");
}
```

# NiagaraInternalType

- **Function Description:** Specifies that the structure is of Niagara's internal type.
- **Usage Location:** USTRUCT
- **Engine Module:** Niagara
- **Metadata Type:** bool
- **Commonality:** Rare

Specifies the type of this structure as Niagara's internal type.

Used to differentiate from user-defined types; users should not manually utilize this metadata.

## Source Code Example:

```
USTRUCT(meta = (DisplayName = "Half", NiagaraInternalType = "true"))
struct FNiagaraHalf
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 value = 0;
};

USTRUCT(meta = (DisplayName = "Half Vector2", NiagaraInternalType = "true"))
struct FNiagaraHalfVector2
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 x = 0;

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 y = 0;
};

USTRUCT(meta = (DisplayName = "Half Vector3", NiagaraInternalType = "true"))
struct FNiagaraHalfVector3
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 x = 0;

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 y = 0;

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 z = 0;
};

USTRUCT(meta = (DisplayName = "Half Vector4", NiagaraInternalType = "true"))
struct FNiagaraHalfVector4
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 x = 0;

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 y = 0;

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 z = 0;

    UPROPERTY(EditAnywhere, Category = Parameters)
    uint16 w = 0;
};
```

# Principle: The Fundamental Concept

```
#if WITH_EDITORONLY_DATA
bool FNiagaraTypeDefinition::IsInternalType() const
{
    if (const UScriptStruct* ScriptStruct = GetScriptStruct())
    {
        return ScriptStruct->GetBoolMetaData(TEXT("NiagaraInternalType"));
    }

    return false;
}
#endif
```

## CtrlMultiplier

- **Function description:** Specifies the multiplier for value changes in a numeric input box when scrolling with the mouse wheel or dragging with the mouse while the Ctrl key is pressed.
- **Usage location:** UPROPERTY
- **Engine module:** Numeric Property
- **Metadata type:** float/int
- **Restricted types:** Data structures: FVector, FRotator, FColor
- **Associated items:** ShiftMultiplier
- **Commonly used:** ★★

Specifies the multiplier for value changes in a numeric input box when scrolling with the mouse wheel or dragging with the mouse while the Ctrl key is pressed.

- The default value for CtrlMultiplier is 0.1f, typically used as a fine-tuning mode.
- Directly setting it to a float attribute has no effect. By default, the CtrlMultiplier on a property is not applied to SNumericEntryBox and SSpinBox, as these do not directly derive meta from the property to set their own Multiplier values.
- In the source code, FMathStructCustomization is found to extract the values of CtrlMultiplier and ShiftMultiplier, allowing settings on mathematical structures such as FVector, FRotator, FColor
- If you define your own Customization and create an SSpinBox, you can manually extract the Multiplier value and apply it to the control.

## Test Code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultiplierTest, meta = (CtrlMultiplier = "5"))
float MyFloat_HasCtrlMultiplier = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultiplierTest, meta = (ShiftMultiplier = "100"))
float MyFloat_HasShiftMultiplier = 100;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultiplierTest)
FVector MyVector_NoMultiplier;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultiplierTest, meta = (CtrlMultiplier = "5"))
FVector MyVector_HasCtrlMultiplier;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultiplierTest, meta = (ShiftMultiplier = "100"))
FVector MyVector_HasShiftMultiplier;
```

## Test Results:

- It is observed that the test on a regular float attribute has no effect; pressing Ctrl and Shift still changes the values to the defaults of 0.1 and 10.f, respectively
- For the standard default FVector, pressing Ctrl and Shift also changes the values to the defaults of 0.1 and 10.f, respectively
- MyVector\_HasCtrlMultiplier: It is noticed that pressing Ctrl results in an immediate change of 5 in the range
- MyVector\_HasShiftMultiplier: It is noticed that pressing Shift results in an immediate change of 100 in the range
- Of course, the same effect occurs when dragging with the mouse, but the change is too abrupt, making the demonstration effect not very clear

Multiplier Test			
My Float Has Ctrl Multiplier	100.0		
My Float Has Shift Multiplier	100.0		
▶ My Vector No Multiplier	0.0	0.0	0.0
▶ My Vector Has Ctrl Multiplier	0.0	0.0	0.0
▶ My Vector Has Shift Multiplier	0.0	0.0	0.0

## Principle:

SNumericEntryBox的构造函数里：

```
, _ShiftMultiplier(10.f)
, _CtrlMultiplier(0.1f)
```

```
void FMathStructCustomization::MakeHeaderRow(TSharedRef<class IPropertyHandle>&
StructPropertyHandle, FDetailWidgetRow& Row)
{
```

```

        for (int32 ChildIndex = 0; ChildIndex < SortedChildHandles.Num();
++ChildIndex)
{
    TSharedRef<IPROPERTYHANDLE> ChildHandle = SortedChildHandles[ChildIndex];

    // Propagate metadata to child properties so that it's reflected in the
    nested, individual spin boxes
    ChildHandle->SetInstanceMetaData(TEXT("UIMin"), StructPropertyHandle-
>GetMetaData(TEXT("UIMin")));
    ChildHandle->SetInstanceMetaData(TEXT("UIMax"), StructPropertyHandle-
>GetMetaData(TEXT("UIMax")));
    ChildHandle->SetInstanceMetaData(TEXT("SliderExponent"),
StructPropertyHandle->GetMetaData(TEXT("SliderExponent")));
    ChildHandle->SetInstanceMetaData(TEXT("Delta"), StructPropertyHandle-
>GetMetaData(TEXT("Delta")));
    ChildHandle->SetInstanceMetaData(TEXT("LinearDeltaSensitivity"),
StructPropertyHandle->GetMetaData(TEXT("LinearDeltaSensitivity")));
    ChildHandle->SetInstanceMetaData(TEXT("ShiftMultiplier"),
StructPropertyHandle->GetMetaData(TEXT("ShiftMultiplier")));
    ChildHandle->SetInstanceMetaData(TEXT("CtrlMultiplier"),
StructPropertyHandle->GetMetaData(TEXT("CtrlMultiplier")));
    ChildHandle->SetInstanceMetaData(TEXT("SupportDynamicsSlider.MaxValue"),
StructPropertyHandle->GetMetaData(TEXT("SupportDynamicsSlider.MaxValue")));
    ChildHandle->SetInstanceMetaData(TEXT("SupportDynamicsSlider.MinValue"),
StructPropertyHandle->GetMetaData(TEXT("SupportDynamicsSlider.MinValue")));
    ChildHandle->SetInstanceMetaData(TEXT("ClampMin"), StructPropertyHandle-
>GetMetaData(TEXT("ClampMin")));
    ChildHandle->SetInstanceMetaData(TEXT("ClampMax"), StructPropertyHandle-
>GetMetaData(TEXT("ClampMax")));
}
}

```

## ShiftMultiplier

- **Function Description:** Specifies the multiplier for value changes in the numeric input box when the mouse wheel scrolls or the mouse is dragged while the Shift key is pressed.
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** float/int
- **Restricted Types:** Data structures: FVector, FRotator, FColor
- **Associated Items:** CtrlMultiplier
- **Commonality:** ★★

The default value is 10.f

Shift mode can be regarded as a rapid adjustment mode, allowing for quick value changes.

# SliderExponent

- **Function Description:** Specifies the exponential distribution for the scroll bar's drag behavior on numeric input fields
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** float/int
- **Restriction Type:** float, int32
- **Commonliness:** ★★★★☆

Specifies the exponential distribution for the scroll bar's drag behavior on numeric input fields. The default value is 1.

This value must be used in conjunction with Min and Max

The term "exponential distribution" refers to how the percentage of the scroll bar changes as the text value being scrolled changes within the range of UIMin and Max. By default, the midpoint value is at 50%. However, we can also specify an exponent to create an exponential distribution curve. On the left side of the number axis, the curve starts off more gradual, allowing for finer adjustments. On the right side, as the curve steepens, the changes become more dramatic, and precision is lost.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = SliderTest, meta =  
(UIMin = "0", UIMax = "1000"))  
    float MyFloat_DefaultSliderExponent = 100;  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = SliderTest, meta =  
(UIMin = "0", UIMax = "1000", SliderExponent = 5))  
    float MyFloat_HasSliderExponent = 100;
```

## Test Results:

It is evident that with SliderExponent=5, a text value of 100 falls within the UI range of 1000, starting close to the 0.3 position, with a fine range of adjustment before 500 and a rapid change thereafter, contrasting with the former scenario.



## Principle:

The default value is 1. If not, the new percentage is calculated using SpinBoxComputeExponentSliderFraction. Readers can refer to the SpinBoxComputeExponentSliderFraction function to understand the details of the exponential distribution.

```

const float CachedSliderExponent = SliderExponent.Get();
if (!FMath::IsNearlyEqual(CachedSliderExponent, 1.f))
{
    if (sliderExponentNeutralValue.IsSet() && SliderExponentNeutralValue.Get() >
GetMinSliderValue() && SliderExponentNeutralValue.Get() < GetMaxSliderValue())
    {
        //Compute a log curve on both side of the neutral value
        float StartFractionFilled =
Fraction((double)SliderExponentNeutralValue.Get(), (double)GetMinSliderValue(),
(double)GetMaxSliderValue());
        FractionFilled = SpinBoxComputeExponentsSliderFraction(FractionFilled,
StartFractionFilled, CachedSliderExponent);
    }
    else
    {
        FractionFilled = 1.0f - FMath::Pow(1.0f - FractionFilled,
CachedSliderExponent);
    }
}

```

## Multiple

---

- **Function Description:** Specifies that the value of a number must be an integer multiple of the value provided by Multiple.
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** int32
- **Restriction Type:** int32
- **Commonality:** ★★★

The value of the specified number must be an integer multiple of the value provided by Multiple.

## Test Code:

---

```

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultipleTest)
    int32 MyInt_NoMultiple = 100;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MultipleTest, meta =
(Multiple = 5))
    int32 MyInt_HasMultiple = 100;

```

## Blueprint Effect:

---

It can be observed that values with Multiple can only increase in increments that are multiples of 5.

▼ Multiple Test	
My Int No Multiple	100
My Int Has Multiple	100

## Principle:

```

template <typename Type>
static Type ClampIntegerValueFromMetaData(Type InValue, FPropertyHandleBase&
InPropertyParams, FPropertyParams& InPropertyParams)
{
    Type RetVal = ClampValueFromMetaData<Type>(InValue, InPropertyParams);

    //if there is "Multiple" meta data, the selected number is a multiple
    const FString& MultipleString =
    InPropertyParams.GetMetaData(TEXT("Multiple"));
    if (MultipleString.Len())
    {
        check(MultipleString.IsNumeric());
        Type MultipleValue;
        TTypeFromString<Type>::FromString(MultipleValue, *MultipleString);
        if (MultipleValue != 0)
        {
            RetVal -= Type(RetVal) % MultipleValue;
        }
    }

    return RetVal;
}

```

## Units

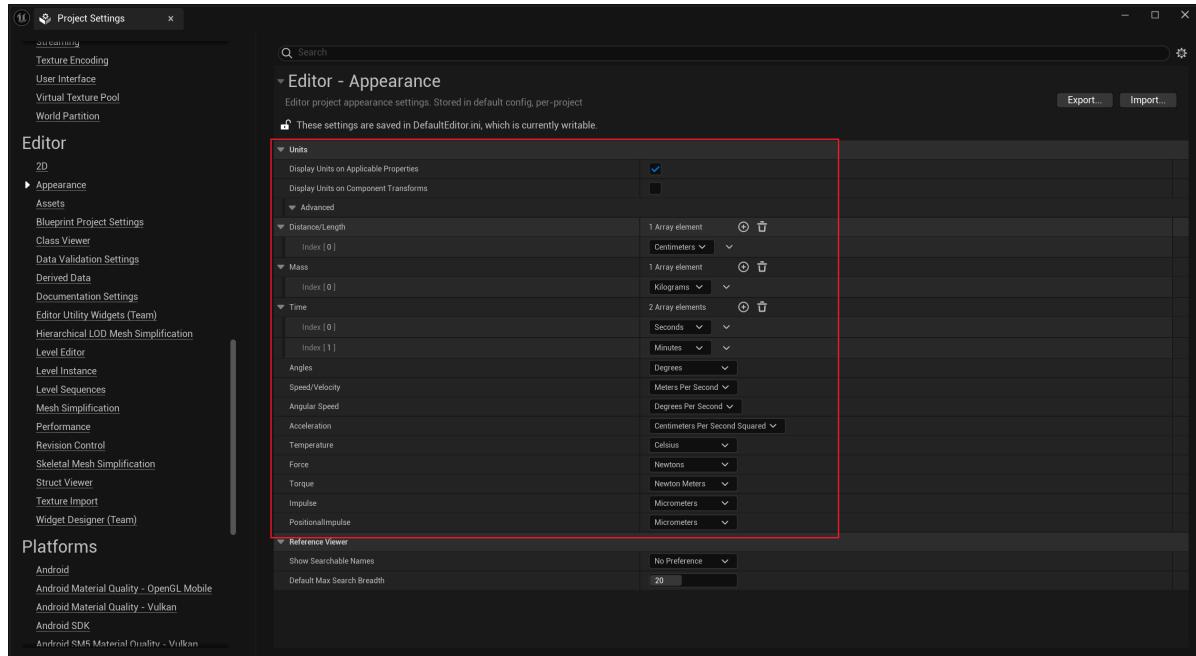
- **Function description:** Defines the unit for attribute values, supporting dynamic adjustment of the display unit based on the numerical input.
- **Use location:** UPROPERTY
- **Engine module:** Numeric Property
- **Metadata type:** string = "abc"
- **Restriction type:** float, int32
- **Related items:** ForceUnits
- **Commonly used:** ★★★

Units set the unit for attribute values. A unit can have multiple aliases, such as Kilograms and kg, Centimeters and cm.

The role of Units is not only to define the unit but also implies that the display unit string can be automatically adjusted to adapt to the user's input value. For instance, 100cm is equivalent to 1m, and 0.5km is 500m.

Moreover, after setting the unit, the system can accept a combination of numbers and units directly entered into the numeric field. For example, typing 1km sets the value to 1 with the unit km. Or 1ft equals 1 foot, which is 30.84cm.

To implement the feature of automatically adjusting the display unit, you first need to set up a series of units in the project settings. For example, the image below sets units for distance in centimeters, meters, kilometers, and millimeters (the order is not important). Subsequently, the numeric field can convert between these four units when displaying the distance unit.



## Test Code:

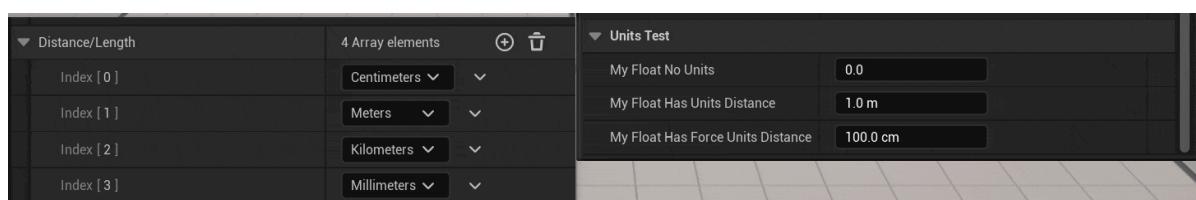
```
UPROPERTY(EditAnywhere, Category = UnitsTest)
float MyFloat_NoUnits = 0.0;

UPROPERTY(EditAnywhere, Category = UnitsTest, Meta = (Units = "cm"))
float MyFloat_HasUnits_Distance = 100.f;

UPROPERTY(EditAnywhere, Category = UnitsTest, Meta = (ForceUnits = "cm"))
float MyFloat_HasForceUnits_Distance = 100.f;
```

## Test Results:

- Enter four units in the project settings: cm, m, km, mm, and begin testing.
- It is observed that properties using Units automatically adjust the unit based on different values and also accept numeric inputs combined with units.
- It is noted that properties using ForceUnits also accept numeric inputs combined with units, but the display always remains in cm, without adjusting to other units.



# Principle:

- If ForceUnits is set, UnderlyingUnits (base units), UserDisplayUnits, and FixedDisplayUnits are also set.
- Otherwise, if Units is set, it is applied only to UnderlyingUnits and FixedDisplayUnits
- Finally, when displaying units, if UserDisplayUnits is present, it is given priority. Then FixedDisplayUnits are considered.
- When using ToString, the value is converted from UnderlyingUnits to UserDisplayUnits or FixedDisplayUnits.
- When the numeric input box is changed, SetupFixedDisplay is triggered, recalculating the appropriate unit value internally and assigning it to FixedDisplayUnits each time. Therefore, as previously stated, if UserDisplayUnits is not set (no ForceUnits), the display unit will be adjusted to the new appropriate unit every time. Otherwise, due to the highest priority of UserDisplayUnits, which always holds a value, the display will remain constant.

```
void SPropertyEditorNumeric<NumericType>::Construct( const FArguments& InArgs,
const TSharedRef<FPropertyEditor>& InPropertyEditor )
{
    // First off, check for ForceUnits= meta data. This meta tag tells us to
    // interpret, and always display the value in these units.
    FUnitConversion::Settings().ShouldDisplayUnits does not apply to such properties
    const FString& ForcedUnits = MetaDataGetter.Execute("ForceUnits");
    TOptional<EUnit> PropertyUnits =
    FUnitConversion::UnitFromString(*ForcedUnits);
    if (PropertyUnits.IsSet())
    {
        // Create the type interface and set up the default input units if
        // they are compatible
        TypeInterface = MakeShareable(new
        TNumericUnitTypeInterface<NumericType>(PropertyUnits.GetValue()));
        TypeInterface->UserDisplayUnits = TypeInterface->FixedDisplayUnits =
        PropertyUnits.GetValue();
    }
    // If that's not set, we fall back to Units=xxx which calculates the most
    // appropriate unit to display in
    else
    {
        if (FUnitConversion::Settings().ShouldDisplayUnits())
        {
            const FString& DynamicUnits = PropertyHandle-
            >GetMetaData(TEXT("Units"));
            if (!DynamicUnits.IsEmpty())
            {
                PropertyUnits =
                FUnitConversion::UnitFromString(*DynamicUnits);
            }
            else
            {
                PropertyUnits =
                FUnitConversion::UnitFromString(*MetaDataGetter.Execute("Units"));
            }
        }
    }
}
```

```

        if (!PropertyUnits.IsSet())
        {
            PropertyUnits = EUnit::Unspecified;
        }
    }

void SPropertyEditorNumeric<NumericType>::OnValueCommitted( NumericType NewValue,
ETextCommit::Type CommitInfo )
{

    if (TypeInterface.IsValid())
    {
        TypeInterface->SetupFixedDisplay(NewValue);
    }
}

template<typename NumericType>
void TNumericUnitTypeInterface<NumericType>::SetupFixedDisplay(const NumericType& InValue)
{
    // We calculate this regardless of whether FixedDisplayUnits is used, so that
    // the moment it is used, it's correct
    EUnit DisplayUnit = FUnitConversion::CalculateDisplayUnit(InValue,
UnderlyingUnits);
    if (DisplayUnit != EUnit::Unspecified)
    {
        FixedDisplayUnits = DisplayUnit;
    }
}

//During conversion,
FString TNumericUnitTypeInterface<NumericType>::ToString(const NumericType& value) const
{
    if (UserDisplayUnits.IsSet())
    {
        auto Converted = FinalValue.ConvertTo(UserDisplayUnits.GetValue());
        if (Converted.IsSet())
        {
            return TUnitString(Converted.GetValue());
        }
    }

    if (FixedDisplayUnits.IsSet())
    {
        auto Converted = FinalValue.ConvertTo(FixedDisplayUnits.GetValue());
        if (Converted.IsSet())
        {
            return TUnitString(Converted.GetValue());
        }
    }
}

```

# ForceUnits

---

- **Function Description:** Ensures that the unit of a fixed attribute value remains constant, without dynamically adjusting the display unit based on the value.
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** string="abc"
- **Restricted Types:** float, int32
- **Associated Items:** Units
- **Commonality:** ★★★

# Delta

---

- **Function description:** Set the change amplitude of the numeric input box to a multiple of Delta
- **Usage location:** UPROPERTY
- **Engine module:** Numeric Property
- **Metadata type:** float/int
- **Restricted types:** float, int32
- **Associated items:** LinearDeltaSensitivity
- **Commonly used:** ★★★

Set the change amplitude of the numeric input box to a multiple of Delta.

## Notes:

---

1. Make the change value a multiple of Delta globally
2. The default value for Delta is 0, indicating no setting is applied; the number will change exponentially.
3. Note the difference from WheelStep: Delta is effective when the mouse is dragged left or right or when the arrow keys on the keyboard are pressed. WheelStep is only effective when the mouse wheel is used. Although both control the change amplitude, their scopes of application differ.

## Test code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest)
float MyFloat_DefaultDelta = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(Delta = 10))
float MyFloat_Delta10 = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(UIMin = "0", UIMax = "1000", Delta = 10))
float MyFloat_Delta10_UIMinMax = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(Delta = 10, LinearDeltaSensitivity = 50))
float MyFloat_Delta10_LinearDeltaSensitivity50 = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(UIMin = "0", UIMax = "1000", Delta = 10, LinearDeltaSensitivity = 50))
float MyFloat_Delta10_LinearDeltaSensitivity50_UIMinMax = 100;
```

## Test effects:

- MyFloat\_DefaultDelta: By default, dragging the mouse to the right causes drastic changes, with values increasing exponentially.
- MyFloat\_Delta10: Dragging the mouse to the right also causes drastic changes (with the final value reaching a very large number), but the changes are always in steps of Delta.
- MyFloat\_Delta10\_UIMinMax: With UIMinMax set, the maximum value is restricted, but the change is actually linear according to the proportional value (SliderExponent defaults to 1 and remains unchanged).
- MyFloat\_Delta10\_LinearDeltaSensitivity50: Without UIMinMax and with LinearDeltaSensitivity set, the change value remains linear throughout the entire process of dragging the mouse to the right. The higher the LinearDeltaSensitivity, the less sensitive it is, resulting in a slow change of 10 each time
- MyFloat\_Delta10\_LinearDeltaSensitivity50\_UIMinMax: If UIMinMax is added to the previous case, the effect of LinearDeltaSensitivity is lost. This is because LinearDeltaSensitivity cannot take effect when a scrollbar is present.

▼ Delta Test	
My Float Default Delta	100.0
My Float Delta 10	100.0
My Float Delta 10 UIMin Max	100.0
My Float Delta 10 Linear Delta Sensitivity 50	100.0
My Float Delta 10 Linear Delta Sensitivity 50 UIMin Max	100.0

## Principle:

- When the up, down, left, or right arrow keys are pressed, the value changes by a positive or negative Delta each time. However, OnKeyDown is not directly bound, so by default, pressing these keys will only shift the focus.
- The base step is 0.1 or 1, unless affected by CtrlMultiplier or ShiftMultiplier.

- By default, the exponential part of the change in value when moving the mouse left or right is calculated as FMath::Pow((double)CurrentValue, (double)SliderExponent.Get()). The default value for SliderExponent is 1, meaning the change in amplitude is greater toward the left and right ends.
- When both LinearDeltaSensitivity and Delta are set, the exponential part of the change in value when moving the mouse left or right is calculated as FMath::Pow((double)Delta.Get(), (double)SliderExponent.Get()), making the change in value linearly consistent across the entire range of the number line from left to right.
- When the final value is submitted, the changed delta value is normalized to a multiple of Delta.

```

, _Delta(0)

virtual FReply SSpinBox<NumericType>::OnMouseMove(const FGeometry& MyGeometry,
const FPointerEvent& MouseEvent) override
{
    if (bUnlimitedSpinRange)
    {
        // If this control has a specified delta and sensitivity then we
        // use that instead of the current value for determining how much to change.
        const double Sign = (MouseEvent.GetCursorDelta().X > 0) ? 1.0 :
        -1.0;

        if (LinearDeltaSensitivity.IsSet() &&
LinearDeltaSensitivity.Get() != 0 && Delta.IsSet() && Delta.Get() > 0)
        {
            const double MouseDelta =
FMath::Abs(MouseEvent.GetCursorDelta().X / (float)LinearDeltaSensitivity.Get());
            NewValue = InternalValue + (Sign * MouseDelta *
FMath::Pow((double)Delta.Get(), (double)SliderExponent.Get())) * Step;
        }
        else
        {
            const double MouseDelta =
FMath::Abs(MouseEvent.GetCursorDelta().X / SliderWidthInSlateUnits);
            const double CurrentValue = FMath::Clamp<double>
(FMath::Abs(InternalValue), 1.0,
(double)std::numeric_limits<NumericType>::max());
            NewValue = InternalValue + (Sign * MouseDelta *
FMath::Pow((double)CurrentValue, (double)SliderExponent.Get())) * Step;
        }
    }
}

virtual FReply SSpinBox<NumericType>::OnKeyDown(const FGeometry& MyGeometry,
const FKeyEvent& InKeyEvent) override
{
    else if (Key == EKeys::Up || Key == EKeys::Right)
    {
        const NumericType LocalValueAttribute = ValueAttribute.Get();
        const NumericType LocalDelta = Delta.Get();
        InternalValue = (double)LocalValueAttribute;
        CommitValue(LocalValueAttribute + LocalDelta, InternalValue +
(double)LocalDelta, CommittedViaArrowKey, ETextCommit::OnEnter);
        ExitTextMode();
    }
}

```

```

        return FReply::Handled();
    }
    else if (Key == EKeys::Down || Key == EKeys::Left)
    {
        const NumericType LocalValueAttribute = ValueAttribute.Get();
        const NumericType LocalDelta = Delta.Get();
        InternalValue = (double)LocalValueAttribute;
        CommitValue(LocalValueAttribute - LocalDelta, InternalValue +
(double)LocalDelta, CommittedViaArrowKey, ETextCommit::OnEnter);
        ExitTextMode();
        return FReply::Handled();
    }
}

void SSpinBox<NumericType>::CommitValue(NumericType NewValue, double
NewSpinValue, ECommitMethod CommitMethod, ETextCommit::Type OriginalCommitInfo)
{
    // If needed, round this value to the delta. Internally the value is not held
    // to the delta but externally it appears to be.
    if (CommitMethod == CommittedviaSpin || CommitMethod == CommittedviaArrowKey
|| bAlwaysUsesDeltaSnap)
    {
        NumericType CurrentDelta = Delta.Get();
        if (CurrentDelta != NumericType())
        {
            NewValue = FMath::GridSnap<NumericType>(NewValue, CurrentDelta); ///
            snap numeric point value to nearest Delta
        }
    }
}

```

## LinearDeltaSensitivity

- **Function description:** After setting the Delta, further configure the numeric input box to change linearly along with its sensitivity to changes (the higher the value, the less sensitive it is)
- **Usage location:** UPROPERTY
- **Engine module:** Numeric Property
- **Metadata type:** float/int
- **Restriction type:** float, int32
- **Associated items:** Delta
- **Commonly used:** ★★★

Conditions for effectiveness:

1. First, set Delta > 0
2. Do not set UIMin, UIMax
3. Set LinearDeltaSensitivity > 0

## Test code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(UIMin = "0", UIMax = "1000", Delta = 10))
float MyFloat_Delta10_UIMinMax = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(Delta = 10, LinearDeltaSensitivity = 50))
float MyFloat_Delta10_LinearDeltaSensitivity50 = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DeltaTest, meta =
(UIMin = "0", UIMax = "1000", Delta = 10, LinearDeltaSensitivity = 50))
float MyFloat_Delta10_LinearDeltaSensitivity50_UIMinMax = 100;
```

## Test results:

For an analysis of the effects, please refer to: Analysis of Delta

▼ Delta Test	
My Float Default Delta	100.0
My Float Delta 10	100.0
My Float Delta 10 UIMin Max	100.0
My Float Delta 10 Linear Delta Sensitivity 50	100.0
My Float Delta 10 Linear Delta Sensitivity 50 UIMin Max	100.0

## Principle:

It is evident that the code branch for linear change can only be entered only when there are no UIMinMax constraints and after Delta has been set.

```
, _Delta(0)
virtual FReply SSpinBox<NumericType>::OnMouseMove(const FGeometry& MyGeometry,
const FPointerEvent& MouseEvent) override
{
    if (bUnlimitedSpinRange)
    {
        // If this control has a specified delta and sensitivity then we
        // use that instead of the current value for determining how much to change.
        const double sign = (MouseEvent.GetCursorDelta().X > 0) ? 1.0 :
        -1.0;

        if (LinearDeltaSensitivity.IsSet() &&
LinearDeltaSensitivity.Get() != 0 && Delta.IsSet() && Delta.Get() > 0)
        {
            const double MouseDelta =
FMath::Abs(MouseEvent.GetCursorDelta().X / (float)LinearDeltaSensitivity.Get());
            NewValue = InternalValue + (Sign * MouseDelta *
FMath::Pow((double)Delta.Get(), (double)SliderExponent.Get()) * Step;
        }
        else
        {
            const double MouseDelta =
FMath::Abs(MouseEvent.GetCursorDelta().X / SliderWidthInSlateUnits);
```

```

        const double CurrentValue = FMath::Clamp<double>
(FMath::Abs(Internalvalue), 1.0,
(double)std::numeric_limits<NumericType>::max());
                NewValue = Internalvalue + (sign * MouseDelta *
FMath::Pow((double)CurrentValue, (double)sliderExponent.Get()) * Step;
}
}
}

```

## UIMin

- **Function Description:** Specifies the minimum range value for the scrollbar drag on the numeric input field
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** float/int
- **Restriction Type:** float, int32
- **Associated Items:** UIMax, ClampMin, ClampMax
- **Commonly Used:** ★★★★

The difference between UIMin-UIMax and ClampMin-ClampMax is that the UI series prevents users from exceeding a certain range while dragging the mouse, but users can still manually input values outside this range. The Clamp series, however, sets actual value constraints, preventing users from exceeding these limits whether they are dragging or manually entering values.

Neither of these constraints can prevent direct value modifications in Blueprints.

## Test Code:

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MinMaxTest)
float MyFloat_NoMinMax = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MinMaxTest, meta =
(UIMin = "0", UIMax = "100"))
float MyFloat_HasMinMax_UI = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MinMaxTest, meta =
(ClampMin = "0", ClampMax = "100"))
float MyFloat_HasMinMax_Clamp = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MinMaxTest, meta =
(ClampMin = "0", ClampMax = "100", UIMin = "20", UIMax = "50"))
float MyFloat_HasMinMax_ClampAndUI = 100;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = MinMaxTest, meta =
(ClampMin = "20", ClampMax = "50", UIMin = "0", UIMax = "100"))
float MyFloat_HasMinMax_ClampAndUI2 = 100;

```

## Test Results:

- From MyFloat\_HasMinMax\_UI, it is observed that UIMin and UIMax limit the scrollbar range of the numeric input field, but values exceeding 999 can still be manually entered

- In MyFloat\_HasMinMax\_Clamp, ClampMin and ClampMax simultaneously limit both the UI and manual input ranges.
- In MyFloat\_HasMinMax\_ClampAndUI and MyFloat\_HasMinMax\_ClampAndUI2, it is noted that the UI scrollbar will adopt the narrower range between the UI constraints and the Clamp constraints, and the actual input value will also be confined within this narrower range.



## Principle:

TNumericPropertyParams acquires certain metadata upon construction to initialize these variables, or they will revert to default values.

Numeric types have actual minimum and maximum values (MinValue-MaxValue), provided by ClampMin and ClampMax. There are also UI minimum and maximum values (MinSliderValue-MaxSliderValue), determined by Max(UImin, ClampMin) and Min(UIMax, ClampMax), ensuring the smallest range for legal input.

```
template<typename NumericType>
struct TNumericPropertyParams
{
    if (MetaDataGetter.IsBound())
    {
        UIMinString = MetaDataGetter.Execute("UIMin");
        UIMaxString = MetaDataGetter.Execute("UIMax");
        SliderExponentString = MetaDataGetter.Execute("SliderExponent");
        LinearDeltaSensitivityString =
            MetaDataGetter.Execute("LinearDeltaSensitivity");
        DeltaString = MetaDataGetter.Execute("Delta");
        ClampMinString = MetaDataGetter.Execute("ClampMin");
        ClampMaxString = MetaDataGetter.Execute("ClampMax");
        ForcedUnits = MetaDataGetter.Execute("ForceUnits");
        WheelStepString = MetaDataGetter.Execute("WheelStep");
    }

    TOptional<NumericType> MinValue;
    TOptional<NumericType> MaxValue;
    TOptional<NumericType> MinSliderValue;
    TOptional<NumericType> MaxSliderValue;
    NumericType SliderExponent;
    NumericType Delta;
    int32 LinearDeltaSensitivity;
    TOptional<NumericType> WheelStep;
}
```

```

//Finally, these values are passed on to
SAssignNew(SpinBox, SSpinBox<NumericType>)
    .Style(InArgs._SpinBoxStyle)
    .Font(InArgs._Font.IsSet() ? InArgs._Font : InArgs._EditableTextBoxStyle->TextStyle.Font)
    .Value(this, &SNumericEntryBox<NumericType>::OnGetValueForSpinBox)
    .Delta(InArgs._Delta)
    .ShiftMultiplier(InArgs._ShiftMultiplier)
    .CtrlMultiplier(InArgs._CtrlMultiplier)
    .LinearDeltaSensitivity(InArgs._LinearDeltaSensitivity)
    .SupportDynamicsSliderMaxValue(InArgs._SupportDynamicsSliderMaxValue)
    .SupportDynamicsSliderMinValue(InArgs._SupportDynamicsSliderMinValue)
    .OnDynamicsSliderMaxValueChanged(InArgs._OnDynamicsSliderMaxValueChanged)
    .OnDynamicsSliderMinValueChanged(InArgs._OnDynamicsSliderMinValueChanged)
    .OnValueChanged(OnValueChanged)
    .OnValueCommitted(OnValueCommitted)
    .MinFractionalDigits(MinFractionalDigits)
    .MaxFractionalDigits(MaxFractionalDigits)
    .MinSliderValue(InArgs._MinSliderValue)
    .MaxSliderValue(InArgs._MaxSliderValue)
    ..MaxValue(InArgs._.MaxValue)
    .MinValue(InArgs._.MinValue)
    .SliderExponent(InArgs._SliderExponent)
    .SliderExponentNeutralValue(InArgs._SliderExponentNeutralValue)
    .EnableWheel(InArgs._AllowWheel)
    .BroadcastValueChangesPerKey(InArgs._BroadcastValueChangesPerKey)
    .WheelStep(InArgs._WheelStep)
    .OnBeginSliderMovement(InArgs._OnBeginSliderMovement)
    .OnEndSliderMovement(InArgs._OnEndSliderMovement)
    .MinDesiredWidth(InArgs._MinDesiredValueWidth)
    .TypeInterface(Interface)
    .ToolTipText(this, &SNumericEntryBox<NumericType>::GetValueAsText);

//In Conclusion
void SSpinBox<NumericType>::CommitValue(NumericType NewValue, double
NewSpinValue, ECommitMethod CommitMethod, ETextCommit::Type OriginalCommitInfo)
{
    if (CommitMethod == CommittedViaSpin || CommitMethod == CommittedViaArrowKey)
    {
        const NumericType LocalMinSliderValue = GetMinSliderValue();
        const NumericType LocalMaxSliderValue = GetMaxSliderValue();
        NewValue = FMath::Clamp<NumericType>(NewValue, LocalMinSliderValue,
LocalMaxSliderValue);
        NewSpinValue = FMath::Clamp<double>(NewSpinValue,
(double)LocalMinSliderValue, (double)LocalMaxSliderValue);
    }

    {
        const NumericType LocalMinValue = GetMinValue();
        const NumericType LocalMaxValue = GetMaxValue();
        NewValue = FMath::Clamp<NumericType>(NewValue, LocalMinValue,
LocalMaxValue);
        NewSpinValue = FMath::Clamp<double>(NewSpinValue, (double)LocalMinValue,
(double)LocalMaxValue);
    }
}

```

```

// update the internal value, this needs to be done before rounding.
Internalvalue = NewSpinValue;

const bool bAlwaysUsesDeltaSnap = GetAlwaysUsesDeltaSnap();
// If needed, round this value to the delta. Internally the value is not held
to the Delta but externally it appears to be.
if (CommitMethod == CommittedviaSpin || CommitMethod == CommittedviaArrowKey
|| bAlwaysUsesDeltaSnap)
{
    NumericType CurrentDelta = Delta.Get();
    if (CurrentDelta != NumericType())
    {
        NewValue = FMath::GridSnap<NumericType>(NewValue, CurrentDelta); ///
snap numeric point value to nearest Delta
    }
}

// update the max slider value based on the current value if we're in dynamic
mode
if (SupportDynamicslider.MaxValue.Get() && ValueAttribute.Get() >
GetMaxSliderValue())
{
    ApplySlider.MaxValueChanged(Float(ValueAttribute.Get() -
GetMaxSliderValue()), true);
}
else if (SupportDynamicslider.MinValue.Get() && ValueAttribute.Get() <
GetMinSliderValue())
{
    ApplySlider.MinValueChanged(Float(ValueAttribute.Get() -
GetMinSliderValue()), true);
}

```

## UIMax

- **Function Description:** Specifies the maximum range value for the scrollbar drag on the numeric input field
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** float/int
- **Restriction Type:** float, int32
- **Associated Items:** UIMin
- **Commonality:** ★★★★☆

# ClampMin

---

- **Function Description:** Specifies the minimum value that the numeric input field will actually accept
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** float/int
- **Restriction Type:** float, int32
- **Associated Items:** UIMin
- **Commonly Used:** ★★★★☆

# ClampMax

---

- **Function Description:** Specifies the maximum value that the numeric input field will actually accept
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** float/int
- **Restriction Type:** float, int32
- **Associated Items:** UIMax
- **Commonality:** ★★★★☆

1 # SupportDynamicSliderMinValue

- **Function description:** Supports dynamically changing the minimum range value of the scrollbar on the numeric input field when the Alt key is pressed
- **Usage location:** UPROPERTY
- **Engine module:** Numeric Property
- **Metadata type:** bool
- **Restriction type:** FVector4
- **Related items:** SupportDynamicSlider.MaxValue
- **Commonly used:** ★

Enables the dynamic adjustment of the minimum range value of the scrollbar on the numeric input field when the Alt key is pressed.

- Must be used in conjunction with UIMin and UIMax, as the scrollbar UI is only available with these
- Usually, the scrollbar range is pre-set, but this setting allows for dynamic changes. The method is to press and hold the Alt key while dragging the mouse.
- Standard float properties do not support this meta, because the default SPropertyEditorNumeric does not extract SupportDynamicSliderMinValue from the property meta, rendering any settings ineffective.

- Currently, only FColorGradingVectorCustomizationBase (which corresponds to FVector4) and inherits from FMathStructCustomization extracts SupportDynamicSliderMinValue, and then creates SNumericEntryBox to correctly set the value of SupportDynamicSliderMinValue.
- Therefore, if you wish to enable this feature for numeric properties in your own structure, you will need to manually create a Customization and within it, manually create a SNumericEntryBox to set the value of SupportDynamicSliderMinValue.

## Test Code:

```

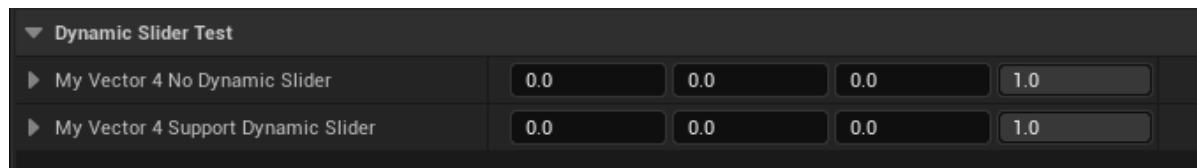
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DynamicsliderTest,
meta = (UIMin = "0", UIMax = "1"))
FVector4 MyVector4_NoDynamicslider;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DynamicsliderTest,
meta = (UIMin = "0", UIMax = "1", SupportDynamicSliderMinValue = "true",
SupportDynamicSliderMaxValue = "true"))
FVector4 MyVector4_SupportDynamicSlider;

```

## Test Results:

It can be observed that MyVector4\_NoDynamicSlider is unable to change the scrollbar range of 0-1. In contrast, MyVector4\_SupportDynamicSlider can adjust the minimum and maximum UI range after pressing and holding the Alt key while dragging the mouse.



## Principle:

The property SupportDynamicSliderMinValue is not retrieved in SPropertyEditorNumeric, hence it is not effective by default.

```

void SPropertyEditorNumeric<NumericType>::Construct( const FArguments& InArgs,
const TSharedRef<FPropertyEditor>& InPropertyEditor )
{
    TNumericPropertyParams<NumericType> NumericPropertyParams(Property,
MetaDataTable);
    ChildSlot
    [
        SAssignNew(PrimaryWidget, SNumericEntryBox<NumericType>)
        // Only allow spinning if we have a single value
        .AllowSpin(bAllowSpin)
        .Value(this, &SPropertyEditorNumeric<NumericType>::OnGetValue)
        .Font(InArgs._Font)
        .MinValue(NumericPropertyParams.MinValue)
        .MaxValue(NumericPropertyParams.MaxValue)
        .MinSliderValue(NumericPropertyParams.MinSliderValue)
        .MaxSliderValue(NumericPropertyParams.MaxSliderValue)
        .SliderExponent(NumericPropertyParams.SliderExponent)
        .Delta(NumericPropertyParams.Delta)
    ]
}

```

```

        // LinearDeltaSensitivity needs to be left unset if not provided, rather
        // than being set to some default

    .LinearDeltaSensitivity(NumericPropertyParams.GetLinearDeltaSensitivityAttribute(
    ))
        .Allowwheel(bAllowSpin)
        .WheelStep(NumericPropertyParams.WheelStep)
        .UndeterminedString(PropertyEditorConstants::DefaultUndeterminedText)
        .OnValueChanged(this,
&SPropertyEditorNumeric<NumericType>::OnValueChanged)
        .OnValueCommitted(this,
&SPropertyEditorNumeric<NumericType>::OnValueCommitted)
        .OnUndeterminedValueCommitted(this,
&SPropertyEditorNumeric<NumericType>::OnUndeterminedValueCommitted)
        .OnBeginsliderMovement(this,
&SPropertyEditorNumeric<NumericType>::OnBeginsliderMovement)
        .OnEndsliderMovement(this,
&SPropertyEditorNumeric<NumericType>::OnEndsliderMovement)
        .TypeInterface(TypeInterface)

    ];
}

virtual FReply OnMouseMove(const FGeometry& MyGeometry, const FPointerEvent&
MouseEvent) override
{
    if (MouseEvent.IsAltDown())
    {
        float DeltaToAdd =
(float)MouseEvent.GetCursorDelta().X / sliderwidthInSlateUnits;

        if (SupportDynamicSliderMaxValue.Get() &&
(NumericType)InternalValue == GetMaxSliderValue())
        {
            ApplySliderMaxValueChanged(DeltaToAdd, false);
        }
        else if (SupportDynamicSliderMinValue.Get() &&
(NumericType)InternalValue == GetMinSliderValue())
        {
            ApplySliderMinValueChanged(DeltaToAdd, false);
        }
    }
}

```

## SupportDynamicSlider.MaxValue

- **Function Description:** Supports dynamically changing the maximum range value of the scrollbar for numeric input boxes when the Alt key is pressed
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** bool
- **Restriction Type:** FVector4
- **Associated Items:** SupportDynamicSliderMinValue

- **Commonality:** ★

# ArrayClamp

- **Function description:** Ensures that the value of an integer attribute is within the valid index range of a specified array, [0, ArrayClamp.Size() - 1]
- **Usage location:** UPROPERTY
- **Engine module:** Numeric Property
- **Metadata type:** int32
- **Restriction type:** int32
- **Commonly used:** ★★★

Ensures that the value of the integer attribute is within the valid index range of the specified array, [0, ArrayClamp.Size() - 1]

## Test code:

```
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ArrayClampTest)
    int32 MyInt_NoArrayClamp = 0;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ArrayClampTest, meta =
    (ArrayClamp = "MyIntArray"))
    int32 MyInt_HasArrayClamp = 0;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ArrayClampTest)
    TArray<int32> MyIntArray;
```

## Test results:

It is evident that the integer value with ArrayClamp is constrained within the array's index range.

▼ Array Clamp Test	
My Int No Array Clamp	0
My Int Has Array Clamp	0
▼ My Int Array	
Index [ 0 ]	0
Index [ 1 ]	0
Index [ 2 ]	0

## Principle:

Locates the Array attribute within the class based on the specified array name and then clamps the value of the integer attribute to the index range of the array.

```
template <typename Type>
static Type ClampIntegerValueFromMetaData(Type InValue, FPropertyHandleBase&
InPropertyHandle, FPropertyNode& InPropertyNode)
{
    Type RetVal = ClampValueFromMetaData<Type>(InValue, InPropertyHandle);

    //enforce array bounds
    const FString& ArrayClampString =
InPropertyHandle.GetMetaData(TEXT("ArrayClamp"));
    if (ArrayClampString.Len())
    {
        FObjectPropertyNode* ObjectPropertyNode =
InPropertyNode.FindObjectItemParent();
        if (ObjectPropertyNode && ObjectPropertyNode->GetNumObjects() == 1)
        {
            Type LastValidIndex = static_cast<Type>
(GetArrayPropertyLastValidIndex(ObjectPropertyNode, ArrayClampString));
            RetVal = FMath::Clamp<Type>(RetVal, 0, LastValidIndex);
        }
        else
        {
            UE_LOG(LogPropertyName, Warning, TEXT("Array Clamping isn't supported
in multi-select (Param Name: %s)"), *InPropertyHandle.GetProperty()->GetName());
        }
    }

    return RetVal;
}
```

## Hide Alpha Channel

- **Function Description:** Hides the Alpha channel of FColor or FLinearColor properties during editing.
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** bool
- **Restricted Types:** FColor, FLinearColor
- **Commonality:** ★★★

Makes the FColor or FLinearColor property hide the alpha channel when editing.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, Category = AlphaTest)  
    FColor MyColor;  
  
    UPROPERTY(EditAnywhere, Category = AlphaTest, meta = (HideAlphaChannel))  
    FColor MyColor_HideAlphaChannel;  
  
    UPROPERTY(EditAnywhere, Category = AlphaTest)  
    FLinearColor MyLinearColor;  
  
    UPROPERTY(EditAnywhere, Category = AlphaTest, meta = (HideAlphaChannel))  
    FLinearColor MyLinearColor_HideAlphaChannel;
```

## Test Results:

It is evident that properties with HideAlphaChannel do not display an Alpha channel.



## Principle:

```
void FColorStructCustomization::CustomizeHeader(TSharedRef<class IPropertyHandle>  
InStructPropertyHandle, class FDetailWidgetRow& InHeaderRow,  
IPropertyTypeCustomizationUtils& StructCustomizationUtils)  
{  
    bIgnoreAlpha = TypeSupportsAlpha() == false || StructPropertyHandle->  
GetProperty()->HasMetaData(TEXT("HideAlphaChannel"));  
}  
  
.AlphaDisplayMode(bIgnoreAlpha ? EColorBlockAlphaDisplayMode::Ignore :  
EColorBlockAlphaDisplayMode::Separate)
```

# AllowPreserveRatio

- **Function Description:** Adds a ratio lock feature for the FVector attribute within the details panel.
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** bool
- **Restricted Type:** FVector
- **Commonality:** ★★★

Adds a ratio lock to the FVector property in the details panel.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, Category = VectorTest)  
    FVector MyVector_Default;  
  
    UPROPERTY(EditAnywhere, Category = VectorTest, meta = (AllowPreserveRatio))  
    FVector MyVector_AllowPreserveRatio;  
  
    UPROPERTY(EditAnywhere, Category = VectorTest, meta = (ShowNormalize))  
    FVector MyVector_ShowNormalize;
```

## Test Results:

It is evident that the value of MyVector\_AllowPreserveRatio maintains a fixed ratio after being locked.



## Principle:

The mechanism involves detecting the AllowPreserveRatio during UI customization and creating a dedicated UI element for it.

```
void FMathStructCustomization::MakeHeaderRow(TSharedRef<class IPropertyHandle>&  
StructPropertyHandle, FDetailWidgetRow& Row)  
{  
    if (StructPropertyHandle->HasMetaData("AllowPreserveRatio"))
```

```

    {
        if (!GConfig->GetBool(TEXT("SelectionDetails"), *
(StructPropertyHandle->GetProperty()->GetName() + TEXT("_PreserveScaleRatio")),
bPreserveScaleRatio, GEditorPerProjectIni))
        {
            bPreserveScaleRatio = true;
        }

        HorizontalBox->AddSlot()
        .Autowidth()
        .Maxwidth(18.0f)
        .VAlign(VAlign_Center)
        [
            // Add a checkbox to toggle between preserving the ratio of x,y,z
components of scale when a value is entered
            SNew(SCheckBox)
            .IsChecked(this,
&FMathStructCustomization::IsPreserveScaleRatioChecked)
            .OnCheckStateChanged(this,
&FMathStructCustomization::OnPreserveScaleRatioToggled, StructweakHandlePtr)
            .Style(FAppStyle::Get(), "TransparentCheckBox")
            .ToolTipText(LOCTEXT("PreserveScaleToolTip", "When locked, scales
uniformly based on the current xyz scale values so the object maintains its shape
in each direction when scaled"))
            [
                SNew(SImage)
                .Image(this,
&FMathStructCustomization::GetPreserveScaleRatioImage)
                .ColorAndOpacity(FSlateColor::UseForeground())
            ]
        ];
    }

}

```

## NoSpinbox

- **Function Description:** Prevents the default UI editing features for drag-and-drop and mouse wheel manipulation of numerical properties, which include the int series and float series.
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** bool
- **Restricted Types:** Numeric Types, int / float
- **Commonality:** ★★

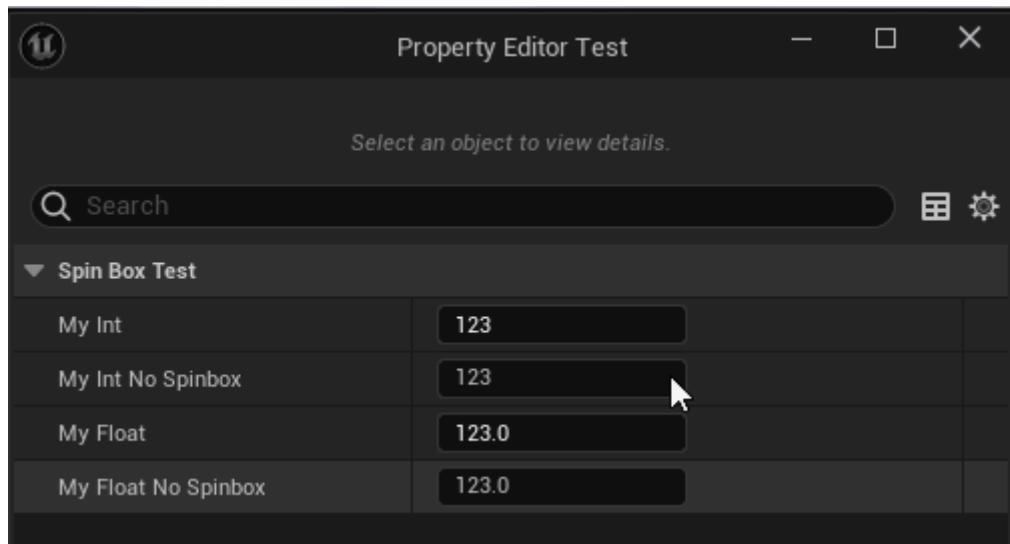
Disables the default UI editing features for drag-and-drop and mouse wheel manipulation of numerical properties, including the int series and float series.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=SpinBoxTest)  
    int32 MyInt = 123;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=SpinBoxTest, meta =  
(NoSpinbox = true))  
    int32 MyInt_NoSpinbox = 123;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=SpinBoxTest)  
    float MyFloat = 123;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=SpinBoxTest, meta =  
(NoSpinbox = true))  
    float MyFloat_NoSpinbox = 123;
```

## Test Effects:

It is observed that properties with NoSpinbox cannot be changed by dragging the mouse left or right, nor can the value be altered using the mouse wheel.



## Principle:

It can be noted that for the UI of numerical values, the function of bAllowSpin directly determines the Widget's AllowWheel and AllowSpin functionalities.

```
virtual TSharedRef<swidget> GetDefaultValueWidget() override  
{  
    const typename TNumericPropertyParams<NumericType>::FMetaDataGetter  
    MetaDataGetter =  
    TNumericPropertyParams<NumericType>::FMetaDataGetter::CreateLambda([&] (const  
    FName& Key)  
    {  
        return (PinProperty) ? PinProperty->GetMetaData(Key) : FString();  
    });  
  
    TNumericPropertyParams<NumericType> NumericPropertyParams(PinProperty,  
    PinProperty ? MetaDataGetter : nullptr);
```

```

const bool bAllowSpin = !(PinProperty && PinProperty-
>GetBoolMetaData("NoSpinbox"));

// Save last committed value to compare when value changes
LastSliderCommittedValue = GetNumericValue().GetValue();

return SNew(SBox)
    .MinDesiredWidth(MinDesiredBoxWidth)
    .MaxDesiredWidth(400)
    [
        SNew(SNumericEntryBox<NumericType>)
            .EditableTextBoxStyle(FAppStyle::Get(), "Graph.EditableTextBox")
            .BorderForegroundColor(FSlateColor::UseForeground())
            .Visibility(this, &SGraphPinNumSlider::GetDefaultValueVisibility)
            .IsEnabled(this, &SGraphPinNumSlider::GetDefaultValueIsEditable)
            .Value(this, &SGraphPinNumSlider::GetNumericValue)
            .MinValue(NumericPropertyParams.MinValue)
            .MaxValue(NumericPropertyParams.MaxValue)
            .MinSliderValue(NumericPropertyParams.MinSliderValue)
            .MaxSliderValue(NumericPropertyParams.MaxSliderValue)
            .SliderExponent(NumericPropertyParams.SliderExponent)
            .Delta(NumericPropertyParams.Delta)

        .LinearDeltaSensitivity(NumericPropertyParams.GetLinearDeltaSensitivityAttribute(
        ))
            .AllowWheel(bAllowSpin)
            .WheelStep(NumericPropertyParams.WheelStep)
            .AllowSpin(bAllowSpin)
            .OnValueCommitted(this, &SGraphPinNumSlider::OnValueCommitted)
            .OnValueChanged(this, &SGraphPinNumSlider::OnValueChanged)
            .OnBeginSliderMovement(this,
&SGraphPinNumSlider::OnBeginSliderMovement)
            .OnEndSliderMovement(this, &SGraphPinNumSlider::OnEndSliderMovement)
        ];
    ];
}

```

## SRGB

---

- **Function Description:** Ensures that FColor or FLinearColor attributes utilize the sRGB color space during editing.
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** bool
- **Restricted Types:** FColor, FLinearColor

Enables the FColor or FLinearColor property to use the sRGB color space when being edited.

However, it does not function as expected during testing.

## Principle:

```
void FColorStructCustomization::CustomizeHeader(TSharedRef<class IPropertyHandle>
InStructPropertyHandle, class FDetailWidgetRow& InHeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    if (StructPropertyHandle->GetProperty()->HasMetaData(TEXT("SRGB")))
    {
        SRGBOverride = StructPropertyHandle->GetProperty()->
>GetBoolMetaData(TEXT("SRGB"));
    }

}
```

## WheelStep

- **Function description:** Specifies the value change induced by mouse wheel scrolling on a numeric input field
- **Usage location:** UPROPERTY
- **Engine module:** Numeric Property
- **Metadata type:** float/int
- **Commonality:** ★★★

Specifies the value change induced by mouse wheel scrolling up or down on a numeric input field.

## Default value rules:

If the property is a floating-point number and the UI scrollbar range is less than 10, then WheelStep=0.1; otherwise, it is 1

If the property is an integer, then WheelStep=1

## Test code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = wheelStepTest)
float MyFloat_DefaultWheelStep = 50;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = wheelStepTest, meta =
(UIMin = "0", UIMax = "10"))
float MyFloat_SmallWheelStep = 1;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = wheelStepTest, meta =
(wheelStep = 10))
float MyFloat_HasWheelStep = 50;
```

## Screenshot:

The default value does not specify UIMin or UIMax, and the value can also be changed by mouse wheel scrolling. The default is 1

The UI range for MyFloat\_SmallWheelStep is only 10, so the default change increment is 0.1

If WheelStep is specified as 10, the value will change by 10 in one go

Wheel Step Test	
My Float Default Wheel Step	50.0
My Float Small Wheel Step	1.0
My Float Has Wheel Step	50.0

## Principle:

As indicated by the code, if WheelStep is set, that value is used.

Otherwise, if it is a floating-point number and the UI scrollbar range is less than 10, then WheelStep=0.1; otherwise, it is 1

Otherwise, if it is an integer, then WheelStep=1

```
virtual FReply SSpinBox<NumericType>::OnMousewheel(const FGeometry&
MyGeometry, const FPointerEvent& MouseEvent) override
{
    if (bEnablewheel && PointerDraggingsliderIndex == INDEX_NONE &&
HasKeyboardFocus())
    {
        // If there is no wheelStep defined, we use 1.0 (or 0.1 if slider
range is <= 10)
        constexpr bool bIsIntegral = TIsIntegral<NumericType>::Value;
        const bool bIsSmallStep = !bIsIntegral && (GetMaxSliderValue() -
GetMinSliderValue()) <= 10.0;
        double Step = WheelStep.IsSet() && wheelStep.Get().IsSet() ?
wheelStep.Get().GetValue() : (bIsSmallStep ? 0.1 : 1.0);

        if (MouseEvent.IsControlDown())
        {
            // If no value is set for wheelSmallStep, we use the DefaultStep
multiplied by the CtrlMultiplier
            Step *= CtrlMultiplier.Get();
        }
        else if (MouseEvent.IsShiftDown())
        {
            // If no value is set for wheelBigStep, we use the DefaultStep
multiplied by the ShiftMultiplier
            Step *= ShiftMultiplier.Get();
        }

        const double Sign = (MouseEvent.GetWheelDelta() > 0) ? 1.0 : -1.0;
        const double NewValue = InternalValue + (Sign * Step);
        const NumericType RoundedNewValue = RoundIfIntegerValue(NewValue);

        return FReply::Handled();
    }

    return FReply::Unhandled();
}
```

# InlineColorPicker

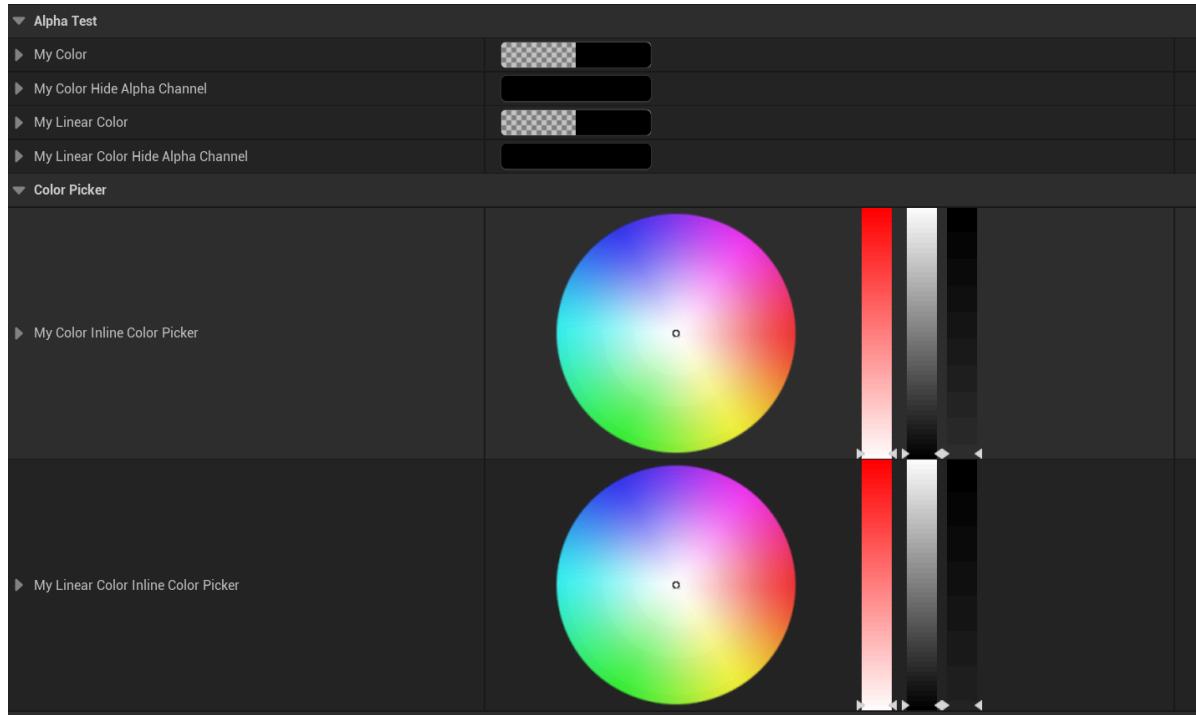
- **Function Description:** Allows direct inline editing of FColor or FLinearColor properties with a color picker during the editing process.
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** boolean
- **Restricted Types:** FColor, FLinearColor
- **Commonly Used:** ★★

Enables direct inline editing of FColor or FLinearColor properties with a color picker during the editing process.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, Category = ColorPicker, meta = (InlineColorPicker))  
    FColor MyColor_InlineColorPicker;  
    UPROPERTY(EditAnywhere, Category = ColorPicker, meta = (InlineColorPicker))  
    FLinearColor MyLinearColor_InlineColorPicker;
```

## Test Results:



## Principle:

creates different ColorWidgets based on various tags.

```

void FColorStructCustomization::MakeHeaderRow(TSharedRef<class IPropertyHandle>&
InStructPropertyHandle, FDetailWidgetRow& Row)
{
    if (InStructPropertyHandle->HasMetaData("InlineColorPicker"))
    {
        ColorWidget = CreateInlineColorPicker(StructWeakHandlePtr);
        ContentWidth = 384.0f;
    }
    else
    {
        ColorWidget = CreateColorWidget(StructWeakHandlePtr);
    }
}

```

## ShowNormalize

- Function Description:** Adds a normalization button for FVector variables in the details panel.
- Usage Location:** UPROPERTY
- Engine Module:** Numeric Property
- Metadata Type:** bool
- Restricted Type:** FVector
- Commonality:** ★★★

Enables a normalization button for FVector variables in the details panel.

## Test Code:

```

UPROPERTY(EditAnywhere, Category = VectorTest)
FVector MyVector_Default;

UPROPERTY(EditAnywhere, Category = VectorTest, meta = (AllowPreserveRatio))
FVector MyVector_AllowPreserveRatio;

UPROPERTY(EditAnywhere, Category = VectorTest, meta = (ShowNormalize))
FVector MyVector_ShowNormalize;

```

## Test Results:

The button to the right of MyVector\_ShowNormalize normalizes the value.



## Principle:

What actually happens is that during UI customization, the presence of ShowNormalize triggers the creation of a dedicated UI element.

```

if (StructPropertyHandle->HasMetaData("ShowNormalize") &&
MathStructCustomization::IsFloatVector(StructPropertyHandle))
{
    HorizontalBox->AddSlot()
        .AutoWidth()
        .MaxWidth(18.0f)
        .VAlign(VAlign_Center)
    [
        // Add a button to scale the vector uniformly to achieve a unit
        // vector
        SNew(SButton)
            .OnClicked(this, &FMathStructCustomization::OnNormalizeClicked,
StructWeakHandlePtr)
                .ButtonStyle(FAppStyle::Get(), "NoBorder")
                .ToolTipText(LOCTEXT("NormalizeToolTip", "when clicked, if the
vector is large enough, it scales the vector uniformly to achieve a unit vector
(vector with a length of 1)"))
        [
            SNew(SImage)
                .ColorAndOpacity(FSlateColor::UseForeground())
                .Image(FAppStyle::GetBrush(TEXT("Icons.Normalize")))
        ]
    ];
}

```

## ColorGradingMode

- **Function Description:** Enables an FVector4 attribute to be displayed as a color
- **Usage Location:** UPROPERTY
- **Engine Module:** Numeric Property
- **Metadata Type:** string = "abc"
- **Restriction Type:** FVector4
- **Commonality:** ★★

Enables an FVector4 attribute to be displayed as a color. As FVector4 aligns perfectly with RGBA.

Must be used in conjunction with UIMin and UIMax to avoid crashes, because FColorGradingVectorCustomization directly accesses the UIMinValue.

## Test Code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = ())
FVector4 MyVector4_NotColor;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = (UIMin = "0", UIMax = "1", ColorGradingMode = "saturation"))
FVector4 MyVector4_Saturation;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = (UIMin = "0", UIMax = "1", ColorGradingMode = "contrast"))
FVector4 MyVector4_Contrast;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = (UIMin = "0", UIMax = "1", ColorGradingMode = "gamma"))
FVector4 MyVector4_Gamma;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = (UIMin = "0", UIMax = "1", ColorGradingMode = "gain"))
FVector4 MyVector4_Gain;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = ColorGradingModeTest,
meta = (UIMin = "0", UIMax = "1", ColorGradingMode = "offset"))
FVector4 MyVector4_Offset;
```

## Test Results:

It can be observed that without ColorGradingMode, it remains a standard FVector4; otherwise, a color wheel is used for display and editing.

▼ Color Grading Mode Test

▼ My Vector 4 Not Color

	0.0	0.0	0.0	1.0
X	0.0			
Y	0.0			
Z	0.0			
W	1.0			

▼ My Vector 4 Saturation



0.0

RGB HSV

R	0.0	
G	0.0	
B	0.0	
Y	1.0	

▼ My Vector 4 Contrast



0.0

RGB HSV

R	0.0	
G	0.0	
B	0.0	
Y	1.0	

▼ My Vector 4 Gamma



0.0

RGB HSV

R	0.0	
G	0.0	
B	0.0	
Y	1.0	

▼ My Vector 4 Gain



0.0

RGB HSV

R	0.0	
G	0.0	
B	0.0	
Y	1.0	

▼ My Vector 4 Offset



0.0

R	0.0
G	0.0
B	0.0
Y	1.0

## Principle:

If the attribute is FVector4, and there is a ColorGradingMode, create FColorGradingVectorCustomization to customize the FVector for color display. Then, determine the EColorGradingModes based on the string, and ultimately create the corresponding specific UI control.

```
void FVector4StructCustomization::CustomizeChildren(TSharedRef<IPropertyHandle>
StructPropertyHandle, IDetailChildrenBuilder& StructBuilder,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    FProperty* Property = StructPropertyHandle->GetProperty();
    if (Property)
    {
        const FString& ColorGradingModeString = Property-
>GetMetaData(TEXT("ColorGradingMode"));
        if (!ColorGradingModeString.IsEmpty())
        {
            //Create our color grading customization shared pointer
            TSharedPtr<FColorGradingVectorCustomization>
ColorGradingCustomization =
GetOrCreateColorGradingVectorCustomization(StructPropertyHandle);

            //Customize the childrens
            ColorGradingVectorCustomization->CustomizeChildren(StructBuilder,
StructCustomizationUtils);

            // We handle the customize Children so just return here
            return;
        }
    }

    //Use the base class customize children
    FMathStructCustomization::CustomizeChildren(StructPropertyHandle,
StructBuilder, StructCustomizationUtils);
}

EColorGradingModes FColorGradingVectorCustomizationBase::GetColorGradingMode()
const
{
    EColorGradingModes ColorGradingMode = EColorGradingModes::Invalid;

    if (ColorGradingPropertyHandle.IsValid())
    {
        //Query all meta data we need
        FProperty* Property = ColorGradingPropertyHandle.Pin()->GetProperty();
        const FString& ColorGradingModeString = Property-
>GetMetaData(TEXT("ColorGradingMode"));

        if (ColorGradingModeString.Len() > 0)
        {
            if (ColorGradingModeString.Compare(TEXT("saturation")) == 0)
            {

```

```

        ColorGradingMode = EColorGradingModes::Saturation;
    }
    else if (ColorGradingModeString.Compare(TEXT("contrast")) == 0)
    {
        ColorGradingMode = EColorGradingModes::Contrast;
    }
    else if (ColorGradingModeString.Compare(TEXT("gamma")) == 0)
    {
        ColorGradingMode = EColorGradingModes::Gamma;
    }
    else if (ColorGradingModeString.Compare(TEXT("gain")) == 0)
    {
        ColorGradingMode = EColorGradingModes::Gain;
    }
    else if (ColorGradingModeString.Compare(TEXT("offset")) == 0)
    {
        ColorGradingMode = EColorGradingModes::Offset;
    }
}
}

return ColorGradingMode;
}

```

## DisplayThumbnail

---

- **Function Description:** Specifies whether to display a thumbnail on the left side of this attribute.
- **Usage Location:** UPROPERTY
- **Engine Module:** Object Property
- **Metadata Type:** bool
- **Restriction Type:** UObject\*
- **Associated Items:** ThumbnailSize
- **Commonliness:** ★★★

Specifies whether to display a thumbnail on the left side of this property.

### Test Code:

---

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_Thumbnail_Test :public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (DisplayThumbnail =
"false"))
    UObject* MyObject;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (DisplayThumbnail =
"true"))
    UObject* MyObject_DisplayThumbnail;

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite)
AActor* MyActor;

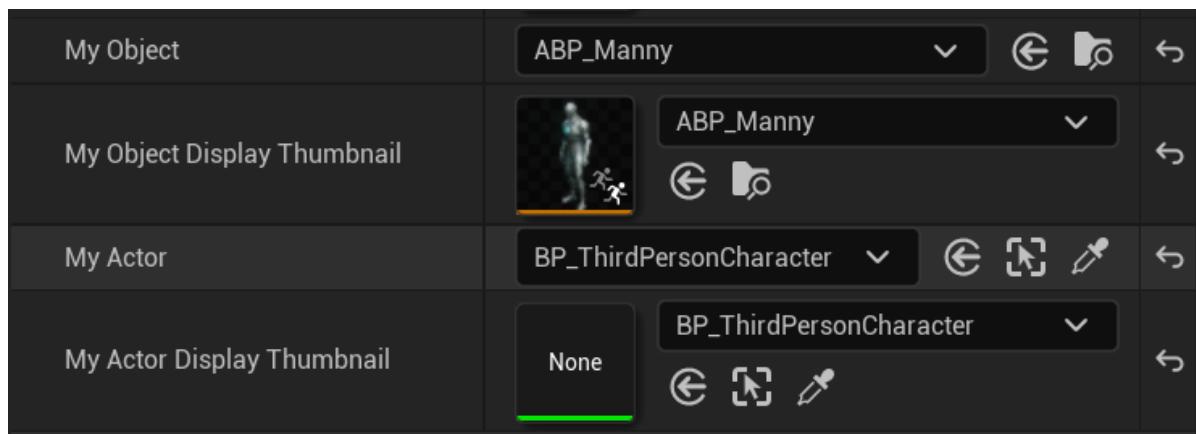
UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (DisplayThumbnail =
"true"))
AActor* MyActor_DisplayThumbnail;
};

```

## Test Code:

Visible that a thumbnail of the selected asset is displayed to the left of MyObject\_DisplayThumbnail, while MyObject does not have one because it is set to false. If DisplayThumbnail is not set to false, a thumbnail will be displayed by default.

MyActor\_DisplayThumbnail shows a thumbnail icon, but it has been noticed that the correct image is not displayed. AActor does not display a thumbnail by default.



## 测试效果:

The function determines whether to display a thumbnail.

By default, non-Actor types are displayed. Moreover, SPropertyEditorAsset is used for asset type properties, which essentially refers to the Object property.

```

bool SPropertyEditorAsset::ShouldDisplayThumbnail(const FArguments& InArgs, const
UClass* InObjectClass) const
{
    if (!InArgs._DisplayThumbnail || !InArgs._ThumbnailPool.IsValid())
    {
        return false;
    }

    bool bShowThumbnail = InObjectClass == nullptr || !InObjectClass-
>IsChildOf(AActor::StaticClass());

    // also check metadata for thumbnail & text display
    const FProperty* PropertyToCheck = nullptr;
    if (PropertyEditor.IsValid())
    {
        PropertyToCheck = PropertyEditor->GetProperty();
    }
    else if (PropertyHandle.IsValid())

```

```

    {
        PropertyToCheck = PropertyHandle->GetProperty();
    }

    if (PropertyToCheck != nullptr)
    {
        PropertyToCheck = GetActualMetadataProperty(PropertyToCheck);

        return GetTagOrBoolMetadata(PropertyToCheck, TEXT("DisplayThumbnail"),
bShowThumbnail);
    }

    return bShowThumbnail;
}

```

## ThumbnailSize

---

- **Function Description:** Modify the thumbnail size.
- **Usage Locations:** UCLASS, UPROPERTY
- **Engine Module:** Object Property
- **Metadata Type:** boolean
- **Associated Items:** DisplayThumbnail

Modifying the thumbnail size does not appear to have any effect.

## Principle:

---

```

void SObjectPropertyEntryBox::Construct( const FArguments& InArgs )
{
    // check if the property metadata wants us to display a thumbnail
    const FString& DisplayThumbnailString = PropertyHandle->GetProperty()->GetMetaData(TEXT("DisplayThumbnail"));
    if(DisplayThumbnailString.Len() > 0)
    {
        bDisplayThumbnail = DisplayThumbnailString == TEXT("true");
    }

    // check if the property metadata has an override to the thumbnail size
    const FString& ThumbnailsizeString = PropertyHandle->GetProperty()->GetMetaData(TEXT("ThumbnailSize"));
    if ( ThumbnailsizeString.Len() > 0 )
    {
        FVector2D ParsedVector;
        if ( ParsedVector.InitFromString(ThumbnailsizeString) )
        {
            Thumbnailsize.X = (int32)ParsedVector.X;
            Thumbnailsize.Y = (int32)ParsedVector.Y;
        }
    }
}

```

# LoadBehavior

---

- **Function Description:** Used on UCLASS to denote the loading behavior of this class, enabling the corresponding TObjectPtr attribute to support lazy loading. The default loading behavior is Eager, which can be changed to LazyOnDemand.
- **Usage Location:** UCLASS
- **Engine Module:** Object Property
- **Metadata Type:** string="abc"
- **Restriction Type:** TObjectPtr
- **Commonality:** ★

Applied to UCLASS to mark the class's loading behavior, allowing the associated TObjectPtr property to support delayed loading. The optional loading behavior defaults to Eager and can be set to LazyOnDemand.

- The default Eager behavior is similar to the common resource hard reference logic, meaning if A has a hard reference to B, B will be loaded recursively when A is loaded.
- The LazyOnDemand behavior means the resource will only be loaded when it is actually required (when Get is triggered). This is also a hard reference but is loaded lazily. Similarly, if A has a hard reference to B, B will not be loaded immediately when A is loaded; instead, the reference information (B's ObjectPath) is recorded first. When A needs to access B, B can be loaded at that time because the location of B has already been recorded in advance. If the loading is fast enough, it will be transparent to the user. LazyOnDemand is only effective in the editor, which allows for quicker editor startup without waiting for all resources to be loaded, as not all resources need to be loaded and parsed immediately for access.
- The difference from FSoftObjectPtr is that it represents a soft reference that requires manual judgment for loading时机. LazyOnDemand, on the other hand, is an automatic delayed loading mechanism that is transparent to the user, requiring no additional actions.
- LoadBehavior only affects TObjectPtr properties; UObject\* properties are always loaded directly. This is because only TObjectPtr has implemented the reference path information encoding for UObject\*, which allows for delayed loading.
- LoadBehavior is also only supported in the editor environment. At runtime, TObjectPtr is downgraded to UObject\*, resulting in direct loading of all instances.
- LoadBehavior is typically marked on asset type classes. Classes marked in the source code include DataAsset, DataTable, CurveTable, SoundCue, SoundWave, DialogueWave, and AnimMontage. Therefore, if you define a custom asset class that contains a lot of data, you can use LazyOnDemand to optimize the loading speed in the editor.
- To test LoadBehavior, enable the engine's LazyLoadImports feature, which is disabled by default. This can be done by adding the setting LazyLoadImports=True under the Core.System.Experimental section in DefaultEngine.ini. For source code reference, see the IsImportLazyLoadEnabled method.
- When testing, be cautious if you open a DataAsset asset by double-clicking, as the property details panel needs to display the property values, which triggers GetObjectPropertyValue\_InContainer and leads to ObjectHandleResolve, causing the resolution of TObjectPtr.

## Test Code:

Here, two DataAsset types, UMyDataAsset\_Eager and UMyDataAsset\_LazyOnDemand, are specifically defined for comparison with different LoadBehavior settings.

```
//(BlueprintType = true, IncludePath = Property/MyProperty_Asset.h,
IsBlueprintBase = true, LoadBehavior = Eager, ModuleRelativePath =
Property/MyProperty_Asset.h)
UCLASS(Blueprintable, Blueprintable, meta = (LoadBehavior = "Eager"))
class INSIDER_API UMyDataAsset_Eager :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score;
};

//(BlueprintType = true, IncludePath = Property/MyProperty_Asset.h,
IsBlueprintBase = true, LoadBehavior = LazyOnDemand, ModuleRelativePath =
Property/MyProperty_Asset.h)
UCLASS(Blueprintable, Blueprintable, meta = (LoadBehavior = "LazyOnDemand"))
class INSIDER_API UMyDataAsset_LazyOnDemand :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float Score;
};

UCLASS(BlueprintType)
class INSIDER_API UMyClass_LoadBehaviorTest :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    TObjectPtr<UMyDataAsset_LazyOnDemand> MyLazyOnDemand_AssetPtr;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    TObjectPtr<UMyDataAsset_Eager> MyEager_AssetPtr;

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere,meta = (LoadBehavior = "Eager"))
    TObjectPtr<UMyDataAsset_LazyOnDemand>
    MyLazyOnDemand_AssetPtr_EagerOnProperty;

    UPROPERTY(BlueprintReadWrite, EditAnywhere,meta = (LoadBehavior =
"LazyOnDemand"))
    TObjectPtr<UMyDataAsset_Eager> MyEager_AssetProperty_LazyOnDemandOnProperty;

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    UMyDataAsset_LazyOnDemand* MyLazyOnDemand_Asset;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    UMyDataAsset_Eager* MyEager_Asset;
```

```

public:
    UFUNCTION(BlueprintCallable)
    static void LoadBehaviorTest();
};

void UMyClass_LoadBehaviorTest::LoadBehaviorTest()
{
    UPackage* pk = LoadPackage(nullptr,
    TEXT("/Game/Class/Behavior/LoadBehavior/DA_LoadBehaviorTest"), 0);
    UMyClass_LoadBehaviorTest* obj = LoadObject<UMyClass_LoadBehaviorTest>(pk,
    TEXT("DA_LoadBehaviorTest"));
}

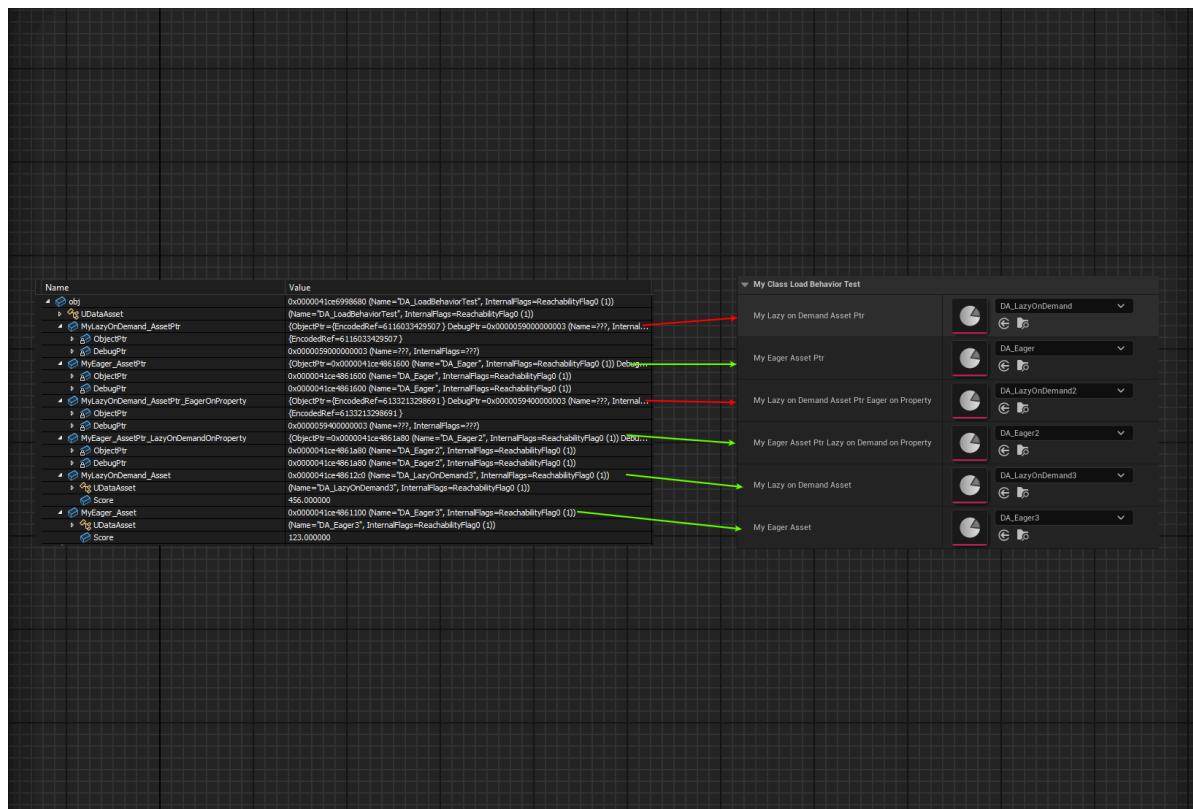
//Enable feature
DefaultEngine.ini
[Core.System.Experimental]
LazyLoadImports=True

```

## Test Results:

After the editor is running, manually call LoadBehaviorTest to load the DataSet of UMyClass\_LoadBehaviorTest. Check the object values for different property types. You will find:

- Among them, the ObjectPtr values for MyLazyOnDemand\_AssetPtr and MyLazyOnDemand\_AssetPtr\_EagerOnProperty are not yet resolved, while others show the values of the direct objects.
- The conclusion is that only marking LazyOnDemand on UCLASS will make delayed loading effective. Marking LoadBehavior on properties does not take effect. Properties of type UObject\* are always loaded directly.



# Principle:

In LinkerLoadImportBehavior.cpp, the FindLoadBehavior method is used to determine LoadBehavior, indicating that it only acts on UCLASS.

Also, in the TObjectPtr Get function, the call to ResolveObjectHandle is found, which is where resolution is triggered.

Note that the UE\_WITH\_OBJECT\_HANDLE\_LATE\_RESOLVE definition is WITH\_EDITORONLY\_DATA, meaning it is effective only in the editor environment.

```
//D:\github\UnrealEngine\Engine\Source\Runtime\CoreUObject\Private\UObject\Linker
LoadImportBehavior.cpp
enum class EImportBehavior : uint8
{
    Eager = 0,
    // @TODO: OBJPTR: we want to permit lazy background loading in the future
    //LazyBackground,
    LazyOnDemand,
};

EImportBehavior FindLoadBehavior(const UClass& Class)
{
    //Package class can't have meta data because of UHT
    if (&Class == UPackage::StaticClass())
    {
        return EImportBehavior::LazyOnDemand;
    }

    static const FName Name_LoadBehavior(TEXT("LoadBehavior"));
    if (const FString* LoadBehaviorMeta = Class.FindMetaData(Name_LoadBehavior))
    {
        if (*LoadBehaviorMeta == TEXT("LazyOnDemand"))
        {
            return EImportBehavior::LazyOnDemand;
        }
        return EImportBehavior::Eager;
    }
    else
    {
        //look in super class to see if it has lazy load on
        const UClass* Super = Class.GetSuperClass();
        if (Super != nullptr)
        {
            return FindLoadBehavior(*Super);
        }
        return EImportBehavior::Eager;
    }
}

#define UE_WITH_OBJECT_HANDLE_LATE_RESOLVE WITH_EDITORONLY_DATA

inline UObject* ResolveObjectHandle(FObjectHandle& Handle)
{
#if UE_WITH_OBJECT_HANDLE_LATE_RESOLVE || UE_WITH_OBJECT_HANDLE_TRACKING
```

```

        UObject* ResolvedObject = ResolveObjectHandleNoRead(Handle);
        UE::CoreUObject::Private::OnHandleRead(ResolvedObject);
        return ResolvedObject;
    #else
        return ReadObjectHandlePointerNoCheck(Handle);
    #endif
}

```

## ShowInnerProperties

- **Function description:** Display the internal properties of object references within the property details panel
- **Usage location:** UPROPERTY
- **Engine module:** Object Property
- **Metadata type:** bool
- **Restriction type:** UObject\*
- **Associated items:** ShowOnlyInnerProperties, FullyExpand, CollapsibleChildProperties
- **Commonality:** ★★★★☆

Displays the internal properties of object references within the property details panel.

By default, the internal properties of object reference attributes are not displayed in the details panel; only the object name is shown. However, if you wish to directly display and edit these internal properties, the ShowInnerProperties meta attribute comes into play.

ShowInnerProperties has two constraints: the attribute must be of type UObject\*, and it cannot be a container.

Note that Struct properties are displayed with internal properties by default, thus there is no need to set ShowInnerProperties for them.

### What is the difference between it and EditInlineNew?

This effect is similar to what is achieved by setting EditInlineNew on a UCLASS and setting Instanced on its object reference property. The difference lies in the fact that setting EditInlineNew on a UCLASS allows the object property reference to create objects within the property panel, while setting Instanced on a UPROPERTY automatically adds the EditInline meta attribute, resulting in the same effect of displaying internal properties. Therefore, fundamentally, what is similar to ShowInnerProperties is the EditInline meta attribute. However, EditInline has an additional layer of functionality, supporting object containers, whereas ShowInnerProperties only supports individual object reference properties.

## Test Code:

```

USTRUCT(BlueprintType)
struct FMyPropertyInner
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 StructInnerInt = 123;
}

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite)
FString StructInnerString;
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_InnerSub :public UObject
{
GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 ObjectInnerInt = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString ObjectInnerString;
};

UCLASS(BlueprintType, EditInlineNew)
class INSIDER_API UMyProperty_InnerSub_EditInlineNew :public UObject
{
GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 ObjectInnerInt = 123;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString ObjectInnerString;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Inner :public UDataAsset
{
GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FMyPropertyInner InnerStruct;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ShowInnerProperties))
    FMyPropertyInner InnerStruct_ShowInnerProperties;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UMyProperty_InnerSub* InnerObject;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ShowInnerProperties))
    UMyProperty_InnerSub* InnerObject_ShowInnerProperties;

    //((Category = MyProperty_Inner, EditInline = , ModuleRelativePath =
    Property/MyProperty_Inner.h)
    //CPF>Edit | CPF_BlueprintVisible | CPF_ZeroConstructor | CPF_NoDestructor |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditInline))
    UMyProperty_InnerSub* InnerObject_EditInline;

    //((Category = MyProperty_Inner, EditInline = true, ModuleRelativePath =
    Property/MyProperty_Inner.h)
    //CPF>Edit | CPF_BlueprintVisible | CPF_ExportObject | CPF_ZeroConstructor |
    CPF_InstancedReference | CPF_NoDestructor | CPF_PersistentInstance |
    CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced)
    UMyProperty_InnerSub* InnerObject_Instanced;
}

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite)
UMyProperty_InnerSub_EditInlineNew* InnerObject_EditInlineNewClass;
UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (EditInline))
UMyProperty_InnerSub_EditInlineNew*
InnerObject_EditInlineNewClass_EditInline;
UPROPERTY(EditAnywhere, BlueprintReadWrite, Instanced)
UMyProperty_InnerSub_EditInlineNew* InnerObject_EditInlineNewClass_Instanced;

public:
    UFUNCTION(CallInEditor)
    void ClearInnerObject();
    UFUNCTION(CallInEditor)
    void InitInnerObject();
};

void UMyProperty_Inner::ClearInnerObject()
{
    InnerObject = nullptr;
    InnerObject_ShowInnerProperties = nullptr;
    InnerObject>EditInline = nullptr;
    InnerObject>Instanced = nullptr;

    InnerObject>EditInlineNewClass = nullptr;
    InnerObject>EditInlineNewClass>EditInline = nullptr;
    InnerObject>EditInlineNewClass>Instanced = nullptr;

    Modify();

    FPropertyEditorModule& PropertyEditorModule =
FModuleManager::GetModuleChecked<FPropertyEditorModule>("PropertyEditor");
    PropertyEditorModule.NotifyCustomizationModuleChanged();
}

void UMyProperty_Inner::InitInnerObject()
{
    InnerObject = NewObject<UMyProperty_InnerSub>(this);
    InnerObject>ShowInnerProperties = NewObject<UMyProperty_InnerSub>(this);
    InnerObject>EditInline = NewObject<UMyProperty_InnerSub>(this);
    InnerObject>Instanced = NewObject<UMyProperty_InnerSub>(this);

    InnerObject>EditInlineNewClass =
NewObject<UMyProperty_InnerSub>EditInlineNew>(this);
    InnerObject>EditInlineNewClass>EditInline =
NewObject<UMyProperty_InnerSub>EditInlineNew>(this);
    //InnerObject>EditInlineNewClass>Instanced =
NewObject<UMyProperty_InnerSub>EditInlineNew>(this);

    Modify();

    FPropertyEditorModule& PropertyEditorModule =
FModuleManager::GetModuleChecked<FPropertyEditorModule>("PropertyEditor");
    PropertyEditorModule.NotifyCustomizationModuleChanged();
}

```

# Blueprint Effect:

The screenshot illustrates the Blueprint Effect in a game editor, showing how inner properties are expanded by default. The interface is organized into sections:

- My Property Inner**:
  - Inner Struct**: Contains "Struct Inner Int" (value: 123) and "Struct Inner String".
  - Inner Struct Show Inner Properties**: Contains "Struct Inner Int" (value: 123) and "Struct Inner String".
  - Inner Object**: Shows a cube icon and a dropdown menu set to "MyProperty\_InnerSub\_0".
  - Inner Object Show Inner Properties**: Shows a cube icon and a dropdown menu set to "MyProperty\_InnerSub\_1".
- Inner Object Edit Inline**: Shows a dropdown menu set to "My Property Inner Sub".
- Inner Object Instanced**: Shows a dropdown menu set to "My Property Inner Sub".
- Inner Object Edit Inline New Class**: Shows a cube icon and a dropdown menu set to "MyProperty\_InnerSub\_EditInlineNew\_0".
- Inner Object Edit Inline New Class Edit Inline**: Shows a dropdown menu set to "My Property Inner Sub Edit Inline New".
- Inner Object Edit Inline New Class Instanced**: Shows a dropdown menu set to "None".

Red boxes highlight the expanded properties in the first four sections. Red arrows point to the expanded properties in the fifth section and to the search dropdown in the eighth section.

Observations include:

- Struct properties are displayed with internal properties expanded by default

- UMyProperty\_InnerSub\* InnerObject\_ShowInnerProperties; supports expanding properties when marked with ShowInnerProperties
- UMyProperty\_InnerSub\* with both EditInline and Instanced also supports expanding internal properties, and their meta attributes are consistent, both showing EditInline = true
- Only UMyProperty\_InnerSub\_EditInlineNew\* InnerObject\_EditInlineNewClass; does not support expanding the reference attribute when EditInlineNew is set on the UCLASS, indicating that setting EditInlineNew on the class has no effect.
- However, we also observed that the InnerObject\_EditInlineNewClass\_Instanced setting supports direct object creation due to the presence of EditInlineNew on its class. Conversely, InnerObject\_Instanced does not support direct object creation because its class UMyProperty\_InnerSub lacks EditInlineNew and will not appear in the selection box.

## Extended Example:

---

Searching the source code reveals that UChildActorComponent::ChildActorTemplate also includes ShowInnerProperties, which is a typical application that allows direct editing of ChildActor property data within the familiar details panel.

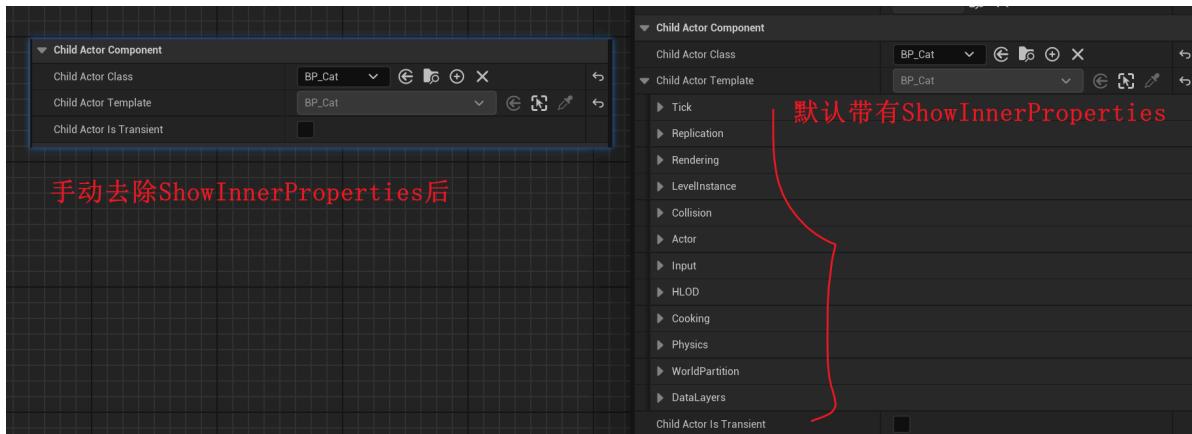
But if we remove ShowInnerProperties, we can compare the effects before and after:

```
class UChildActorComponent : public USceneComponent
{
    UPROPERTY(VisibleDefaultsOnly, DuplicateTransient,
Category=ChildActorComponent, meta=(ShowInnerProperties))
    TObjectPtr<AActor> ChildActorTemplate;
}

void UMyProperty_Inner::RemoveActorMeta()
{
    FProperty* prop = UChildActorComponent::StaticClass()-
>FindPropertyByName(TEXT("ChildActorTemplate"));
    prop->RemoveMetaData(TEXT("ShowInnerProperties"));
}

void UMyProperty_Inner::AddActorMeta()
{
    FProperty* prop = UChildActorComponent::StaticClass()-
>FindPropertyByName(TEXT("ChildActorTemplate"));
    prop->SetMetaData(TEXT("ShowInnerProperties"), TEXT(""));
}
```

# Comparison Effect:



It can be seen that after removing `ShowInnerProperties`, the `ChildActorTemplate` property reverts to a standard object reference, and we can no longer directly edit the object's internal properties.

## Principle:

The most typical example in the source code is `ChildActorTemplate`, which allows the internal properties to be displayed directly.

```
class UChildActorComponent : public USceneComponent
{
    UPROPERTY(VisibleDefaultsOnly, DuplicateTransient,
Category=ChildActorComponent, meta=(ShowInnerProperties))
    TObjectPtr<AActor> ChildActorTemplate;
}
```

The source code for the function:

```
void FPropertyNode::InitNode(const FPropertyParams& InitParams)
{
    const bool bIsObjectOrInterface = CastField<FOBJECTPROPERTYBASE>
(MyProperty) || CastField<FINTERFACEPROPERTY>(MyProperty);
    // we are EditInlineNew if this property has the flag, or if inside a
    // container that has the flag.
    bIsEditInlineNew = GotReadAddresses && bIsObjectOrInterface &&
!MyProperty->HasMetaData(NAME_NoEditInline) &&
    (MyProperty->HasMetaData(NAME_EditInline) || (bIsInsideContainer &&
OwnerProperty->HasMetaData(NAME_EditInline)));
    bShowInnerObjectProperties = bIsObjectOrInterface && MyProperty-
>HasMetaData(NAME_ShowInnerProperties);

    if (bIsEditInlineNew)
    {
        SetNodeFlags(EPropertyNodeFlags::EditInlineNew, true);
    }
    else if (bShowInnerObjectProperties)
    {
        SetNodeFlags(EPropertyNodeFlags::ShowInnerObjectProperties, true);
    }
}
```

```

void FItemPropertyNode::InitExpansionFlags(void)
{
    FProperty* MyProperty = GetProperty();

    if (TSharedPtr<FPropertyNode>& valueNode = GetOrCreateOptionalValueNode())
    {
        // This is a set optional, so check its SetValue instead.
        MyProperty = valueNode->GetProperty();
    }

    bool bExpandableType = CastField<FStructProperty>(MyProperty)
        || (CastField<FArrayProperty>(MyProperty) || CastField<FSetProperty>(MyProperty) || CastField<FMapProperty>(MyProperty));
}

if (bExpandableType
    || HasNodeFlags(EPropertyNodeFlags::EditInlineNew)
    || HasNodeFlags(EPropertyNodeFlags::ShowInnerObjectProperties)
    || (MyProperty->ArrayDim > 1 && ArrayIndex == -1))
{
    SetNodeFlags(EPropertyNodeFlags::CanBeExpanded, true);
}
}

void FPropertyNode::RebuildChildren()
{
    if (HasNodeFlags(EPropertyNodeFlags::CanBeExpanded) && (childNodes.Num() == 0))
    {
        InitChildNodes();
        if (ExpandedPropertyItemSet.size() > 0)
        {
            FPropertyNodeUtils::SetExpandedItems(ThisAsSharedRef,
ExpandedPropertyItemSet);
        }
    }
}

```

Note that the condition for bShowInnerObjectProperties here is bIsObjectOrInterface and the presence of a meta attribute, so this feature only applies to object references. If EPropertyNodeFlags::ShowInnerObjectProperties is detected, then EPropertyNodeFlags::CanBeExpanded is set, ultimately allowing the properties of the object to be expanded.

## ShowOnlyInnerProperties

---

- **Function Description:** Promotes the internal properties of a struct attribute to the immediate parent level for direct display, rather than grouping them under an expandable parent structure as is the default behavior
- **Usage Location:** UPROPERTY
- **Metadata Type:** bool
- **Restriction Type:** FStruct attribute
- **Related Items:** ShowInnerProperties
- **Commonality:** ★★★

The internal properties of struct attributes are directly elevated one level to be displayed, rather than being nested under a collapsible parent structure by default.

## Test Code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite)
FMyPropertyInner InnerStruct;

UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ShowOnlyInnerProperties))
FMyPropertyInner InnerStruct_ShowOnlyInnerProperties;
```

## Effect Comparison:



You can observe that the internal properties of InnerStruct\_ShowOnlyInnerProperties are directly shown at the object's current level, whereas the properties of InnerStruct are categorized under a structure name that can be expanded.

## Principle:

When encountering an FStructProperty, the system will check ShowOnlyInnerProperties to determine whether to create an expandable Category or to display the internal properties directly. With ShowOnlyInnerProperties set, it will recursively iterate through its internal properties.

```
void DetailLayoutHelpers::updateSinglePropertyMapRecursive(FPropertyNode& InNode,
FName CurCategory, FComplexPropertyNode* CurobjectNode, FUpdatePropertyMapArgs&
InUpdateArgs)
{
    static FName ShowOnlyInners("ShowOnlyInnerProperties");
    // Whether or not to push out struct properties to their own categories
    // or show them inside an expandable struct
    // This recursively applies for any nested structs that have the
    ShowOnlyInners metadata
    const bool bPushoutStructProps = bIsStruct && !bIsCustomizedStruct &&
    Property->HasMetaData(ShowOnlyInners);

    if (bRecurseIntoChildren || LocalUpdateFavoriteSystemOnly)
    {
        // Built in struct properties or children of arrays
        updateSinglePropertyMapRecursive(ChildNode, CurCategory,
        CurobjectNode, ChildArgs);
    }
}
```

```

void FObjectPropertyNode::GetCategoryProperties(const TSet<UClass*>&
ClassesToConsider, const FProperty* CurrentProperty, bool
bShouldShowDisableEditOnInstance, bool bShouldShowHiddenProperties,
const TSet< FName>& CategoriesFromBlueprints, TSet< FName>&
CategoriesFromProperties, TArray< FName>& SortedCategories)
{
    if (CurrentProperty->HasMetaData(NAME_ShowOnlyInnerProperties))
    {
        const FStructProperty* StructProperty = CastField<const
FStructProperty>(CurrentProperty);
        if (StructProperty)
        {
            for (TFieldIterator< FProperty> It(StructProperty->Struct);
It; ++It)
            {
                GetCategoryProperties(ClassesToConsider, *It,
bShouldShowDisableEditOnInstance, bShouldShowHiddenProperties,
CategoriesFromBlueprints, CategoriesFromProperties, SortedCategories);
            }
        }
    }
}

```

## FullyExpand

- **Usage Location:** UPROPERTY
- **Metadata Type:** boolean
- **Associated Items:** ShowInnerProperties

However, the underlying code utilizing this Meta was not found.

Searches within the source code revealed multiple instances of its application, yet no underlying principle code was identified.

```

/** The options that are available on the node. */
UPROPERTY(EditAnywhere, Instanced, Category = "Options", meta=
>ShowInnerProperties, FullyExpand="true")
TObjectPtr<UMovieGraphValueContainer> SelectOptions;

/** The currently selected option. */
UPROPERTY(EditAnywhere, Instanced, Category = "Options", meta=
>ShowInnerProperties, FullyExpand="true")
TObjectPtr<UMovieGraphValueContainer> SelectedOption;

```

## CollapsableChildProperties

- **Function Description:** A newly added meta in the TextureGraph module, utilized for collapsing the internal properties of a structure.
- **Usage Location:** UPROPERTY
- **Metadata Type:** bool

- **Restriction Type:** For use within the TextureGraph plugin only
- **Associated Items:** ShowInnerProperties
- **Commonality:** 0

Added as a new meta in the TextureGraph module, this is used to collapse the internal properties of a structure.

## Source Code:

```
bool STG_GraphPinOutputSettings::CollapsibleChildProperties() const
{
    FProperty* Property = GetPinProperty();
    bool Collapsible = false;
    // check if there is a display name defined for the property, we use that as
    // the Pin Name
    if (Property && Property->HasMetaData("CollapsibleChildProperties"))
    {
        Collapsible = true;
    }
    return Collapsible;
}

UPROPERTY(EditAnywhere, Category = NoCategory, meta = (TGType = "TG_Input",
    CollapsibleChildProperties, ShowOnlyInnerProperties, FullyExpand,
    NoResetToDefault, PinDisplayName = "Settings" ) )
FTG_OutputSettings OutputSettings;
```

## Untracked

- **Function Description:** Allows properties of the soft object reference types TSoftObjectPtr and FSoftObjectPath to not be tracked and recorded for asset dependencies.
- **Usage Location:** UPROPERTY
- **Engine Module:** Object Property
- **Metadata Type:** bool
- **Restricted Types:** TSoftObjectPtr, FSoftObjectPath
- **Commonality:** ★

Properties of the soft object reference types TSoftObjectPtr and FSoftObjectPath will not track asset dependencies.

By default, soft object references on properties will also create asset dependency references, although during the loading process, they do not load other soft reference objects as hard references do. However, since the reference relationship still exists, these soft reference objects are checked during cooking or asset redirection to ensure they are also cooked or handled properly.

If you want to record "references" to some assets on properties for later use without creating actual asset dependency references, you can use untracked. This is not commonly used in the source code and is a relatively rare scenario.

The difference from the transient tag is that transient properties are not serialized during serialization and their values are lost after saving with **ctrl+S** and restarting the editor. Transient properties do not create asset dependency relationships nor are they serialized to save values. Untracked properties are serialized to save values but do not create asset dependency relationships.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyProperty_Soft :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TSoftObjectPtr<UStaticMesh> MyStaticMesh;

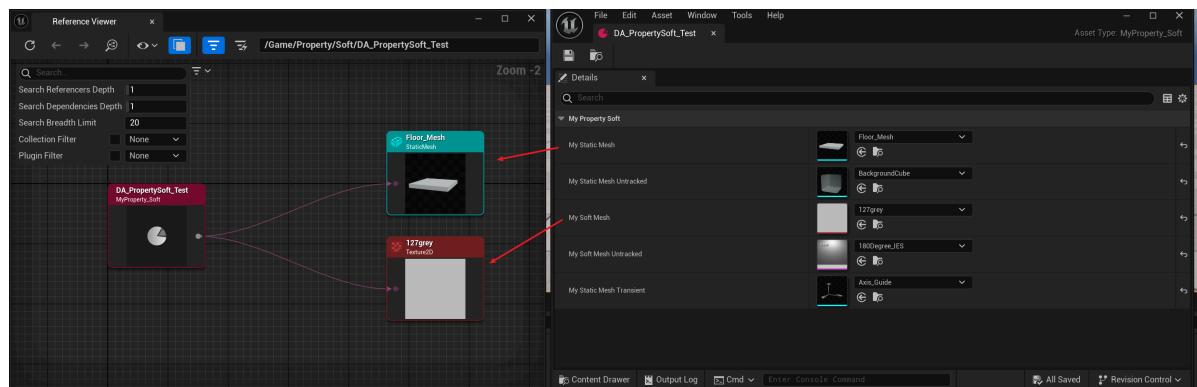
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (Untracked))
    TSoftObjectPtr<UStaticMesh> MyStaticMeshUntracked;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FSoftObjectPath MySoftMesh;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (Untracked))
    FSoftObjectPath MySoftMeshUntracked;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient)
    TSoftObjectPtr<UStaticMesh> MyStaticMeshTransient;
};
```

## Blueprint Effect:

In the Blueprint, create a UMyProperty\_Soft DataAsset and set its property values. When checking the referenced resources, you will find that the Untracked property does not include the set assets in the dependency relationship. Of course, transient properties are also not included in the dependency relationship.



## Principle:

Untracked metadata is set to the `ESoftObjectPathCollectType::NeverCollect` option. Further searching will reveal that `FSoftObjectPath` with `NeverCollect` does not include the asset package in the dependency calculation, thus not adding it to the upackage Import table. There are multiple instances in the source code with similar judgments using `NeverCollect`.

```

bool FSoftObjectPathThreadContext::GetSerializationOptions(FName& OutPackageName,
FName& OutPropertyName, ESoftObjectPathCollectType& OutCollectType,
ESoftObjectPathSerializeType& OutSerializeType, FArchive* Archive) const
{
#if WITH_EDITOR
    bEditorOnly = Archive->IsEditorOnlyPropertyOnTheStack();

    static FName UntrackedName = TEXT("Untracked");
    if (CurrentProperty && CurrentProperty->GetOwnerProperty()-
>HasMetaData(UntrackedName))
    {
        // Property has the Untracked metadata, so set to never collect
        // references if it's higher than NeverCollect
        CurrentCollectType =
FMath::Min(ESoftObjectPathCollectType::NeverCollect, CurrentCollectType);
    }
#endif
}

FArchive& FImportExportCollector::operator<<(FSoftObjectPath& Value)
{
    FName CurrentPackage;
    FNamePropertyName;
    ESoftObjectPathCollectType CollectType;
    ESoftObjectPathSerializeType SerializeType;
    FSoftObjectPathThreadContext& ThreadContext =
FSoftObjectPathThreadContext::Get();
    ThreadContext.GetSerializationOptions(CurrentPackage, PropertyName,
CollectType, SerializeType, this);

    if (CollectType != ESoftObjectPathCollectType::NeverCollect && CollectType != ESoftObjectPathCollectType::NonPackage)
    {
        FName PackageName = Value.GetLongPackageName();
        if (PackageName != RootPackageName && !PackageName.IsNone())
        {
            AddImport(value, CollectType);
        }
    }
    return *this;
}

```

## HideAssetPicker

---

- **Function Description:** Hides the selection list of the AssetPicker on an Object type pin
- **Usage Location:** UFUNCTION
- **Engine Module:** Object Property
- **Metadata Type:** strings = "a, b, c"
- **Restriction Type:** UObject\*
- **Commonliness:** ★★

Hides the selection list of the AssetPicker on Object type pins. This is particularly useful when we want to pass an Object reference directly without allowing the user to select other assets within the engine. Since the Asset type is a subclass of Object, the parameter for an Object reference type is referred to as HideAssetPicker.

There is no usage found in the source code, but this feature is functional.

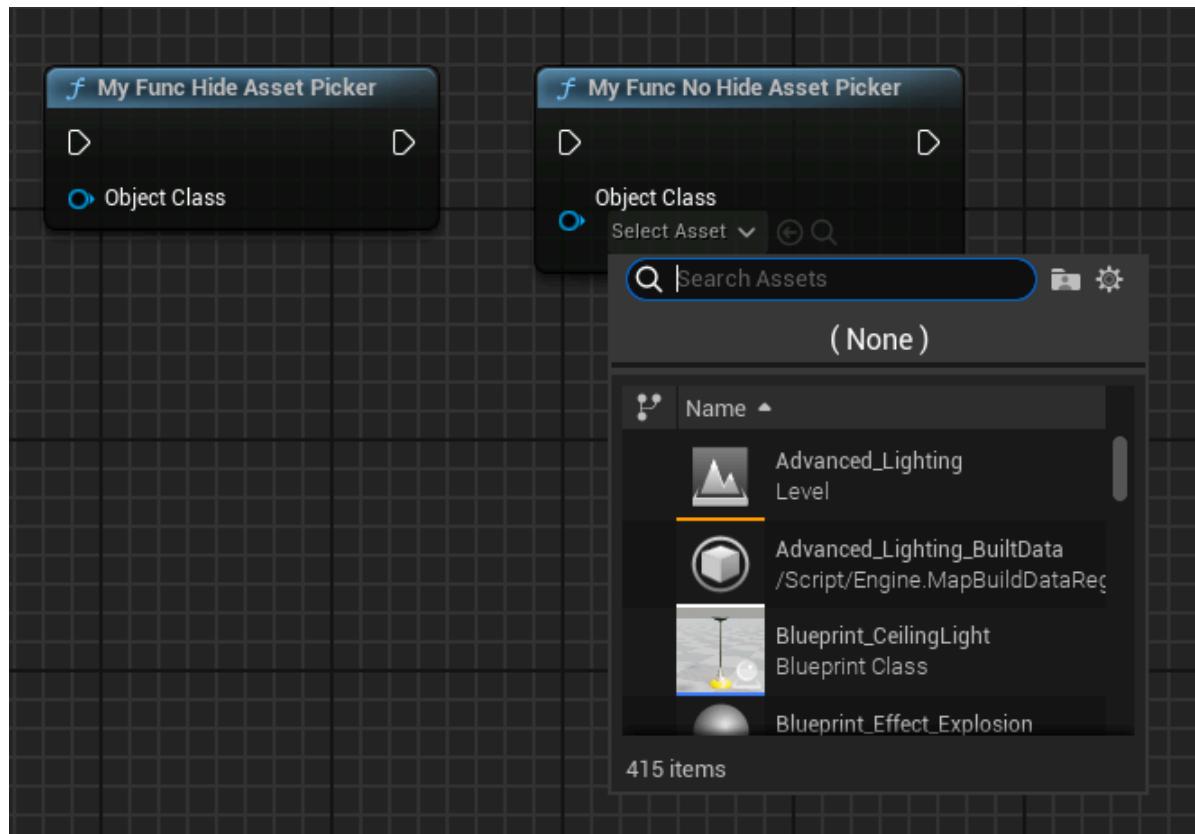
## Test Code:

```
UFUNCTION(BlueprintCallable)
static void MyFunc_NoHideAssetPicker(UObject* ObjectClass) {}

UFUNCTION(BlueprintCallable, meta = (HideAssetPicker = "ObjectClass"))
static void MyFunc_HideAssetPicker(UObject* ObjectClass) {}
```

## Blueprint Effect:

In the default scenario, MyFunc\_NoHideAssetPicker can pop up a selection list. However, MyFunc\_HideAssetPicker remains hidden.



## Principle:

The logic for determining whether a function pin can open the AssetPicker is as follows:

- It must be an object type
- If it is a UActorComponent, it will not be displayed
- If it is an Actor type, it must be within a level blueprint and the Actor must be marked as placeable to be displayed.
- If the parameter is explicitly specified with HideAssetPicker, it will also not be displayed.

```

bool UEdGraphSchema_K2::ShouldShowAssetPickerForPin(UEdGraphPin* Pin) const
{
    bool bShow = true;
    if (Pin->PinType.PinCategory == PC_Object)
    {
        UClass* ObjectClass = Cast<UClass>(Pin-
>PinType.PinSubCategoryObject.Get());
        if (ObjectClass)
        {
            // Don't show literal buttons for component type objects
            bShow = !ObjectClass->IsChildOf(UActorComponent::StaticClass());

            if (bShow && ObjectClass->IsChildOf(AActor::StaticClass()))
            {
                // only show the picker for Actor classes if the class is
                placeable and we are in the level script
                bShow = !ObjectClass->HasAllClassFlags(CLASS_NotPlaceable)
                    &&
FBlueprintEditorUtils::IsLevelScriptBlueprint(FBlueprintEditorUtils::FindBlueprint
tForNode(Pin->GetOwningNode()));
            }

            if (bShow)
            {
                if (UK2Node_CallFunction* CallFunctionNode =
Cast<UK2Node_CallFunction>(Pin->GetOwningNode()))
                {
                    if (UFunction* FunctionRef = CallFunctionNode-
>GetTargetFunction())
                    {
                        const UEdGraphPin* WorldContextPin = CallFunctionNode-
>FindPin(FunctionRef->GetMetaData(FBlueprintMetadata::MD_WorldContext));
                        bShow = (WorldContextPin != Pin);

                        // Check if we have explicitly marked this pin as hiding
                        the asset picker
                        const FString& HideAssetPickerMetaData = FunctionRef-
>GetMetaData(FBlueprintMetadata::MD_HideAssetPicker);
                        if (!HideAssetPickerMetaData.IsEmpty())
                        {
                            TArray< FString> PinNames;
                            HideAssetPickerMetaData.ParseIntoArray(PinNames,
TEXT("",), true);
                            const FString PinName = Pin->GetName();
                            for (FString& ParamNameToHide : PinNames)
                            {
                                ParamNameToHide.TrimStartAndEndInline();
                                if (ParamNameToHide == PinName)
                                {
                                    bShow = false;
                                    break;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        else if (Cast<UK2Node_CreateDelegate>( Pin->GetOwningNode()))
    {
        bShow = false;
    }
}
return bShow;
}

```

## AssetBundles

- **Function description:** Indicates which assets the attribute refers to belong to which AssetBundles.
- **Use location:** UPROPERTY
- **Engine module:** Object Property
- **Metadata type:** strings = "a, b, c"
- **Restriction type:** UPrimaryDataAsset internal FSoftObjectPtr, FSoftObjectPath
- **Related items:** IncludeAssetBundles
- **Commonly used:** ★★★

Used for SoftObjectPtr or SoftObjectPath properties within UPrimaryDataAsset, indicating which AssetBundles the assets they refer to belong to.

To understand the purpose of this, you need to first understand some basic concepts:

- PrimaryAsset refers to items that can be manually loaded/released in the game, including level files (.umap) and some game-related objects, such as characters or items in backpacks. As the name implies, main assets are the primary root assets in the game, with a large number of other assets under their reference tree. On the other hand, we often actively load or release these main assets, such as loading levels, loading monster characters, or loading drop items. However, we generally do not load assets like materials, textures, or sounds directly, as most of them are referenced by main assets. When we load main assets, these secondary assets are loaded automatically.
- SecondaryAsset refers to other assets, such as textures and sounds. These types of assets are automatically loaded based on the PrimaryAsset. We generally do not need to manage secondary assets, as they are automatically loaded by the main assets based on the reference relationship.
- AssetBundle can be referred to as an asset package, which is essentially a list of assets. We name each asset package to distinguish it, such as UI and Game, which also categorizes assets. Here, we do not differentiate between PrimaryAsset and SecondaryAsset, as the categorization is based on usage rather than loading method. The role of AssetBundle is that when loading a PrimaryAsset, the PrimaryAsset itself may reference other SecondaryAssets with different purposes. We can group these SecondaryAssets into different AssetBundles, allowing us to control the loading of these SecondaryAssets more precisely by providing an additional AssetBundleName when loading the PrimaryAsset.

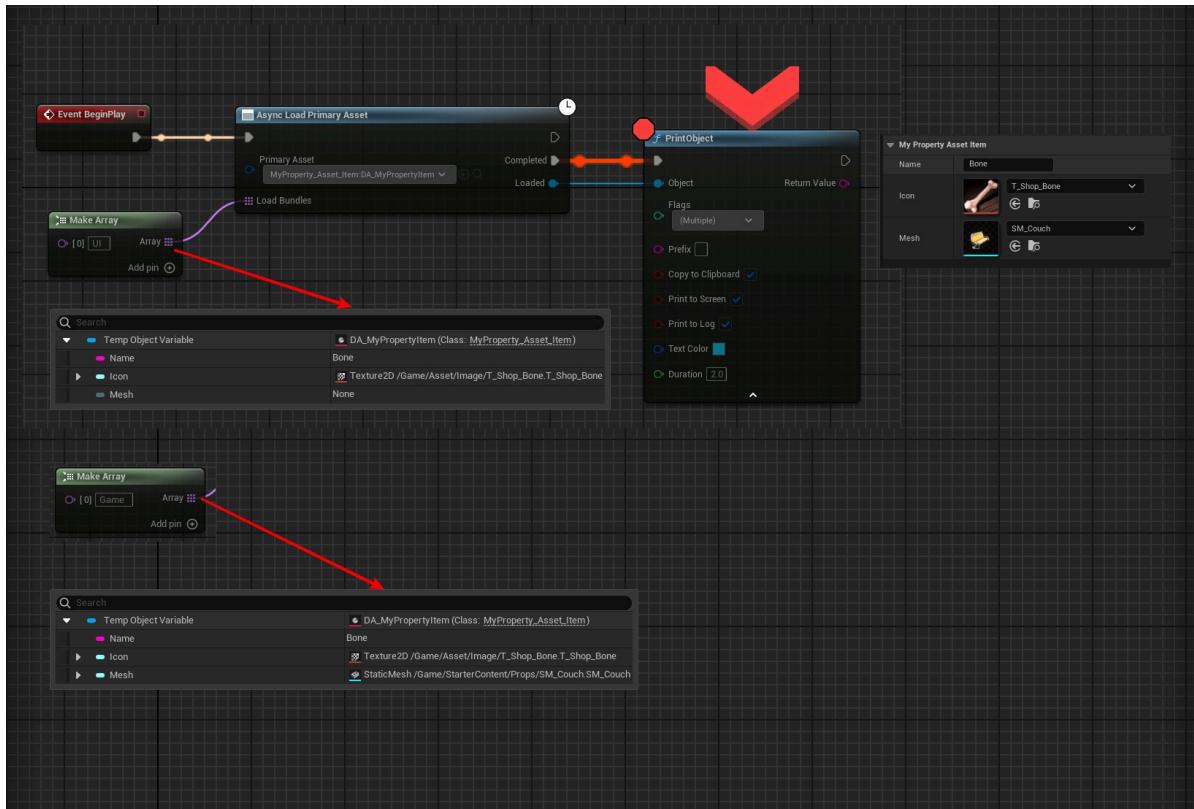
- For a PrimaryAsset, the specified AssetBundle's asset property must be a soft reference; otherwise, it will be loaded regardless if it is a hard reference. Soft reference assets need to be loaded manually by default. By attaching an AssetBundle, the soft reference asset can be loaded incidentally when the PrimaryAsset is loaded.

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Asset_Item :public UPrimaryDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
        FString Name;
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta=(AssetBundles="UI,Game"))
        TSoftObjectPtr<UTexture2D> Icon;
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta=(AssetBundles="Game"))
        TSoftObjectPtr<UStaticMesh> Mesh;
public:
    virtual FPrimaryAssetId GetPrimaryAssetId() const override;
};
```

## Test Results:

- First, we defined the UMyProperty\_Asset\_Item asset in BP and configured the corresponding reference objects. As shown in the figure, there is an icon for UI and Game use, and there is a Mesh specifically for Game use. Imagine that in some interfaces, we only need the icon of the item.
- Then, when loading the PrimaryAsset, we can specify the name of LoadBundles to load only specific Bundles. As shown in the figure below.
- When specifying the Bundle as UI, you can see that the Mesh is not loaded.
- When specifying the Bundle as Game, you can see that both the Icon and Mesh are loaded.
- Note that when testing in the editor, if the Mesh has been loaded previously, it may still reside in the editor's memory. Therefore, even when using the name UI, you may still find that the Mesh can be referenced.



## Principle:

First, UPrimaryDataAsset contains an AssetBundleData that stores information about the currently referenced AssetBundle. This information is saved during the PreSave process in the editor environment and is analyzed and mapped in UAssetManager::InitializeAssetBundlesFromMetadata\_Recursive. Subsequently, when UAssetManager loads the PrimaryAsset, it internally calls ChangeBundleStateForPrimaryAssets, checks the AssetBundle, and adds other additional assets to be loaded together to PathsToLoad, thus completing the logic of loading together.

```

void UAssetManager::InitializeAssetBundlesFromMetadata_Recursive(const UStruct* Struct, const void* StructValue, FAssetBundleData& AssetBundle, FName DebugName, TSet<const void*>& AllVisitedStructValues) const
{
    static FName AssetBundlesName = TEXT("AssetBundles");
    static FName IncludeAssetBundlesName = TEXT("IncludeAssetBundles");

    //Based on the current object's value, search for the value of properties
    //that own AssetBundles. Finally, AddBundleAsset, where BundleName is the set
    //value, and FoundRef is the asset path of the referenced object
    TSet<FName> BundleSet;
    TArray<const FProperty*> PropertyChain;
    It.GetPropertyChain(PropertyChain);

    for (const FProperty* PropertyToSearch : PropertyChain)
    {
        if (PropertyToSearch->HasMetaData(AssetBundlesName))
        {
            TSet<FName> LocalBundleSet;
            TArray< FString > BundleList;
            const FString& BundleString = PropertyToSearch-
>GetMetaData(AssetBundlesName);
    }
}

```

```

        BundleString.ParseToArrayWS(BundleList, TEXT(","));
    }

    for (const FString& BundleNameString : BundleList)
    {
        LocalBundleSet.Add(FName(*BundleNameString));
    }

    // If Set is empty, initialize. Otherwise intersect
    if (BundleSet.Num() == 0)
    {
        BundleSet = LocalBundleSet;
    }
    else
    {
        BundleSet = BundleSet.Intersect(LocalBundleSet);
    }
}

for (const FName& BundleName : BundleSet)
{
    AssetBundle.AddBundleAsset(BundleName,
    FoundRef.GetAssetPath());
}

#endif WITH_EDITORONLY_DATA
void UPrimaryDataAsset::UpdateAssetBundleData()
{
    // By default parse the metadata
    if (UAssetManager::IsInitialized())
    {
        AssetBundleData.Reset();
        UAssetManager::Get().InitializeAssetBundlesFromMetadata(this,
        AssetBundleData);
    }
}

void UPrimaryDataAsset::PreSave(FObjectPreSaveContext ObjectSaveContext)
{
    Super::PreSave(ObjectSaveContext);

    UpdateAssetBundleData();

    if (UAssetManager::IsInitialized())
    {
        // Bundles may have changed, refresh
        UAssetManager::Get().RefreshAssetData(this);
    }
}
#endif

void UPrimaryDataAsset::PostLoad()
{
    Super::PostLoad();
}

```

```

#ifndef WITH_EDITORONLY_DATA
    FAssetBundleData OldData = AssetBundleData;
}

UpdateAssetBundleData();

if (UAssetManager::IsInitialized() && OldData != AssetBundleData)
{
    // Bundles changed, refresh
    UAssetManager::Get().RefreshAssetData(this);
}
#endif
}

//when loading an asset, if there is an FAssetBundleEntry, it is added to
PathsToLoad together
TSharedPtr<FStreamableHandle>
UAssetManager::ChangeBundleStateForPrimaryAssets(const TArray<FPrimaryAssetId>&
AssetsToChange, const TArray<FName>& AddBundles, const TArray<FName>&
RemoveBundles, bool bRemoveAllBundles, FStreamableDelegate DelegateToCall,
TAsyncLoadPriority Priority)
{
    if (!AssetPath.IsNull())
    {
        // Dynamic types can have no base asset path
        PathsToLoad.Add(AssetPath);
    }

    for (const FName& BundleName : NewBundleState)
    {
        FAssetBundleEntry Entry = GetAssetBundleEntry(PrimaryAssetId,
        BundleName);

        if (Entry.IsValid())
        {
            for (const FTopLevelAssetPath & Path : Entry.AssetPaths)
            {
                PathsToLoad.AddUnique(FSoftObjectPath(Path));
            }
        }
        else
        {
            UE_LOG(LogAssetManager, Verbose,
TEXT("ChangeBundleStateForPrimaryAssets: No assets for bundle %s::%s"),
*PrimaryAssetId.ToString(), *BundleName.ToString());
        }
    }
}

```

Reference documentation: [https://dev.epicgames.com/documentation/en-us/unreal-engine/asset-management-in-unreal-engine?application\\_version=5.4](https://dev.epicgames.com/documentation/en-us/unreal-engine/asset-management-in-unreal-engine?application_version=5.4)

# IncludeAssetBundles

---

- **Function Description:** Used for sub-object attributes of UPrimaryDataAsset to specify that recursion should continue into the sub-object to detect AssetBundle data.
- **Usage Location:** UPROPERTY
- **Engine Module:** Object Property
- **Metadata Type:** string="abc"
- **Restriction Type:** ObjectPtr within UPrimaryDataAsset
- **Associated Items:** AssetBundles
- **Commonly Used:** ★★

Used for sub-object attributes of UPrimaryDataAsset to specify that recursion should continue into the sub-object to detect AssetBundle data.

By doing this, the FSoftObjectPtr or FSoftObjectPath properties within the object, marked with AssetBundle data, will be parsed and added to the AssetBundleData of UPrimaryDataAsset.

- By default, InitializeAssetBundlesFromMetadata\_Recursive will only analyze the attributes at the level of UPrimaryDataAsset itself, such as the Icon and Mesh properties below.
- However, if there is another level of nesting, meaning UPrimaryDataAsset has a child object, UMyProperty\_Asset\_ChildObject, and UMyProperty\_Asset\_ChildObject contains FSoftObjectPath, it is desired that it be considered part of the AssetBundles and loaded simultaneously with UPrimaryDataAsset. In this case, the engine needs to be informed to continue analyzing this child object.
- Note that UMyProperty\_Asset\_ChildObject is always a hard reference using TObjectPtr, and this object will be loaded when UMyProperty\_Asset\_Item is loaded. Therefore, UMyProperty\_Asset\_ChildObject will always be loaded. However, the ChildIcon within UMyProperty\_Asset\_ChildObject uses TSofObjectPtr, which is a soft reference, and thus relies on the AssetBundle mechanism to be loaded.

## Test Code:

---

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Asset_ChildObject :public UDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (AssetBundles = "Client"))
    TSofObjectPtr<UTexture2D> ChildIcon;
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Asset_Item :public UPrimaryDataAsset
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString Name;
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (AssetBundles =
"UI,Game"))
}
```

```

TSoftObjectPtr<UTexture2D> Icon;
UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (AssetBundles = "Game"))
TSoftObjectPtr<UStaticMesh> Mesh;

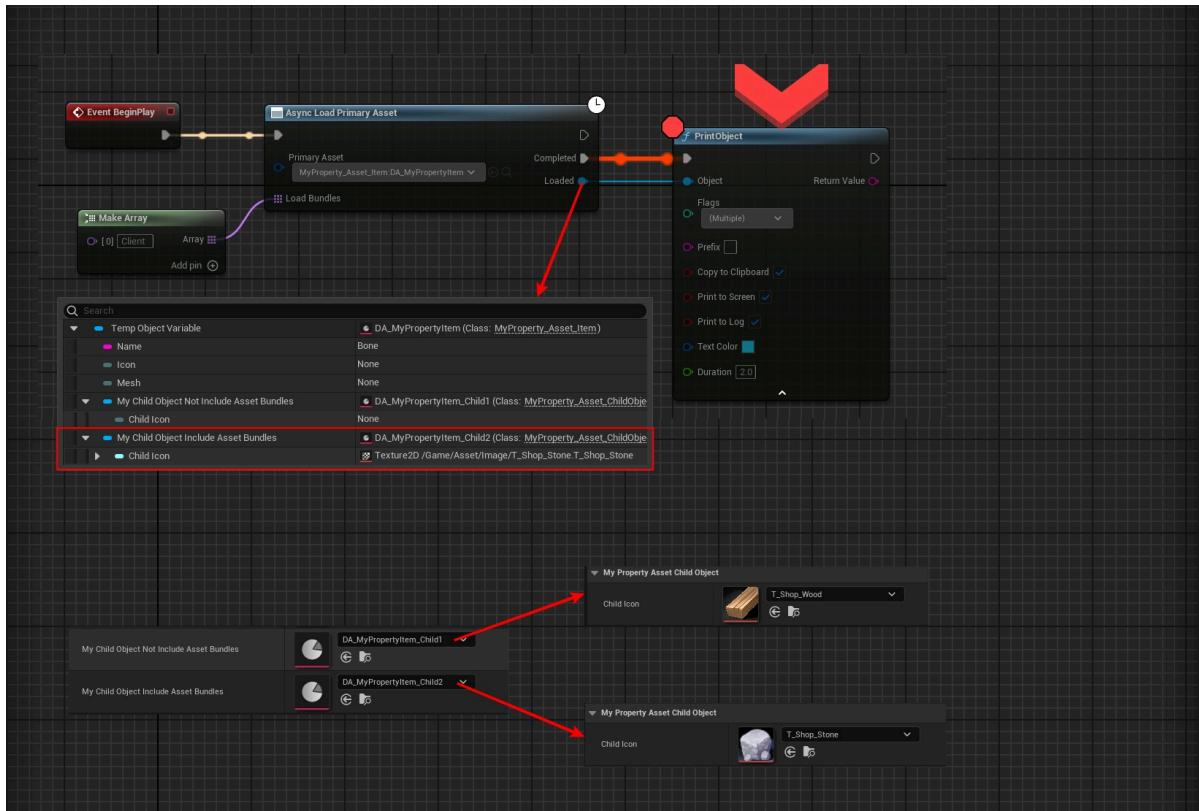
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    TObjectPtr<UMyProperty_Asset_ChildObject>
    MyChildObject_NotIncludeAssetBundles;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (IncludeAssetBundles))
    TObjectPtr<UMyProperty_Asset_ChildObject> MyChildObject_IncludeAssetBundles;
public:
    virtual FPrimaryAssetId GetPrimaryAssetId() const override;
};

```

## Test Results:

In the lower part of the configured data graph, two images are set. However, after loading LoadPrimaryAsset, only the ChildIcon within MyChildObject\_IncludeAssetBundles is loaded.



If analyzing the AssetBunbleData of UMyProperty\_Asset\_Item, it will be found that the Client contains only the path to the second Stone image. This is because only the second image is analyzed and included.

```

{
    BundleName = "Client";
    BundleAssets =
    {
        {
            AssetPath =
            {
                PackageName = "/Game/Asset/Image/T_Shop_Stone";
                AssetName = "T_Shop_Stone";
            }
        }
    }
}

```

```

        };
        SubPathString = "";
    };
},
AssetPaths =
{
{
    PackageName = "/Game/Asset/Image/T_Shop_Stone";
    AssetName = "T_Shop_Stone";
};
},
};

```

## Principle:

For properties under UPrimaryDataAsset that are object properties, recursion will only continue downwards when IncludeAssetBundles is present.

```

void UAssetManager::InitializeAssetBundlesFromMetadata_Recursive(const UStruct* Struct, const void* StructValue, FAssetBundleData& AssetBundle, FName DebugName, TSet<const void*>& AllVisitedStructValues) const
{
    static FName AssetBundlesName = TEXT("AssetBundles");
    static FName IncludeAssetBundlesName = TEXT("IncludeAssetBundles");

    //Based on the current object's value, search for the value of properties
    //that have AssetBundles, and finally, AddBundleAsset, where BundleName is the set
    //value, and FoundRef is the asset path of the referenced object
    else if (const FObjectProperty* ObjectProperty = CastField<FObjectProperty>(Property))
    {
        if (ObjectProperty->PropertyFlags & CPF_InstancedReference || ObjectProperty->GetOwnerProperty()->HasMetaData(IncludeAssetBundlesName))
        {
            const Uobject* Object = ObjectProperty-
>GetObjectPropertyValue(PropertyValue);
            if (Object != nullptr)
            {
                InitializeAssetBundlesFromMetadata_Recursive(Object->GetClass(),
Object, AssetBundle, Object->GetFName(), AllVisitedStructValues);
            }
        }
    }
}

```

## MustBeLevelActor

- **Usage Location:** UPROPERTY
- **Engine Module:** Object Property
- **Metadata Type:** bool

Indicates that this must be an Actor within the scene, as opposed to a LevelScriptActor.

The trigger occurs when the currently selected Actor is being highlighted with the arrow cursor.

## Found in the Source Code:

```
if (FObjectPropertyBase* ObjectProperty = CastField< FObjectPropertyBase>
(Property))
{
    ObjectClass = ObjectProperty->PropertyClass;
    bMustBeLevelActor = ObjectProperty->GetOwnerProperty()-
>GetBoolMetaData(TEXT("MustBeLevelActor"));
    RequiredInterface = ObjectProperty->GetOwnerProperty()-
>GetClassMetaData(TEXT("MustImplement"));
}
```

## ExposeFunctionCategories

- **Function description:** Specifies that certain functions within directories of the class to which this Object attribute belongs can be directly exposed on this class.
- **Use location:** UPROPERTY
- **Engine module:** Object Property
- **Metadata type:** strings = "a, b, c"
- **Restriction type:** UObject\*
- **Commonly used:** ★★★

Functions within certain directories of the class to which the Object attribute belongs can be directly exposed on this class.

Initially, it may be difficult to grasp the meaning and function, but this is actually a feature for convenience. For instance, if class A defines some functions and class B has an instance of A, if you want to call A's functions on the B object, you would normally have to manually drag out B.ObjA and then drag out the functions within it. We aim to allow, within the current context of B, certain functions from A to be exposed more conveniently for calling within B.

Effectively, the engine automates the step of dragging out B.ObjA for you. If you wish to call additional functions from A, you can still manually drag and right-click on B.ObjA to expose more functions.

This kind of application is also common in the source code. A convenient example is the following source code snippet, which allows functions defined in the directories specified by ExposeFunctionCategories in USkeletalMeshComponent to be directly exposed on ASkeletalMeshActor.

```
UCLASS(ClassGroup=ISkeletalMeshes, Blueprintable, Componentwrapperclass,
ConversionRoot, meta=(ChildCanTick), MinimalAPI)
class ASkeletalMeshActor : public AActor
{
private:
    UPROPERTY(Category = SkeletalMeshActor, VisibleAnywhere, BlueprintReadOnly,
meta = (ExposeFunctionCategories =
"Mesh,Components|skeletalMesh,Animation,Physics", AllowPrivateAccess = "true"))
    TobjectPtr<class USkeletalMeshComponent> SkeletalMeshComponent;
}
```

## Test Code:

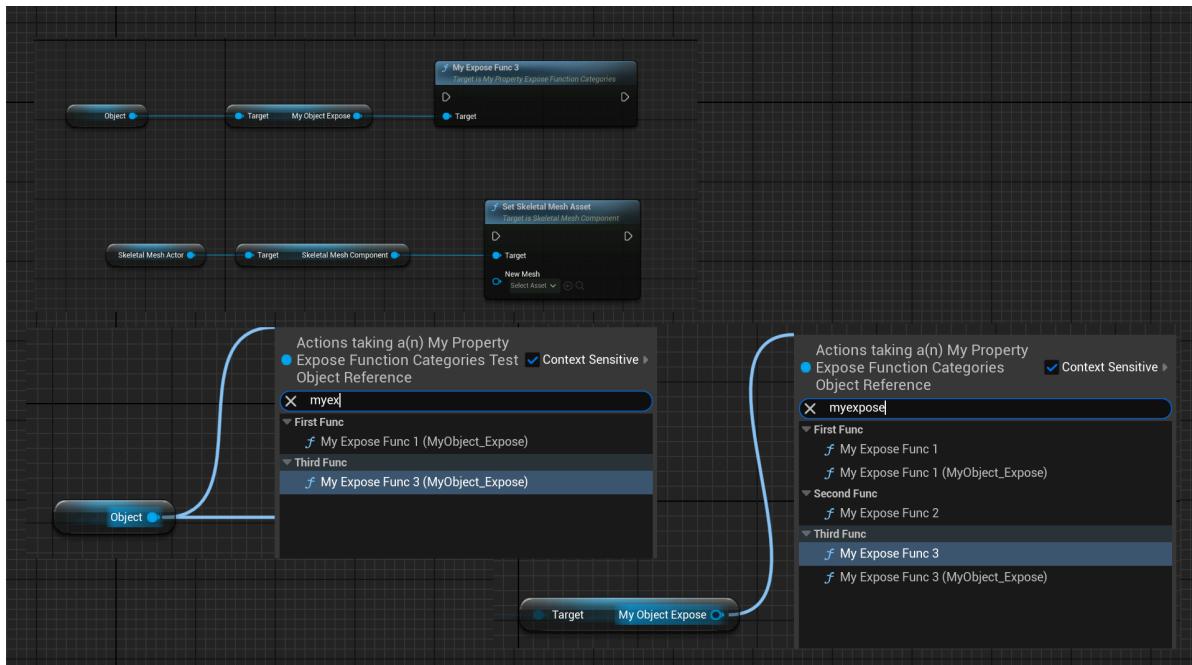
```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_ExposeFunctionCategories :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, Category = "FirstFunc")
    void MyExposeFunc1() {}
    UFUNCTION(BlueprintCallable, Category = "SecondFunc")
    void MyExposeFunc2() {}
    UFUNCTION(BlueprintCallable, Category = "ThirdFunc")
    void MyExposeFunc3() {}
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_ExposeFunctionCategories_Test :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadOnly, meta = (ExposeFunctionCategories =
"FirstFunc,ThirdFunc"))
    UMyProperty_ExposeFunctionCategories* MyObject_Expose;
};
```

## Test Results:

You can see that on an Object of type UMyProperty\_ExposeFunctionCategories\_Test, typing "MyExposeFunc" directly will pop up the functions in the "FirstFunc" and "ThirdFunc" directories, but "MyExposeFunc2" will not appear directly because it has not been exposed.

If you right-click and drag on an internal object like MyObject\_Expose, you will see all the functions defined internally. Note that although there are two entries for MyExposeFunc1, the function called is actually the same, with no practical difference.



## Principle:

During the construction of the right-click menu in the blueprint, it is determined whether an action should be filtered out. The IsUnexposedMemberAction function is used to decide if a function should be filtered. The general logic involves retrieving the attributes corresponding to the function, such as in the UMyProperty\_ExposeFunctionCategories\_Test object, where recursively, three functions will be considered for testing. These three functions (MyExposeFunc 1, 2, 3) each have their own Category but correspond to the same MyObject\_Expose property, so their AllExposedCategories value is the array we defined, "FirstFunc, ThirdFunc". In the end, only two functions pass the test, and thus only the functions 1 and 3 are displayed.

```
static bool
BlueprintActionMenuUtilsImpl::IsUnexposedMemberAction(FBlueprintActionFilter
const& Filter, FBlueprintActionInfo& BlueprintAction)
{
    bool bIsFilteredout = false;

    if (UFunction const* Function = BlueprintAction.GetAssociatedFunction())
    {
        TArray< FString> AllExposedCategories;
        for (FBindingObject Binding : BlueprintAction.GetBindings())
        {
            if (FProperty* Property = Binding.Get< FProperty>())
            {
                const FString& ExposedCategoryMetadata = Property-
>GetMetaData(FBlueprintMetadata::MD_ExposeFunctionCategories);
                if (ExposedCategoryMetadata.IsEmpty())
                {
                    continue;
                }

                TArray< FString> PropertyExposedCategories;
                ExposedCategoryMetadata.ParseIntoArray(PropertyExposedCategories,
TEXT(","), true);
                AllExposedCategories.Append(PropertyExposedCategories);
            }
        }
    }

    const FString& FunctionCategory = Function-
>GetMetaData(FBlueprintMetadata::MD_FunctionCategory);
    bIsFilteredout = !AllExposedCategories.Contains(FunctionCategory);
}
return bIsFilteredout;
}
```

## ContentDir

- **Function description:** Use the UE style to select Content and its subdirectories.
- **Use location:** UPROPERTY
- **Engine module:** Path Property
- **Metadata type:** bool

- **Restriction type:** FDirectoryPath
- **Associated items:** RelativePath, RelativeToGameContentDir
- **Commonly used:** ★★★

Use the UE style to select Content and its subdirectories.

By default, selecting a directory will prompt the Windows default directory selection dialog, as FDirectoryPath can indeed be used to select any directory within the Windows system (which may be what your project requires). However, if you specifically want to select a directory under UE Content, specifying ContentDir will allow UE to pop up a dedicated UE directory selection dialog, which is more convenient and reduces errors.

When using FDirectoryPath, ContentDir and LongPackageName are equivalent.

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Path :public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DirectoryPath)
    FDirectoryPath MyDirectory_Default;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DirectoryPath, meta = (ContentDir))
        FDirectoryPath MyDirectory_ContentDir;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DirectoryPath, meta = (LongPackageName))
        FDirectoryPath MyDirectory_LongPackageName;

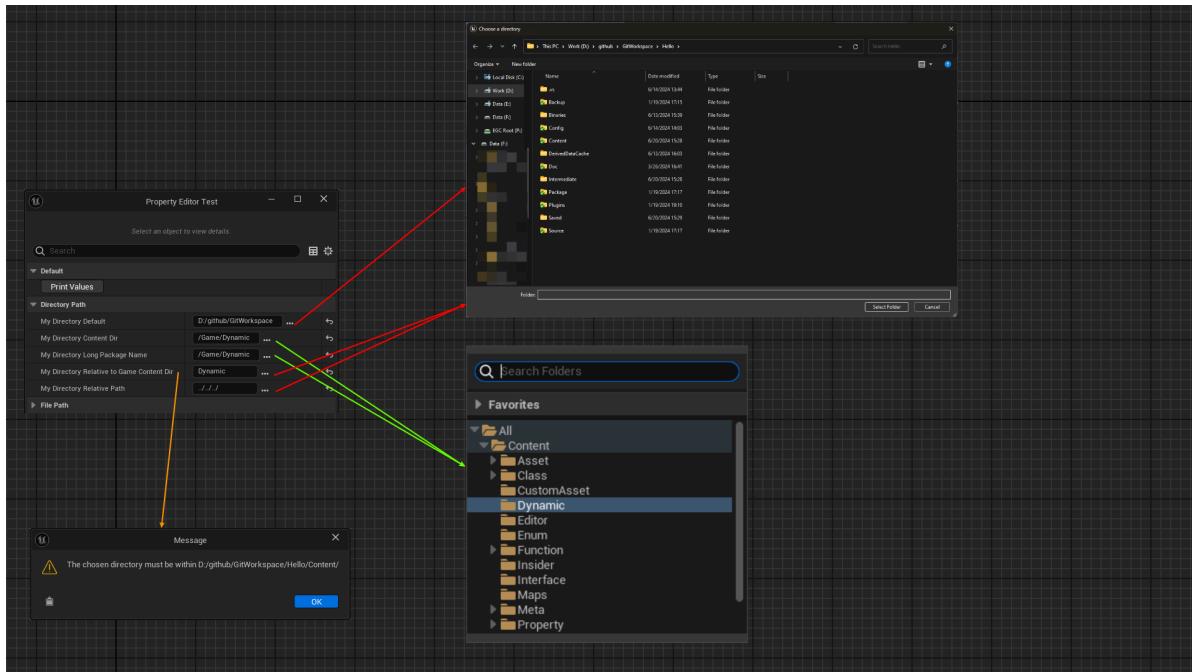
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DirectoryPath, meta = (RelativeToGameContentDir))
        FDirectoryPath MyDirectory_RelativeToGameContentDir;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = DirectoryPath, meta = (RelativePath))
        FDirectoryPath MyDirectory_RelativePath;
};
```

## Test Results:

- By default, MyDirectory\_Default will open the system dialog, allowing any directory to be selected.
- MyDirectory\_ContentDir and MyDirectory\_LongPackageName will, as shown in the figure, pop up a UE-style dialog to select the directory.
- MyDirectory\_RelativeToGameContentDir and MyDirectory\_RelativePath will both pop up the system dialog. The difference is that the final directory for MyDirectory\_RelativeToGameContentDir will be restricted to the Content directory (if another directory is selected, an error warning will appear), resulting in a relative path.

MyDirectory\_RelativePath also results in a relative path but allows any directory to be selected.



## Principle:

The editing of FDirectoryPath is customized with FDirectoryPathStructCustomization. As the code shows, if there is a ContentDir or LongPackageName, OnPickContent is used to select the directory. Internally, ContentBrowserModule.Get().CreatePathPicker(PathPickerConfig) is used to create a specialized directory selection menu.

Otherwise, the OnPickDirectory branch is taken, and DesktopPlatform->OpenDirectoryDialog is used to open the system dialog.

This can also be observed in the source code:

bRelativeToGameContentDir causes Directory.RightChopInline(AbsoluteGameContentDir.Len(), EAllowShrinking::No); to cut off the left part of the Content path.

bUseRelativePath triggers Directory = IFileManager::Get().ConvertToRelativePath(\*Directory); to convert the path to a relative path.

```
/** Structure for directory paths that are displayed in the editor with a picker
UI. */
USTRUCT(BlueprintType)
struct FDirectoryPath
{
    GENERATED_BODY()

    /**
     * The path to the directory.
     */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = Path)
    FString Path;
};
```

```

RegisterCustomPropertyTypeLayout("DirectoryPath",
FOnGetPropertyTypeCustomizationInstance::CreateStatic(&FDirectoryPathStructCustomization::MakeInstance));
RegisterCustomPropertyTypeLayout("FilePath",
FOnGetPropertyTypeCustomizationInstance::CreateStatic(&FFFilePathStructCustomization::MakeInstance));

void FDirectoryPathStructCustomization::CustomizeHeader(
TSharedRef<IPropertyHandle> StructPropertyHandle, class FDetailWidgetRow&
HeaderRow, IPropertyTypeCustomizationUtils& StructCustomizationUtils )
{
    TSharedPtr<IPropertyHandle> PathProperty = StructPropertyHandle-
>GetChildHandle("Path");

    const bool bRelativeToGameContentDir = StructPropertyHandle->HasMetaData(
TEXT("RelativeToGameContentDir") );
    const bool bUseRelativePath = StructPropertyHandle->HasMetaData(
TEXT("RelativePath") );
    const bool bContentDir = StructPropertyHandle->HasMetaData(
TEXT("ContentDir") ) || StructPropertyHandle-
>HasMetaData(TEXT("LongPackageName"));

    AbsoluteGameContentDir =
FPaths::ConvertRelativePathToFull(FPaths::ProjectContentDir());

    if(bContentDir)
    {
        PickerWidget = SAssignNew(PickerButton, SButton)
            .ButtonStyle( FAppStyle::Get(), "HoverHintOnly" )
            .ToolTipText( LOCTEXT( "FolderComboToolTipText", "Choose a content
directory" ) )
            .OnClicked( this, &FDirectoryPathStructCustomization::OnPickContent,
PathProperty.ToSharedRef() )
            .ContentPadding(2.0f)
            .ForegroundColor( FSlateColor::UseForeground() )
            .IsFocusable(false)
            .IsEnabled(this, &FDirectoryPathStructCustomization::IsBrowseEnabled,
StructPropertyHandle)
        [
            SNew(SImage)
            .Image(FAppStyle::GetBrush("Propertywindow.Button_Ellipsis"))
            .ColorAndOpacity(FSlateColor::UseForeground())
        ];
    }
    else
    {
        PickerWidget = SAssignNew(BrowseButton, SButton)
            .ButtonStyle( FAppStyle::Get(), "HoverHintOnly" )
            .ToolTipText( LOCTEXT( "FolderButtonToolTipText", "Choose a directory
from this computer" ) )
            .OnClicked( this, &FDirectoryPathStructCustomization::OnPickDirectory,
PathProperty.ToSharedRef(), bRelativeToGameContentDir, bUseRelativePath )
            .ContentPadding( 2.0f )
            .ForegroundColor( FSlateColor::UseForeground() )
            .IsFocusable( false )
    }
}

```

```

        .IsEnabled( this, &FDirectoryPathStructCustomization::IsBrowseEnabled,
StructPropertyHandle )
    [
        SNew( SImage )
        .Image( FAppStyle::GetBrush("Propertywindow.Button_Ellipsis") )
        .ColorAndOpacity( FSlateColor::UseForeground() )
    ];
}
}

FReply
FDirectoryPathStructCustomization::OnPickContent(TSharedRef<IPropertyHandle>
PropertyHandle)
{
    FContentBrowserModule& ContentBrowserModule =
FModuleManager::LoadModuleChecked<FCContentBrowserModule>("ContentBrowser");
    FPathPickerConfig PathPickerConfig;
    PropertyHandle->GetValue(PathPickerConfig.DefaultPath);
    PathPickerConfig.bAllowContextMenu = false;
    PathPickerConfig.OnPathSelected = FOnPathSelected::CreateSP(this,
&FDirectoryPathStructCustomization::OnPathPicked, PropertyHandle);

    FMenuBuilder MenuBuilder(true, NULL);
    MenuBuilder.AddWidget(SNew(SBox)
        .WidthOverride(300.0f)
        .HeightOverride(300.0f)
    [
        ContentBrowserModule.Get().CreatePathPicker(PathPickerConfig)
    ], FText());
}

PickerMenu = FSlateApplication::Get().PushMenu(PickerButton.TosharedRef(),
FWidgetPath(),
MenuBuilder.Makewidget(),
FSlateApplication::Get().GetCursorPos(),
FPopupTransitionEffect(FPopupTransitionEffect::ContextMenu)
);

return FReply::Handled();
}

FReply
FDirectoryPathStructCustomization::OnPickDirectory(TSharedRef<IPropertyHandle>
PropertyHandle, const bool bRelativeToGameContentDir, const bool
buseRelativePath) const
{
    FString Directory;
    IDesktopPlatform* DesktopPlatform = FDesktopPlatformModule::Get();
    if (DesktopPlatform)
    {

        TSharedPtr<SWindow> ParentWindow =
FSlateApplication::Get().FindWidgetWindow(BrowseButton.TosharedRef());
        void* ParentwindowHandle = (ParentWindow.IsValid() && Parentwindow-
>GetNativeWindow().IsValid()) ? ParentWindow->GetNativeWindow()-
>GetOSWindowHandle() : nullptr;
    }
}
```

```

FString StartDirectory =
FEditorDirectories::Get().GetLastDirectory(ELastDirectory::GENERIC_IMPORT);
    if (bRelativeToGameContentDir && !IsValidPath(StartDirectory,
bRelativeToGameContentDir))
    {
        StartDirectory = AbsoluteGameContentDir;
    }

    // Loop until; a) the user cancels (OpenDirectoryDialog returns false),
or, b) the chosen path is valid (IsValidPath returns true)
    for (;;)
    {
        if (DesktopPlatform->OpenDirectoryDialog(ParentWindowHandle,
LOCTEXT("FolderDialogTitle", "Choose a directory").ToString(), StartDirectory,
Directory))
        {
            FText FailureReason;
            if (IsValidPath(Directory, bRelativeToGameContentDir,
&FailureReason))
            {
                FEditorDirectories::Get().SetLastDirectory(ELastDirectory::GENERIC_IMPORT,
Directory);

                if (bRelativeToGameContentDir)
                {
                    Directory.RightChopInline(AbsoluteGameContentDir.Len(),
EAllowShrinking::No);
                }
                else if (bUseRelativePath)
                {
                    Directory =
IFileManager::Get().ConvertToRelativePath(*Directory);
                }

                PropertyHandle->SetValue(Directory);
            }
            else
            {
                StartDirectory = Directory;
                FMessageDialog::Open(EAppMsgType::Ok, FailureReason);
                continue;
            }
        }
        break;
    }

    return FReply::Handled();
}

```

# RelativePath

- **Function Description:** Ensures that the result of the system directory selection dialog is the relative path to the current executing exe.
- **Usage Location:** UPROPERTY
- **Engine Module:** Path Property
- **Metadata Type:** bool
- **Restriction Type:** FDirectoryPath
- **Associated Item:** ContentDir

The current directory is: D:\github\GitWorkspace\Hello\Binaries\Win64, which is the working directory of the exe. Any selected directory will be converted to a relative path.

```
Directory = IFileManager::Get().ConvertToRelativePath(*Directory);
```

# RelativeToGameContentDir

- **Function Description:** Ensures that the result of the system directory selection dialog is a relative path to the Content directory.
- **Usage Location:** UPROPERTY
- **Engine Module:** Path Property
- **Metadata Type:** bool
- **Restriction Type:** FDirectoryPath
- **Associated Item:** ContentDir

Restricts the directory selection result to the Content directory or its subdirectories; otherwise, an error message will be displayed. The logic for conversion involves truncating the left part of the Content path.

```
Directory.RightChopInline(AbsoluteGameContentDir.Len(), EAllowShrinking::No);
```

# RelativeToGameDir

- **Function Description:** If the result of the system directory selection dialog is a subdirectory of the Project, it is converted to a relative path; otherwise, an absolute path is returned.
- **Usage Location:** UPROPERTY
- **Engine Module:** Path Property
- **Metadata Type:** bool
- **Restriction Type:** FFilePath
- **Commonly Used:** ★★★

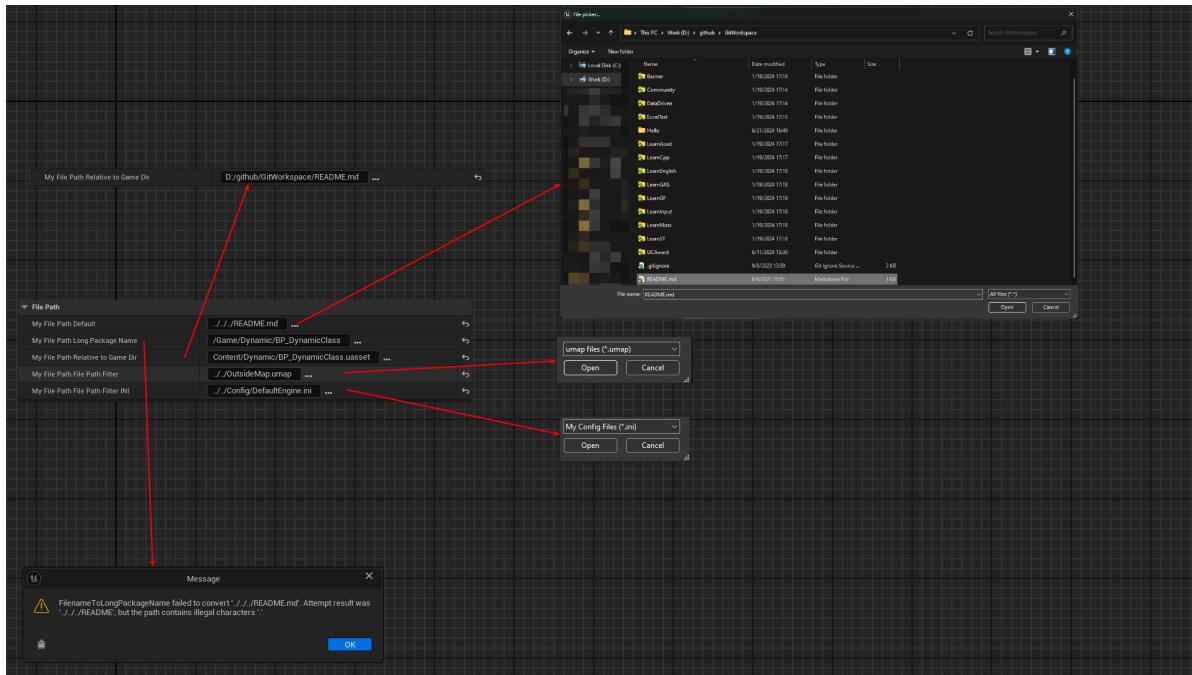
If the result of the system directory selection dialog is a subdirectory of the Project, it is converted to a relative path; otherwise, an absolute path is returned.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath)  
    FFilePath MyFilePath_Default;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath, meta =  
(LongPackageName))  
    FFilePath MyFilePath_LongPackageName;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath, meta =  
(RelativeToGameDir))  
    FFilePath MyFilePath_RelativeToGameDir;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath, meta =  
(FilePathFilter = "umap"))  
    FFilePath MyFilePath_FilePathFilter;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath, meta =  
(FilePathFilter = "My Config Files|*.ini"))  
    FFilePath MyFilePath_FilePathFilter_INI;
```

## Test Results:

- The FFilePath dialog box that pops up is the default file selection dialog of the Windows system.
- MyFilePath\_Default, pops up the default system file selection dialog, allowing selection of any file from any path.
- MyFilePath\_LongPackageName, restricts the selection to assets under Content; otherwise, an error message will appear. The resulting path will be converted to a long package name format like /Game/ObjectPath.
- MyFilePath\_RelativeToGameDir, if the selection result is a subfile of the Project directory (the directory where the uproject file is located), the returned path will be relative; otherwise, an absolute path is returned directly.
- MyFilePath\_FilePathFilter, allows selection of files with a specified extension from any directory. The example in the code is umap, which restricts selection to level files only.
- MyFilePath\_FilePathFilter\_INI, demonstrates the selection of only INI files. The FilePathFilter allows us to write our own filtering method using the format "Description | \*.extension," following the same rules as the Windows file selection dialog, and multiple extensions can be specified simultaneously.



## Principle:

The processing logic for FilePathFilter, bLongPackageName, and bRelativeToGameDir can be seen in the code below.

- The FileTypeFilter sets the extension in SFilePathPicker
- bLongPackageName triggers the conversion of the path using TryConvertFilenameToLongPackageName.
- bRelativeToGameDir triggers the transformation into a relative path using AbsolutePickedPath.RightChop(AbsoluteProjectDir.Len());

```
USTRUCT(BlueprintType)
struct FFilePath
{
    GENERATED_BODY()

    /**
     * The path to the file.
     */
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = FilePath)
    FString FilePath;
};

void FFilePathStructCustomization::CustomizeHeader( TSharedRef<IPropertyHandle>
StructPropertyHandle, class FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils )
{
    const FString& MetaData = StructPropertyHandle-
>GetMetaData(TEXT("FilePathFilter"));
    bLongPackageName = StructPropertyHandle-
>HasMetaData(TEXT("LongPackageName"));
    bRelativeToGameDir = StructPropertyHandle-
>HasMetaData(TEXT("RelativeToGameDir"));

    if (MetaData.IsEmpty())

```

```

    {
        FileTypeFilter = TEXT("All files (*.*)|*.*");
    }
    else
    {
        if (MetaData.Contains(TEXT(" | "))) // If MetaData follows the
        Description|ExtensionList format, use it as is
        {
            FileTypeFilter = MetaData;
        }
        else
        {
            FileTypeFilter = FString::Printf(TEXT("%s files (*.%s)|*.%s"),
*MetaData, *MetaData, *MetaData);
        }
    }
}

void FFilePathStructCustomization::HandleFilePathPickerPathPicked( const FString&
PickedPath )
{
    FString FinalPath = PickedPath;
    if (bLongPackageName)
    {
        FString LongPackageName;
        FString StringFailureReason;
        if (FPackageName::TryConvertFilenameToLongPackageName(PickedPath,
LongPackageName, &StringFailureReason) == false)
        {
            FMessageDialog::Open(EAppMsgType::Ok,
FText::FromString(StringFailureReason));
        }
        FinalPath = LongPackageName;
    }
    else if (bRelativeToGameDir && !PickedPath.IsEmpty())
    {
        //A filepath under the project directory will be made relative to the
        project directory
        //Otherwise, the absolute path will be returned unless it doesn't exist,
        the current path will
        //be kept. This can happen if it's already relative to project dir
        (tabbing when selected)

        const FString ProjectDir = FPaths::ProjectDir();
        const FString AbsoluteProjectDir =
FPaths::ConvertRelativePathToFull(ProjectDir);
        const FString AbsolutePickedPath =
FPaths::ConvertRelativePathToFull(PickedPath);

        //Verify if absolute path to file exists. If it was already relative to
        content directory
        //the absolute will be to binaries and will possibly be garbage
        if (FPaths::FileExists(AbsolutePickedPath))
        {
            //If file is part of the project dir, chop the project dir part
            //otherwise, use the absolute path
        }
    }
}

```

```

    if (AbsolutePickedPath.StartsWith(AbsoluteProjectDir))
    {
        FinalPath =
AbsolutePickedPath.RightChop(AbsoluteProjectDir.Len());
    }
    else
    {
        FinalPath = AbsolutePickedPath;
    }
}
else
{
    //If absolute file doesn't exist, it might already be relative to
    project dir
    //If not, then it might be a manual entry, so keep it untouched
    either way
    FinalPath = PickedPath;
}
}

PathStringProperty->SetValue(FinalPath);
FEditorDirectories::Get().SetLastDirectory(ELastDirectory::GENERIC_OPEN,
FPaths::GetPath(PickedPath));
}

```

## LongPackageName

---

- **Function description:** Utilizes the UE style for selecting Content and its subdirectories, or converts file paths into long package names.
- **Usage location:** UPROPERTY
- **Engine module:** Path Property
- **Metadata type:** bool
- **Restricted types:** FDirectoryPath, FFilePath
- **Commonly used:** ★★★

LongPackageName can be applied to both FDirectoryPath and FFilePath, both of which restrict the selection to within the Content directory.

When applied to FDirectoryPath, it restricts directories to Content or its subdirectories.

When used with FFilePath, it limits the selection to assets within Content and ultimately converts the selected file path into an object path of the form "/Game/ObjectPath".

## FilePathFilter

---

- **Function description:** Specifies the file extension for the file selector, adhering to the format specifications of the system dialog box. Multiple extensions can be entered.
- **Usage location:** UPROPERTY
- **Engine module:** Path Property
- **Metadata type:** string="abc"
- **Restriction type:** FFilePath

- **Commonly used:** ★★☆

Usual extensions include ".umap" and ".uasset". However, it also supports custom filtering by using the "description | \*.extension" format, following the same rules as the Windows file selection dialog, and multiple extensions can be specified simultaneously.

## HidePin

---

- **Function Description:** Used in function calls to specify parameter names to be hidden, as well as to hide return values. Multiple parameters can be hidden
- **Usage Location:** UFUNCTION
- **Engine Module:** Pin
- **Metadata Type:** strings="a, b, c"
- **Associated Items:** InternalUseParam
- **Commonality:** ★★

In the source code, it is often found to be in a contract comparison with DefaultToSelf. It is both hidden and has a default value. The combined effect is to restrict a static function call to an external calling environment object directly as a parameter.

The value of HidePin is frequently WorldContextObject

```
meta = (HidePin = "WorldContextObject", DefaultToSelf = "WorldContextObject")
```

## C++ Test Code:

---

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_HidePinTest :public AActor
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    int MyFunc_Default(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (HidePin = "options"))
    int MyFunc_HidePin(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options,comment"))
    int MyFunc_HidePin2(FName name, float value, FString options,FString comment)
    { return 0; }

    UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options"))
    int MyFunc_InternalUseParam(FName name, float value, FString options) {
    return 0; }

    UFUNCTION(BlueprintCallable, meta = (HidePin = "ReturnValue"))
    int MyFunc_HideReturn(FName name, float value, FString options, FString&
otherReturn) { return 0; }

public:
    UFUNCTION(BlueprintPure)
    int MyPure_Default(FName name, float value, FString options) { return 0; }
```

```

UFUNCTION(BlueprintPure, meta = (HidePin = "options"))
int MyPure_HidePin(FName name, float value, FString options) { return 0; }

UFUNCTION(BlueprintPure, meta = (InternalUseParam = "options"))
int MyPure_InternalUseParam(FName name, float value, FString options) {
return 0; }

UFUNCTION(BlueprintPure, meta = (HidePin = "ReturnValue"))
int MyPure_HideReturn(FName name, float value, FString options, FString&
otherReturn) { return 0; }

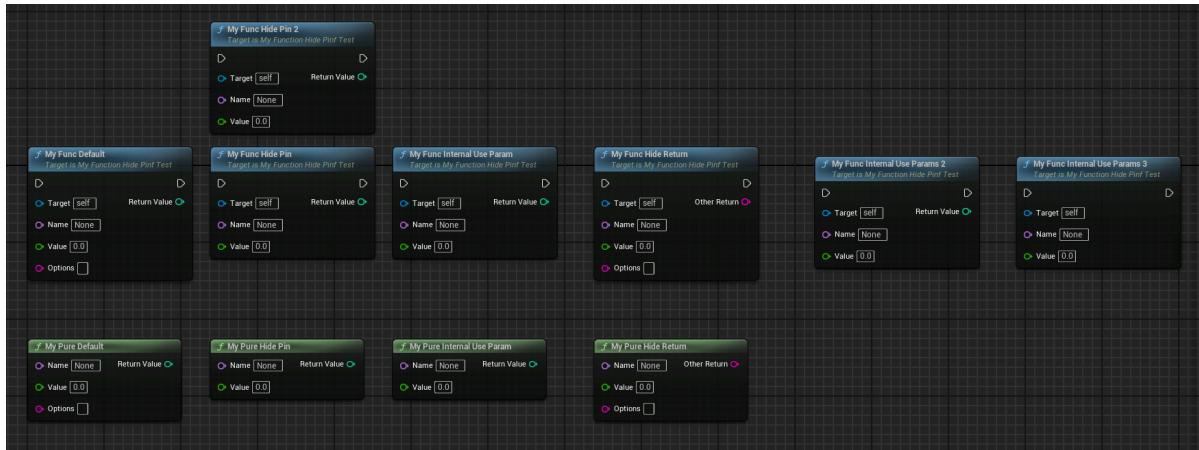
public:
UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options,comment"))
int MyFunc_InternalUseParams2(FName name, float value, FString
options,FString comment) { return 0; }

UFUNCTION(BlueprintCallable, meta = (InternalUseParam =
"options,comment,ReturnValue"))
int MyFunc_InternalUseParams3(FName name, float value, FString
options,FString comment) { return 0; }

};


```

## Blueprint Test Results:



It can be observed that both BlueprintCallable and BlueprintPure are actually applicable. Additionally, ReturnValue is the default name for the return value, which can also be hidden using this feature.

## Principle:

When searching through the source code, the only usage found is:

The following types of Pins are automatically hidden:

- LatentInfo="ParameterName"
- HidePin="ParameterName"
- InternalUseParam="ParameterName1, ParameterName2", multiple entries are allowed
- PinNames specified in ExpandEnumAsExecs or ExpandBoolAsExecs

- WorldContext="ParameterName", when a member function is called and the C++ base class has an implementation of GetWorld, the WorldContext can be automatically assigned the correct World value without the need for display.

This function is invoked by CreatePinsForFunctionCall to filter the internal properties of a Function, which include Params and ReturnValue. Therefore, HidePin cannot be used to hide Pins such as Target; this requirement should be addressed by HideSelfPin.

```
// Gets a list of pins that should be hidden for a given function
void FBlueprintEditorUtils::GetHiddenPinsForFunction(UEdGraph const* Graph,
UFunction const* Function, TSet<FName>& HiddenPins, TSet<FName>* OutInternalPins)
{
    check(Function != nullptr);
    TMap<FName, FString>* MetaData = UMetaData::GetMapForObject(Function);
    if (MetaData != nullptr)
    {
        for (TMap<FName, FString>::TConstIterator It(*MetaData); It; ++It)
        {
            const FName& Key = It.Key();

            if (Key == BlueprintMetadata::MD_LatentInfo)
            {
                HiddenPins.Add(*It.Value());
            }
            else if (Key == BlueprintMetadata::MD_HidePin)
            {
                TArray<FString> HiddenPinNames;
                It.Value().ParseIntoArray(HiddenPinNames, TEXT(","));
                for (FString& HiddenPinName : HiddenPinNames)
                {
                    HiddenPinName.TrimStartAndEndInline();
                    HiddenPins.Add(*HiddenPinName);
                }
            }
            else if (Key == BlueprintMetadata::MD_ExpandEnumAsExecs ||
                     Key == BlueprintMetadata::MD_ExpandBoolAsExecs)
            {
                TArray<FName> EnumPinNames;
                UK2Node_CallFunction::GetExpandEnumPinNames(Function,
                                                EnumPinNames);

                for (const FName& EnumName : EnumPinNames)
                {
                    HiddenPins.Add(EnumName);
                }
            }
            else if (Key == BlueprintMetadata::MD_InternalUseParam)
            {
                TArray<FString> HiddenPinNames;
                It.Value().ParseIntoArray(HiddenPinNames, TEXT(","));
                for (FString& HiddenPinName : HiddenPinNames)
                {
                    HiddenPinName.TrimStartAndEndInline();
                    FName HiddenPinName(*HiddenPinName);
                }
            }
        }
    }
}
```

```

        HiddenPins.Add(HiddenPinFName);

        if (OutInternalPins)
        {
            OutInternalPins->Add(HiddenPinFName);
        }
    }

    else if (Key == FBlueprintMetadata::MD_WorldContext)
    {
        const UEdGraphSchema_K2* K2Schema = GetDefault<UEdGraphSchema_K2>
() ;

        if(!K2Schema->IsStaticFunctionGraph(Graph))
        {
            bool bHasIntrinsicWorldContext = false;

            UBlueprint const* CallingContext =
FindBlueprintForGraph(Graph);
            if (CallingContext && CallingContext->ParentClass)
            {
                UClass* NativeOwner = CallingContext->ParentClass;
                while(NativeOwner && !NativeOwner->IsNative())
                {
                    NativeOwner = NativeOwner->GetSuperClass();
                }

                if(NativeOwner)
                {
                    bHasIntrinsicWorldContext = NativeOwner-
>GetDefaultObject()->ImplementsGetWorld();
                }
            }

            // if the blueprint has world context that we can lookup with
            "self",
            // then we can hide this pin (and default it to self)
            if (bHasIntrinsicWorldContext)
            {
                HiddenPins.Add(*It.Value());
            }
        }
    }
}
}

```

# InternalUseParam

- **Function Description:** Used in function calls to specify parameter names that should be hidden, and it can also hide return values. Multiple parameters can be concealed
  - **Usage Location:** UFUNCTION
  - **Engine Module:** Pin
  - **Metadata Type:** strings = "a, b, c"

- **Associated Items:** HidePin

- **Commonly Used:** ★★

This metadata is equivalent to HidePin.

## C++ Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_HidePinTest :public AActor
{
public:
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    int MyFunc_Default(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (HidePin = "options"))
    int MyFunc_HidePin(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options,comment"))
    int MyFunc_HidePin2(FName name, float value, FString options,FString comment)
    { return 0; }

    UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options"))
    int MyFunc_InternalUseParam(FName name, float value, FString options) {
    return 0; }

    UFUNCTION(BlueprintCallable, meta = (HidePin = "ReturnValue"))
    int MyFunc_HideReturn(FName name, float value, FString options, FString&
otherReturn) { return 0; }

public:
    UFUNCTION(BlueprintPure)
    int MyPure_Default(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintPure, meta = (HidePin = "options"))
    int MyPure_HidePin(FName name, float value, FString options) { return 0; }

    UFUNCTION(BlueprintPure, meta = (InternalUseParam = "options"))
    int MyPure_InternalUseParam(FName name, float value, FString options) {
    return 0; }

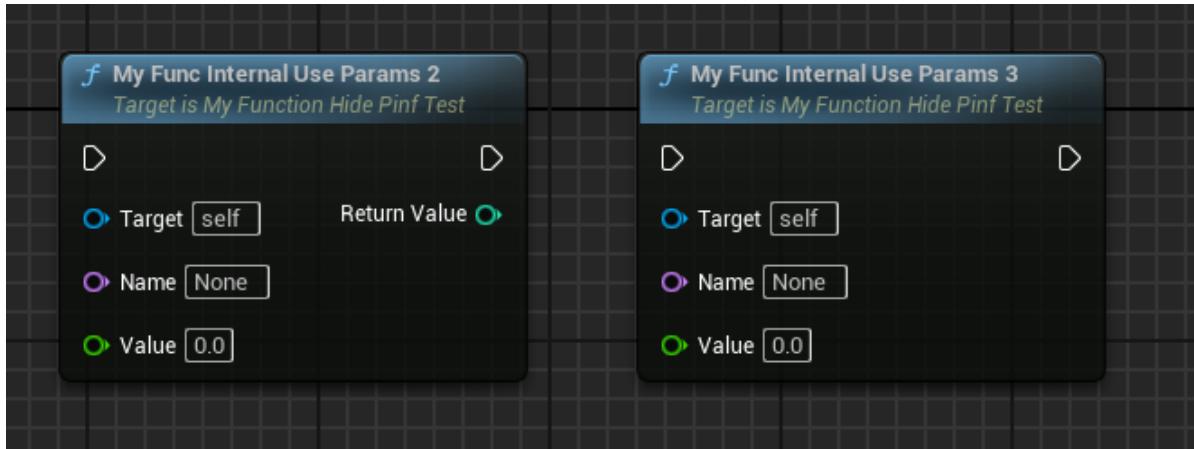
    UFUNCTION(BlueprintPure, meta = (HidePin = "ReturnValue"))
    int MyPure_HideReturn(FName name, float value, FString options, FString&
otherReturn) { return 0; }

public:
    UFUNCTION(BlueprintCallable, meta = (InternalUseParam = "options,comment"))
    int MyFunc_InternalUseParams2(FName name, float value, FString
options,FString comment) { return 0; }

    UFUNCTION(BlueprintCallable, meta = (InternalUseParam =
"options,comment,ReturnValue"))
    int MyFunc_InternalUseParams3(FName name, float value, FString
options,FString comment) { return 0; }
```

```
};
```

## Blueprint Test Results:



It can be observed that both BlueprintCallable and BlueprintPure are applicable. Moreover, ReturnValue is the default name for the return value, which can also be hidden using this feature.

## Principle:

The use of MD\_InternalUseParam is also for hiding pins.

```
// Gets a list of pins that should be hidden for a given function
void FBlueprintEditorUtils::GetHiddenPinsForFunction(UEdGraph const* Graph,
UFunction const* Function, TSet<FName>& HiddenPins, TSet<FName>* OutInternalPins)
{
    check(Function != nullptr);
    TMap<FName, FString>* MetaData = UMetaData::GetMapForObject(Function);
    if (MetaData != nullptr)
    {
        for (TMap<FName, FString>::TConstIterator It(*MetaData); It; ++It)
        {
            const FName& Key = It.Key();

            if (Key == FBlueprintMetadata::MD_LatentInfo)
            {
                HiddenPins.Add(*It.Value());
            }
            else if (Key == FBlueprintMetadata::MD_HidePin)
            {
                TArray<FString> HiddenPinNames;
                It.Value().ParseIntoArray(HiddenPinNames, TEXT(","));
                for (FString& HiddenPinName : HiddenPinNames)
                {
                    HiddenPinName.TrimStartAndEndInline();
                    HiddenPins.Add(*HiddenPinName);
                }
            }
            else if (Key == FBlueprintMetadata::MD_ExpandEnumAsExecs ||
                     Key == FBlueprintMetadata::MD_ExpandBoolAsExecs)
            {
                TArray<FName> EnumPinNames;
```

```

UK2Node_CallFunction::GetExpandEnumPinNames(Function,
EnumPinNames);

    for (const FName& EnumName : EnumPinNames)
    {
        HiddenPins.Add(EnumName);
    }
}

else if (Key == FBlueprintMetadata::MD_InternalUseParam)
{
    TArray< FString> HiddenPinNames;
    It.Value().ParseIntoArray(HiddenPinNames, TEXT(","));
    for (FString& HiddenPinName : HiddenPinNames)
    {
        HiddenPinName.TrimStartAndEndInline();

        FName HiddenPinName(*HiddenPinName);
        HiddenPins.Add(HiddenPinName);

        if (OutInternalPins)
        {
            OutInternalPins->Add(HiddenPinName);
        }
    }
}

else if (Key == FBlueprintMetadata::MD_WorldContext)
{
    const UEdGraphSchema_K2* K2Schema = GetDefault<UEdGraphSchema_K2>
();

    if (!K2Schema->IsStaticFunctionGraph(Graph))
    {
        bool bHasIntrinsicWorldContext = false;

        UBlueprint const* CallingContext =
FindBlueprintForGraph(Graph);
        if (CallingContext && CallingContext->ParentClass)
        {
            UClass* NativeOwner = CallingContext->ParentClass;
            while(NativeOwner && !NativeOwner->IsNative())
            {
                NativeOwner = NativeOwner->GetSuperClass();
            }

            if(NativeOwner)
            {
                bHasIntrinsicWorldContext = NativeOwner-
>GetDefaultObject()->ImplementsGetWorld();
            }
        }

        // if the blueprint has world context that we can lookup with
"self",
        // then we can hide this pin (and default it to self)
        if (bHasIntrinsicWorldContext)
        {
            HiddenPins.Add(*It.Value());
        }
    }
}

```

```
    }  
    }  
    }  
}
```

# HideSelfPin

- **Function Description:** Used in function calls to conceal the default SelfPin, which is the Target, thereby restricting the function to being called only within the OwnerClass.
  - **Usage Location:** UFUNCTION
  - **Engine Module:** Pin
  - **Metadata Type:** bool
  - **Commonality:** ★★

Used in function calls to hide the default SelfPin, which is the Target, resulting in the function being callable only within the OwnerClass.

Comments indicate it is typically used in conjunction with the DefaultToSelf specifier, but no examples were found in the source code.

Similar to HidePin and InternalUseParam, but while the latter can hide other parameters, HideSelfPin can only hide the SelfPin

## Logic Code:

It can be observed that the visibility of SelfPin's bHidden is influenced by certain conditions:

1. If the function is Static (either a function in the blueprint function library or a static function in C++), SelfPin is hidden by default since no Target is required for invocation.
  2. If the function is marked with HideSelfPin, it is also hidden by default and cannot be connected externally, limiting the function to being called only within the class.
  3. If the function is a BlueprintPure function and is being called within the OwnerClass, there is no need to display the SelfPin.

Only this particular instance of judgment and application is found in the source code. Therefore, it can be inferred that HideSelfPin only conceals the Self, meaning it only hides the This pointer (which is the Self, or Target) when a class member function is called, but does not hide the parameters when a Static function is called, even if those parameters are marked with DefaultToSelf. Being marked with DefaultToSelf indicates that the parameter's default value is the Self value of the external calling environment, not the SelfPin on this function node. The SelfPin of a static function is always hidden. The parameter marked with DefaultToSelf may have a value equal to Self, but it is not the SelfPin.

```
bool UK2Node_CallFunction::CreatePinsForFunctionCall(const UFunction* Function)
{
//...
    if (bIsStaticFunc)
    {
        // For static methods, wire up the self to the CDO of the class if
it's not us
```

```

        if (!bIsFunctionCompatibleWithSelf)
    {
        UClass* AuthoritativeClass = FunctionOwnerClass-
>GetAuthoritativeClass();
        SelfPin->DefaultObject = AuthoritativeClass->GetDefaultObject();
    }

        // Purity doesn't matter with a static function, we can always hide
        the self pin since we know how to call the method
        SelfPin->bHidden = true;
    }
else
{
    if (Function->GetBoolMetaData(FBlueprintMetadata::MD_HideSelfPin))
    {
        SelfPin->bHidden = true;
        SelfPin->bNotConnectable = true;
    }
else
{
    // Hide the self pin if the function is compatible with the
    blueprint class and pure (the !bIsConstFunc portion should be going away soon too
    hopefully)
    SelfPin->bHidden = (bIsFunctionCompatibleWithSelf && bIsPureFunc
&& !bIsConstFunc);
}
}
}
}

```

## C++ Test Code:

```

UCLASS()
class INSIDER_API UMyFunctionLibrary_SelfPinTest :public
UBlueprintFunctionLibrary
{
GENERATED_BODY()

UFUNCTION(BlueprintPure,meta=(DefaultToSelf="myOwner",hidePin="myOwner"))
static FString PrintProperty_HasDefaultToSelf_ButHide(Uobject* myOwner,FName
propertyName);

UFUNCTION(BlueprintPure,meta=(DefaultToSelf="myOwner",HideSelfPin="true"))
static FString PrintProperty_HasDefaultToSelf_AndHideSelf(Uobject*
myOwner,FName propertyName);

UFUNCTION(BlueprintPure,meta=
(DefaultToSelf="myOwner",InternalUseParam="myOwner"))
static FString PrintProperty_HasDefaultToSelf_ButInternal(Uobject*
myOwner,FName propertyName);
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyFunction_HideselfTest :public AActor
{
public:

```

```

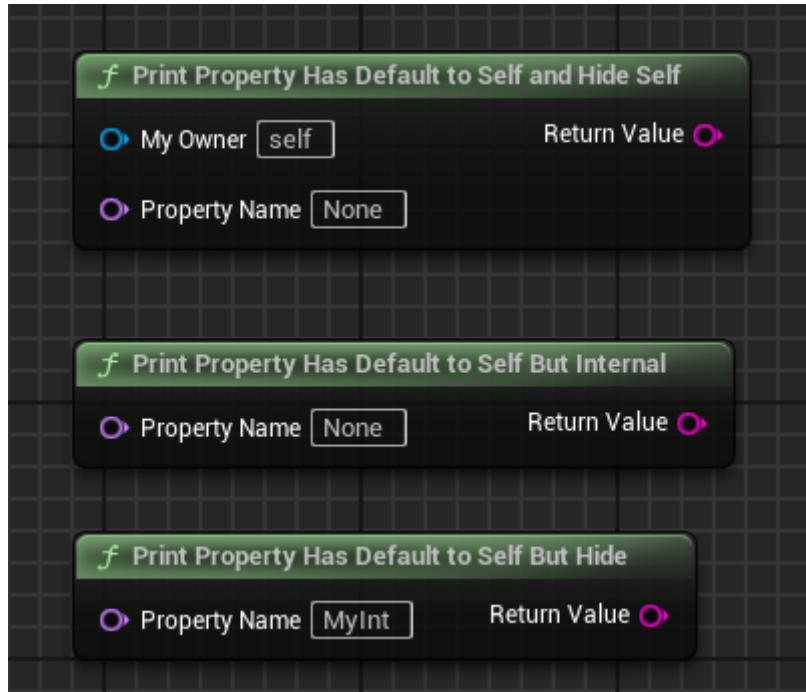
GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    void MyFunc_Default(int value){}

    UFUNCTION(BlueprintCallable,meta=(HideSelfPin="true"))
    void MyFunc_HideSelfPin(int value){}
};

```

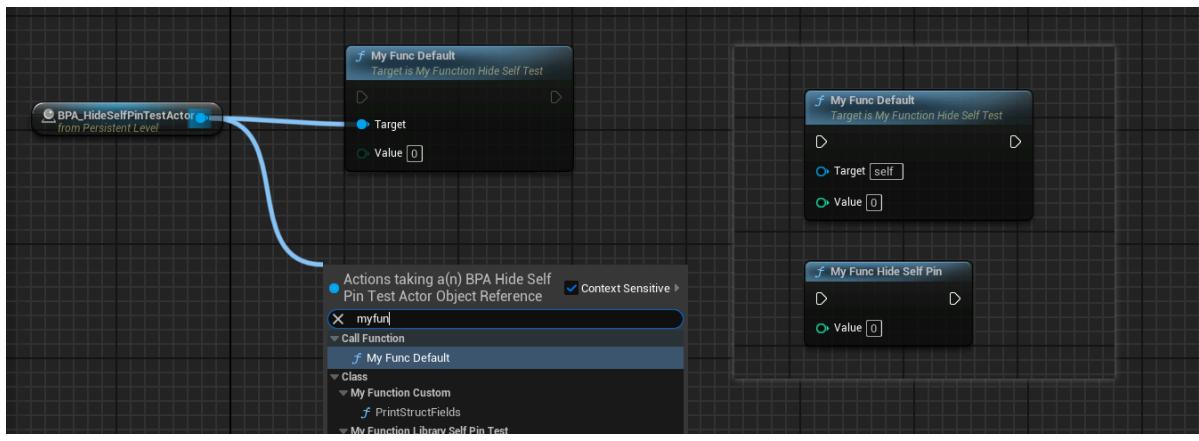
## Test Effect in Blueprint:

The first image shows that HideSelfPin has no effect on Static functions, while InternalUseParam can hide pins.



For the test results of class member functions, it can be observed that:

- When called within the class, Self can be hidden but remains callable. The difference is that the MyFunc\_Default version, by default, can be invoked by different instances of AMyFunction\_HideSelfTest of the same type. MyFunc\_HideSelfPin, however, can only be called by the current object.
- In the level blueprint on the left, when attempting to call these two functions through an AMyFunction\_HideSelfTest object, MyFunc\_Default can be called, but the MyFunc\_HideSelfPin function node cannot be created. Even if it is artificially created through copy-pasting, it cannot be connected due to the absence of the Self Target Pin, thus preventing the call.



## #DataTablePin

- **Function Description:** Specifies a function parameter as DataTable or CurveTable type to allow selection of RowNameList for other parameters of FName.
- **Usage Location:** UFUNCTION
- **Engine Module:** Pin
- **Metadata Type:** string="abc"
- **Restricted Types:** DataTable, CurveTable
- **Commonliness:** ★★

Designates a function parameter as UDataTable type, enabling the loading of data from the DataTable to provide a List for RowName selection, instead of manual input. Supported types include DataTable and CurveTable

For DataTablePin, UK2Node\_CallDataTableFunction is used to generate a blueprint node, which triggers the refresh of RowNameList when the DataTable Pin connection parameters change.

UK2Node\_GetDataTableRow is also a standalone blueprint node.

However, RowName is not specified with meta? There is a function parameter type check; if it is FName, it is treated as RowName. (This simple and rough check is due to the intention not to provide this Meta for user use.) Therefore, there can actually be multiple FName parameters in the function, all of which are automatically assigned to RowNameList

The only place this is used in the source code is UDataTableFunctionLibrary

```
UFUNCTION(BlueprintCallable, Category = "DataTable", meta =
(ExpandEnumAsExecs="OutResult", DataTablePin="CurveTable"))
static ENGINE_API void EvaluateCurveTableRow(UCurveTable* CurveTable, FName
RowName, float InXY, TEnumAsByte<EEvaluateCurveTableResult::Type>& OutResult,
float& OutXY, const FString& ContextString);
```

## Test Code:

```
UFUNCTION(BlueprintCallable, meta = (DataTablePin="CurveTable"))
static void GetMyCurveTableRow(UCurveTable* CurveTable, FName MyRowName,
float InXY, float& OutXY, const FString& ContextString){}

UFUNCTION(BlueprintCallable, meta = (DataTablePin="TargetTable"))
static bool HasMyDataTableRow(UDataTable* TargetTable, FName MyRowName, FName
OtherRowName){return false;}
```

## Blueprint Effect:

The function node on the left is user-defined. You can see that the names on the blueprint nodes on the left have changed to the RowNameList from CurveTable and DataTable, even though these FName parameters have no special designations. The blueprint system automatically recognizes the FName type and changes the actual Pin Widget.



## Principle:

If an FName node is detected, it attempts to find the DataTablePin and then adjusts the Pin type of the FName according to the type of the DataTablePin.

```
TSharedPtr<class SGraphPin> FBlueprintGraphPanelPinFactory::CreatePin(class UEdGraphPin* InPin) const
{
    if (InPin->PinType.PinCategory == UEdGraphSchema_K2::PC_Name)
    {
        UObject* Outer = InPin->GetOuter();

        // Create drop down combo boxes for DataTable and CurveTable RowName pins
        const UEdGraphPin* DataTablePin = nullptr;
        if (Outer->IsA(UK2Node_CallFunction::StaticClass()))
        {
            const UK2Node_CallFunction* CallFunctionNode =
            CastChecked<UK2Node_CallFunction>(Outer);
            if (CallFunctionNode)
            {
                const UFunction* FunctionToCall = CallFunctionNode-
                >GetTargetFunction();
                if (FunctionToCall)
                {
                    const FString& DataTablePinName = FunctionToCall-
                    >GetMetaData(FBlueprintMetadata::MD_DataTablePin);
```

```

        DataTablePin = CallFunctionNode->FindPin(DataTablePinName);
    }
}
else if (Outer->IsA(UK2Node_GetDataRow::StaticClass()))
{
    const UK2Node_GetDataRow* GetDataRowNode =
CastChecked<UK2Node_GetDataRow>(Outer);
    DataTablePin = GetDataRowNode->GetDataTablePin();
}

if (DataTablePin)
{
    if (DataTablePin->DefaultObject != nullptr && DataTablePin-
>LinkedTo.Num() == 0)
    {
        if (auto DataTable = Cast<UDataTable>(DataTablePin-
>DefaultObject))
        {
            return SNew(SGraphPinDataTableRowName, InPin, DataTable);
        }
        if (DataTablePin->DefaultObject->IsA(UCurveTable::StaticClass()))
        {
            UCurveTable* CurveTable = (UCurveTable*)DataTablePin-
>DefaultObject;
            if (CurveTable)
            {
                TArray<TSharedPtr<FName>> RowNames;
                /** Extract all the row names from the RowMap */
                for (TMap<FName, FRealCurve*>::TConstIterator
Iterator(CurveTable->GetRowMap()); Iterator; ++Iterator)
                {
                    /** Create a simple array of the row names */
                    TSharedPtr<FName> RowNameItem = MakeShareable(new
FName(Iterator.Key()));
                    RowNames.Add(RowNameItem);
                }
                return SNew(SGraphPinNameList, InPin, RowNames);
            }
        }
    }
}

return nullptr;
}

```

## DisableSplitPin

---

- **Function Description:** Disable the split feature of a Struct
- **Usage Location:** USTRUCT
- **Engine Module:** Pin

- **Metadata Type:** bool
- **Commonliness:** ★★

For some Structs, especially structures with only one member variable, sometimes it will look strange if you expand them by default. At this time, I hope to disable this function. But note that you can still manually access member variables through Break in the blueprint. If you don't want to expose member variable access in the blueprint, you should not add BlueprintReadWrite/BlueprintReadOnly to UPROPERTY

Search the source code for examples like FGameplayTag, FPostProcessSettings, FSlatePostSettings

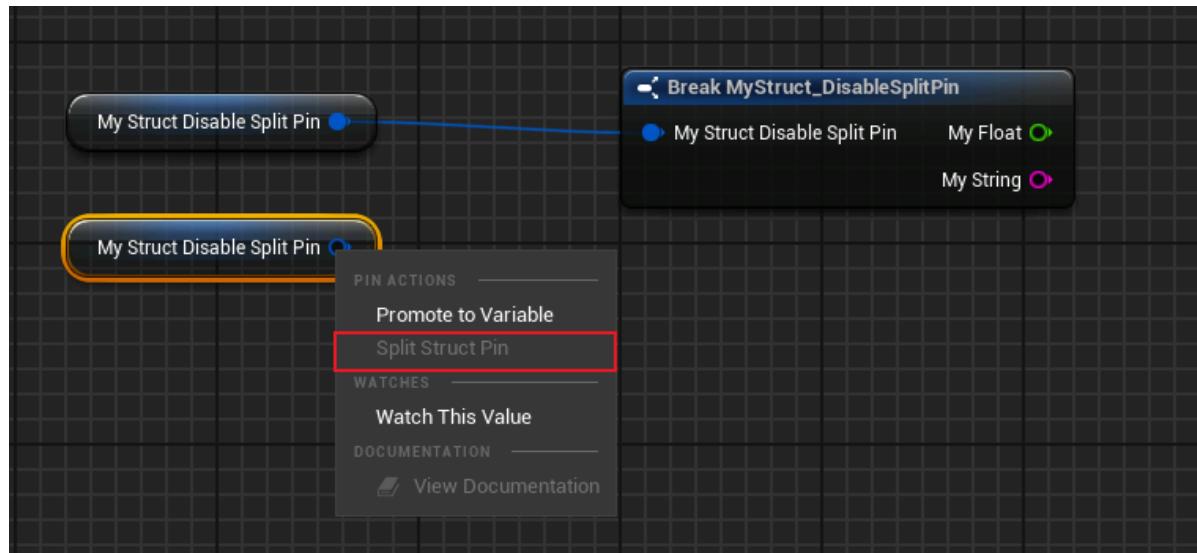
## Test Code:

```
USTRUCT(BlueprintType, meta = (DisableSplitPin))
struct INSIDER_API FMyStruct_DisableSplitPin
{
    GENERATED_BODY()

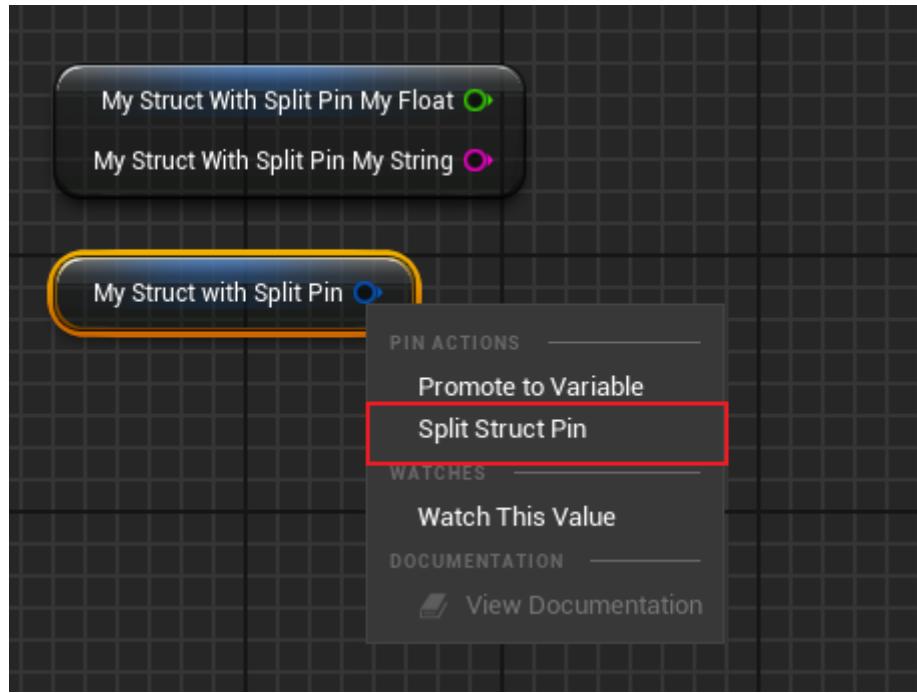
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_WithSplitPin
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};
```



Allowed comparisons



## HiddenByDefault

- **Function Description:** Pins within the "Make Struct" and "Break Struct" nodes of a Struct are set to a hidden state by default
- **Usage Location:** USTRUCT
- **Engine Module:** Pin
- **Metadata Type:** bool
- **Commonality:** ★

### Test Code:

```
//(BlueprintType = true, HiddenByDefault = , ModuleRelativePath =
Struct/MyStruct_HiddenByDefault.h)
USTRUCT(BlueprintType, meta = (HiddenByDefault))
struct INSIDER_API FMyStruct_HiddenByDefault
{
    GENERATED_BODY()

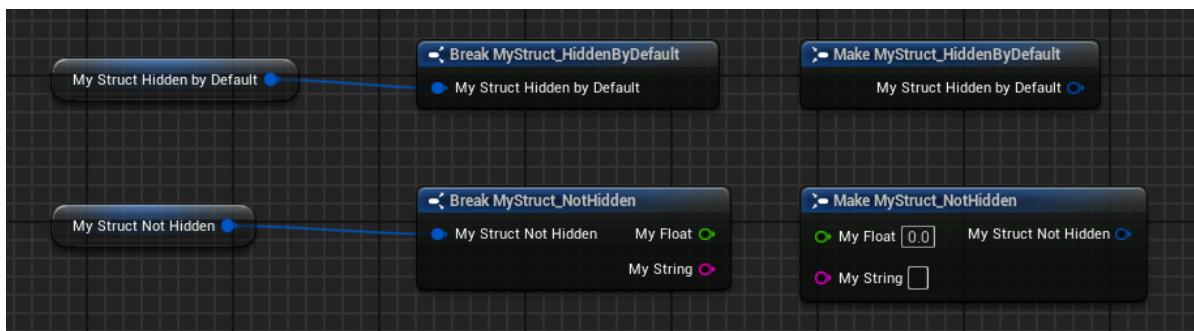
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_NotHidden
{
    GENERATED_BODY()

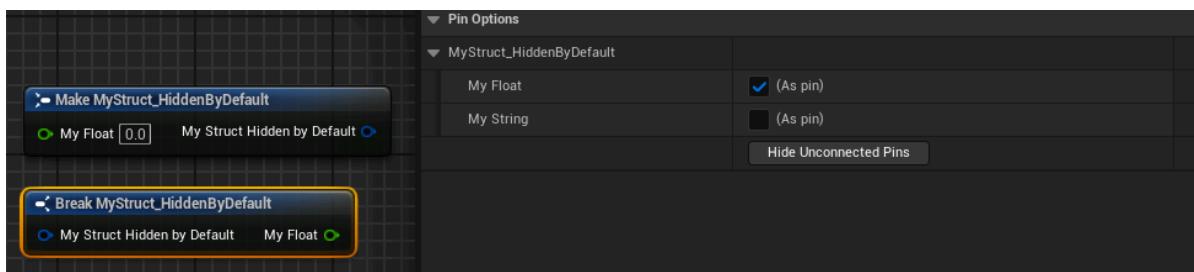
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
```

```
};
```

## Blueprint Results:



What is meant by "hidden" is that certain properties need to be manually selected in the node's details panel, rather than being automatically displayed in full as with the default setting.



## AlwaysAsPin

- **Function Description:** Ensures that a specific attribute of an animation node is always exposed as a pin in the animation blueprint
- **Usage Location:** UPROPERTY
- **Engine Module:** Pin
- **Metadata Type:** bool
- **Restriction Type:** FAnimNode\_Base
- **Associated Items:** PinShownByDefault
- **Commonality:** ★★★

The difference from PinShownByDefault is that AlwaysAsPin will always display the attribute as a pin, whereas PinShownByDefault displays the pin by default but can be changed subsequently.

## Test Code:

```
USTRUCT(BlueprintInternalUseOnly)
struct INSIDEREDITOR_API FAnimNode_MyTestPinShown : public FAnimNode_Base
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest)
    int32 MyInt_NotShown = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
    meta = (PinShownByDefault))
    int32 MyInt_PinShownByDefault = 123;
```

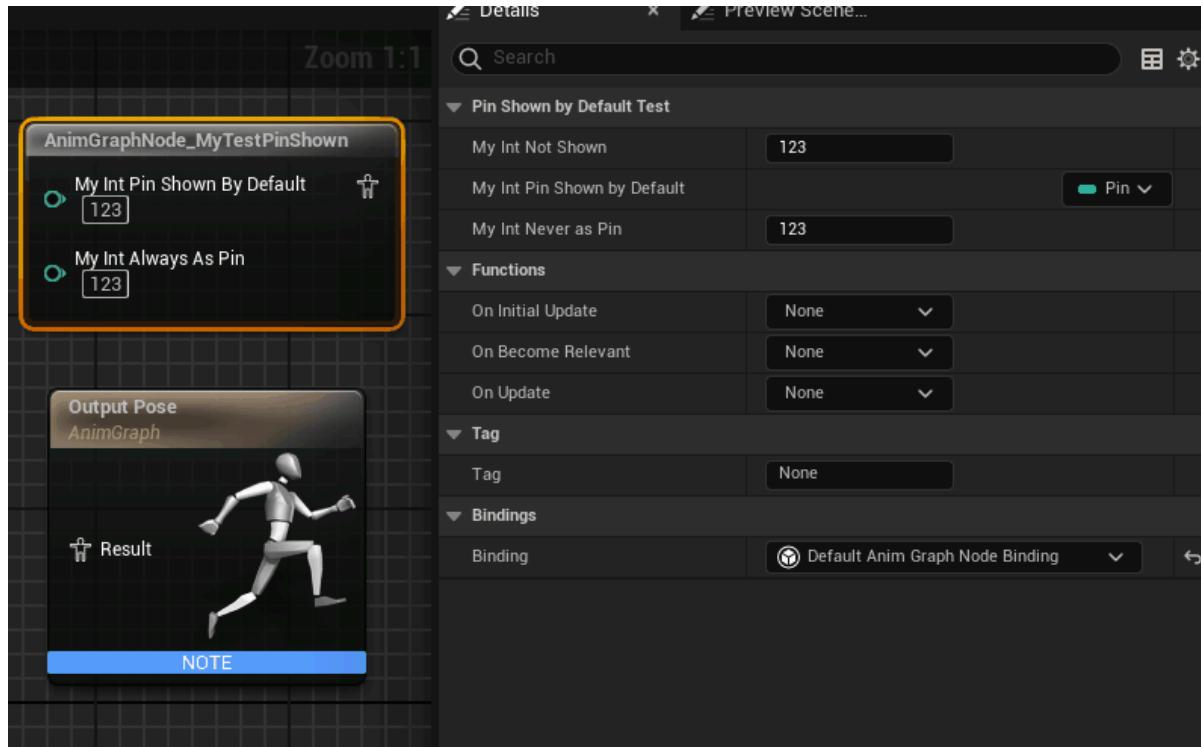
```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (AlwaysAsPin))
int32 MyInt_AlwaysAsPin = 123;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (NeverAsPin))
int32 MyInt_NeverAsPin = 123;
};

```

## Test Results:



## Principle:

As evident from the logic in the source code, bAlwaysShow will result in bShowPin being true, differing from PinShownByDefault in that AlwaysAsPin will always display the attribute as a pin. PinShownByDefault shows the pin by default and can be modified later.

```

void FAnimBlueprintNodeOptionalPinManager::GetRecordDefaults(FProperty* TestProperty, FOptionalPinFromProperty& Record) const
{
    const UAnimationGraphSchema* Schema = GetDefault<UAnimationGraphSchema>();

    // Determine if this is a pose or array of poses
    FArrayProperty* ArrayProp = CastField<FArrayProperty>(TestProperty);
    FStructProperty* StructProp = CastField<FStructProperty>(ArrayProp ? ArrayProp->Inner : TestProperty);
    const bool bIsPoseInput = (StructProp && StructProp->Struct->IsChildOf(FPoseLinkBase::StaticStruct()));

    // @TODO: Error if they specified two or more of these flags
    const bool bAlwaysShow = TestProperty->HasMetaData(Schema->NAME_AlwaysAsPin)
        || bIsPoseInput;
}

```

```

        const bool bOptional_ShowByDefault = TestProperty->HasMetaData(Schema-
>NAME_PinShownByDefault);
        const bool bOptional_HideByDefault = TestProperty->HasMetaData(Schema-
>NAME_PinHiddenByDefault);
        const bool bNeverShow = TestProperty->HasMetaData(Schema->NAME_NeverAsPin);
        const bool bPropertyIsCustomized = TestProperty->HasMetaData(Schema-
>NAME_CustomizeProperty);
        const bool bCanTreatPropertyAsOptional =
CanTreatPropertyAsOptional(TestProperty);

        Record.bCanToggleVisibility = bCanTreatPropertyAsOptional &&
(bOptional_ShowByDefault || bOptional_HideByDefault);
        Record.bShowPin = bAlwaysShow || bOptional_ShowByDefault;
        Record.bPropertyIsCustomized = bPropertyIsCustomized;
    }
}

```

## NeverAsPin

- **Function Description:** Ensures that a specific attribute of an animation node is never exposed as a pin in the animation blueprint
- **Usage Location:** UPROPERTY
- **Engine Module:** Pin
- **Metadata Type:** bool
- **Restriction Type:** FAnimNode\_Base
- **Associated Items:** PinShownByDefault
- **Commonly Used:** ★★★

The NeverAsPin feature is not utilized in the source code because, by default, it is not supported as a pin, making its inclusion optional.

## Test Code:

```

USTRUCT(BlueprintInternalUseOnly)
struct INSIDEREDITOR_API FAnimNode_MyTestPinShown : public FAnimNode_Base
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest)
    int32 MyInt_NotShown = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (PinShownByDefault))
    int32 MyInt_PinShownByDefault = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (AlwaysAsPin))
    int32 MyInt_AlwaysAsPin = 123;

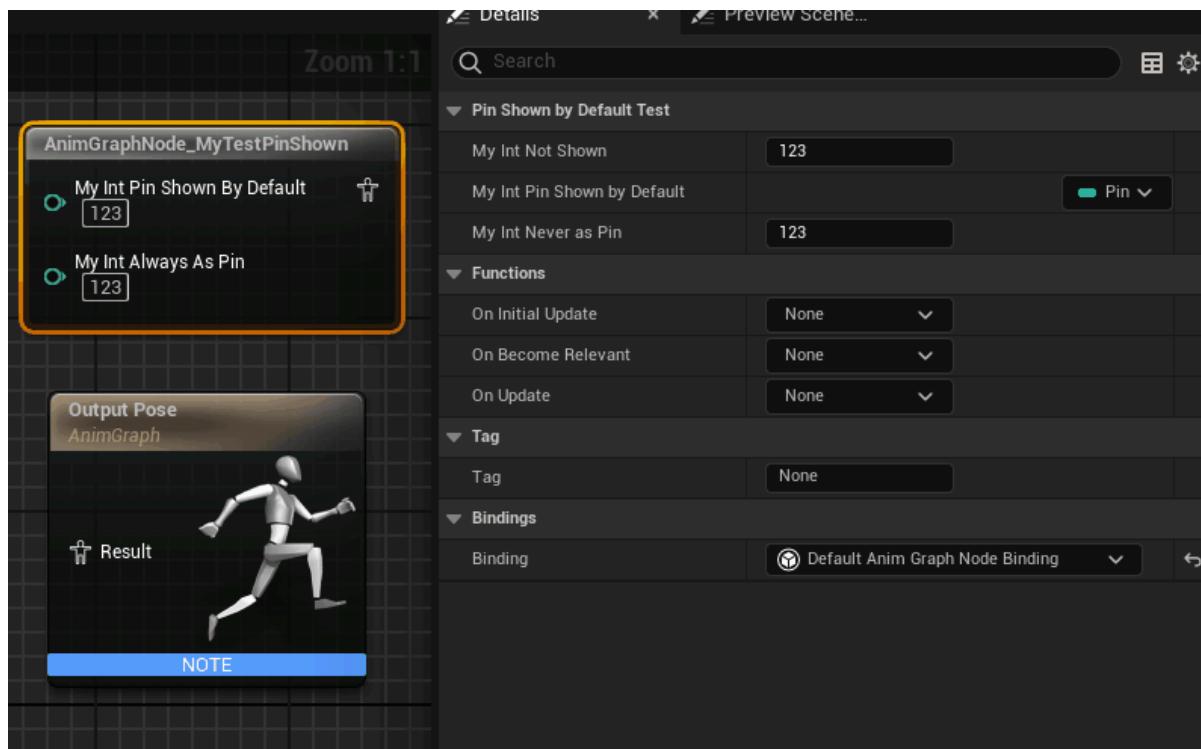
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = PinShownByDefaultTest,
meta = (NeverAsPin))
    int32 MyInt_NeverAsPin = 123;
}

```

```
};
```

## Testing Code:

The attribute MyInt\_NeverAsPin can only be displayed like the default attribute on the right and cannot be shown as a pin.



## 测试效果:

It has been observed that bNeverShow is not in use, as the attribute is inherently not supported as a pin by default.

```
void FAnimBlueprintNodeOptionalPinManager::GetRecordDefaults(FProperty*  
TestProperty, FOptionalPinFromProperty& Record) const  
{  
    const UAnimationGraphSchema* Schema = GetDefault<UAnimationGraphSchema>();  
  
    // Determine if this is a pose or array of poses  
    FArrayProperty* ArrayProp = CastField<FArrayProperty>(TestProperty);  
    FStructProperty* StructProp = CastField<FStructProperty>(ArrayProp ?  
        ArrayProp->Inner : TestProperty);  
    const bool bIsPoseInput = (StructProp && StructProp->Struct-  
    >IsChildOf(FPoseLinkBase::StaticStruct()));  
  
    // TODO: Error if they specified two or more of these flags  
    const bool bAlwaysShow = TestProperty->HasMetaData(Schema->NAME_AlwaysAsPin)  
    || bIsPoseInput;  
    const bool bOptional_ShowByDefault = TestProperty->HasMetaData(Schema-  
    >NAME_PinShownByDefault);  
    const bool bOptional_HideByDefault = TestProperty->HasMetaData(Schema-  
    >NAME_PinHiddenByDefault);  
    const bool bNeverShow = TestProperty->HasMetaData(Schema->NAME_NeverAsPin);  
    const bool bPropertyIsCustomized = TestProperty->HasMetaData(Schema-  
    >NAME_CustomizeProperty);
```

```

    const bool bCanTreatPropertyAsOptional =
CanTreatPropertyAsOptional(TestProperty);

    Record.bCanToggleVisibility = bCanTreatPropertyAsOptional &&
(boptional_ShowByDefault || boptional_HideByDefault);
    Record.bShowPin = bAlwaysShow || boptional_ShowByDefault;
    Record.bPropertyIsCustomized = bPropertyIsCustomized;
}

```

## PinHiddenByDefault

- **Function description:** Causes properties within this structure to be hidden by default when represented as pins in blueprints.
- **Use location:** UPROPERTY
- **Engine module:** Pin
- **Metadata type:** bool
- **Restriction type:** struct member property
- **Commonly used:** ★★

Makes the properties within this structure hidden by default when represented as pins in blueprints.

Please note that this metadata only affects the member properties of the structure and is only active within blueprint nodes. In some cases, a structure may contain multiple properties, but not all properties need to be exposed for user editing at once. Some properties might be advanced and should initially remain hidden.

This attribute can also be applied in Animation Blueprints to prevent certain properties of animation nodes from being exposed as pins.

## Test Code:

```

USTRUCT(BlueprintType)
struct FMyPinHiddenTestStruct
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=PinHiddenByDefaultTest)
    int32 MyInt_NotHidden = 123;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category=PinHiddenByDefaultTest,
meta = (PinHiddenByDefault))
    int32 MyInt_PinHiddenByDefault = 123;
};

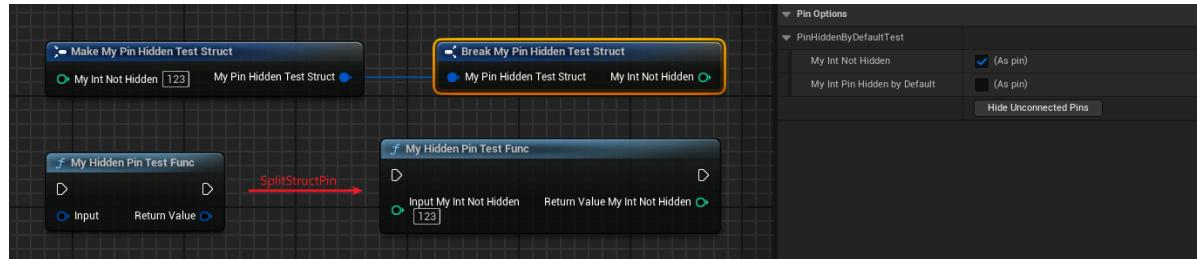
UFUNCTION(BlueprintCallable)
static FMyPinHiddenTestStruct MyHiddenPinTestFunc(FMyPinHiddenTestStruct
Input);

```

# Test Results:

It can be observed that for the MakeStruct and BreakStruct nodes, only MyInt\_NotHidden is displayed by default. When the blueprint node is selected, the MyInt\_PinHiddenByDefault property appears unchecked in the details panel on the right, illustrating the difference.

The same applies when the structure is used as a function input and output parameter. When the SplitStructPin is used to expand the structure node, MyInt\_PinHiddenByDefault is also hidden.



## Principle:

As revealed in the source code, FStructOperationOptionalPinManager utilizes this metadata, and both FMakeStructPinManager and FBreakStructPinManager inherit from it, ensuring that the PinHiddenByDefault pin is not displayed initially.

```
struct FStructOperationOptionalPinManager : public FOptionalPinManager
{
    //~ Begin FOptionalPinsUpdater Interface
    virtual void GetRecordDefaults(FProperty* TestProperty,
FOptionalPinFromPropertyParams& Record) const override
    {
        Record.bCanToggleVisibility = true;
        Record.bShowPin = true;
        if (TestProperty)
        {
            Record.bShowPin = !TestProperty->HasMetaData(TEXT("PinHiddenByDefault"));
            if (Record.bShowPin)
            {
                if (UStruct* OwnerStruct = TestProperty->GetOwnerStruct())
                {
                    Record.bShowPin = !OwnerStruct->HasMetaData(TEXT("HiddenByDefault"));
                }
            }
        }
    }

    virtual void CustomizePinData(UEdGraphPin* Pin, FName SourcePropertyName,
int32 ArrayIndex, FProperty* Property) const override;
    // End of FOptionalPinsUpdater interface
};

struct FMakeStructPinManager : public FStructOperationOptionalPinManager
{}
struct FBreakStructPinManager : public FStructOperationOptionalPinManager
{}
```

# Input

- **Function Description:** Designate this attribute under FRigUnit as an input pin.
- **Usage Location:** UPROPERTY
- **Engine Module:** RigVMStruct
- **Metadata Type:** bool
- **Restriction Type:** Attribute within FRigUnit
- **Associated Items:** Output, Visible, Hidden, DetailsOnly, Constant
- **Commonliness:** ★★★★☆

This attribute under FRigUnit is specified as an input pin.

It is worth noting that if a pin has both Input and Output designations, it becomes an IO pin, serving as both input and output simultaneously.

## Test Code:

```
USTRUCT(meta = (DisplayName="MyRig"))
struct INSIDER_API FRigUnit_MyRig : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
    virtual void Execute() override;

public:
    UPROPERTY()
    float MyFloat_Normal;

    UPROPERTY(meta = (Input))
    float MyFloat_Input;

    UPROPERTY(meta = (Output))
    float MyFloat_Output;

    UPROPERTY(meta = (Input, Output))
    float MyFloat_IO;

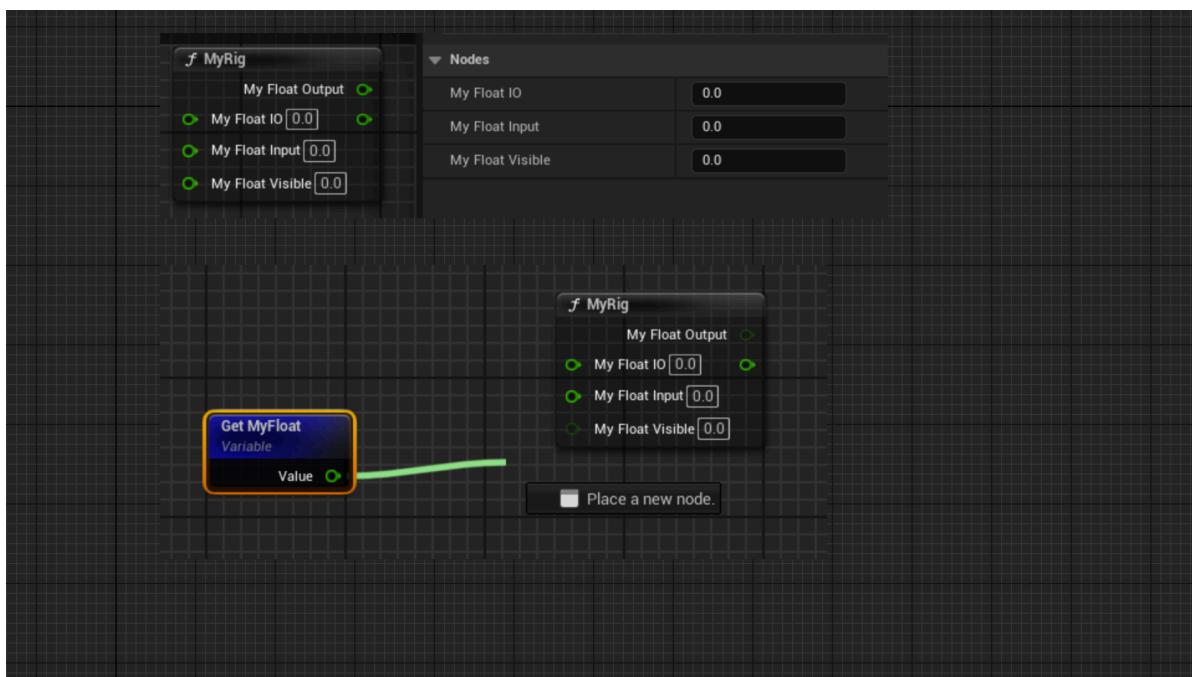
    UPROPERTY(meta = (Visible))
    float MyFloat_Visible;

    UPROPERTY(meta = (Hidden))
    float MyFloat_Hidden;
};
```

## Test Results:

In the ControlRig blueprint, you can invoke the MyRig node. Observe the pin behavior of the attribute on the blueprint node and its display in the details panel on the right.

- MyFloat\_Normal is not marked with meta and does not appear in either location.
- MyFloat\_Input serves as an input pin and is also displayed in the right details panel.
- MyFloat\_Output serves as an output pin and is not shown in the right details panel.
- MyFloat\_IO can function as both an input and output pin and is displayed in the right details panel.
- MyFloat\_Visible can be displayed as an input pin and is shown in the right details panel. However, it cannot be connected to a variable, meaning it can only be used as a constant.
- MyFloat\_Hidden, like MyFloat\_Normal, is hidden in both the blueprint node and the details panel, used only as its internal value.



## Principle:

The direction of the pin is determined by the Meta tag on the property. You can refer to the various types of ERigVMPinDirection in the source code.

```
UENUM(BlueprintType)
enum class ERigVMPinDirection : uint8
{
    Input, // A const input value
    Output, // A mutable output value
    IO, // A mutable input and output value
    Visible, // A const value that cannot be connected to
    Hidden, // A mutable hidden value (used for internal state)
    Invalid // The max value for this enum - used for guarding.
};

ERigVMPinDirection FRigVMStruct::GetPinDirectionFromProperty(FProperty* InProperty)
```

```

bool bIsInput = InProperty->HasMetaData(InputMetaName);
bool bIsOutput = InProperty->HasMetaData(OutputMetaName);
bool bIsVisible = InProperty->HasMetaData(VisibleMetaName);

if (bIsVisible)
{
    return ERigVMPinDirection::Visible;
}

if (bIsInput)
{
    return bIsOutput ? ERigVMPinDirection::IO : ERigVMPinDirection::Input;
}

if(bIsOutput)
{
    return ERigVMPinDirection::Output;
}

return ERigVMPinDirection::Hidden;
}

```

## Constant

- **Function Description:** Indicates that a property is to be treated as a constant pin.
- **Usage Locations:** UPROPERTY, USTRUCT
- **Engine Module:** RigVMStruct
- **Metadata Type:** bool
- **Associated Items:** Input
- **Commonality:** ★★★

When applied to UPROPERTY, similar to Visible, it designates a property as a constant pin.

When applied to USTRUCT, it is observed in functions like IsDefinedAsConstant, but no calls to it were found using F5.

```

USTRUCT(meta = (DisplayName = "Rotation Order", Category = "Math|Quaternion",
Constant))
struct RIGVM_API FRigVMFunction_MathQuaternionRotationOrder : public
FRigVMFunction_MathBase
{
}

```

## Output

- **Function Description:** Designates the specified attribute under FRigUnit as an output pin.
- **Usage Location:** UPROPERTY
- **Engine Module:** RigVMStruct
- **Metadata Type:** bool
- **Restriction Type:** Attribute within FRigUnit

- **Associated Items:** Input

- **Commonality:** ★★★★☆

Designates the specified attribute under FRigUnit as an output pin.

## Visible

- **Function Description:** The attribute within FRigUnit is designated as a constant pin, which cannot be linked to variables.
- **Use Location:** UPROPERTY
- **Engine Module:** RigVMStruct
- **Metadata Type:** Boolean
- **Restriction Type:** Attribute within FRigUnit
- **Associated Items:** Input
- **Commonality:** ★★★

Designate the attribute under FRigUnit as a constant pin, disallowing connection to variables.

The "Visible" and "Input+Constant" settings yield identical effects.

## Test Code:

```
USTRUCT(meta = (DisplayName = "MyRig"))
struct INSIDER_API FRigUnit_MyRig : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;

public:
    UPROPERTY()
    float MyFloat_Normal;

    UPROPERTY(meta = (Input))
    float MyFloat_Input;

    UPROPERTY(meta = (output))
    float MyFloat_Output;

    UPROPERTY(meta = (Input, output))
    float MyFloat_IO;

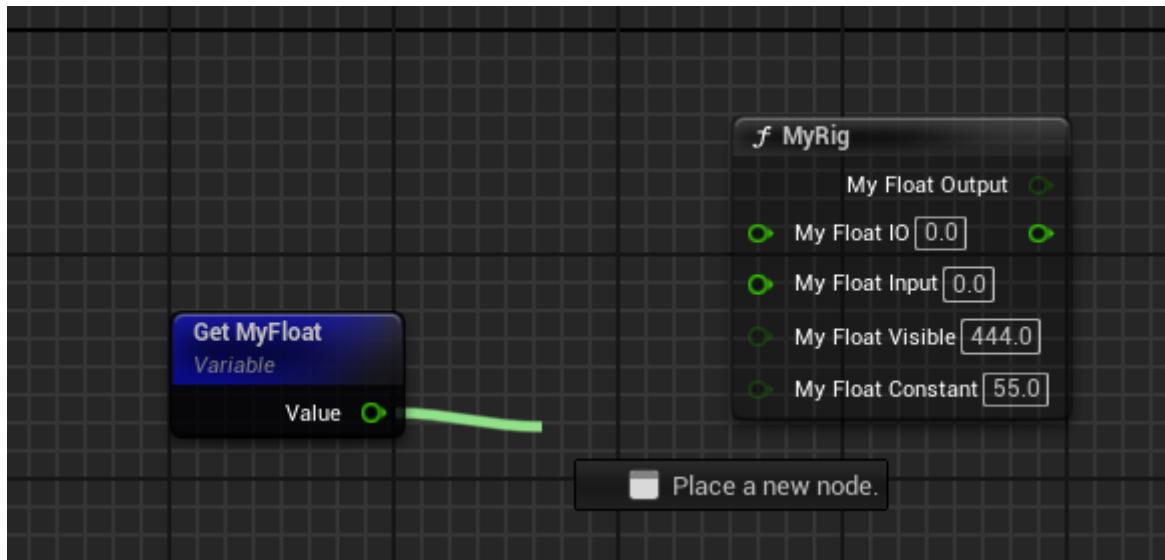
    UPROPERTY(meta = (Hidden))
    float MyFloat_Hidden;

    UPROPERTY(meta = (visible))
    float MyFloat_Visible;

    UPROPERTY(meta = (Input, Constant))
    float MyFloat_Constant;
};
```

## Test Effects:

Both "Visible" and "Input+Constant" not only have the same effect but also establish the attribute as a constant.



## Principle:

```
UENUM(BlueprintType)
enum class ERigVMPinDirection : uint8
{
    Input, // A const input value
    Output, // A mutable output value
    IO, // A mutable input and output value
    Visible, // A const value that cannot be connected to
    Hidden, // A mutable hidden value (used for internal state)
    Invalid // The max value for this enum - used for guarding.
};

FRigVMPinInfo::FRigVMPinInfo(FProperty* InProperty, ERigVMPinDirection InDirection, int32 InParentIndex, const uint8* InDefaultValueMemory)
{
    bIsConstant = InProperty->HasMetaData(TEXT("Constant"));
}

void URigVMController::ConfigurePinFromProperty(FProperty* InProperty, URigVMPin* InOutPin, ERigVMPinDirection InPinDirection) const
{
    InOutPin->bIsConstant = InProperty->HasMetaData(TEXT("Constant"));
}

bool URigVMPin::CanBeBoundToVariable(const FRigVMEExternalVariable& InExternalVariable, const FString& InSegmentPath) const
{
    if (bIsConstant)
    {
        return false;
    }
}
```

# Hidden

- **Function Description:** Specifies that the attribute under FRigUnit should be hidden
- **Usage Location:** UPROPERTY
- **Engine Module:** RigVMStruct
- **Metadata Type:** boolean
- **Restriction Type:** Attribute within FRigUnit
- **Associated Items:** Input
- **Commonality:** ★★★

# DetailsOnly

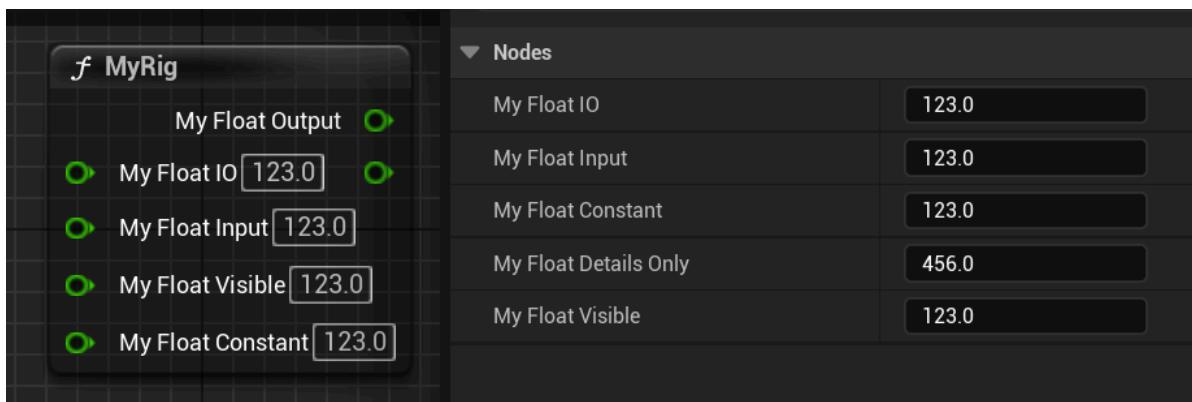
- **Function Description:** Specifies that the attribute under FRigUnit is displayed only in the Details panel.
- **Use Location:** UPROPERTY
- **Engine Module:** RigVMStruct
- **Metadata Type:** Boolean
- **Restriction Type:** Attributes within FRigUnit
- **Associated Items:** Input
- **Usage Frequency:** ★★★

Specifies that this attribute under FRigUnit is to be displayed exclusively in the Details panel.

## Test Code:

```
UPROPERTY(meta = (Input, DetailsOnly))
float MyFloat_DetailsOnly = 456.f;
```

## Test Results:



## Principle:

Based on the DetailsOnly attribute, determine whether to return ShowInDetailsPanelOnly.

```
bool URigVMPin::ShowInDetailsPanelOnly() const
{
```

```

#if WITH_EDITOR
    if (GetParentPin() == nullptr)
    {
        if (URigVMUnitNode* UnitNode = Cast<URigVMUnitNode>(GetNode()))
        {
            if (UScriptStruct* ScriptStruct = UnitNode->GetScriptStruct())
            {
                if (FProperty* Property = ScriptStruct-
>FindPropertyByName(GetFName()))
                {
                    if (Property->HasMetaData(FRigVMStruct::DetailsOnlyMetaName))
                    {
                        return true;
                    }
                }
            }
        }
        else if(const URigVMTemplateNode* TemplateNode = Cast<URigVMTemplateNode>(GetNode()))
        {
            if(const FRigVMTemplate* Template = TemplateNode->GetTemplate())
            {
                return !Template->GetArgumentMetaData(GetFName(),
FRigVMStruct::DetailsOnlyMetaName).IsEmpty();
            }
        }
    }
#endif
    return false;
}

```

## TemplateName

---

- **Function description:** Designates this FRigUnit as a generic template node.
- **Usage location:** USTRUCT
- **Engine module:** RigVMStruct
- **Metadata type:** string="abc"
- **Restriction type:** FRigUnit
- **Commonly used:** ★★★

Designates this FRigUnit as a generic template node.

When different FRigUnits are assigned the same TemplateName, they analyze the complete function signature of their Input and Output properties, ultimately identifying which properties are generic pins (i.e., properties with the same name but different types). During invocation, the input is the TemplateNode, which is the node formed by the TemplateName. Pins are then manually connected to determine the final function type, thus ultimately deciding which FRigUnit node should be applied in practice.

This feature is particularly useful when implementing logic that is identical but with slightly different parameter types. Typically, FRigUnit\_MyTemplate\_Float and FRigUnit\_MyTemplate\_Int inherit from a base class (though this is not mandatory), where common logic or properties are implemented.

## Test Code:

```
USTRUCT(meta = (DisplayName = "Set My float", TemplateName = "SetMyTemplate"))
struct INSIDER_API FRigUnit_MyTemplate_Float : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;

public:
    UPROPERTY(meta = (Input))
    float MyValue;

    UPROPERTY(meta = (Output))
    FString MyStringResult;
};

USTRUCT(meta = (DisplayName = "Set My int", TemplateName = "SetMyTemplate"))
struct INSIDER_API FRigUnit_MyTemplate_Int : public FRigUnit
{
    GENERATED_BODY()

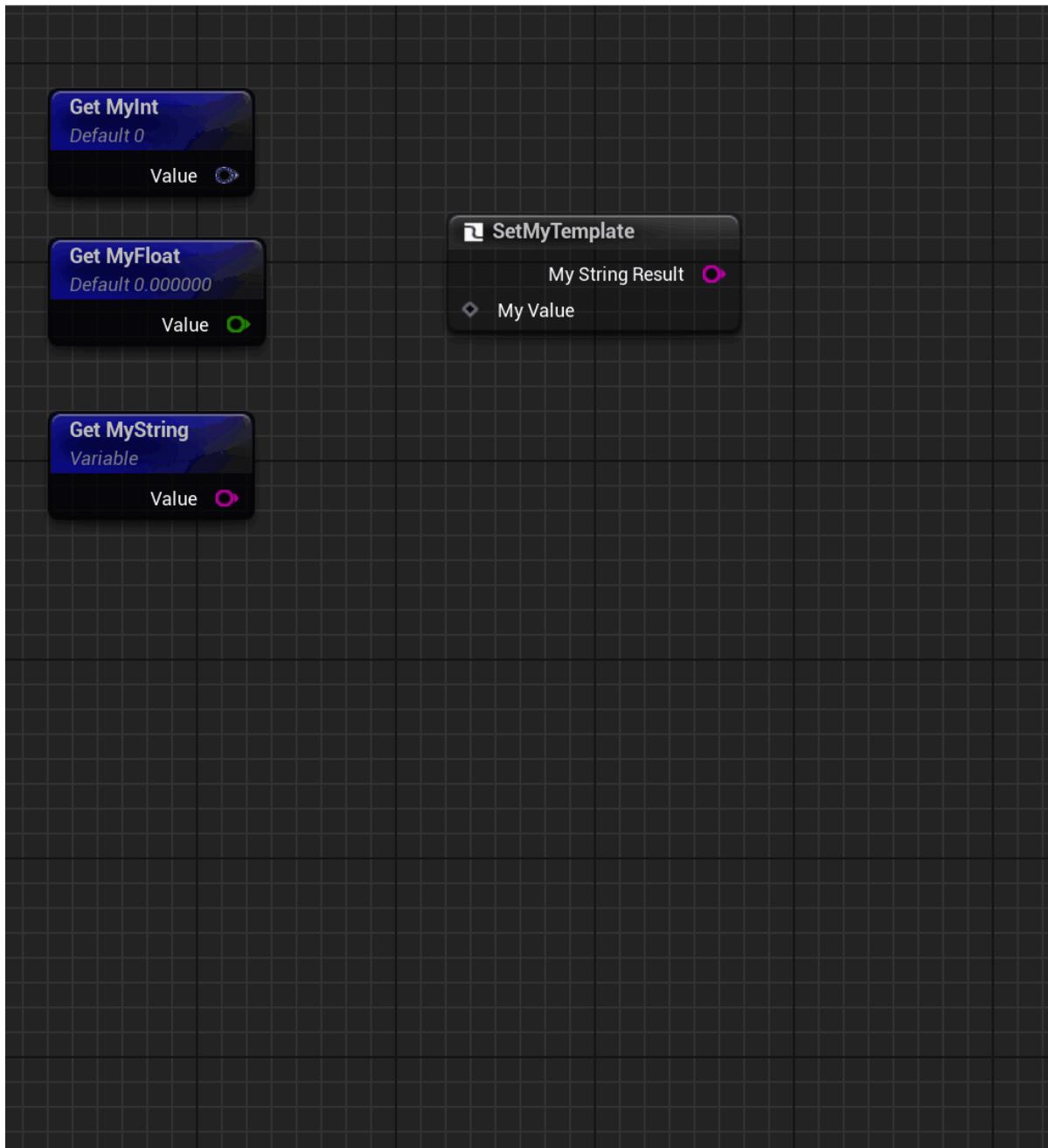
    RIGVM_METHOD()
        virtual void Execute() override;

public:
    UPROPERTY(meta = (Input))
    int32 MyValue;

    UPROPERTY(meta = (Output))
    FString MyStringResult;
};
```

## Test Results:

It can be observed that the initial node is SetMyTemplate, and depending on the pin type, it is actually resolved into either FRigUnit\_MyTemplate\_Float or FRigUnit\_MyTemplate\_Int. Since SetMyString has not been implemented, the FString type cannot be connected to the pin.



## Principle:

The source code involves a considerable amount of code related to this functionality. The general logic is that when FRigUnit initializes, it registers with FRigVMRegistry. If a TemplateName is present, a FRigTemplate is created. When creating via a right-click in the blueprint, a URigTemplateNameNode is actually created, which is then dispatched to the actual final node by FRigDispatch.

```
void FRigVMRegistry::Register(const TCHAR* InName, FRigVMFunctionPtr  
InFunctionPtr, UScriptStruct* InStruct, const TArray<FRigVMFunctionArgument>&  
InArguments)  
{  
    FString TemplateMetadata;  
    if (InStruct->GetStringMetaDataHierarchical(TemplateNameMetaName,  
&TemplateMetadata))  
    {  
    }  
}
```

# CustomWidget

- **Function Description:** Specifies that the properties within this FRigUnit should be edited using a custom control.
- **Usage Location:** UPROPERTY
- **Engine Module:** RigVMStruct
- **Metadata Type:** string="abc"
- **Restriction Type:** Attributes within FRigUnit
- **Commonality:** ★★

Indicates that the properties in the FRigUnit should be edited using a custom control.

The value for CustomWidget is chosen from a selection of options. These custom controls are already implemented within the engine.

The list of available options includes: BoneName, ControlName, SpaceName/NullName, CurveName, ElementName, ConnectorName, DrawingName, ShapeName, AnimationChannelName, MetadataName, MetadataTagName.

In the test code, BoneName is used solely for demonstration purposes:

```
USTRUCT(meta = (DisplayName = "MyRigCustomWidget"))
struct INSIDER_API FRigUnit_MyRigCustomWidget : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;
public:
    UPROPERTY(meta = (Input))
    FString MyString;

    UPROPERTY(meta = (Input, CustomWidget = "BoneName"))
    FString MyString_Custom;

    UPROPERTY(meta = (output))
    float MyFloat_Output = 123.f;
};
```

## Test Results:

The Pin type for MyString\_Custom is changed to a selectable list of BoneName.

## f MyRigCustomWidget

My Float Output 

 My String

 My String Custom   

Search

- None
- ball\_l
- ball\_r
- calf\_l
- calf\_r
- calf\_twist\_01\_l
- calf\_twist\_01\_r
- clavicle\_l
- clavicle\_r
- foot\_l
- foot\_r
- hand\_l
- hand\_r
- head
- ik\_foot\_l
- ik\_foot\_r
- ik\_foot\_root
- ik\_hand\_gun
- ik\_hand\_l
- ik\_hand\_r
- ik\_hand\_root
- index\_01\_l
- index\_01\_r
- index\_02\_l
- index\_02\_r
- index\_03\_l
- index\_03\_r
- lowerarm\_l
- lowerarm\_r
- lowerarm\_twist\_01\_l

## Principle:

```
TSharedPtr<SGraphPin>
FControlRigGraphPanelPinFactory::CreatePin_Internal(UEdGraphPin* InPin) const
{
    if (CustomWidgetName == TEXT("BoneName"))
    {
        return SNew(SRigVMGraphPinNameList, InPin)
            .ModelPin(ModelPin)
            .OnGetNameFromSelection_UObject(RigGraph,
&UControlRigGraph::GetSelectedElementsNameList)
            .OnGetNameListContent_UObject(RigGraph,
&UControlRigGraph::GetBoneNameList)
            .OnGetSelectedClicked_UObject(RigGraph,
&UControlRigGraph::HandleGetSelectedClicked)
            .OnBrowseClicked_UObject(RigGraph,
&UControlRigGraph::HandleBrowseClicked);
    }
    //Additional details to follow
}
```

## ExpandByDefault

- **Function Description:** Defaults to expanding the attribute pins within FRigUnit.
- **Usage Location:** UPROPERTY
- **Engine Module:** RigVMSStruct
- **Metadata Type:** boolean
- **Commonly Used:** ★★★

Defaults to expanding the attribute pins within FRigUnit.

## Test Code:

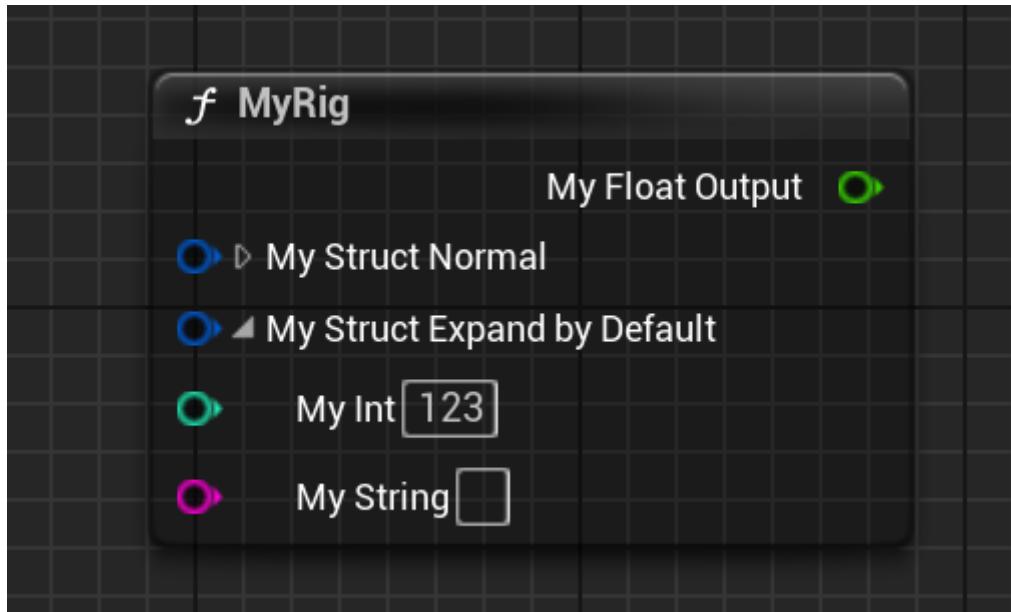
```
USTRUCT(meta = (DisplayName = "MyRig"))
struct INSIDER_API FRigUnit_MyRig : public FRigUnit
{
    UPROPERTY(meta = (Input))
    FMyCommonStruct MyStruct_Normal;

    UPROPERTY(meta = (Input, ExpandByDefault))
    FMyCommonStruct MyStruct_ExpandByDefault;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
}
```

## Test Code:

Visible that MyStruct\_ExpandByDefault expands the structure by default.



## 测试效果:

Identifies the metadata and sets the bIsExpanded state for the pin.

```
FRigVMPinInfo::FRigVMPinInfo(FProperty* InProperty, ERigVMPinDirection InDirection, int32 InParentIndex, const uint8* InDefaultValueMemory)
{
    if (InProperty->HasMetaData(FRigVMStruct::ExpandPinByDefaultMetaName))
    {
        bIsExpanded = true;
    }
}
```

## Aggregate

- **Function Description:** Specifies that an attribute pin within an FRigUnit can serve as an operand for an extensible continuous binary operator.
- **Usage Location:** UPROPERTY
- **Engine Module:** RigVMStruct
- **Metadata Type:** bool
- **Restriction Type:** Attributes under FRigUnit
- **Commonly Used:** ★★★

Specifies that the attribute pin in an FRigUnit is to be used as an operand for an extensible continuous binary operator.

Remember to add Aggregate to both the Input and Output.

## Test Code:

```
USTRUCT(meta = (DisplayName = "MyRigAggregate"))
struct INSIDER_API FRigUnit_MyRigAggregate : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;

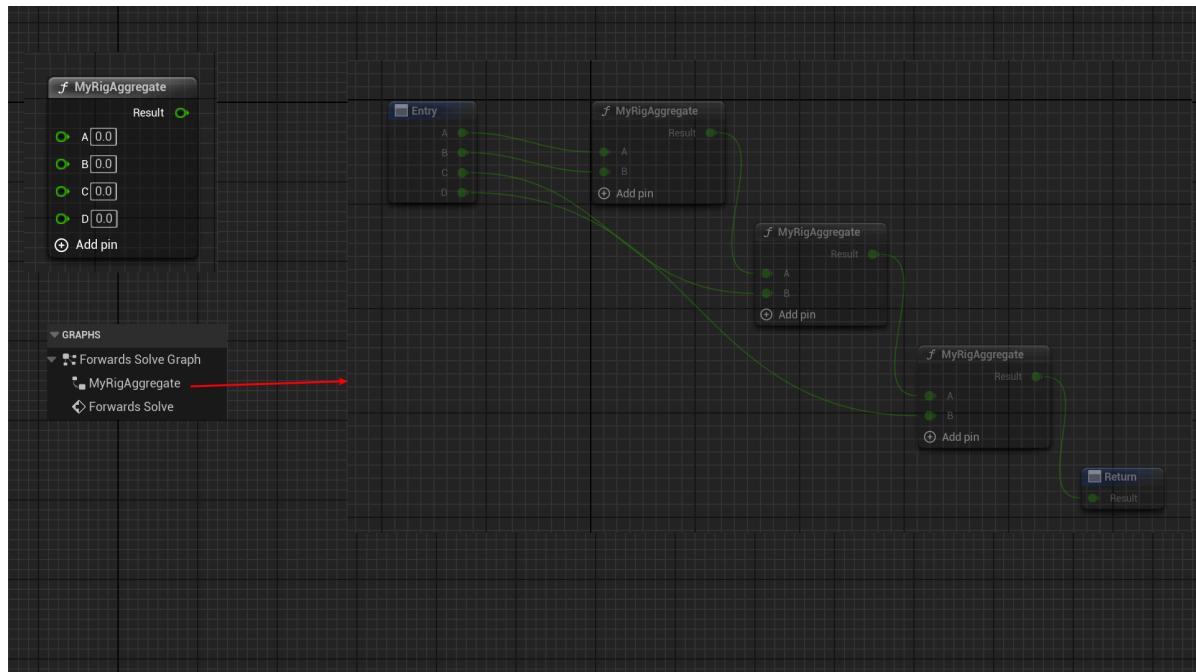
public:
    UPROPERTY(meta = (Input, Aggregate))
    float A = 0.f;

    UPROPERTY(meta = (Input, Aggregate))
    float B = 0.f;

    UPROPERTY(meta = (Output, Aggregate))
    float Result = 0.f;
};
```

## Test Results:

It is apparent that after adding Aggregate, dynamic AddPin can be continued on the blueprint node. Additionally, a MyRigAggregate node is created in the Graph on the left. Upon opening it, you can observe that it essentially continues to assemble the original binary operations to achieve the effect of further AddPin.



## Principle:

Identify the Meta and then add the pins to AggregateInputs and AggregateOutputs.

```
TArray<URigVMPin*> URigVMUnitNode::GetAggregateInputs() const
{
    TArray<URigVMPin*> AggregateInputs;
    #if UE_RIGVM.Aggregate_Nodes_Enabled
```

```

    if (const UScriptStruct* Struct = GetScriptStruct())
    {
        for (URigVMPin* Pin : GetPins())
        {
            if (Pin->GetDirection() == ERigVMPinDirection::Input)
            {
                if (const FProperty* Property = Struct->FindPropertyByName(Pin->GetFName()))
                {
                    if (Property->HasMetaData(FRigVMStruct::AggregateMetaName))
                    {
                        AggregateInputs.Add(Pin);
                    }
                }
            }
        }
    }
    else
    {
        return Super::GetAggregateInputs();
    }
#endif
    return AggregateInputs;
}

```

## Varying

---

- **Function Description:** ScriptStruct /Script/RigVM.RigVMFunction\_GetDeltaTime
- **Usage Location:** UCLASS
- **Engine Module:** RigVMStruct
- **Metadata Type:** bool
- **Commonality:** 0

When placed on USTRUCT, it was discovered being used in functions like IsDefinedAsVarying, but no calls were found using F5.

## MenuDescSuffix

---

- **Function Description:** Identifies the name suffix for the FRigUnit in the context menu of the Blueprint when right-clicked.
- **Usage Location:** USTRUCT
- **Engine Module:** RigVMStruct
- **Metadata Type:** Boolean
- **Restriction Type:** Applicable to FRigUnit type
- **Commonly Used:** ★★★

Identifies the name suffix for the FRigUnit in the context menu of the Blueprint when right-clicked.

## Test Code:

```
USTRUCT(meta = (DisplayName = "MyRigSuffix", MenuDescSuffix = "(MyVector)"))
struct INSIDER_API FRigUnit_MyRigSuffix: public FRigUnit
{
    GENERATED_BODY()

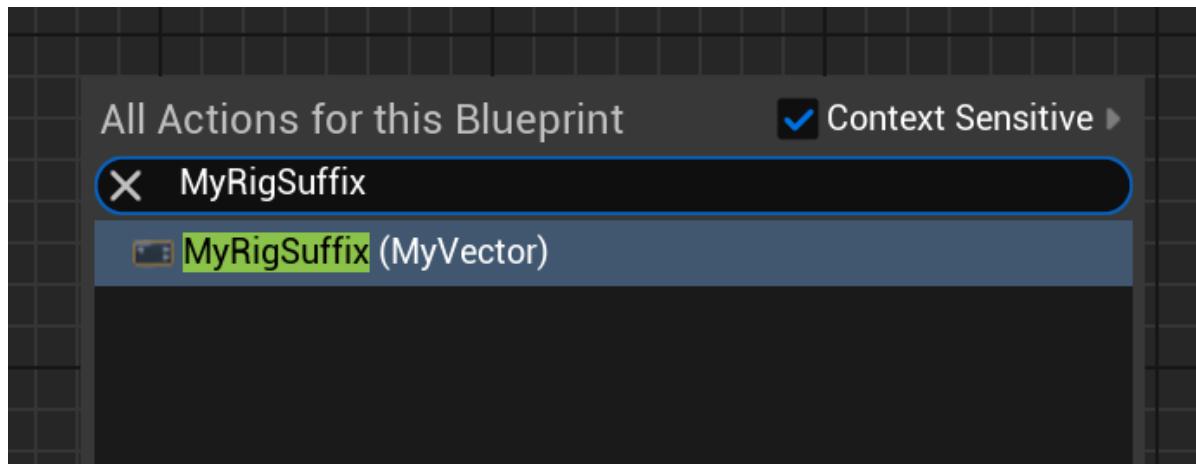
    RIGVM_METHOD()
        virtual void Execute() override;

public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};
```

## Test Effects:

It is visible that the suffix "(MyVector)" has appeared.



## Principle:

The data is obtained and then appended to the DisplayName.

```
FString CategoryMetadata, DisplayNameMetadata, MenuDescSuffixMetadata;
Struct->GetStringMetaDataHierarchical(FRigVMStruct::CategoryMetaName,
&CategoryMetadata);
Struct->GetStringMetaDataHierarchical(FRigVMStruct::DisplayNameMetaName,
&DisplayNameMetadata);
Struct->GetStringMetaDataHierarchical(FRigVMStruct::MenuDescSuffixMetaName,
&MenuDescSuffixMetadata);

if(DisplayNameMetadata.IsEmpty())
{
    DisplayNameMetadata = Struct->GetDisplayNameText().ToString();
}
if (!MenuDescSuffixMetadata.IsEmpty())
{
    MenuDescSuffixMetadata = TEXT(" ") + MenuDescSuffixMetadata;
}
```

```
FText MenuDesc = FText::FromString(DisplayNameMetadata + MenuDescSuffixMetadata);
```

# NodeColor

- **Function Description:** Specifies the RGB color value for the FRigUnit Blueprint node.
- **Usage Location:** USTRUCT
- **Engine Module:** RigVMStruct
- **Metadata Type:** string="abc"
- **Restriction Type:** FRigUnit
- **Commonliness:** ★★

Specifies the RGB color value of the FRigUnit Blueprint node.

## Test Code:

```
USTRUCT(meta = (DisplayName = "MyRigColor", NodeColor="1 0 0"))
struct INSIDER_API FRigUnit_MyRigColor : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;
public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};
```

## Testing Code:

With NodeColor added, the color transitions from left to right.



## 测试效果:

Extracts the color value from the Meta data.

```

FLinearColor FRigVMDispatchFactory::GetNodeColor() const
{
    if(const UScriptStruct* ScriptStruct = GetScriptStruct())
    {
        FString NodeColor;
        if (ScriptStruct->GetStringMetaDataHierarchical(FRigVMStruct::NodeColorMetaName, &NodeColor))
        {
            return FRigVMTemplate::GetColorFromMetadata(NodeColor);
        }
    }
    return FLinearColor::White;
}

```

## Icon

- **Function description:** Sets the icon for the FRigUnit blueprint node.
- **Usage location:** USTRUCT
- **Engine module:** RigVMStruct
- **Metadata type:** string="abc"
- **Restriction type:** FRigUnit
- **Commonality:** ★★

Sets the icon for the FRigUnit blueprint node.

As indicated by the comments in the source code, the format for the Icon is "StyleSetName|StyleName|SmallStyleName|StatusOverlayStyleName", with the last two being optional. For more information, refer to the FSlateIcon documentation.

## Test Code:

```

USTRUCT(meta = (DisplayName =
"MyRigIcon", Icon="EditorStyle|GraphEditor.Macro.ForEach_16x"))
struct INSIDER_API FRigUnit_MyRigIcon: public FRigUnit
{
    GENERATED_BODY()

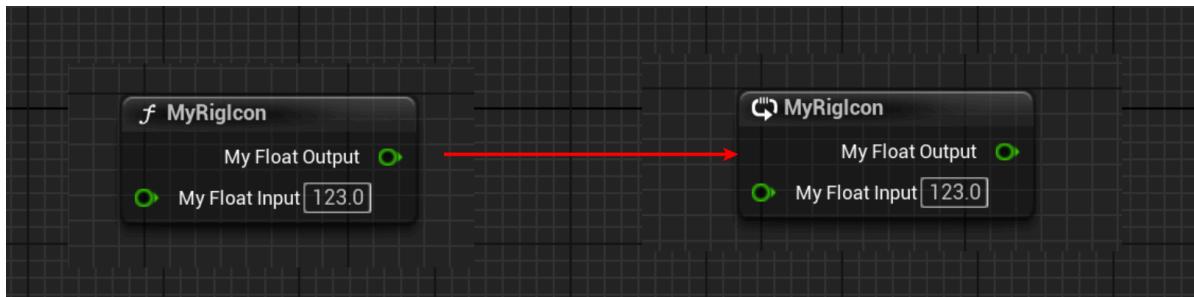
    RIGVM_METHOD()
    virtual void Execute() override;
public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};

```

## Test Results:

It is evident that after adding the Icon, the upper-left corner icon changes to a different one, no longer the default function target 'f'.



## Principle:

```
FSlateIcon URigVMEdGraphNode::GetIconAndTint(FLinearColor& outColor) const
{
    if(MetadataScriptStruct && MetadataScriptStruct->HasMetaDataHierarchical(FRigVMStruct::IconMetaName))
    {
        FString IconPath;
        const int32 NumOfIconPathNames = 4;

        FName IconPathNames[NumOfIconPathNames] = {
            NAME_None, // StyleSetName
            NAME_None, // StyleName
            NAME_None, // SmallStyleName
            NAME_None // StatusOverlayStyleName
        };

        // icon path format:
        styleSetName|styleName|smallStyleName|statusOverlayStyleName
        // the last two names are optional, see FSlateIcon() for reference
        MetadataScriptStruct->GetStringMetaDataHierarchical(FRigVMStruct::IconMetaName, &IconPath);
        return FSlateIcon(IconPathNames[0], IconPathNames[1],
        IconPathNames[2], IconPathNames[3]);
    }
}
```

## Obsolete

- **Function Description:** Indicates that this FRigUnit is deprecated and will not be displayed in the blueprint's right-click menu.
- **Usage Location:** USTRUCT
- **Engine Module:** RigVMStruct
- **Metadata Type:** bool
- **Type Constraint:** Applicable to FRigUnit type
- **Commonality:** ★★

Indicates that this FRigUnit is deprecated and will not be shown in the blueprint's right-click menu.

However, if it has already been used in a blueprint, it can continue to be used.

Attention should be given to implementing GetUpgradeInfo() accordingly, otherwise an error will occur.

## Test Code:

```
USTRUCT(meta = (DisplayName = "MyRigDeprecated", Deprecated))
struct INSIDER_API FRigUnit_MyRigDeprecated : public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
    virtual void Execute() override;

    RIGVM_METHOD()
    virtual FRigVMStructUpgradeInfo GetUpgradeInfo() const override;
public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};
```

## Test Outcomes:



## Principle:

Deprecated nodes are skipped when constructing menu items.

```
void FRigVMEditorModule::GetTypeActions(URigVMBLueprint* RigVMBLueprint,
FBlueprintActionDatabaseRegistrar& ActionRegistrar)
{
    // Add all rig units
    for(const FRigVMFunction& Function : Registry.GetFunctions())
    {
        // skip deprecated units
        if(Function.Struct->HasMetaData(FRigVMStruct::DeprecatedMetaName))
        {
            continue;
        }
    }
}
```

# Abstract

- **Function Description:** Identifies this FRigUnit as an abstract class, which does not require the implementation of Execute.
- **Usage Location:** USTRUCT
- **Engine Module:** RigVMStruct
- **Metadata Type:** bool
- **Restriction Type:** Applicable to FRigUnit type
- **Commonly Used:** ★★

Indicates that this FRigUnit is an abstract class and does not need to implement Execute, often serving as a base class for other FRigUnit classes.

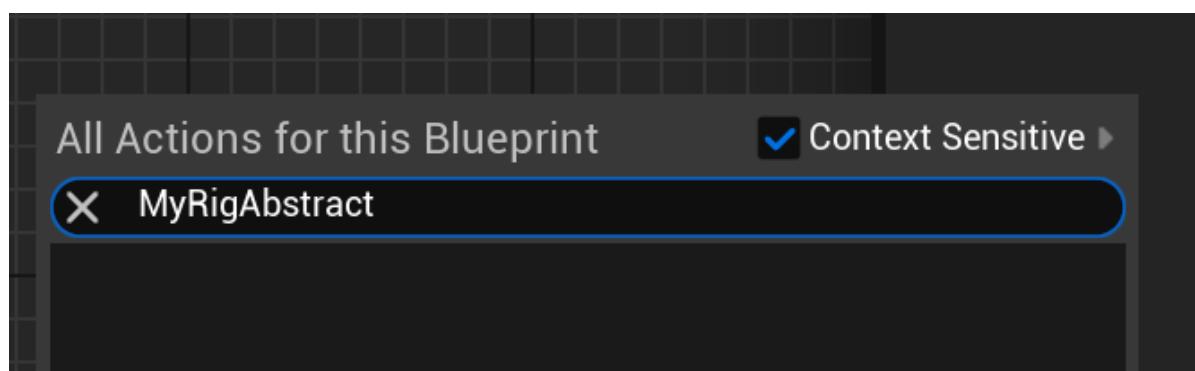
However, if Execute is still implemented, it can indeed be invoked within blueprints.

## Test Code:

```
USTRUCT(meta = (DisplayName = "MyRigAbstract", Abstract))
struct INSIDER_API FRigUnit_MyRigAbstract: public FRigUnit
{
    GENERATED_BODY()
public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};
```

## Test Results:



## Principle:

During certain internal processes, this abstract base class will naturally be overlooked.

```
void FRigVMBuildingBlocks::ForAllRigVMStructs(TFunction<void(UScriptStruct*)>
InFunction)
{
    // Run over all unit types
    for(TObjectIterator<USTRUCT> StructIt; StructIt; ++StructIt)
    {
```

```

    if (*StructIt)
    {
        if(StructIt->IsChildOf(FRigVMStruct::StaticStruct()) && !StructIt-
>HasMetaData(FRigVMStruct::AbstractMetaName))
        {
            if (UScriptStruct* ScriptStruct = Cast<UScriptStruct>(*StructIt))
            {
                InFunction(ScriptStruct);
            }
        }
    }
}

```

## RigVMTypAllowed

- **Function Description:** Specifies a UENUM that can be selected for the UEnum\* property within an FRigUnit.
- **Usage Location:** UENUM
- **Engine Module:** RigVMStruct
- **Metadata Type:** bool
- **Commonality:** ★★

Specifies that a UENUM can be chosen for the UEnum\* attribute of an FRigUnit.

### Test Code:

```

UENUM(BlueprintType)
enum class ERigMyEnum : uint8
{
    First,
    Second,
    Third,
};

UENUM(BlueprintType, meta = (RigVMTypAllowed))
enum class ERigMyEnumAllowed : uint8
{
    Cat,
    Dog,
    Tiger,
};

USTRUCT(meta = (DisplayName = "MyRigEnum"))
struct INSIDER_API FRigUnit_MyRigEnum : public FRigUnit
{
    GENERATED_BODY()

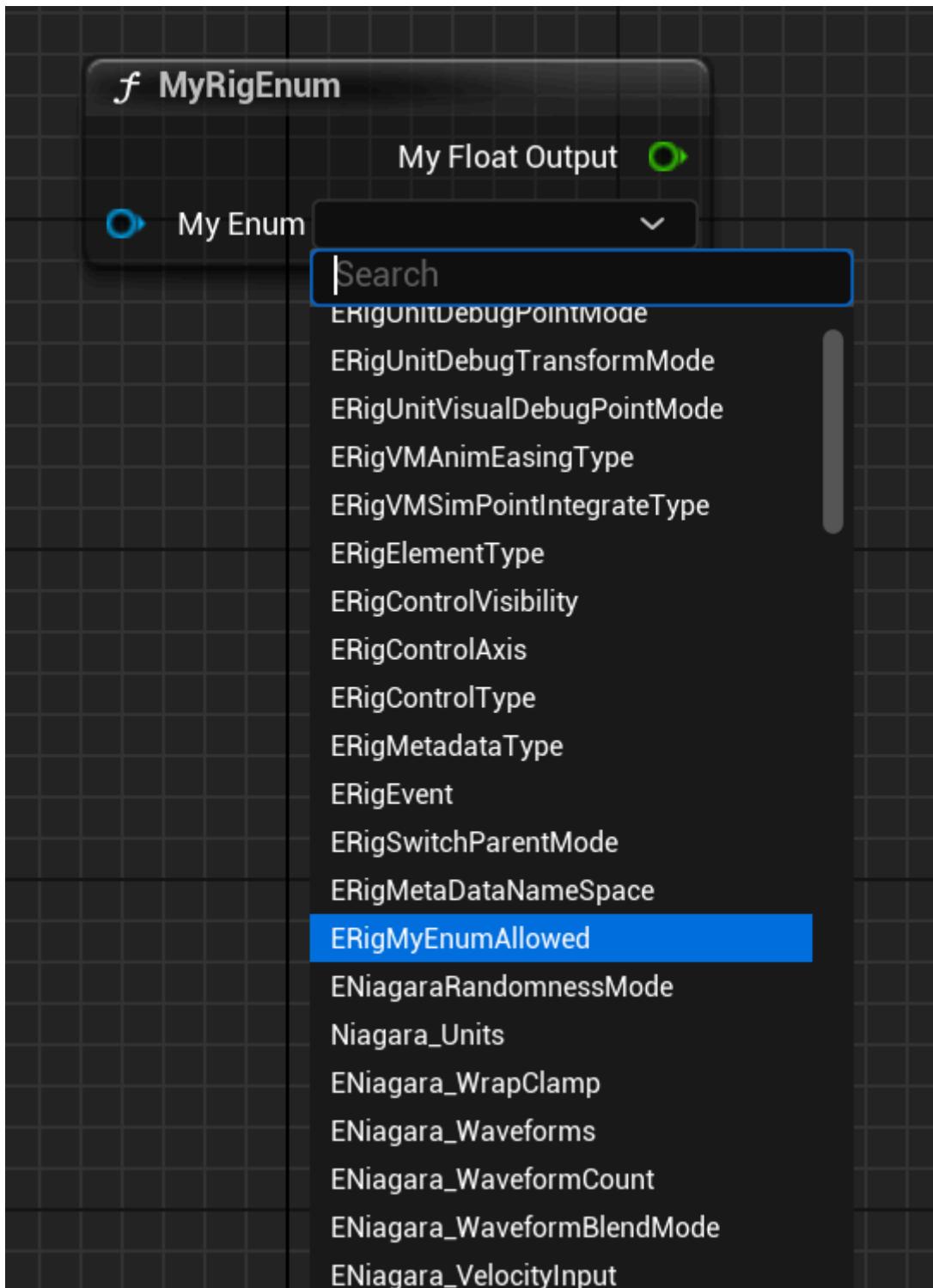
    RIGVM_METHOD()
        virtual void Execute() override;
public:
    UPROPERTY(meta = (Input))

```

```
UEnum* MyEnum;  
  
UPROPERTY(meta = (Output))  
float MyFloat_Output = 123.f;  
};
```

## Test Code:

Only ERigMyEnumAllowed is visible in the options list, with ERigMyEnum being absent.



## 测试效果：

When generating options, check if !Enum->IsAsset() indicates an enumeration in C++, and then RigVMTypAllowed must be present.

```
void SRigVMEnumPicker::PopulateEnumOptions()
{
    EnumOptions.Reset();
    EnumOptions.Add(MakeShareable(new FString(TEXT("None"))));
    for (TObjectIterator<UEnum> EnumIt; EnumIt; ++EnumIt)
    {
        UEnum* Enum = *EnumIt;

        if (Enum->HasAnyFlags(RF_BeginDestroyed | RF_FinishDestroyed) || !Enum->HasAllFlags(RF_Public))
        {
            continue;
        }

        // Any asset based enum is valid
        if (!Enum->IsAsset())
        {
            // Native enums only allowed if contain RigVMTypAllowed metadata
            if (!Enum->HasMetaData(TEXT("RigVMTypAllowed")))
            {
                continue;
            }
        }

        EnumOptions.Add(MakeShareable(new FString(Enum->GetPathName())));
    }
}
```

## Keywords

- **Function Description:** Defines keywords for FRigUnit blueprint nodes in the right-click menu to simplify search and input.
- **Usage Location:** USTRUCT
- **Engine Module:** RigVMStruct
- **Metadata Type:** strings = "a, b, c"
- **Restriction Type:** FRigUnit
- **Commonality:** ★★★

Settings the keywords for FRigUnit blueprint nodes in the right-click menu to make it easier to search and input.

Functions similarly to the Keywords on Functions.

## Test Code:

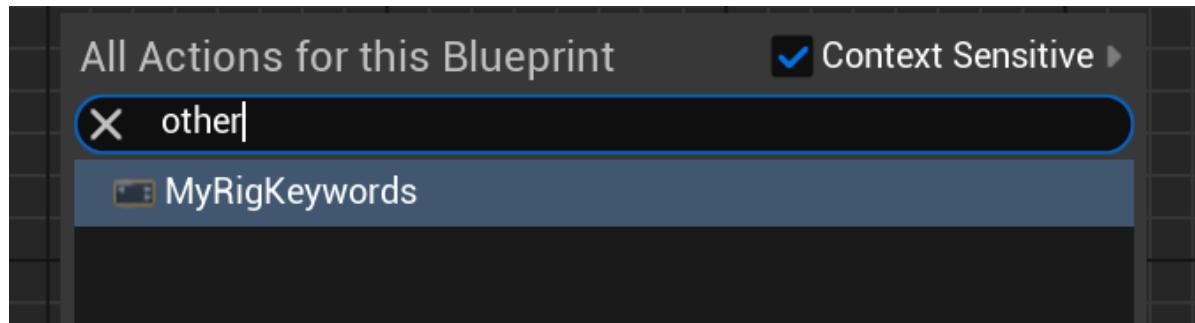
```
USTRUCT(meta = (DisplayName = "MyRigKeywords", Keywords="MyKey,Otherword"))
struct INSIDER_API FRigUnit_MyRigKeywords: public FRigUnit
{
    GENERATED_BODY()

    RIGVM_METHOD()
        virtual void Execute() override;
public:
    UPROPERTY(meta = (Input))
    float MyFloat_Input = 123.f;

    UPROPERTY(meta = (Output))
    float MyFloat_Output = 123.f;
};
```

## Test Effects:

Nodes can be located by entering the characters specified in the Keywords.



## Principle:

```
URigVMEdGraphUnitNodeSpawner*
URigVMEdGraphUnitNodeSpawner::CreateFromStruct(UscriptStruct* InStruct, const
FName& InMethodName, const FText& InMenuDesc, const FText& InCategory, const
FText& InTooltip)
{
    FString KeywordsMetadata, TemplateNameMetadata;
    InStruct->GetStringMetaDataHierarchical(FRigVMStruct::KeywordsMetaName,
    &KeywordsMetadata);
    if(!TemplateNameMetadata.IsEmpty())
    {
        if(KeywordsMetadata.IsEmpty())
        {
            KeywordsMetadata = TemplateNameMetadata;
        }
        else
        {
            KeywordsMetadata = KeywordsMetadata + TEXT(",") +
            TemplateNameMetadata;
        }
    }
    MenuSignature.Keywords = FText::FromString(KeywordsMetadata);
```

```
}
```

# MakeEditWidget

- **Function Description:** Allows FVector and FTransform to appear as draggable widgets within the scene editor.
- **Usage Location:** UPROPERTY
- **Engine Module:** Scene
- **Metadata Type:** bool
- **Restricted Types:** FVector, FTransform
- **Associated Items:** ValidateWidgetUsing
- **Commonly Used:** ★★★

Enables FVector and FTransform to appear as draggable widgets within the scene editor.

Offering a more intuitive experience compared to direct numerical editing.

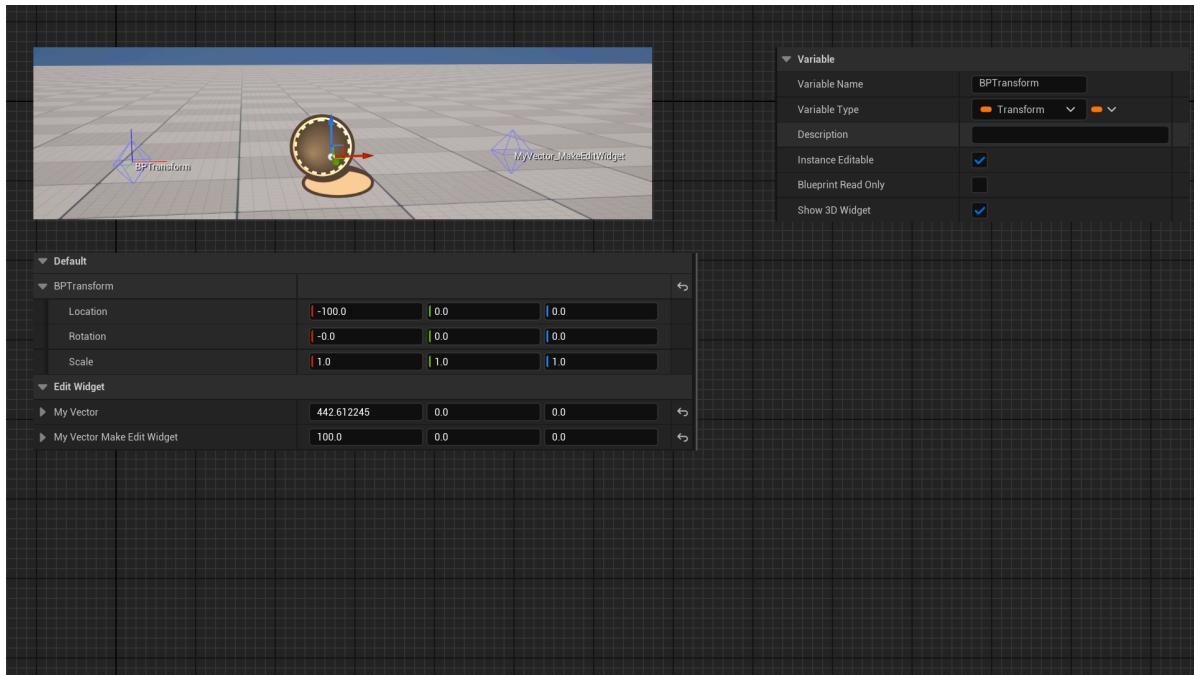
## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API AMyActor_Editwidget :public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="EditWidget")
    FVector MyVector;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="EditWidget", meta=(MakeEditWidget))
    FVector MyVector_MakeEditwidget;
};
```

## Test Results:

In the AMyActor\_EditWidget subclass inherited in the blueprint, adding another FTransform variable allows the "Show 3D Widget" option to be visible. Both this and MyVector\_MakeEditWidget are represented as draggable widgets within the scene.



## Principle:

If it is determined to be FVector or FTransform and possesses the MakeEditWidget attribute, a widget can be created.

```
/** Value of UPROPERTY can be edited with a widget in the editor (translation,
rotation) */
static UNREALED_API const FName MD_MakeEditWidget;
/** Specifies a function used for validation of the current value of a property.
The function returns a string that is empty if the value is valid, or contains an
error description if the value is invalid */
static UNREALED_API const FName MD_ValidatewidgetUsing;

bool FLegacyEdModeWidgetHelper::CanCreateWidgetForStructure(const Ustruct*
InPropStruct)
{
    return InPropStruct && (InPropStruct->GetFName() == NAME_Vector ||
InPropStruct->GetFName() == NAME_Transform);
}

bool FLegacyEdModeWidgetHelper::ShouldCreateWidgetForProperty(FProperty* InProp)
{
    return CanCreateWidgetForProperty(InProp) && InProp-
>HasMetaData(MD_MakeEditWidget);
}
```

## ValidateWidgetUsing

- Function Description:** Provides a function to validate the legitimacy of the current attribute value.
- Usage Location:** UPROPERTY
- Engine Module:** Scene
- Metadata Type:** bool

- **Restriction Type:** FVector and FTransform with MakeEditWidget attribute
- **Associated Items:** MakeEditWidget
- **Commonliness:** ★★★

ValidateWidgetUsing offers a function to validate the legitimacy of the current property value.

- The current property must be tagged with MakeEditWidget
- The function prototype is FString MyFunc(), which returns a non-empty string to indicate an error message.

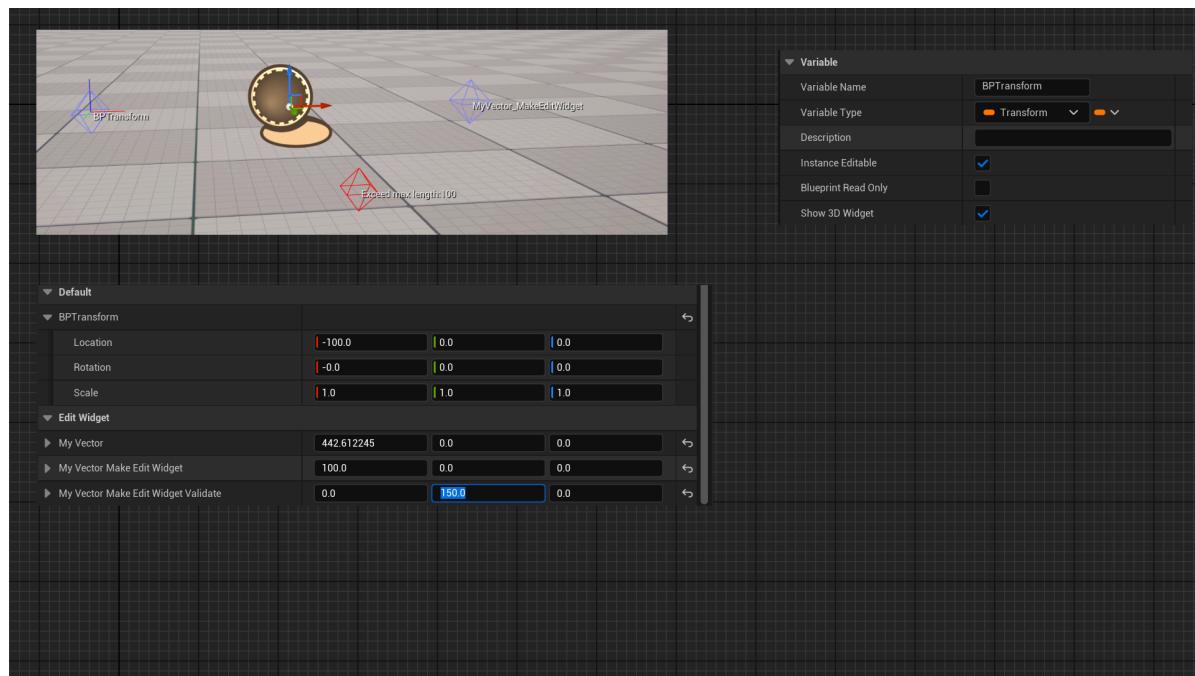
## Test Code:

```
UFUNCTION()
FString validateMyVector()
{
    if (MyVector_MakeEditWidget_validate.Length()>100.f)
    {
        return TEXT("Exceed max length:100");
    }
    return TEXT("");
}

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Editwidget", meta =
(MakeEditWidget, ValidateWidgetUsing = "ValidateMyVector"))
FVector MyVector_MakeEditWidget_validate;
```

## Test Results:

Visible MyVector\_MakeEditWidget\_Validate changes the control color to red and displays an error message on the control when the length exceeds 100.



# Principle:

The logic is straightforward. If a validation function is detected, it is invoked to perform the validation. If an error message is present, the final output color and text are altered accordingly.

```
static FLegacyEdModeWidgetHelper::FPropertyWidgetInfo CreateWidgetInfo(const
TArray<FPropertyWidgetInfoChainElement>& Chain, bool bIsTransform, FProperty*
CurrentProp, int32 Index = INDEX_NONE)
{
    check(CurrentProp);
    FEdMode::FPropertyWidgetInfo WidgetInfo;
    WidgetInfo.PropertyValidationName = FName(*CurrentProp-
>GetMetaData(FEdMode::MD_ValidateWidgetUsing));

    return WidgetInfo;
}

void
FLegacyEdModeWidgetHelper::FPropertyWidgetInfo::GetTransformAndColor(UObject*
BestSelectedItem, bool bIsSelected, FTransform& OutLocalTransform, FString&
OutValidationMessage, FColor& OutDrawColor) const
{
    // Determine the desired color
    if (PropertyValidationName != NAME_None)
    {
        if (UFunction* ValidateFunc = BestSelectedItem-
>FindFunction(PropertyValidationName))
        {
            BestSelectedItem->ProcessEvent(ValidateFunc, &OutValidationMessage);

            // if we have a negative result, the widget color is red.
            OutDrawColor = OutValidationMessage.IsEmpty() ? OutDrawColor :
FColor::Red;
        }
    }
}

void FLegacyEdModeWidgetHelper::DrawHUD(FEditorViewportClient* ViewportClient,
FViewport* Viewport, const FSceneView* View, FCanvas* Canvas)
{
    FTransform LocalWidgetTransform;
    FString ValidationMessage;
    FColor WidgetColor;
    WidgetInfo.GetTransformAndColor(BestSelectedItem, bSelected, /*out*/
LocalWidgetTransform, /*out*/ ValidationMessage, /*out*/ WidgetColor);

    Canvas->DrawItem(TextItem);
}
```

# AllowedLocators

- **Function Description:** Used for Sequencer to locate objects that can be bound
- **Usage Location:** UPROPERTY
- **Engine Module:** Scene
- **Metadata Type:** string="abc"
- **Restriction Type:** FUniversalObjectLocator
- **Commonly Used:** ★

Used to identify objects that can be bound for the Sequencer.

Appears to be an auxiliary locator used by the Sequencer for binding properties to Actors. It is typically only utilized within the predefined attribute FUniversalObjectLocator, and generally, there is no need to expand this section; hence, it is marked as OnlyInternal.

## Searchable in the source code:

```
// Helper struct for Binding Properties UI for locators.  
USTRUCT()  
struct FMovieSceneUniversalLocatorInfo  
{  
    GENERATED_BODY()  
  
    // Locator for the entry  
    UPROPERTY(EditAnywhere, Category = "Default", meta=(AllowedLocators="Actor"))  
    FUniversalObjectLocator Locator;  
  
    // Flags for how to resolve the locator  
    UPROPERTY()  
    ELocatorResolveFlags ResolveFlags = ELocatorResolveFlags::None;  
};
```

Seems to be the type of object that is allowed to be located.

## Principle:

```
TMap<FName, TSharedPtr<ILocatorEditor>> ApplicableLocators;  
  
void  
FUniversalObjectLocatorCustomization::CustomizeHeader(TSharedRef<IPROPERTYHandle>  
StructPropertyHandle, FDetailWidgetRow& HeaderRow,  
IPropertyTypeCustomizationUtils& StructCustomizationUtils)  
{  
    TArray< FString > AllowedTypes;  
    if (PropertyHandle->HasMetaData("AllowedLocators"))  
    {  
        PropertyHandle->  
        GetMetaData("AllowedLocators").ParseIntoArray(AllowedTypes, TEXT(","));  
    }  
}
```

```

FUniversalObjectLocatorEditorModule& Module =
FModuleManager::Get().LoadModuleChecked< FUniversalObjectLocatorEditorModule>
("UniversalObjectLocatorEditor");
for (TPair< FName, TSharedPtr< ILocatorEditor>> Pair : Module.LocatorEditors)
{
    if (AllowedTypes.Num() == 0 || AllowedTypes.Contains(Pair.Key.ToString()))
    {
        ApplicableLocators.Add(Pair.Key, Pair.Value);
    }
}

```

## ScriptName

- **Function Description:** The name used when exporting to script
- **Usage Location:** Any
- **Engine Module:** Script
- **Metadata Type:** string="abc"
- **Commonly Used:** ★★★

Specify the name to be used when exporting to script.

- Can be used with UCLASS, USTRUCT, UENUM, UFUNCTION, and UPROPERTY to change the exported script name.
- If no custom name is provided using ScriptName, the exported name will be the default Pythonized name. For instance, MyFunc() becomes my\_func().

When testing Python, remember to enable the Python plugin.

You can find numerous pre-written test cases in  
 \UnrealEngine\Engine\Plugins\Experimental\PythonScriptPlugin\Source\PythonScriptPlugin\Private\PyTest.h.

## Test Code:

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPythonTestLibrary2 :public UBlueprintFunctionLibrary
{
    GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType, meta=(ScriptName="MyPythonLib"))
class INSIDER_API UMyPythonTestLibrary :public UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    //unreal.MyPythonLib.my_script_func_default()
    UFUNCTION(BlueprintCallable, meta=())
    static void MyScriptFuncDefault()
    {
        UInsiderSubsystem::Get().PrintStringEx(nullptr,
TEXT("MyScriptFuncDefault"));
    }
}

```

```
//unreal.MyPythonLib.my_script_func()
UFUNCTION(BlueprintCallable,meta=(ScriptName="MyScriptFunc"))
static void MyScriptFunc_ScriptName()
{
    UInsiderSubsystem::Get().PrintStringEx(nullptr,
TEXT("MyScriptFunc_ScriptName"));
}
};
```

## Test Results:

After opening the editor, the engine will automatically generate the corresponding export-to-Python glue code based on type data reflection. You can view the exported script code for the classes we define in C++ in \Intermediate\PythonStub[unreal.py](<http://unreal.py/>).

For the class mentioned above, the Python code generated in unreal.py is as follows:

- As seen, UMyPythonTestLibrary2 does not have a ScriptName and retains the default name, while UMyPythonTestLibrary's name is changed to MyPythonLib.
- The export script name for MyScriptFuncDefault is my\_script\_func\_default, and because MyScriptFunc\_ScriptName has a ScriptName specified, it becomes MyScriptFunc

```
class MyPythonTestLibrary2(BlueprintFunctionLibrary):
    r"""
    My Python Test Library 2

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPythonTest.h

    ...
    ...

class MyPythonLib(BlueprintFunctionLibrary):
    r"""
    My Python Test Library

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_Test.h

    ...
    @classmethod
    def my_script_func_default(cls) -> None:
        r"""
        X.my_script_func_default() -> None
        My Script Func Default
        ...
        ...

    @classmethod
    def my_script_func(cls) -> None:
```

```

r"""
X.my_script_func() -> None
My Script Func Script Name
"""

...

```

## Principle:

When retrieving the name of each type, the ScriptName is checked first. If it is present, that name is used. Otherwise, the name is Pythonized in GetFieldPythonNameImpl.

```

\Engine\Plugins\Experimental\PythonScriptPlugin\Source\PythonScriptPlugin\Private
\PyGenUtil.cpp

const FName ScriptNameMetaDataKey = TEXT("ScriptName");

FString GetClassPythonName(const UClass* InClass)
{
    return GetFieldPythonNameImpl(InClass, ScriptNameMetaDataKey);
}

TArray<FString> GetDeprecatedClassPythonNames(const UClass* InClass)
{
    return GetDeprecatedFieldPythonNamesImpl(InClass, ScriptNameMetaDataKey);
}

FString GetStructPythonName(const UScriptStruct* InStruct)
{
    return GetFieldPythonNameImpl(InStruct, ScriptNameMetaDataKey);
}

TArray<FString> GetDeprecatedStructPythonNames(const UScriptStruct* InStruct)
{
    return GetDeprecatedFieldPythonNamesImpl(InStruct, ScriptNameMetaDataKey);
}

FString GetEnumPythonName(const UEnum* InEnum)
{
    return GetFieldPythonNameImpl(InEnum, ScriptNameMetaDataKey);
}

TArray<FString> GetDeprecatedEnumPythonNames(const UEnum* InEnum)
{
    return GetDeprecatedFieldPythonNamesImpl(InEnum, ScriptNameMetaDataKey);
}

FString GetFieldPythonNameImpl(const FFieldVariant& InField, const FName
InMetaDataKey)
{
    FString FieldName;

    // First see if we have a name override in the meta-data
    if (GetFieldPythonNameFromMetaDataImpl(InField, InMetaDataKey, FieldName))
    {
        return FieldName;
    }
}

```

```
}

//...
}
```

## ScriptNoExport

- **Function Description:** Prevents the export of this function or property to the script.
- **Usage Locations:** UFUNCTION, UPROPERTY
- **Engine Module:** Script
- **Metadata Type:** bool
- **Commonality:** ★★★

Do not export this function or property to the script.

### Test Code:

```
UCLASS(Blueprintable, BlueprintType, meta = (ScriptName = "MyPythonLib"))
class INSIDER_API UMyPythonTestLibrary :public UBlueprintFunctionLibrary
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    static void MyScriptFunc_None();

    UFUNCTION(BlueprintCallable, meta = (ScriptNoExport))
    static void MyScriptFunc_NoExport();
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat = 123.f;

    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (ScriptNoExport))
    float MyFloat_NoExport = 123.f;
};
```

### 测试效果 Py 代码:

It is evident that default functions and properties are exported to the script, whereas MyScriptFunc\_NoExport and MyFloat\_NoExport are not present in the script.

```
class MyPythonLib(BlueprintFunctionLibrary):
    r"""
    My Python Test Library

    ***C++ Source:***

    - ***Module***: Insider
    - ***File***: MyPythonTest.h

    .....
    @property
    def my_float(self) -> float:
```

```

r"""
    (float): [Read-Write]
"""

...
@my_float.setter
def my_float(self, value: float) -> None:
    ...

@classmethod
def my_script_func_none(cls) -> None:
    r"""
        X.my_script_func_none() -> None
        My Script Func None
    """
...

```

## Principle:

---

The export status of a property or function is determined by the presence or absence of `ScriptNoExport`.

```

bool IsScriptExposedProperty(const FProperty* InProp)
{
    return !InProp->HasMetaData(ScriptNoExportMetaDataKey)
        && InProp->HasAnyPropertyParams(CPF_BlueprintVisible | CPF_BlueprintAssignable);
}

bool IsScriptExposedFunction(const UFunction* InFunc)
{
    return !InFunc->HasMetaData(ScriptNoExportMetaDataKey)
        && InFunc->HasAnyFunctionFlags(FUNC_BlueprintCallable | FUNC_BlueprintEvent)
        && !InFunc->HasMetaData(BlueprintGetterMetaDataKey)
        && !InFunc->HasMetaData(BlueprintSetterMetaDataKey)
        && !InFunc->HasMetaData(BlueprintInternalUseOnlyMetaDataKey)
        && !InFunc->HasMetaData(CustomThunkMetaDataKey)
        && !InFunc->HasMetaData(NativeBreakFuncMetaDataKey)
        && !InFunc->HasMetaData(NativeMakeFuncMetaDataKey);
}

```

## ScriptConstant

---

- **Function Description:** Packages the return value of a static function into a constant value.
- **Usage Location:** UFUNCTION
- **Engine Module:** Script
- **Metadata Type:** string="abc"
- **Associated Items:** ScriptConstantHost
- **Commonality:** ★★★

Wraps the return value of a static function into a constant value.

- The function's name serves as the default name for the constant, but ScriptConstant can also specify a custom name.
- The constant's scope is, by default, within the outer class of the static function, but it can also be directed to another type using ScriptConstantHost.

## Test Code:

```

USTRUCT(BlueprintType)
struct INSIDER_API FMyPythonConstantStruct
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_ConstantOwner :public UObject
{
    GENERATED_BODY()
public:
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_Constant_Test :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintPure, meta = (ScriptConstant))
    static int32 MyIntConst() { return 123; }

    UFUNCTION(BlueprintPure, meta = (ScriptConstant = "MyOtherIntConst"))
    static int32 MyIntConst2() { return 456; }

    UFUNCTION(BlueprintPure, meta = (ScriptConstant))
    static FMyPythonConstantStruct MyStructConst() { return
        FMyPythonConstantStruct{ TEXT("Hello") }; }

    UFUNCTION(BlueprintPure, meta = (ScriptConstant = "MyOtherStructConst"))
    static FMyPythonConstantStruct MyStructConst2() { return
        FMyPythonConstantStruct{ TEXT("World") }; }

public:
    UFUNCTION(BlueprintPure, meta = (ScriptConstant="FirstString",
    ScriptConstantHost = "/Script/Insider.MyPython_ConstantOwner"))
    static FString MyStringConst() { return TEXT("First"); }
****};

```

## Generated Py Code:

```
class MyPython_Constant_Test(Object):
    r"""
    My Python Constant Test

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptConstant.h

    """
    MY_OTHER_STRUCT_CONST: MyPythonConstantStruct #: (MyPythonConstantStruct): My
    Struct Const 2
    MY_STRUCT_CONST: MyPythonConstantStruct #: (MyPythonConstantStruct): My
    Struct Const
    MY_OTHER_INT_CONST: int #: (int32): My Int Const 2
    MY_INT_CONST: int #: (int32): My Int Const

class MyPython_ConstantOwner(Object):
    r"""
    My Python Constant Owner

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptConstant.h

    """
    FIRST_STRING: str #: (str): My String Const
```

## Result of Execution:

As seen, the corresponding constants are generated within the class. MyStringConst is generated in a different class due to the specification of ScriptConstantHost.

```
LogPython: print(unreal.MyPython_Constant_Test.MY_INT_CONST)
LogPython: 123
LogPython: print(unreal.MyPython_Constant_Test.MY_OTHER_INT_CONST)
LogPython: 456
LogPython: print(unreal.MyPython_Constant_Test.MY_OTHER_STRUCT_CONST)
LogPython: <Struct 'MyPythonConstantStruct' (0x00000A0FC4051F00) {my_string:
"world">
LogPython: print(unreal.MyPython_Constant_Test.MY_STRUCT_CONST)
LogPython: <Struct 'MyPythonConstantStruct' (0x00000A0FC4051EA0) {my_string:
"Hello">
LogPython: print(unreal.MyPython_ConstantOwner.FIRST_STRING)
LogPython: First
```

## Principle:

The logic for generating constants is encapsulated within the GenerateWrappedConstant function.

```
auto GeneratewrappedConstant = [this, &GeneratedwrappedType,
&OutGeneratedwrappedTypeReferences, &OutDirtyModules](const UFunction* InFunc)
{}
```

## ScriptConstantHost

- **Function Description:** Extending ScriptConstant, this specifies the type in which the constant is generated.
- **Usage Location:** UFUNCTION
- **Engine Module:** Script
- **Metadata Type:** string="abc"
- **Associated Items:** ScriptConstant
- **Commonality:** ★

Building upon ScriptConstant, this designates the type where the constant is to be generated.

Refer to the test code in ScriptConstant. The string specified by ScriptConstantHost should represent an object path.

```
UFUNCTION(BlueprintPure, meta = (ScriptConstant="FirstString",
ScriptConstantHost = "/Script/Insider.MyPython_ConstantOwner"))
static FString MyStringConst() { return TEXT("First"); }
```

## ScriptMethod

- **Function Description:** Converts a static function into a member function with the first parameter as its owner.
- **Usage Location:** UFUNCTION
- **Engine Module:** Script
- **Metadata Type:** string="a;b;c"
- **Restriction Type:** static function
- **Associated Items:** ScriptMethodMutable, ScriptMethodSelfReturn
- **Commonliness:** ★★★

Converts a static function into a member function with the first parameter.

- Transforms func(A, B) into A.func(B), allowing the addition of member functions to object A, similar to extension methods in C#.
- An alternative name can also be provided directly to change the name of the wrapped member function. Note the distinction from ScriptName, which changes the name exported to the script, whereas ScriptMethod changes the name of the resulting member function. Changes func(A, B) to A.OtherFunc(B).

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_ScriptMethod :public UObject
{
    GENERATED_BODY()
public:
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyPythonStruct_ScriptMethod
{
    GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_ScriptMethod_Test :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod))
    static void MyFuncOnobject(UMyPython_ScriptMethod* obj, FString val);

    UFUNCTION(BlueprintCallable, meta = (ScriptMethod =
"MySuperFunOnObject;MyOtherFunOnObject"))
    static void MyFuncOnobject2(UMyPython_ScriptMethod* obj, FString val);

public:
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod))
    static void MyFuncOnStruct(const FMyPythonStruct_ScriptMethod& mystruct,
FString val);;
```

## Test Results:

Visible in MyPythonStruct\_ScriptMethod, the method my\_func\_on\_struct has been added, and in MyPython\_ScriptMethod, the method my\_func\_on\_object has been added. Thus, in Python, these two functions can be called as if they were member functions.

Moreover, two ScriptMethod aliases are set on MyFuncOnObject2, which are also visible in MyPython\_ScriptMethod.

```
class MyPythonStruct_ScriptMethod(StructBase):
    """
    My Python Struct Script Method

    **C++ Source:**
    - **Module**: Insider
    - **File**: MyPython_ScriptMethod.h

    ...
    def __init__(self) -> None:
```

```

    ...
def my_func_on_struct(self, val: str) -> None:
    r"""
        x.my_func_on_struct(val) -> None
    My Func on Struct

    Args:
        val (str):
    """
    ...


class MyPython_ScriptMethod(Object):
    r"""
        My Python Script Method

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptMethod.h

    """
    ...

    def my_super_func_on_object(self, val: str) -> None:
        r"""
            x.my_super_func_on_object(val) -> None
        My Func on Object 2

        Args:
            val (str):
        """
        ...

    def my_other_func_on_object(self, val: str) -> None:
        r"""
            deprecated: 'my_other_func_on_object' was renamed to
            'my_super_func_on_object'.
        """
        ...

    def my_func_on_object(self, val: str) -> None:
        r"""
            x.my_func_on_object(val) -> None
        My Func on Object

        Args:
            val (str):
        """
        ...


class MyPython_ScriptMethod_Test(Object):
    r"""
        My Python Script Method Test

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptMethod.h

    """
    ...

    @classmethod

```

```

def my_func_on_struct(cls, my_struct: MyPythonStruct_ScriptMethod, val: str)
-> None:
    r"""
        X.my_func_on_struct(my_struct, val) -> None
        My Func on Struct

    Args:
        my_struct (MyPythonStruct_ScriptMethod):
        val (str):
    """

    ...

    @classmethod
def my_func_on_object2(cls, obj: MyPython_ScriptMethod, val: str) -> None:
    r"""
        X.my_func_on_object2(obj, val) -> None
        My Func on Object 2

    Args:
        obj (MyPython_ScriptMethod):
        val (str):
    """

    ...

    @classmethod
def my_func_on_object(cls, obj: MyPython_ScriptMethod, val: str) -> None:
    r"""
        X.my_func_on_object(obj, val) -> None
        My Func on Object

    Args:
        obj (MyPython_ScriptMethod):
        val (str):
    """

    ...

```

## Principle:

The detailed process of wrapping static functions into member functions is described in `GenerateWrappedDynamicMethod`. Those interested are encouraged to read further.

```

PyTypeObject* FPyWrapperTypeRegistry::GenerateWrappedClassType(const UClass*
InClass, FGeneratedWrappedTypeReferences& OutGeneratedWrappedTypeReferences,
TSet<FName>& OutDirtyModules, const EPyTypeGenerationFlags InGenerationFlags)
{
    // Should this function also be hoisted as a struct method or operator?
    if (InFunc->HasMetaData(PyGenUtil::ScriptMethodMetaDataKey))
    {
        GenerateWrappedDynamicMethod(InFunc, GeneratedWrappedMethodCopy);
    }
}

```

# ScriptMethodMutable

- **Function description:** Modify the first const structure parameter of ScriptMethod to a reference parameter during the call. Changes made to this parameter within the function will be preserved.
- **Usage Location:** UFUNCTION
- **Engine Module:** Script
- **Metadata Type:** bool
- **Restriction Type:** The first parameter must be a structure type
- **Associated Items:** ScriptMethod
- **Commonality:** ★★

Modify the first const structure parameter of ScriptMethod to a reference parameter during the call. Changes made to this parameter within the function will be preserved.

- To alter the value of a const parameter, it is still necessary to declare it as mutable in C++.
- Although the Python code generated is identical, calling ScriptMethodMutable will actually modify the parameter's value, while a function without ScriptMethodMutable will not alter the original parameter value.
- Both ScriptMethodMutable and UPARAM(ref) can modify parameter values during function calls. However, UPARAM(ref) generates Python code that returns the first parameter as the result.

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyPythonStruct_ScriptMethod
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    mutable FString MyString;

};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_ScriptMethod_Test :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod))
    static void SetStringOnStruct(const FMyPythonStruct_ScriptMethod& myStruct,
        FString val);

    UFUNCTION(BlueprintCallable, meta = (ScriptMethod, ScriptMethodMutable))
    static void SetStringOnStructMutable(const FMyPythonStruct_ScriptMethod&
        myStruct, FString val);

    UFUNCTION(BlueprintCallable, meta = (ScriptMethod, ScriptMethodMutable))
    static void SetStringOnStructViaRef(UPARAM(ref) FMyPythonStruct_ScriptMethod&
        myStruct, FString val);
};
```

## Test Results:

Examining the generated Python code reveals consistency. When using UPARAM(ref), the generated my\_func\_on\_struct\_via\_ref within MyPython\_ScriptMethod\_Test returns the structure MyPythonStruct\_ScriptMethod to achieve the reference effect.

However, my\_func\_on\_struct mutable returns None, making it indistinguishable from my\_func\_on\_struct without ScriptMethodMutable. However, a real difference emerges during actual function calls.

```
class MyPythonStruct_ScriptMethod(StructBase):
    r"""
    My Python Struct Script Method

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptMethod.h

    ...
    def __init__(self) -> None:
        ...

    def my_func_on_struct_via_ref(self, val: str) -> None:
        r"""
        x.my_func_on_struct_via_ref(val) -> None
        My Func on Struct Via Ref

        Args:
            val (str):
        ...
        ...

    def my_func_on_struct Mutable(self, val: str) -> None:
        r"""
        x.my_func_on_struct Mutable(val) -> None
        My Func on Struct Mutable

        Args:
            val (str):
        ...
        ...

    def my_func_on_struct(self, val: str) -> None:
        r"""
        x.my_func_on_struct(val) -> None
        My Func on Struct

        Args:
            val (str):
        ...
        ...

class MyPython_ScriptMethod_Test(object):
    r"""
    My Python Script Method Test
```

```

**C++ Source:**

- Module: Insider
- File: MyPython_ScriptMethod.h

"""

@classmethod
def my_func_on_struct_via_ref(cls, my_struct: MyPythonStruct_ScriptMethod,
val: str) -> MyPythonStruct_ScriptMethod:
    r"""
        X.my_func_on_struct_via_ref(my_struct, val) ->
    MyPythonStruct_ScriptMethod
        My Func on Struct Via Ref

    Args:
        my_struct (MyPythonStruct_ScriptMethod):
            val (str):

    Returns:
        MyPythonStruct_ScriptMethod:

        my_struct (MyPythonStruct_ScriptMethod):
    """

    ...

@classmethod
def my_func_on_struct mutable(cls, my_struct: MyPythonStruct_ScriptMethod,
val: str) -> None:
    r"""
        X.my_func_on_struct mutable(my_struct, val) -> None
    My Func on Struct Mutable

    Args:
        my_struct (MyPythonStruct_ScriptMethod):
            val (str):
    """

    ...

@classmethod
def my_func_on_struct(cls, my_struct: MyPythonStruct_ScriptMethod, val: str)
-> None:
    r"""
        X.my_func_on_struct(my_struct, val) -> None
    My Func on Struct

    Args:
        my_struct (MyPythonStruct_ScriptMethod):
            val (str):
    """

```

Call records in the UE Python console and analysis of the calling sequence:

- Initially calling set\_string\_on\_struct mutable and then printing (a) outputs "Hello," indicating that the value is successfully set within the a structure.

- Attempting set\_string\_on\_struct again and then printing (a) fails to produce "FFF," suggesting that the value was not set in the a structure. This implies that Python likely constructs a temporary value as the call object during the function call, and the new value after the call is not assigned to the original a object.
- Trying set\_string\_on\_struct\_via\_ref followed by printing (a) outputs "First," demonstrating that UPARAM(ref) can also achieve the effect of modifying parameters.

```

LogPython: a=unreal.MyPythonStruct_ScriptMethod()
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x0000092D08CD6ED0) {my_string:
""}>
LogPython: a.set_string_on_structMutable("Hello")
LogBlueprintUserMessages: [None]
UMyPython_ScriptMethod_Test::SetStringOnStructMutable
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x0000092D08CD6ED0) {my_string:
"Hello"}>
LogPython: a.set_string_on_struct("FFF")
LogBlueprintUserMessages: [None] UMyPython_ScriptMethod_Test::SetStringOnStruct
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x0000092D08CD6ED0) {my_string:
"Hello"}>
LogPython: a.set_string_on_struct_via_ref("First")
LogBlueprintUserMessages: [None]
UMyPython_ScriptMethod_Test::SetStringOnStructViaRef
LogPython: print(a)
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x0000092D08CD6ED0) {my_string:
"First"}>

```

## Principle:

It is determined that if ScriptMethodMutable is present, SelfReturn is set, effectively copying the temporary value from within the function call to the original parameter value, thus achieving the desired variable reference call behavior.

```

// The function may have been flagged as mutable, in which case we always
// consider it to need a 'self' return
if (!GeneratedwrappedDynamicMethod.SelfReturn.ParamProp && InFunc-
>HasMetaData(PyGenUtil::ScriptMethodMutableMetadataKey))
{
    if (!SelfParam.ParamProp->IsA<FStructProperty>())
    {
        REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodMutable' but the 'self' argument is not a struct."), *InFunc-
>GetOwnerClass()->GetName(), *InFunc->GetName());
        return;
    }
    GeneratedwrappedDynamicMethod.SelfReturn = SelfParam;
}

```

# ScriptMethodSelfReturn

- **Function description:** In the case of ScriptMethod , specify that the return value of this function should overwrite the first parameter of the function.
- **Usage Location:** UFUNCTION
- **Engine Module:** Script
- **Metadata Type:** bool
- **Associated Items:** ScriptMethod
- **Commonality:** ★★

In the case of ScriptMethod, specify that the return value of this function should overwrite the first parameter of the function.

In such cases, the original function does not return a value. The effect is as if:

```
C Func(A,B) -> void A::Func2(B)  
调用的时候:  
从 C=A.Func(B) ->  
void A::Func2(B)  
{  
    A=A.Func(B)  
}
```

## Test Code:

Note that because the AppendStringOnStructViaRef parameter is a reference parameter, in order to apply the result to myStruct, there is no need to create a temporary value in the function body, and it can be modified directly on myStruct. If temporary values are also used, myStruct cannot be modified, and the meaning of the ref parameter is lost.

```
public:  
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod))  
    static FMyPythonStruct_ScriptMethod AppendStringOnStruct(const  
FMyPythonStruct_ScriptMethod& myStruct, FString val)  
    {  
        FMyPythonStruct_ScriptMethod Result = myStruct;  
        Result.MyString += val;  
        return Result;  
    }  
  
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod,ScriptMethodSelfReturn))  
    static FMyPythonStruct_ScriptMethod AppendStringOnStructReturn(const  
FMyPythonStruct_ScriptMethod& myStruct, FString val)  
    {  
        FMyPythonStruct_ScriptMethod Result = myStruct;  
        Result.MyString += val;  
        return Result;  
    }  
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod, ScriptMethodMutable))  
    static FMyPythonStruct_ScriptMethod AppendStringOnStructViaRef(UPARAM(ref)  
FMyPythonStruct_ScriptMethod& myStruct, FString val)  
    {
```

```

        mystruct.MyString += val;
    return myStruct;
}

//LogPython: Error: Function
'MyPython_ScriptMethod_Test.AppendStringOnStructViaRefReturn' is marked as
'ScriptMethodSelfReturn' but the 'self' argument is also marked as UPARAM(ref).
This is not allowed.
/UFUNCTION(BlueprintCallable, meta = (ScriptMethod,
ScriptMethodMutable,ScriptMethodSelfReturn))
//static FMyPythonStruct_ScriptMethod
AppendStringOnStructViaRefReturn(UPARAM(ref) FMyPythonStruct_ScriptMethod&
myStruct, FString val);

```

## Generated Py Code:

Visible append\_string\_on\_struct\_return There is no return value. And append\_string\_on\_struct has a return value. append\_string\_on\_struct\_via\_ref also has a return value.

```

class MyPythonStruct_ScriptMethod(StructBase):
    def append_string_on_struct_return(self, val: str) -> None:
        r"""
        x.append_string_on_struct_return(val) -> None
        Append String on Struct Return

        Args:
            val (str):

        Returns:
            MyPythonStruct_ScriptMethod:
            """
        ...

    def append_string_on_struct(self, val: str) -> MyPythonStruct_ScriptMethod:
        r"""
        x.append_string_on_struct(val) -> MyPythonStruct_ScriptMethod
        Append String on Struct

        Args:
            val (str):

        Returns:
            MyPythonStruct_ScriptMethod:
            """
        ...

    def append_string_on_struct_via_ref(self, val: str) ->
MyPythonStruct_ScriptMethod:
        r"""
        x.append_string_on_struct_via_ref(val) -> MyPythonStruct_ScriptMethod
        Append String on Struct Via Ref

        Args:
            val (str):

        Returns:
            MyPythonStruct_ScriptMethod:
            """

```

....  
...

Test Code: Observe the running results and the object's memory address.

- It can be seen that append\_string\_on\_struct has a return value, but the result of the change is not applied to parameter a.
- append\_string\_on\_struct\_return can be applied to parameter a, but there is no return value.
- append\_string\_on\_struct\_via\_ref can be applied to parameter a and also has a return value. But note that the return value and a are not actually the same object because the memory addresses are different.
- But note ScriptMethodSelfReturn and UPARAM ( ref ) cannot be mixed, otherwise an error will be reported: LogPython: Error: Function 'MyPython\_ScriptMethod\_Test.AppendStringOnStructViaRefReturn' is marked as 'ScriptMethodSelfReturn' but the ' self ' argument is also marked as UPARAM ( ref ). This is not allowed .

```
LogPython: a=unreal.MyPythonStruct_ScriptMethod()  
LogPython: print(a)  
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEBAB08ED0) {my_string:  
""}>  
LogPython: b=a.append_string_on_struct("Hello")  
LogPython: print(b)  
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEBAB04010) {my_string:  
"Hello"}>  
LogPython: print(a)  
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEBAB08ED0) {my_string:  
""}>  
LogPython: c=a.append_string_on_struct_return("Hello")  
LogPython: print(c)  
LogPython: None  
LogPython: print(a)  
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEBAB08ED0) {my_string:  
"Hello"}>  
LogPython: d=a.append_string_on_struct_via_ref("World")  
LogPython: print(d)  
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEBAB06110) {my_string:  
"HelloWorld"}>  
LogPython: print(a)  
LogPython: <Struct 'MyPythonStruct_ScriptMethod' (0x000008DEBAB08ED0) {my_string:  
"HelloWorld"}>
```

## Principle:

Treat the first output parameter as the return parameter. The output parameters are actually the return values in the function. SelfReturn means that this value will later overwrite the value of the calling object, which is the object where the call occurred.

```
// The function may also have been flagged as having a 'self' return  
if (InFunc->HasMetaData(PyGenUtil::ScriptMethodSelfReturnMetaDataKey))  
{  
    if (GeneratedwrappedDynamicMethod.SelfReturn.ParamProp)
```

```

{
    REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodSelfReturn' but the 'self' argument is also marked as UPARAM(ref).
This is not allowed."), *InFunc->GetOwnerClass()->GetName(), *InFunc->GetName());
    return;
}
else if (GeneratedwrappedDynamicMethod.MethodFunc.OutputParams.Num() == 0 ||
GeneratedwrappedDynamicMethod.MethodFunc.OutputParams[0].ParamProp-
>HasAnyPropertyFlags(CPF_ReturnParm))
{
    REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodSelfReturn' but has no return value."), *InFunc->GetOwnerClass()-
>GetName(), *InFunc->GetName());
    return;
}
else if (!SelfParam.ParamProp->IsA<FStructProperty>())
{
    REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodSelfReturn' but the 'self' argument is not a struct."), *InFunc-
>GetOwnerClass()->GetName(), *InFunc->GetName());
    return;
}
else if (!GeneratedwrappedDynamicMethod.MethodFunc.OutputParams[0].ParamProp-
>IsA<FStructProperty>())
{
    REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodSelfReturn' but the return value is not a struct."), *InFunc-
>GetOwnerClass()->GetName(), *InFunc->GetName());
    return;
}
else if (CastFieldChecked<const FStructProperty>
(GeneratedwrappedDynamicMethod.MethodFunc.OutputParams[0].ParamProp)->Struct !=

CastFieldChecked<const FStructProperty>(SelfParam.ParamProp)->Struct)
{
    REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as
'ScriptMethodSelfReturn' but the return value is not the same type as the 'self'
argument."), *InFunc->GetOwnerClass()->GetName(), *InFunc->GetName());
    return;
}
else
{
    GeneratedwrappedDynamicMethod.SelfReturn =
MoveTemp(GeneratedwrappedDynamicMethod.MethodFunc.OutputParams[0]);
    GeneratedwrappedDynamicMethod.MethodFunc.OutputParams.RemoveAt(0, 1,
EAllowShrinking::No);
}
}

```

# ScriptOperator

---

- **Function Description:** Wraps a static function with its first parameter as a structure into an operator for that structure.
- **Usage Location:** UFUNCTION
- **Engine Module:** Script
- **Metadata Type:** string="a;b;c"
- **Commonality:** ★★★

Wraps a static function with its first parameter as a structure into an operator for the structure.

- Can include multiple operators.

Different operators need to match different function signatures. The rules are as follows:

- bool Operator: bool
  - bool FuncName(const FMyStruct& Value); // The type of Value can be const FMyStruct& or simply FMyStruct
- Unary Operator: neg (negation)
  - FMyStruct FuncName(const FMyStruct&);
- Comparison Operators: (==, !=, <, <=, >, >=)
  - bool FuncName(const FMyStruct&, OtherType); // OtherType can be any other type
- Mathematical Operators: (+, -, \*, /, %, &, |, ^, >>, <<)
  - ReturnType FuncName(const FMyStruct&, OtherType); // ReturnType and OtherType can be any other type
- Mathematical Assignment Operators: (+=, -=, \*=, /=, %=, &=, |=, ^=, >>=, <<=)
  - FMyStruct FuncName(const FMyStruct&, OtherType); // OtherType can be any other type

It is evident that if a function is to support both standard mathematical operators and assignment operators, the function signature can be:

FMyStruct FuncName(const FMyStruct&, OtherType); // Here, OtherType can be any type, or even FMyStruct

This is often used in conjunction with ScriptMethod, allowing a structure to provide an operation member function, which can also be customized in name through ScriptMethod.

There are also common examples of using ScriptMethodSelfReturn in the source code, such as with the += operator. However, ScriptMethodSelfReturn is not mandatory, as the return value is naturally applied to the first parameter when using +=.

## Test Code:

---

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyPythonMathStruct
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 value = 0;
```

```

};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_Operator_Test :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable, meta = (ScriptMethod=HasValue, ScriptOperator =
"bool"))
        static bool IsValid(const FMyPythonMathStruct& InStruct) { return
InStruct.value != 0; }

    UFUNCTION(BlueprintCallable, meta = (ScriptOperator = "neg"))
        static FMyPythonMathStruct Neg(const FMyPythonMathStruct& InStruct) { return
{ -InStruct.value }; }

    UFUNCTION(BlueprintCallable, meta = (ScriptOperator = "=="))
        static bool IsEqual(const FMyPythonMathStruct& A, const FMyPythonMathStruct&
B) { return A.Value == B.Value; }

    UFUNCTION(BlueprintCallable, meta = (ScriptOperator = "+;+="))
        static FMyPythonMathStruct AddInt(FMyPythonMathStruct InStruct, const int32
InValue) { InStruct.value += InValue; return InStruct; }
};

```

## Generated Py Code:

It can be observed that the functions **bool**, **eq**, **add**, **iadd**, and **neg** have been generated in Python. Additionally, with **IsValid** annotated with **ScriptMethod**, an additional **has\_value** function is provided.

```

class MyPythonMathStruct(structBase):
    r"""
    My Python Math Struct

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptOperator.h

    **Editor Properties:** (see get_editor_property/set_editor_property)

    - ``value`` (int32): [Read-write]
    """

    def __init__(self, value: int = 0) -> None:
        ...
        @property
        def value(self) -> int:
            r"""
            (int32): [Read-write]
            """

            ...
            @value.setter
            def value(self, value: int) -> None:

```

```

    ...
def has_value(self) -> bool:
    r"""
    x.has_value() -> bool
    Is valid

    Returns:
        bool:
    """

    ...

def __bool__(self) -> bool:
    r"""
    Is valid
    """

    ...

def __eq__(self, other: object) -> bool:
    r"""
    **Overloads:**

    - ``MyPythonMathStruct`` Is Equal
    """

    ...

def __add__(self, other: MyPythonMathStruct) -> None:
    r"""
    **Overloads:**

    - ``int32`` Add Int
    """

    ...

def __iadd__(self, other: MyPythonMathStruct) -> None:
    r"""
    **Overloads:**

    - ``int32`` Add Int
    """

    ...

def __neg__(self) -> None:
    r"""
    Neg
    """

    ...

```

## Running the Test:

---

It is confirmed that the mathematical `+=` operator and boolean comparisons are indeed supported.

```

LogPython: a=unreal.MyPythonMathStruct(3)
LogPython: print(a)
LogPython: <Struct 'MyPythonMathStruct' (0x0000074C90D5DCF0) {value: 3}>
LogPython: print(not a)
LogPython: False
LogPython: a+=3
LogPython: print(a)
LogPython: <Struct 'MyPythonMathStruct' (0x0000074C90D5DCF0) {value: 6}>
LogPython: print(-a)
LogPython: <Struct 'MyPythonMathStruct' (0x0000074C90D5DCF0) {value: -6}>

```

## Principle:

---

The specific wrapping functions are located within GenerateWrappedOperator. For those who wish to delve deeper, the details can be found there.

```

auto GenerateWrappedOperator = [this, &OutGeneratedWrappedTypeReferences,
&OutDirtyModules](const UFunction* InFunc, const
PyGenUtil::FGeneratedWrappedMethod& InTypeMethod)
{
    // only static functions can be hoisted onto other types
    if (!InFunc->HasAnyFunctionFlags(FUNC_Static))
    {
        REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Non-static function '%s.%s' is marked as 'ScriptOperator' but only static functions can be hoisted."), *InFunc->GetOwnerClass()->GetName(), *InFunc->GetName());
        return;
    }

    // Get the list of operators to apply this function to
    TArray<FString> ScriptOperators;
    {
        const FString& ScriptOperatorsStr = InFunc->GetMetaData(PyGenUtil::ScriptOperatorMetaDataKey);
        ScriptOperatorsStr.ParseIntoArray(ScriptOperators, TEXT(";"));
    }

    // Go through and try and create a function for each operator, validating that the signature matches what the operator expects
    for (const FString& ScriptOperator : ScriptOperators)
    {
        PyGenUtil::FGeneratedWrappedOperatorSignature OpSignature;
        if (!PyGenUtil::FGeneratedWrappedOperatorSignature::StringToSignature(*ScriptOperator, OpSignature))
        {
            REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is marked as 'ScriptOperator' but uses an unknown operator type '%s'."), *InFunc->GetOwnerClass()->GetName(), *InFunc->GetName(), *ScriptOperator);
            continue;
        }

        PyGenUtil::FGeneratedWrappedOperatorFunction OpFunc;
        {

```

```

        FString SignatureError;
        if (!OpFunc.SetFunction(InTypeMethod.MethodFunc, Opsignature,
&SignatureError))
        {
            REPORT_PYTHON_GENERATION_ISSUE(Error, TEXT("Function '%s.%s' is
marked as 'ScriptOperator' but has an invalid signature for the '%s' operator:
%s."), *InFunc->GetOwnerClass()->GetName(), *InFunc->GetName(), *ScriptOperator,
*SignatureError);
            continue;
        }
    }

    // Ensure that we've generated a finalized Python type for this struct
    // since we'll be adding this function as a operator on that type
    const UScriptStruct* HostedStruct = CastFieldChecked<const
FStructProperty>(OpFunc.SelfParam.ParamProp)->Struct;
    if (GenerateWrappedStructType(HostedStruct,
OutGeneratedWrappedTypeReferences, OutDirtyModules,
EPyTypeGenerationFlags::ForceShouldExport))
    {
        // Find the wrapped type for the struct as that's what we'll actually
        // add the operator to (via its meta-data)
        TSharedPtr<PyGenUtil::FGeneratedwrappedStructType>
HostedStructGeneratedWrappedType =
        StaticCastSharedPtr<PyGenUtil::FGeneratedwrappedStructType>
(GeneratedWrappedTypes.FindRef(PyGenUtil::GetTypeRegistryName(HostedStruct)));
        check(HostedStructGeneratedWrappedType.IsValid());
        StaticCastSharedPtr<FPyWrapperStructMetaData>
(HostedStructGeneratedWrappedType->MetaData)-
>OpStacks[(int32)Opsignature.OpType].Funcs.Add(MoveTemp(OpFunc));
    }
}
};
```

## ScriptDefaultMake

- **Function description:** Disable structural HasNativeMake , do not call the NativeMake function in C++ when constructing in the script, and use the script's built-in default initialization method.
- **Use location:** USTRUCT
- **Engine module:** Script
- **Metadata type:** bool
- **Related items:** ScriptDefaultBreak
- **Frequency of use:** ★

Disable HasNativeMake on the structure, do not call the NativeMake function in C++ when constructing in the script, and use the script's built-in default initialization method.

The same goes for ScriptDefaultBreak.

## Test code:

```
USTRUCT(BlueprintType, meta = (ScriptDefaultMake,
ScriptDefaultBreak, HasNativeMake =
"/Script/Insider.MyPython_MakeBreak_Test.MyNativeMake", HasNativeBreak =
"/Script/Insider.MyPython_MakeBreak_Test.MyNativeBreak"))
struct INSIDER_API FMyPythonMBStructNative
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    int32 MyInt = 0;

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyPython_MakeBreak_Test :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintPure, meta = ())
    static FMyPythonMBStructNative MyNativeMake(int32 InInt) { return
FMyPythonMBStructNative{ InInt, TEXT("Hello") }; }

    UFUNCTION(BlueprintPure, meta = ())
    static void MyNativeBreak(const FMyPythonMBStructNative& InStruct, int&
outInt) { outInt = InStruct.MyInt + 123; }
};
```

## Generated Python code:

Regardless of whether ScriptDefaultMake , ScriptDefaultBreak or MyPythonMBStructNative is added by the code is actually the same. The difference lies in the result when forming the sum to\_tuple .

```
class MyPythonMBStructNative(StructBase):
    r"""
    My Python MBStruct Native

    **C++ Source:**

    - **Module**: Insider
    - **File**: MyPython_ScriptMakeBreak.h

    **Editor Properties:** (see get_editor_property/set_editor_property)

    - ``my_int`` (int32): [Read-Write]
    - ``my_string`` (str): [Read-Write]
    """

    def __init__(self, int: int = 0) -> None:
```

```

...
@property
def my_int(self) -> int:
    r"""
    (int32): [Read-write]
"""

...
@my_int.setter
def my_int(self, value: int) -> None:
    ...

@property
def my_string(self) -> str:
    r"""
    (str): [Read-write]
"""

...
@my_string.setter
def my_string(self, value: str) -> None:
    ...

```

## Running results:

- The second paragraph is the effect after adding ScriptDefaultMake and ScriptDefaultBreak. I deliberately made some differences in the Make and Break functions of C++, so that I can observe the functions being called in C++.
- The first paragraph is the result of calling ScriptDefaultMake and ScriptDefaultBreak (retaining HasNativeMake and HasNativeBreak) in the code. It can be seen that the Make/Break function in C++ is no longer called.

```

LogPython: b=unreal.MyPythonMBStructNative()
LogPython: print(b)
LogPython: <Struct 'MyPythonMBStructNative' (0x0000085F2EE9E680) {my_int: 0,
my_string: "Hello"}>
LogPython: print(b.to_tuple())
LogPython: (123,)

LogPython: b=unreal.MyPythonMBStructNative()
LogPython: print(b)
LogPython: <Struct 'MyPythonMBStructNative' (0x000005E6C3AAFD0) {my_int: 0,
my_string: ""}>
LogPython: print(b.to_tuple())
LogPython: (0, '')

```

## Principle:

In the FindMakeBreakFunction function, if a ScriptDefaultMake or ScriptDefaultBreak tag is found, the functions specified by HasNativeMake and HasNativeBreak in C++ will not be used.

In addition py The structure initialization in ! will call the default init or make function of the structure, and to\_tuple equivalent to break the function will call the default each attribute to\_tuple or the custom break function of the structure.

```
const FName ScriptDefaultMakeMetaDataKey = TEXT("ScriptDefaultMake");
```

```

const FName ScriptDefaultBreakMetaKey = TEXT("ScriptDefaultBreak");

namespace UE::Python
{
    /**
     * Finds the UFunction corresponding to the name specified by 'HasNativeMake' or
     'HasNativeBreak' meta data key.
     * @param The structure to inspect for the 'HasNativeMake' or 'HasNativeBreak'
     meta data keys.
     * @param InNativeMetaKey The native meta data key name. Can only be
     'HasNativeMake' or 'HasNativeBreak'.
     * @param InScriptDefaultMetaKey The script default meta data key name. Can
     only be 'ScriptDefaultMake' or 'ScriptDefaultBreak'.
     * @param NotFoundFn Function invoked if the structure specifies as Make or Break
     function, but the function couldn't be found.
     * @return The function, if the struct has the meta key and if the function was
     found. Null otherwise.
    */
    template<typename NotFoundFuncT>
    UFunction* FindMakeBreakFunction(const UScriptStruct* InStruct, const FName&
    InNativeMetaKey, const FName& InScriptDefaultMetaKey, const
    NotFoundFuncT& NotFoundFn)
    {
        check(InNativeMetaKey == PyGenUtil::HasNativeMakeMetaKey ||
        InNativeMetaKey == PyGenUtil::HasNativeBreakMetaKey);
        check(InScriptDefaultMetaKey == PyGenUtil::ScriptDefaultMakeMetaKey
        || InScriptDefaultMetaKey == PyGenUtil::ScriptDefaultBreakMetaKey);

        UFunction* MakeBreakFunc = nullptr;
        if (!InStruct->HasMetaData(InScriptDefaultMetaKey)) // <--- with default
            , null will be returned directly
        {
            const FString MakeBreakFunctionName = InStruct-
            >GetMetaData(InNativeMetaKey);
            if (!MakeBreakFunctionName.IsEmpty())
            {
                // Find the function.
                MakeBreakFunc = FindObject<UFunction>(*Outer*/nullptr,
                *MakeBreakFunctionName, /*ExactClass*/true);
                if (!MakeBreakFunc)
                {
                    NotFoundFn(MakeBreakFunctionName);
                }
            }
        }
        return MakeBreakFunc;
    }

    struct FF funcs
    {
        static int Init(FPyWrapperStruct* InSelf, PyObject* InArgs, PyObject* InKwds)
        {
            const int SuperResult = PyWrapperStructType.tp_init((PyObject*)InSelf,
            InArgs, InKwds);
        }
    };
}

```

```

    if (SuperResult != 0)
    {
        return SuperResult;
    }

    return FPyWrapperStruct::MakeStruct(InSelf, InArgs, InKwds);
}
};

GeneratedWrappedType->PyType.tp_init = (initproc)&FFuncs::Init;

// python wrapper A to_tuple function is mapped to each type, and the break
function of the type will be called to convert to tuple
static PyObject* ToTuple(FPyWrapperStruct* InSelf)
{
    return FPyWrapperStruct::BreakStruct(InSelf);
}

.....
{ "to_tuple", PyCFunctionCast(&FMethods::ToTuple), METH_NOARGS,
"to_tuple(self) -> Tuple[object, ...] -- break this Unreal struct into a tuple of
its properties" },

```

## ScriptDefaultBreak

---

- **Usage Location:** USTRUCT
- **Engine Module:** Script
- **Metadata Type:** bool
- **Associated Items:** ScriptDefaultMake
- **Commonality:** ★

Refer to the principles and test code for ScriptDefaultMake.

## TakeRecorderDisplayName

---

- **Function Description:** Specifies the display name for UTakeRecorderSource.
- **Usage Location:** UCLASS
- **Engine Module:** Sequencer
- **Metadata Type:** string="abc"
- **Restriction Type:** Applicable to subclasses of UTakeRecorderSource
- **Commonality:** ★★

Specifies the display name for UTakeRecorderSource.

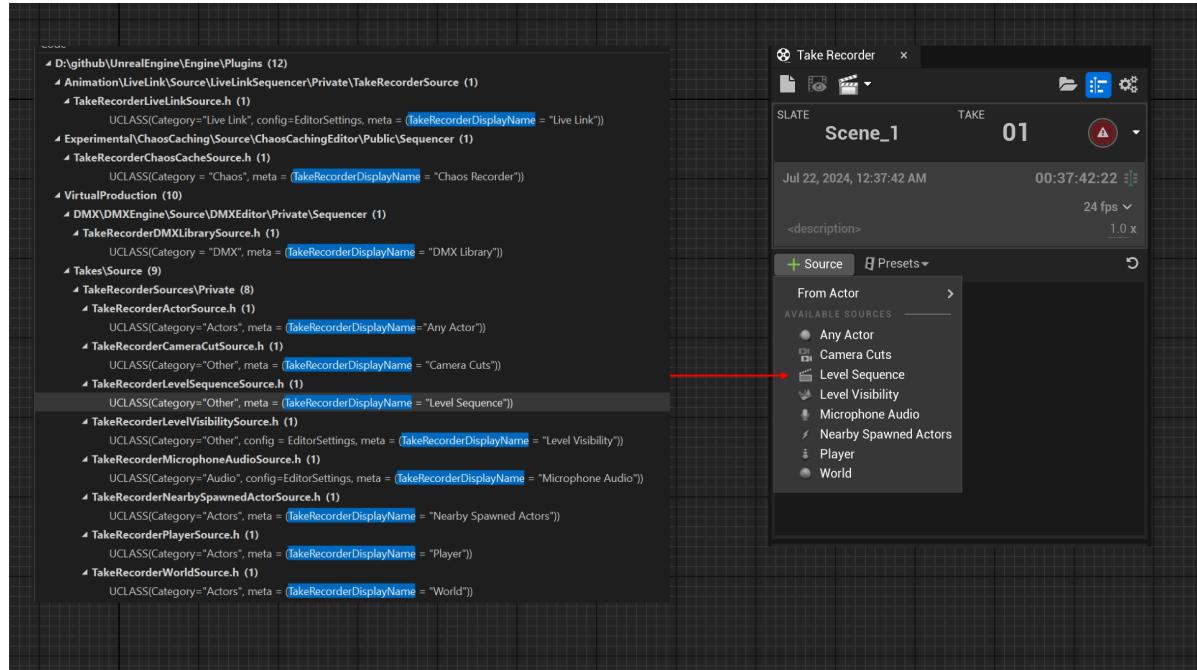
Usually, this is used internally by the engine, and it only becomes relevant if you intend to customize UTakeRecorderSource. Given the simplicity of its concept and demonstration, there is no need to construct test code myself.

# Source Code Example:

```
UCLASS(Category="Actors", meta = (TakeRecorderDisplayName = "Player"))
class UTakeRecorderPlayerSource : public UTakeRecorderSource
{}
```

## Test Effects:

You can see multiple instances of UTakeRecorderSource within the engine source code, all of which are labeled with names.



## Principle:

The name specified by TakeRecorderDisplayName is used as the name for the menu item.

```
TSharedRef<SWidget> SLevelSequenceTakeEditor::OnGenerateSourcesMenu()
{
    for (UClass* class : SourceClasses)
    {
        TSubclassof<UTakeRecorderSource> subclassof = class;

        MenuBuilder.AddMenuEntry(
            FText::FromString(class-
>GetMetaData(TEXT("TakeRecorderDisplayName"))),
            Class->GetToolTipText(true),
            FSlateIconFinder::FindIconForClass(class),
            FUIAction(
                FExecuteAction::CreateSP(this,
&SLevelSequenceTakeEditor::AddSourceFromClass, subclassof),
                FCanExecuteAction::CreateSP(this,
&SLevelSequenceTakeEditor::CanAddSourceFromClass, subclassof)
            )
        );
    }
}
```

# SequencerBindingResolverLibrary

- **Function Description:** Treat UBlueprintFunctionLibrary marked with SequencerBindingResolverLibrary as a dynamically bindable class.
- **Usage Location:** UClass
- **Engine Module:** Sequencer
- **Metadata Type:** bool
- **Restriction Type:** Applies to UClass, but typically used with UBlueprintFunctionLibrary
- **Commonality:** ★★

Consider UBlueprintFunctionLibrary with the SequencerBindingResolverLibrary tag as dynamically bindable. Only add its functions to the context menu accessible via a right-click.

Dynamic binding is a new Sequencer feature that allows pre-set trajectory changes to be dynamically applied to other Actors at runtime. This is particularly useful for seamless transitions between Gameplay and Sequences. For a more detailed explanation of its usage, please refer to the official documentation: <https://dev.epicgames.com/documentation/zh-cn/unreal-engine/dynamic-binding-in-sequencer>

## Test Code:

```
UCLASS(meta=(SequencerBindingResolverLibrary), MinimalAPI)
class UMySequencerBindingResolverLibrary : public UBlueprintFunctionLibrary
{
    GENERATED_BODY()

public:

    /** Resolve the bound object to the player's pawn */
    UFUNCTION(BlueprintPure, Category="Sequencer|Insider", meta=
    (WorldContext="WorldContextObject"))
        static FMovieSceneDynamicBindingResolveResult ResolveToMyActor(UObject*
    WorldContextObject, FString ActorTag);
};
```

Source Code:

```

UCLASS(meta=(SequencerBindingResolverLibrary), MinimalAPI)
class UBuiltInDynamicBindingResolverLibrary : public UBlueprintFunctionLibrary
{
    GENERATED_BODY()

public:

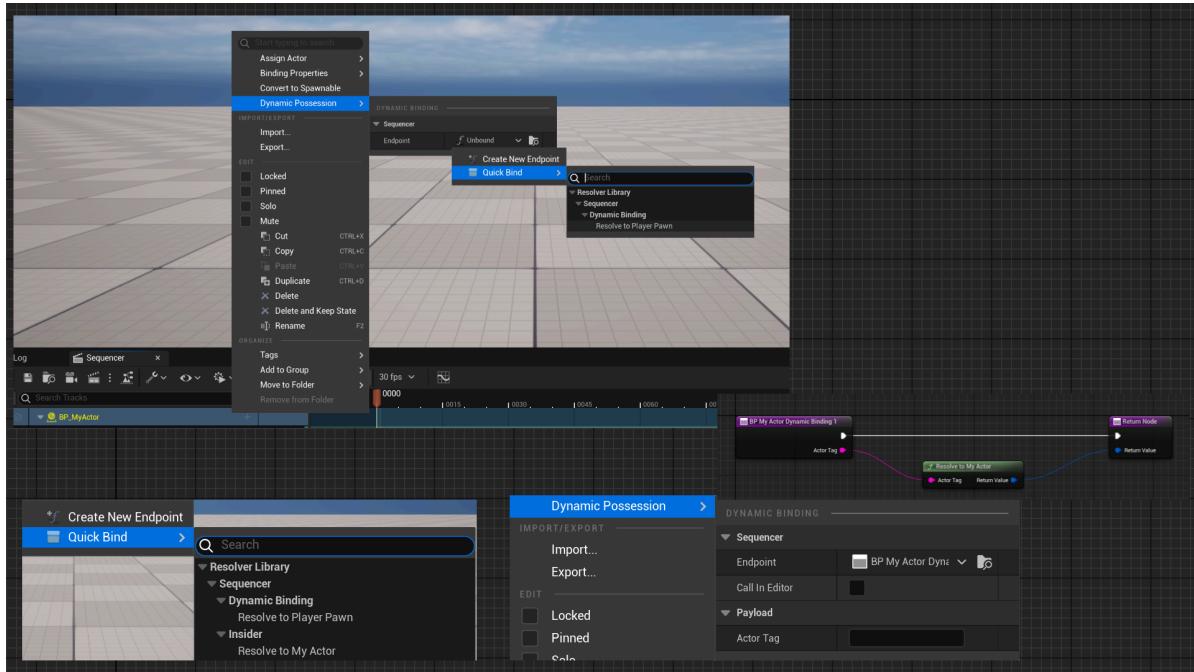
    /** Resolve the bound object to the player's pawn */
    UFUNCTION(BlueprintPure, Category="Sequencer|Dynamic Binding", meta=
    (worldContext="WorldContextObject"))
        static MOVIESCENE_API FMovieSceneDynamicBindingResolveResult
    ResolveToPlayerPawn(UObject* WorldContextObject, int32 PlayerControllerIndex =
    0);
};

```

## Test Results:

Before UMySequencerBindingResolverLibrary was defined, the engine had a built-in ResolveToPlayerPawn function, which could convert the PlayerControllerIndex to a Pawn for dynamic binding to the player's Pawn.

Thus, we can also define our own dynamic binding functions to resolve an FString into an Actor, as demonstrated by the ResolveToMyActor function in the code.



## Principle:

FMovieSceneDynamicBindingCustomization searches for all classes within the engine, but to narrow down the scope, it only looks for Resolver functions under classes tagged with SequencerBindingResolverLibrary.

```

void
FMovieSceneDynamicBindingCustomization::CollectResolverLibraryBindActions(UBlueprint*
int* Blueprint, FBlueprintActionMenuBuilder& MenuBuilder, bool bIsRebinding)
{
}

```

```

    // Add any class that has the "SequencerBindingResolverLibrary" meta as a
    target class.
    //
    // We don't consider *all* blueprint function libraries because there are many,
    many of them that expose
    // functions that are, technically speaking, compatible with bound object
    resolution (i.e. they return
    // a UObject pointer) but that are completely non-sensical in this context.
    const static FName
    SequencerBindingResolverLibraryMeta("SequencerBindingResolverLibrary");
    for (TObjectIterator<UClass> ClassIt; ClassIt; ++ClassIt)
    {
        UClass* CurrentClass = *ClassIt;
        if (CurrentClass->HasMetaData(SequencerBindingResolverLibraryMeta))
        {
            FBlueprintActionFilter::Add(MenuFilter.TargetClasses, CurrentClass);
        }
    }
}

```

## CommandLineID

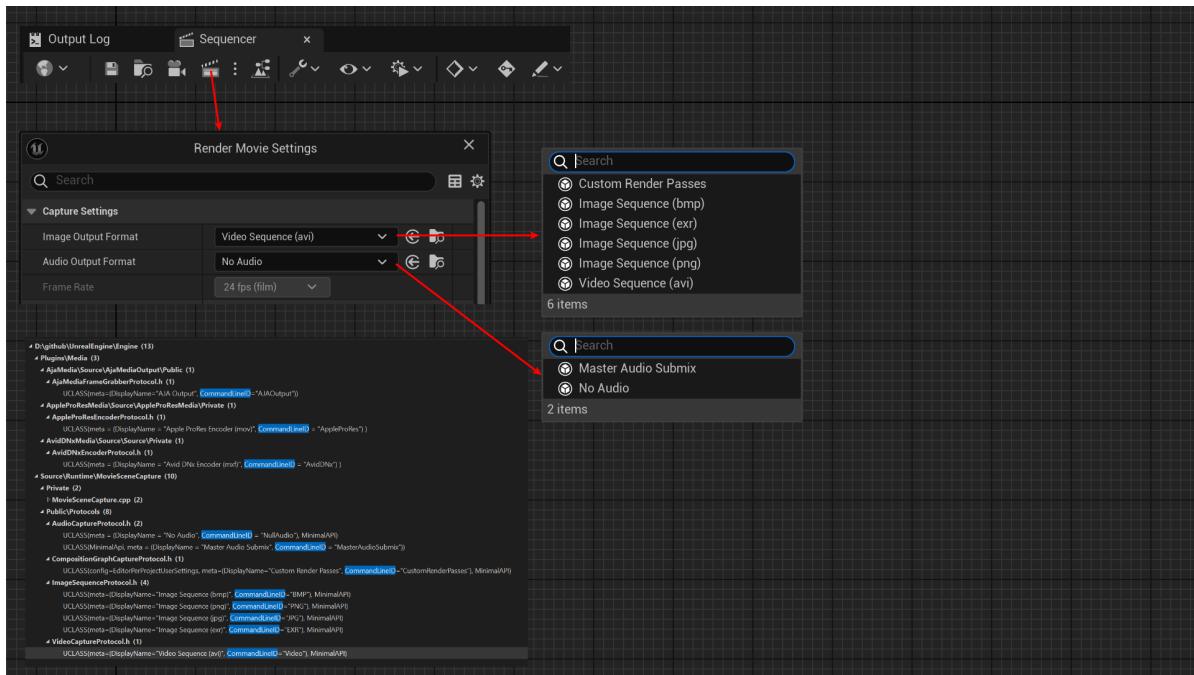
---

- **Function description:** Indicates the protocol type for subclasses of UMovieSceneCaptureProtocolBase.
- **Usage location:** UCLASS
- **Engine module:** Sequencer
- **Metadata type:** string="abc"
- **Restriction type:** Applicable to subclasses of UMovieSceneCaptureProtocolBase
- **Commonality:** ★★

Indicates the protocol type for subclasses of UMovieSceneCaptureProtocolBase.

Used to select the correct handler class during Sequencer rendering and export. Typically used within the engine, unless custom rendering output format protocol classes are being defined.

# Test Results:



## Principle:

In essence, it involves finding the relevant ProtocolType Class based on the chosen format name

```
void UMovieSceneCapture::Initialize(TSharedPtr<FSceneViewport> InSceneViewport,
int32 PIEInstance)
{
    FString ImageProtocolOverrideString;
    if ( FParse::Value( FCommandLine::Get(), TEXT( "-MovieFormat=" ),
ImageProtocolOverrideString )
        || FParse::Value( FCommandLine::Get(), TEXT( "-ImageCaptureProtocol=" )
), ImageProtocolOverrideString ) )
    {
        static const TCHAR* const CommandLineIDString =
TEXT("CommandLineID");
        TArray<UClass*> AllProtocolTypes =
FindAllCaptureProtocolClasses();
        for (UClass* Class : AllProtocolTypes)
        {
            bool bMetaDataMatch = Class->GetMetaData(CommandLineIDString)
== ImageProtocolOverrideString;
            if ( bMetaDataMatch || class->GetName() ==
ImageProtocolOverrideString )
            {
                OverrideClass = Class;
            }
        }
        ImageCaptureProtocolType = OverrideClass;
    }

    if ( FParse::Value( FCommandLine::Get(), TEXT( "-AudioCaptureProtocol=" ),
AudioProtocolOverrideString ) )
    {
```

```

        static const TCHAR* const CommandLineIDString =
TEXT("CommandLineID");
    }
}

```

# SkipUCSModifiedProperties

- **Function description:** Allows properties within an ActorComponent to be retained after modification in the Actor's constructor.
- **Use location:** UPROPERTY
- **Engine module:** Serialization
- **Metadata type:** bool
- **Restriction type:** Properties under ActorComponent
- **Commonly used:** 1

Allows properties within an ActorComponent to be retained after modification in the Actor's constructor.

By default, these properties are not serialized and saved. It is presumed that the developers believed these properties would be reassigned in the Actor's constructor, rendering serialization unnecessary. However, in certain scenarios, such as when the constructor is executed only once, or when the Actor is dynamically created with component properties (e.g., PCG), it is desirable for these Component properties to be serialized and saved.

## Test Code:

```

UCLASS(Blueprintable, BlueprintType, meta=(BlueprintSpawnableComponent))
class INSIDER_API UMyComponent_SkipUCSModifiedProperties :public UActorComponent
{
public:
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString MyString_Default = TEXT("Hello");
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta =
(skipUCSModifiedProperties))
    FString MyString_Skip = TEXT("Hello");
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyProperty_SkipUCSModifiedProperties_BaseActor :public AActor
{
public:
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditDefaultOnly)
    bool CanCallConstruction=false;
};

UCLASS(Blueprintable, BlueprintType)

```

```

class INSIDER_API AMyProperty_SkipUCSModifiedProperties_TestActor :public AActor
{
public:
    GENERATED_BODY()

    UPROPERTY(EditAnywhere)
    TSubclassOf ActorClass;

    UFUNCTION(CallInEditor)
    void CreateActor();

    UFUNCTION(CallInEditor)
    void CleanupActor();
};

void AMyProperty_SkipUCSModifiedProperties_TestActor::CreateActor()
{
    Uworld* editorworld = this->GetWorld();
    FActorSpawnParameters params;
    params.Template = (AActor*)ActorClass->GetDefaultObject();
    params.OverrideLevel=GetLevel();

    params.SpawnCollisionHandlingOverride=ESpawnActorCollisionHandlingMethod::AlwaysSS
pawn;
    params.bDeferConstruction=true;
    FTransform t;

    AMyProperty_SkipUCSModifiedProperties_BaseActor* newActor
    =Cast(editorworld-
>SpawnActor(ActorClass,&t, params));//cannot call user construction script in BP
actor
    newActor->CanCallConstruction=true;
    newActor->FinishSpawning(t);
}

void AMyProperty_SkipUCSModifiedProperties_TestActor::CleanupActor()
{
    Uworld* editorworld = this->GetWorld();

    for (TActorIterator<AActor> It(editorworld, ActorClass); It; ++It)
    {
        AActor* Actor = *It;
        editorWorld->DestroyActor(Actor);
    }
}

```

## Test Results:



Operation steps:



It can be observed that at the beginning, both properties within the component of a dynamically created Actor in the scene are modified by the constructor to be First and Second. However, after saving the level Map, only the value of MyString\_Default reverts to the default (not serialized), while the value of MyString\_Skip is preserved.

In the sample code, I added a variable CanCallConstruction to the Actor with the modifier EditDefaultsOnly, which is crucial. If set to EditAnywhere, its value would be saved on the instance of the Actor in the scene. For variables with EditDefaultsOnly, their values are saved only on the CDO. Thus, I use this value to ensure that the Actor's constructor is executed only once after I manually create the Actor, and it does not trigger the assignment logic following the constructor upon subsequent Map reloads. This highlights the difference in the values of these two properties, as otherwise, the assignment logic after the constructor would always be executed first, and the serialized values in the map would be overwritten.

## Principle:

---

Under UActorComponent, there is an AllUCSModifiedProperties list that records all properties under UActorComponent modified in the constructor of the hosting Actor. The purpose is that the modified values of these properties do not need to be serialized into the level's Actor instance.

As seen in the code of FComponentPropertyReader and FComponentPropertyWriter, properties listed in AllUCSModifiedProperties do not participate in serialization. Therefore, SkipUCSModifiedProperties serves to allow certain properties under an ActorComponent to be serialized and saved even after modification in the Actor's constructor.

```
class UActorComponent : public UObject, public IInterface_AssetData
{
    static ENGINE_API TMap<UActorComponent*, TArray<FSimpleMemberReference>>
AllUCSModifiedProperties;
}

void UActorComponent::GetUCSModifiedProperties(TSet<const FProperty*>&
ModifiedProperties) const
{
    FRWScopeLock Lock(AllUCSModifiedPropertiesLock, SLT_ReadOnly);
    if (TArray<FSimpleMemberReference>* UCSModifiedProperties =
AllUCSModifiedProperties.Find(this))
    {
        for (const FSimpleMemberReference& MemberReference :
*UCSModifiedProperties)
        {

            ModifiedProperties.Add(FMemberReference::ResolveSimpleMemberReference<FProperty>
(MemberReference));
        }
    }
}

class FDataCachePropertyReader : public FObjectReader
{
public:
    FDataCachePropertyReader(FInstanceCacheDataBase& InInstanceData)
        : FObjectReader(InInstanceData.SavedProperties)
        , InstanceData(InInstanceData)
```

```

    {
        // Include properties that would normally skip tagged serialization (e.g.
        // bulk serialization of array properties).
        ArPortFlags |= PPF_ForceTaggedSerialization;
    }

    virtual bool ShouldSkipProperty(const FProperty* InProperty) const override
    {
        return PropertiesToSkip.Contains(InProperty);
    }

};

class FComponentPropertyReader : public FDataCachePropertyReader
{
public:
    FComponentPropertyReader(UActorComponent* InComponent,
    FActorComponentInstanceData& InInstanceId)
        : FDataCachePropertyReader(InInstanceId)
    {
        InComponent->GetUCSModifiedProperties(PropertiesToSkip);

        UClass* Class = InComponent->GetClass();
        Class->SerializeTaggedProperties(*this, (uint8*)InComponent, Class,
        (uint8*)InComponent->GetArchetype());
    }
};

class FComponentPropertyWriter : public FDataCachePropertyWriter
{
public:
    FComponentPropertyWriter(const UActorComponent* Component,
    FActorComponentInstanceData& InInstanceId)
        : FDataCachePropertyWriter(Component, InInstanceId)
    {
        if (Component)
        {
            Component->GetUCSModifiedProperties(PropertiesToSkip);

            if (AActor* ComponentOwner = Component->GetOwner())
            {
                // If this is the owning Actor's root scene component, don't
                // include relative transform properties. This is handled elsewhere.
                if (Component == ComponentOwner->GetRootComponent())
                {
                    UClass* ComponentClass = Component->GetClass();
                    PropertiesToSkip.Add(ComponentClass-
>FindPropertyByName(USceneComponent::GetRelativeLocationPropertyName()));
                    PropertiesToSkip.Add(ComponentClass-
>FindPropertyByName(USceneComponent::GetRelativeRotationPropertyName()));
                    PropertiesToSkip.Add(ComponentClass-
>FindPropertyByName(USceneComponent::GetRelativeScale3DPropertyName()));
                }
            }
        }
    }
};

```

```

        SerializeProperties();
    }
};

void UActorComponent::DetermineUCSModifiedProperties()
{
    class FComponentPropertySkipper : public FArchive
    {
public:
    FComponentPropertySkipper()
        : FArchive()
    {
        this->SetIsSaving(true);

        // Include properties that would normally skip tagged serialization
        // (e.g. bulk serialization of array properties).
        ArPortFlags |= PPF_ForceTaggedSerialization;
    }

    virtual bool ShouldSkipProperty(const FProperty* InProperty) const
override
    {
        static const FName
MD_SkipUCSModifiedProperties(TEXT("SkipUCSModifiedProperties"));
        return (InProperty->HasAnyPropertyFlags(CPF_Transient)
            || !InProperty->HasAnyPropertyFlags(CPF_Edit | CPF_Interp)
            || InProperty->IsA<FMulticastDelegateProperty>()

#if WITH_EDITOR
            || InProperty->HasMetaData(MD_SkipUCSModifiedProperties)
#endif
        );
    }
} PropertySkipper;

UClass* ComponentClass = GetClass();
UObject* ComponentArchetype = GetArchetype();

for (TFieldIterator<FProperty> It(ComponentClass); It; ++It)
{
    FProperty* Property = *It;
    if( Property->ShouldSerializeValue(PropertySkipper) )
    {
        for( int32 Idx=0; Idx<Property->ArrayDim; Idx++ )
        {
            uint8* DataPtr      = Property->ContainerPtrToValuePtr
<uint8>((uint8*)this, Idx);
            uint8* DefaultValue = Property-
>ContainerPtrToValuePtrForDefaults<uint8>(ComponentClass,
(uint8*)ComponentArchetype, Idx);
            if (!Property->Identical( DataPtr, DefaultValue,
PPF_DeepCompareInstances))
            {
                UCSModifiedProperties.Add(FSimpleMemberReference());
            }
        }
    }
}

```

```

FMemberReference::FillSimpleMemberReference<FProperty>
(Property, UCSModifiedProperties.Last());
    break;
}
}
}

FRWScopeLock Lock(AllUCSModifiedPropertiesLock, SLT_Write);
if (UCSModifiedProperties.Num() > 0)
{
    AllUCSModifiedProperties.Add(this, MoveTemp(UCSModifiedProperties));
}
else
{
    AllUCSModifiedProperties.Remove(this);
}
}

```

In the source code, it is noted that this is only used in BodyInstance within UPrimitiveComponent.

```

UCLASS(Abstract, HideCategories=(Mobility, VirtualTexture), ShowCategories=
(PhysicsVolume), MinimalAPI)
class UPrimitiveComponent : public USceneComponent, public INavRelevantInterface,
public IInterface_AsyncCompilation, public IPhysicsComponent
{
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category=Collision, meta=
>ShowOnlyInnerProperties, skipUCSModifiedProperties)
        FBodyInstance BodyInstance;
}

```

## Acknowledgments:

Thanks to **Xu Ruoji** for the feedback, corrections, and example provision!

## MatchedSerializers

- **Function Description:** Used exclusively in NoExportTypes.h to denote the employment of a structured serializer. Indicates support for text-based import and export functionality
- **Usage Location:** UCLASS
- **Engine Module:** Serialization
- **Metadata Type:** Boolean
- **Associated Items:**  
UCLASS: MatchedSerializers
- **Commonality:** 0

```

if (!GetUnrealSourceFile().IsNoExportTypes())
{
    LogError(TEXT("The 'MatchedSerializers' class specifier is only valid in the
NoExportTypes.h file"));
}
ParsedClassFlags |= CLASS_MatchedSerializers;

```

Marking MatchedSerializers in a class is equivalent in effect

## NoGetter

- **Function description:** Prevent UHT from generating a C++ getter function for this attribute, effective only for properties within the structure data of sparse classes.
- **Use location:** UPROPERTY
- **Engine module:** SparseDataType
- **Metadata type:** bool
- **Associated items:**  
UCLASS: SparseClassDataType
- **Commonality:** ★

Prevents UHT from generating a C++ getter function for this attribute, effective only for properties within the structure data of sparse classes.

This should be used in conjunction with the usage of SparseClassDataTypes, and NoGetter does not impact the property's accessibility in blueprints.

## Test Code:

```

USTRUCT(BlueprintType)
struct FMySparseClassData
{
    GENERATED_BODY()

    UPROPERTY(EditDefaultsOnly)
    int32 MyInt_EditDefaultOnly = 123;

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
    int32 MyInt_BlueprintReadOnly = 1024;

    // "GetByRef" means that Blueprint graphs access a const ref instead of a
    // copy.
    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, meta = (GetByRef))
    FString MyString_EditDefault_ReadOnly = TEXT("MyName");

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, meta = (NoGetter))
    FString MyString_EditDefault_NoGetter = TEXT("MyName");
};

```

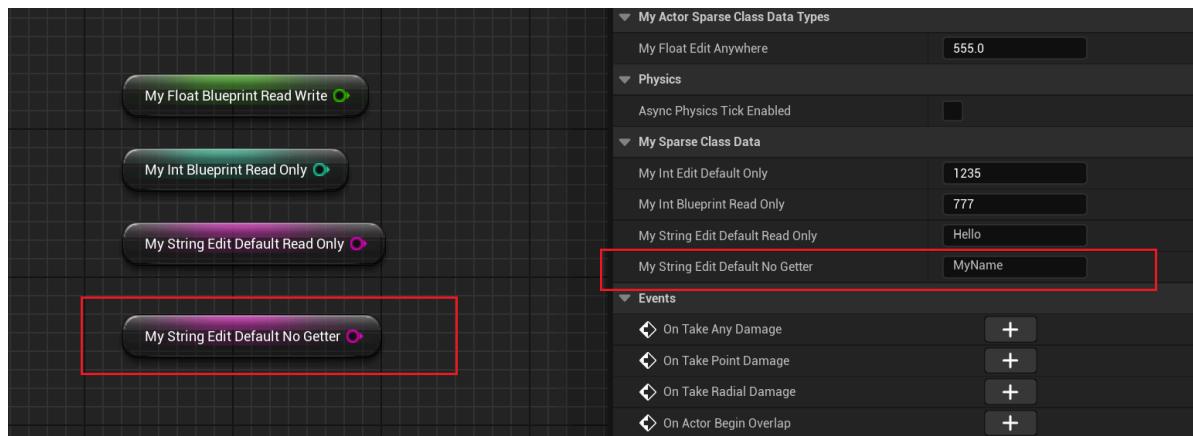
# Test Results:

In the generated .generated.h", you will find that MyString\_EditDefault\_NoGetter does not generate the corresponding C++ get function.

```
// "MyClass_SparseClassDataTypes.generated.h"

#define
FID_Hello_Source_Insider_Class_Trait_MyClass_SparseClassDataTypes_h_33_SPARSE_DATA_PROPERTY_ACCESSORS \
int32 GetMyInt_EditDefaultOnly() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
>MyInt_EditDefaultOnly; } \
int32 GetMyInt_BlueprintReadOnly() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
>MyInt_BlueprintReadOnly; } \
const FString& GetMyString_EditDefault_ReadOnly() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
>MyString_EditDefault_ReadOnly; }
```

And it is still accessible in blueprints:



## Principle:

UHT After recognizing SparseDataStruct , it will call AppendSparseDeclarations to generate the corresponding C++ attribute Get function (that is, FID\_Hello\_Source\_Insider\_Class\_Trait\_MyClass\_SparseClassDataTypes\_h\_33\_SPARSE\_DATA\_PROPERTY\_ACCESSORS those) for its internal attributes. And if the attribute is marked with NoGetter , the attribute will be filtered out SparseDataStruct .

```

private static IEnumerable<UhtProperty>
EnumerateSparseDataStructProperties(IEnumerable<UhtScriptStruct>
sparseScriptStructs)
{
    foreach (UhtScriptStruct sparseScriptStruct in sparseScriptStructs)
    {
        foreach (UhtProperty property in sparseScriptStruct.Properties)
        {
            if (!property.MetaData.ContainsKey(UhtNames.NoGetter))
            {
                yield return property;
            }
        }
    }
}

```

However, the attributes in the blueprint's property details panel still exist because the blueprint system analyzes all attributes within SparseDataStruct and adds them to the details panel. This part of the logic does not consider NoGetter, hence NoGetter does not affect the property's accessibility in blueprints, only the C++ getter function.

```

if (UScriptStruct* SparseClassDataStruct = ResolvedBaseClass-
>GetSparseClassDataStruct())
{
    SparseClassDataInstances.Add(ResolvedBaseClass, TTuple<UScriptStruct*, void*>
(SparseClassDataStruct, ResolvedBaseClass->GetOrCreateSparseClassData()));

    for (TFieldIterator<FProperty> It(SparseClassDataStruct); It; ++It)
    {
        GetCategoryProperties(ClassesToConsider, *It,
bShouldShowDisableEditOnInstance, bShouldShowHiddenProperties,
CategoriesFromBlueprints, CategoriesFromProperties, SortedCategories);
    }
}

```

## PasswordField

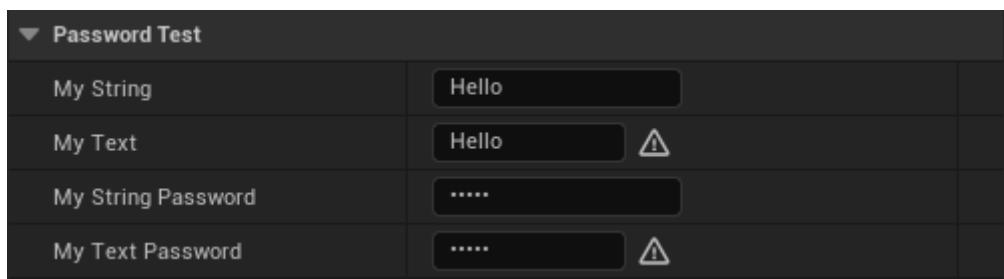
- **Function Description:** Enables the text attribute to be displayed as a password field
- **Usage Location:** UPROPERTY
- **Engine Module:** String/Text Property
- **Metadata Type:** bool
- **Restriction Type:** FName/FString/Fext
- **Commonality:** ★★★★☆

Displays the text attribute as a password field. Note that the value of the text is stored directly in memory without encryption, so security concerns must be addressed separately.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, Category = PasswordTest)  
    FString MyString = TEXT("Hello");  
  
    UPROPERTY(EditAnywhere, Category = PasswordTest)  
    FText MyText = INVTEXT("Hello");  
public:  
    UPROPERTY(EditAnywhere, Category = PasswordTest, meta = (PasswordField =  
true))  
    FString MyString_Password = TEXT("Hello");  
  
    UPROPERTY(EditAnywhere, Category = PasswordTest, meta = (PasswordField =  
true))  
    FText MyText_Password = INVTEXT("Hello");
```

## Test Results:



## Principle:

This attribute sets the Widget's IsPassword property to true. It is also worth noting from the source code that the PasswordField cannot be used concurrently with MultiLine, as this will render the password functionality ineffective.

```
void SPropertyEditorText::Construct( const FArguments& InArgs, const TSharedRef<  
class FPropertyEditor >& InPropertyEditor )  
{  
    const bool bIsPassword = PropertyHandle-  
>GetBoolMetaData(NAME_PasswordField);  
  
    if(bIsMultiLine)  
    {  
        ChildSlot  
        [  
            SAssignNew(HorizontalBox, SHorizontalBox)  
            +SHorizontalBox::Slot()  
            .FillWidth(1.0f)  
            [  
                SAssignNew(MultiLineWidget, SMultiLineEditableTextBox)  
                .Text(InPropertyEditor, &FPropertyEditor::GetValueAsText)  
                .Font(InArgs._Font)  
                .SelectAllTextWhenFocused(false)  
                .ClearKeyboardFocusOnCommit(false)  
                .OnTextCommitted(this, &SPropertyEditorText::OnTextCommitted)
```

```

        .OnVerifyTextChanged(this,
&SPropertyEditorText::OnVerifyTextChanged)
        .SelectAllTextOnCommit(false)
        .IsReadOnly(this, &SPropertyEditorText::IsReadOnly)
        .AutoWrapText(true)
        .ModifierKeyForNewLine(EModifierKey::Shift)
        // .IsPassword( bIsPassword )
    ]
];

PrimaryWidget = MultiLineWidget;
}

else
{
    ChildsSlot
[
    SAssignNew(HorizontalBox, SHorizontalBox)
    +SHorizontalBox::Slot()
    .FillWidth(1.0f)
    [
        SAssignNew( SingleLineWidget, SEditableTextBox )
        .Text( InPropertyParams, &FPropertyParams::GetValueAsText )
        .Font( InPropertyParams._Font )
        .SelectAllTextWhenFocused( true )
        .ClearKeyboardFocusOnCommit(false)
        .OnTextCommitted( this, &SPropertyEditorText::OnTextCommitted
)
        .OnVerifyTextChanged( this,
&SPropertyEditorText::OnVerifyTextChanged )
        .SelectAllTextOnCommit( true )
        .IsReadOnly(this, &SPropertyEditorText::IsReadOnly)
        .IsPassword( bIsPassword )
    ]
];
}

PrimaryWidget = SingleLineWidget;
}
}

```

## PropertyValidator

---

- **Function Description:** Specifies a UFUNCTION to validate text by name
- **Usage Location:** UPROPERTY
- **Engine Module:** String/Text Property
- **Metadata Type:** string="abc"
- **Restriction Type:** FName/FString/Fext
- **Commonality:** ★★★

Specifies a UFUNCTION by name to validate text.

This function must be decorated with UFUNCTION to be locatable by name. Since the search scope is within the class, the function must be defined within the class itself. Otherwise, an error will occur: "LogPropertyNode: Warning: PropertyValidator ufunction 'MyValidateMyString' on UMyProperty\_Text::MyString\_PropertyValidator not found."

The function signature is as shown in the following code. An empty FText return indicates no error, while a non-empty one represents an error message.

## Test Code:

```
UPROPERTY(EditAnywhere, Category = PropertyValidatorTest, meta =
(PropertyValidator = "MyValidateMyString"))
FString MyString_PropertyValidator;

UFUNCTION()
static FText MyValidateMyString(const FString& Value)
{
    if (Value.Len() <= 5 && Value.Contains("A"))
    {
        return FText();
    }
    return INVTEXT("This is invalid string");
}
```

## Test Results:



## Principle:

The principle is relatively straightforward, consisting of two parts: how to locate and create the UFUNCTION, and how to invoke the function to validate the string.

```
const FString PropertyValidatorFunctionName = PropertyHandle-
>GetMetaData(NAME_PropertyValidator);
const UClass* OuterBaseClass = PropertyHandle->GetOuterBaseClass();
if (!PropertyValidatorFunctionName.IsEmpty() && OuterBaseClass)
{
    static TSet<FString> LoggedWarnings;

    UObject* ValidatorObject = OuterBaseClass->GetDefaultObject<UObject>();
    const UFunction* PropertyValidatorFunction = ValidatorObject-
>FindFunction(*PropertyValidatorFunctionName);
    if (PropertyValidatorFunction)
    {
        if (PropertyValidatorFunction->FunctionFlags & FUNC_Static)
        {
            PropertyValidatorFunc =
FPropertyValidatorFunc::Create(ValidatorObject,
PropertyValidatorFunction->GetFName());
        }
    }
}
```

```

bool SPropertyEditorText::OnVerifyTextChanged(const FText& Text, FText& OutError)
{
    const FString& TextString = Text.ToString();

    if (PropertyValidatorFunc.IsBound())
    {
        FText Result = PropertyValidatorFunc.Execute(TextString);
        if (!Result.IsEmpty())
        {
            OutError = Result;
            return false;
        }
    }

    return true;
}

```

## MultiLine

---

- **Function Description:** Allows the text attribute editing box to support line breaks.
- **Usage Location:** UPROPERTY
- **Engine Module:** String/Text Property
- **Metadata Type:** bool
- **Restriction Type:** FName/FString/Fext
- **Commonality:** ★★★★★

Enables line breaks in the text attribute editing box. The string after line breaks is separated by "\r\n".

## Test Code:

---

```

UPROPERTY(EditAnywhere, Category = MultiLineTest, meta = (MultiLine = true))
FString MyString_MultiLine = TEXT("Hello");

UPROPERTY(EditAnywhere, Category = MultiLineTest, meta = (MultiLine = true))
FText MyText_MultiLine = INVTEXT("Hello");

UPROPERTY(EditAnywhere, Category = MultiLineTest, meta = (MultiLine = true,
PasswordField = true))
FString MyString_MultiLine_Password = TEXT("Hello");

UPROPERTY(EditAnywhere, Category = MultiLineTest, meta = (MultiLine = true,
PasswordField = true))
FText MyText_MultiLine_Password = INVTEXT("Hello");

```

# Test Results:

Press Shift+Enter to insert a line break.

Password Test		
My String Password	.....	
My Text Password	.....	⚠
Multi Line Test		
My String Multi Line	Hello fer	↶
My Text Multi Line	Hello erer	⚠ ↶
My String Multi Line Password	Hello ewrer	↶
My Text Multi Line Password	Hello werert	⚠ ↶

## Principle:

The principle is quite simple, involving the creation of a specific multi-line editing control, SMultiLineEditText, based on the bIsMultiLine flag.

```
void SPropertyEditorText::Construct( const FArguments& InArgs, const TSharedRef<  
    class FPropertyEditor >& InPropertyEditor )  
{  
    bIsMultiLine = PropertyHandle->GetBoolMetaData(NAME_MultiLine);  
  
    if(bIsMultiLine)  
    {  
        ChildSlot  
        [  
            SAssignNew(HorizontalBox, SHorizontalBox)  
            +SHorizontalBox::Slot()  
            .Fillwidth(1.0f)  
            [  
                SAssignNew(MultiLineWidget, SMultiLineEditText)  
                .Text(InPropertyEditor, &FPropertyEditor::GetValueAsText)  
                .Font(InArgs._Font)  
                .SelectAllTextWhenFocused(false)  
                .ClearKeyboardFocusOnCommit(false)  
                .OnTextCommitted(this, &SPropertyEditorText::OnTextCommitted)  
                .OnVerifyTextChanged(this,  
                    &SPropertyEditorText::OnVerifyTextChanged)  
                .SelectAllTextOnCommit(false)  
                .IsReadOnly(this, &SPropertyEditorText::IsReadOnly)  
                .AutoWrapText(true)  
                .ModifierKeyForNewLine(EModifierKey::Shift)  
                // .IsPassword( bIsPassword )  
            ]  
        ];  
  
        PrimaryWidget = MultiLineWidget;  
    }  
}
```

```
}
```

# AllowedCharacters

- **Function description:** The text box only permits the input of these specific characters.
- **Usage location:** UPROPERTY
- **Engine module:** String/Text Property
- **Metadata type:** string = "abc"
- **Restriction type:** FName/FString/Fext
- **Commonality:** ★★★

Only these characters are allowed to be entered in the text box.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, Category = AllowedCharactersTest, meta =  
(AllowedCharacters = "abcde"))  
    FString Mystring_AllowedCharacters;  
    UPROPERTY(EditAnywhere, Category = AllowedCharactersTest, meta =  
(AllowedCharacters = "你好"))  
    FString Mystring_AllowedCharacters_Chinese;
```

## Test Code:

It is evident that the first field only accepts input of abcde; any attempt to input fgh results in an error. When testing Chinese characters, if the corresponding Chinese characters are pasted, they are accepted without issue. Otherwise, an error is generated, and input is disallowed.



## 测试效果:

The SPropertyEditorText actually stores an FCharRangeList named AllowedCharacters to restrict input characters. Similarly, when the string is modified, it validates whether the characters are valid.

```
FCharRangeList AllowedCharacters;  
  
AllowedCharacters.InitializeFromString(PropertyHandle->GetMetaData(NAME_AllowedCharacters));  
  
bool SPropertyEditorText::OnVerifyTextChanged(const FText& Text, FText& OutError)  
{  
    const FString& TextString = Text.ToString();  
  
    if (MaxLength > 0 && TextString.Len() > MaxLength)
```

```

    }

    OutError = FText::Format(LOCTEXT("PropertyTextTooLongError", "This value
is too long ({0}/{1} characters)", TextString.Len(), MaxLength);
    return false;
}

if (!AllowedCharacters.IsEmpty())
{
    if (!TextString.IsEmpty() &&
!AllowedCharacters.AreAllCharsIncluded(TextString))
    {
        TSet<TCHAR> InvalidCharacters =
AllowedCharacters.FindCharsNotIncluded(TextString);
        FString InvalidCharactersString;
        for (TCHAR Char : InvalidCharacters)
        {
            if (!InvalidCharactersString.IsEmpty())
            {
                InvalidCharactersString.AppendChar(TEXT(' '));
            }
            InvalidCharactersString.AppendChar(Char);
        }
        OutError =
FText::Format(LOCTEXT("PropertyTextCharactersNotAllowedError", "The value may not
contain the following characters: {0}"),
FText::FromString(InvalidCharactersString));
        return false;
    }
}

if (PropertyValidatorFunc.IsBound())
{
    FText Result = PropertyValidatorFunc.Execute(TextString);
    if (!Result.IsEmpty())
    {
        OutError = Result;
        return false;
    }
}

return true;
}

```

Test Results:

```

/** Initializes this instance with the character ranges represented by the passed
definition string.
* A definition string contains characters and ranges of characters, one after
another with no special separators between them.
* Characters - and \ must be escaped like this: \- and \\.
*
* Examples:
*     "aT._" <-- Letters 'a' and 'T', dot and underscore.
*     "a-zA-Z0-9._" <-- All letters from 'a' to 'z', letter 'T', dot and underscore.
*     "a-zA-Z0-9\-\\" <-- All lowercase and uppercase letters, all digits, dot
and underscore.
*     "a-zA-Z0-9\-\\" <-- All lowercase and uppercase letters, all digits,
minus sign, backslash, dot and underscore.
*/

```

## GetOptions

- **Function Description:** Specifies a function from an external class to provide options for the FName or FString attributes, offering a list of values in the dropdown menu within the details panel.
- **Usage Locations:** UPARAM, UPROPERTY
- **Engine Module:** String/Text Property
- **Metadata Type:** string="abc"
- **Restricted Types:** FString, FName
- **Associated Items:** GetKeyOptions, GetValueOptions
- **Commonly Used:** ★★★★☆

Specifies a function from an external class to provide options for FName or FString attributes, presenting a list of values in the dropdown menu within the details panel.

- Only affects FName or FString properties; FText is not supported.
- Also applicable to containers such as TArray, TMap, and TSet.
- Also can be used on variables within internal structures. The key point is that when searching for a function, it is done by finding OuterObject::Function, allowing even variables within internal structures to access functions from the external class. However, for another unrelated class, the "Module.Class.Function" format must be used to find it, or it will only return an empty result.
- The function prototype is TArray FuncName(), which returns a string type, even if the property type is FName, as the engine internally performs the conversion.
- The function can be a member function or a static function.

## Test Code:

```

USTRUCT(BlueprintType)
struct INSIDER_API FMyOptionsTest
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, meta = (Getoptions = "MyGetoptions_Static"))

```

```

FString MyString_GetOptions;

UPROPERTY(EditAnywhere, meta = (GetOptions = "MyGetOptions_Static"))
TArray<FString> MyArray_GetOptions;

UPROPERTY(EditAnywhere, meta = (GetOptions = "MyGetOptions_Static"))
TSet<FString> MySet_GetOptions;

UPROPERTY(EditAnywhere, meta = (GetOptions = "MyGetOptions_Static"))
TMap<FString, int32> MyMap_GetOptions;
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Text :public UObject
{
public:
    UPROPERTY(EditAnywhere, Category = GetOptions)
    FString MyString_NoOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    FString MyString_GetOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    FName MyName_GetOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    FText MyText_GetOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    TArray<FString> MyArray_GetOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    TSet<FString> MySet_GetOptions;

    UPROPERTY(EditAnywhere, Category = GetOptions, meta = (GetOptions =
"MyGetOptions"))
    TMap<FString, int32> MyMap_GetOptions;

    UFUNCTION()
    static TArray<FString> MyGetOptions_Static() { return TArray<FString>{"Cat",
"Dog"}; }

    UFUNCTION()
    TArray<FString> MyGetOptions() { return TArray<FString>{"First", "Second",
"Third"}; }
public:
    UPROPERTY(EditAnywhere, Category = GetOptionsStruct)
    FMyOptionsTest MyStruct_GetOptions;

    UPROPERTY(EditAnywhere, Category = GetKeyValueOptions, meta = (GetKeyOptions
= "MyGetOptions", GetValueOptions="MyGetOptions_Static"))

```

```

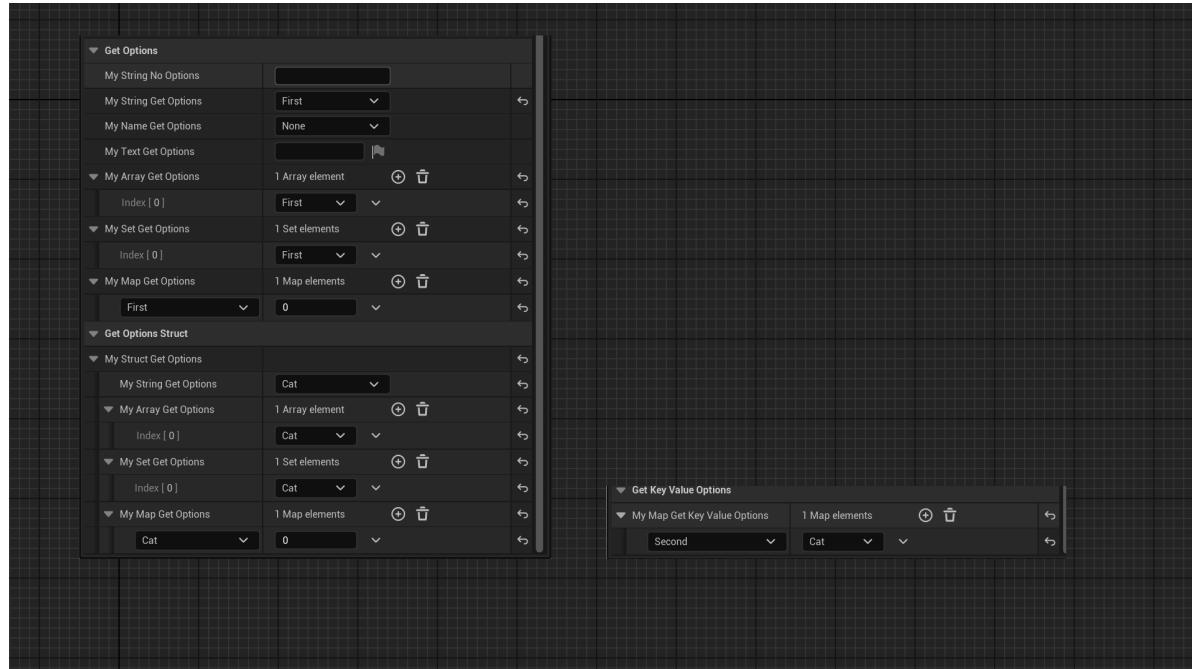
    TMap< FString, FName> MyMap_GetKeyValueOptions;
}

```

## Test Results:

As shown in the figure below, FText does not take effect. All other properties marked with GetOptions have a dropdown menu in the details panel.

When using TMap, you can also use GetKeyOptions and GetValueOptions to provide different option lists for the Key and Value respectively, as seen in MyMap\_GetKeyValueOptions.



## Principle:

The general process involves using GetPropertyOptionsMetaDataKey to determine if a property supports option box editing, then calling

GetPropertyOptions to invoke the specified function to obtain the option list, and finally building the ComboBoxWidget based on the values in this list.

```

void PropertyEditorUtils::GetPropertyOptions(TArray<UObject*>&
InOutContainers, FString& InOutPropertyName,
TArray<TSharedPtr<FString>>& InOutOptions)
{
    // Check for external function references
    if (InOutPropertyName.Contains(TEXT(".")))
    {
        InOutContainers.Empty();
        UFunction* GetOptionsFunction = FindObject<UFunction>(nullptr,
*InOutPropertyName, true);

        if (ensureMsgf(GetOptionsFunction && GetOptionsFunction-
>HasAnyFunctionFlags(EFunctionFlags::FUNC_Static), TEXT("Invalid GetOptions:
%s")), *InOutPropertyName))
        {
            UObject* GetOptionsCDO = GetOptionsFunction->GetOuterUClass()-
>GetDefaultObject();
            GetOptionsFunction->GetName(InOutPropertyName);
        }
    }
}

```

```

        InOutContainers.Add(GetOptionsCDO);
    }

}

if (InOutContainers.Num() > 0)
{
    TArray< FString> OptionIntersection;
    TSet< FString> OptionIntersectionSet;

    for (UObject* Target : InOutContainers)
    {
        TArray< FString> StringOptions;
        {
            FEditorScriptExecutionGuard ScriptExecutionGuard;

            FCachedPropertyPath Path(InOutPropertyName);
            if (!PropertyPathHelpers::GetPropertyValues(Target, Path,
StringOptions))
            {
                TArray< FName> NameOptions;
                if (PropertyPathHelpers::GetPropertyValues(Target, Path,
NameOptions))
                {
                    Algo::Transform(NameOptions, StringOptions, []() (const
FName& InName) { return InName.ToString(); });
                }
            }
        }

        // If this is the first time there won't be any options.
        if (OptionIntersection.Num() == 0)
        {
            OptionIntersection = StringOptions;
            OptionIntersectionSet = TSet< FString>(StringOptions);
        }
        else
        {
            TSet< FString> StringOptionsSet(StringOptions);
            OptionIntersectionSet =
StringOptionsSet.Intersect(OptionIntersectionSet);
            OptionIntersection.RemoveAll([&OptionIntersectionSet] (const
FString& Option){ return !OptionIntersectionSet.Contains(Option); });
        }

        // If we're out of possible intersected options, we can stop.
        if (OptionIntersection.Num() == 0)
        {
            break;
        }
    }

    Algo::Transform(OptionIntersection, InOutOptions, []() (const FString&
InString) { return MakeShared< FString>(InString); });
}

```

```

FName GetPropertyOptionsMetaDataKey(const FProperty* Property)
{
    // only string and name properties can have options
    if (Property->IsA(FStrPropertyParams::StaticClass()) || Property-
>IsA(FNamePropertyParams::StaticClass()))
    {
        const FProperty* OwnerProperty = Property->GetOwnerProperty();
        static const FName GetOptionsName("GetOptions");
        if (OwnerProperty->HasMetaData(GetOptionsName))
        {
            return GetOptionsName;
        }

        // Map properties can have separate options for keys and values
        const FMapPropertyParams* MapProperty = CastField<FMapPropertyParams>(OwnerProperty);
        if (MapProperty)
        {
            static const FName GetKeyOptionsName("GetKeyOptions");
            if (MapProperty->HasMetaData(GetKeyOptionsName) && MapProperty-
>GetKeyPropertyParams() == Property)
            {
                return GetKeyOptionsName;
            }

            static const FName GetValueOptionsName("GetValueOptions");
            if (MapProperty->HasMetaData(GetValueOptionsName) && MapProperty-
>GetValuePropertyParams() == Property)
            {
                return GetValueOptionsName;
            }
        }
    }

    return NAME_None;
}

TSharedPtr<SWidget> SGraphPinString::TryBuildComboBoxWidget()
{
    PropertyEditorUtils::GetPropertyOptions(PropertyContainers,
    GetOptionsFunctionName, ComboBoxOptions);
}

```

## GetKeyOptions

---

- **Function Description:** Provides the option values for the option box within the details panel for FName/FString keys within a TMap
- **Usage Location:** UPROPERTY
- **Engine Module:** String/Text Property
- **Metadata Type:** string="abc"
- **Restriction Type:** Keys in TMap must be FName/FString
- **Related Items:** GetOptions

# GetValueOptions

- **Function description:** To provide the option values for the option boxes within the detail panel, specifically for the FName/FString values in the TMap
- **Usage Location:** UPROPERTY
- **Engine Module:** String/Text Property
- **Metadata Type:** string="abc"
- **Restriction type:** The restriction applies to the FName/FString values used as the Value within the TMap
- **Related Items:** GetOptions

# MaxLength

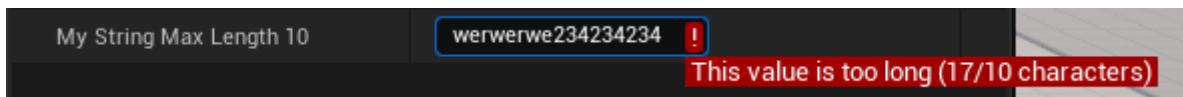
- **Function Description:** Limits the maximum length of text within a text edit box
- **Usage Location:** UPROPERTY
- **Engine Module:** String/Text Property
- **Metadata Type:** int32
- **Restriction Type:** FName/FString/Fext
- **Commonality:** ★★★★☆

Enforces a maximum length for text within a text edit box. However, it is still possible to input any value freely at the C++ or blueprint level.

## Test Code:

```
UPROPERTY(EditAnywhere, Category = MaxLengthTest, meta = (maxLength = 10))
FString MyString_MaxLength10 = TEXT("Hello");
```

## Test Code:



## 测试效果:

When the string in the text box is modified, the current length is checked, and an error is reported if it exceeds the limit.

For FName attributes, MaxLength is further restricted to within NAME\_SIZE (1024).

```
MaxLength = PropertyHandle->GetIntMetaData(NAME_MaxLength);
if (InPropertyEditor->PropertyIsA(FNameProperty::StaticClass()))
{
    MaxLength = MaxLength <= 0 ? NAME_SIZE - 1 : FMath::Min(MaxLength, NAME_SIZE
    - 1);
}

bool SPropertyEditorText::OnVerifyTextChanged(const FText& Text, FText& OutError)
```

```

{
    const FString& TextString = Text.ToString();

    if (MaxLength > 0 && TextString.Len() > MaxLength)
    {
        OutError = FText::Format(LOCTEXT("PropertyTextTooLongError", "This
value is too long ({0}/{1} characters)", TextString.Len(), MaxLength);
        return false;
    }
}

```

## MakeStructureDefaultValue

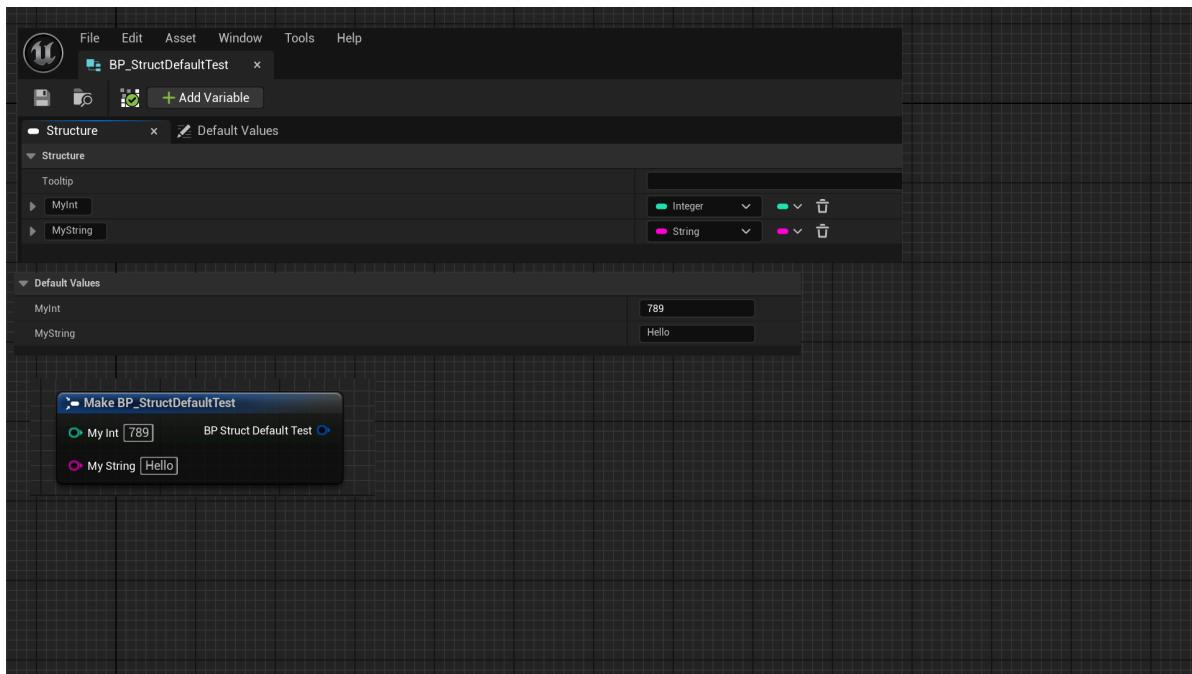
- **Function Description:** Stores the default values for attributes within custom structures in BP.
- **Usage Location:** UPROPERTY
- **Engine Module:** Struct
- **Metadata Type:** bool
- **Restriction Type:** User-defined Struct in BP
- **Commonality:** ★

Stores the default values of attributes in custom structures within BP.

- In C++, the default values for attributes in the USTRUCT we define do not need to be saved in metadata, as the structure's constructor is automatically called when an instance is created, thus properly initializing the values.
  - However, user-defined structures in Blueprints lack a constructor mechanism. Therefore, we require a dedicated tab for entering default attribute values. These default values are then stored in the attribute's metadata.
- 

## Test Code:

Define a structure named BP\_StructDefaultTest in the blueprint and populate it with default values.



## Test Results:

Print the relevant information using the test command line. You will see the actual property names of MyInt and MyString, as well as the value of MakeStructureDefaultValue.

```
[struct BP_StructDefaultTest   UserDefinedStruct->ScriptStruct->Struct->Field->Object /Game/Struct/BP_StructDefaultTest.BP_StructDefaultTest]
(BlueprintType = true, Tooltip = )
    ObjectFlags: RF_Public | RF_Standalone | RF_Transactional | RF_WasLoaded |
    RF_LoadCompleted
        Outer: Package /Game/Struct/BP_StructDefaultTest
    StructFlags: STRUCT_NoFlags
    Size: 24
{
    (DisplayName = MyInt, Tooltip = , MakeStructureDefaultValue = 789)
    0-[4] int32 MyInt_3_CC664A574A072369083883B38EA2F129;
        PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
        CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash
        ObjectFlags: RF_Public | RF_LoadCompleted
        Outer: UserDefinedStruct
    /Game/Struct/BP_StructDefaultTest.BP_StructDefaultTest
        Path: IntProperty
    /Game/Struct/BP_StructDefaultTest.BP_StructDefaultTest:MyInt_3_CC664A574A07236908
    3883B38EA2F129
        (DisplayName = MyString, Tooltip = , MakeStructureDefaultValue = Hello)
        8-[16] FString MyString_6_D8FAF5D6454C781C2D5175ACF266C394;
            PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
            CPF_HasGetValueTypeHash
            ObjectFlags: RF_Public | RF_LoadCompleted
            Outer: UserDefinedStruct
    /Game/Struct/BP_StructDefaultTest.BP_StructDefaultTest
        Path: StrProperty
    /Game/Struct/BP_StructDefaultTest.BP_StructDefaultTest:MyString_6_D8FAF5D6454C781
    C2D5175ACF266C394
};
```

## Principle:

When a structure is created and saved in the blueprint, if a default value is not empty, it will be assigned to MakeStructureDefaultValue. This value can then be used during the MakeStruct operation.

```
void UK2Node_MakeStruct::FMakeStructPinManager::CustomizePinData(UEdGraphPin* Pin, FName SourcePropertyName, int32 ArrayIndex, FProperty* Property) const
{
    const FString& MetadataDefaultValue = Property->GetMetaData(TEXT("MakeStructureDefaultValue"));
    if (!MetadataDefaultValue.IsEmpty())
    {
        Schema->SetPinAutogeneratedDefaultValue(Pin, MetadataDefaultValue);
        return;
    }
}

static void
FUserDefinedStructureCompilerInner::CreateVariables(UUserDefinedStruct* Struct,
const class UEdGraphSchema_K2* Schema, FCompilerResultsLog& MessageLog)
{
    if (!VarDesc.DefaultValue.IsEmpty())
    {
        VarProperty->SetMetaData(TEXT("MakeStructureDefaultValue"),
*VarDesc.DefaultValue);
    }
}
```

## IgnoreForMemberInitializationTest

- **Functional Description:** Allows this property to bypass the uninitialized validation for the structure.
- **Usage Location:** UPROPERTY
- **Engine Module:** Struct
- **Metadata Type:** bool
- **Restricted Type:** Attributes within C++ structures
- **Commonality:** ★★

Allows this property to bypass the uninitialized validation for the structure.

- What is meant by "uninitialized" refers to variables in a C++ structure that are not initialized in the constructor and do not have initial values assigned directly
- Uninitialized structure validation refers to the verification tool provided by the engine, which can be invoked using the console command "CoreUObject.AttemptToFindUninitializedScriptStructMembers". This will output information on all uninitialized variables within the engine.
- In UE, USTRUCT is simply a pure C++ structure, unlike classes defined with UCLASS, which are all UObject, and whose UPROPERTY attributes are automatically initialized to 0. UPROPERTY within structures, however, are not automatically initialized and require manual initialization.

- In practice, if a developer is aware that the uninitialized state of a variable will not impact logic, there is no issue with it remaining uninitialized. However, in reality, it is often due to sheer laziness or forgetfulness that attributes are not initialized. It is therefore recommended to initialize all attributes within a structure. There are, however, special cases where certain attributes should not be initialized, such as some FGuid variables in source code examples, which are only assigned values when they are actually used. Initializing them before use holds little significance. In such cases, IgnoreForMemberInitializationTest can be used to make the property skip this validation, thereby avoiding error messages.

## Test Code:

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_InitTest
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta=IgnoreForMemberInitializationTest)
    int32 MyProperty_IgnoreTest;
};
```

## Test Results:

It is evident that MyProperty encountered an error due to the absence of IgnoreForMemberInitializationTest.

在控制台调用CoreUObject::AttemptToFindUninitializedScriptStructMembers后：

```
LogClass: Error: IntProperty FMyStruct_InitTest::MyProperty is not initialized
properly. Module:Insider File:Property/Struct/MyProperty_Struct.h
```

## Principle:

This command line invokes AttemptToFindUninitializedScriptStructMembers and subsequently calls FindUninitializedScriptStructMembers to identify uninitialized variables within UScriptStruct. The specific method of identification can be examined within this function.

```
static void FindUninitializedScriptStructMembers(UScriptStruct* ScriptStruct,
EScriptStructTestCtorSyntax ConstructorSyntax, TSet<const FProperty*>&
OutUninitializedProperties)
{
    for (const FProperty* Property : TFieldRange<FProperty>(ScriptStruct,
EFieldIteratorFlags::ExcludeSuper))
    {
        #if WITH_EDITORONLY_DATA
```

```

        static const FName
NAME_IgnoreForMemberInitializationTest(TEXT("IgnoreForMemberInitializationTest"))
;
    if (Property->HasMetaData(NAME_IgnoreForMemberInitializationTest))
    {
        continue;
    }
#endif // WITH_EDITORONLY_DATA

}

//called by this
FStructutils::AttemptToFindUninitializedScriptStructMembers();

// Command line invocation
CoreUObject::AttemptToFindUninitializedScriptStructMembers

```

## HasNativeBreak

- **Function description:** Specifies a C++ UFunction as the implementation for the Break node within this structure
- **Use location:** USTRUCT
- **Engine module:** Struct
- **Metadata type:** string = "abc"
- **Related items:** HasNativeMake
- **Frequency:** ★★★★☆

Specifies a C++ UFunction as the implementation for the Break node within this structure

Enter the full path value for a static *UFunction*, typically in the format "/Script/Module.Class.Function"

This function is usually BlueprintThreadSafe, as pure Make and Break functions typically have no side effects, allowing them to be called across threads without issues.

### Test Code:

```

//(BlueprintType = true, HasNativeBreak =
/Script/Insider.MyHasNativeStructHelperLibrary.BreakMyHasNativeStruct,
HasNativeMake =
/Script/Insider.MyHasNativeStructHelperLibrary.MakeMyHasNativeStruct,
ModuleRelativePath = Struct/MyStruct_NativeMakeBreak.h)
USTRUCT(BlueprintType, meta = (HasNativeBreak =
"/Script/Insider.MyHasNativeStructHelperLibrary.BreakMyHasNativeStruct",
HasNativeMake =
"/Script/Insider.MyHasNativeStructHelperLibrary.MakeMyHasNativeStruct"))
struct INSIDER_API FMyStruct_HasNative
{
    GENERATED_BODY()

    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyReadwrite;
    UPROPERTY(BlueprintReadOnly, EditAnywhere)
    float MyReadOnly;
}

```

```

UPROPERTY(EditAnywhere)
float MyNotBlueprint;
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyStruct_HasDefaultMakeBreak
{
GENERATED_BODY()

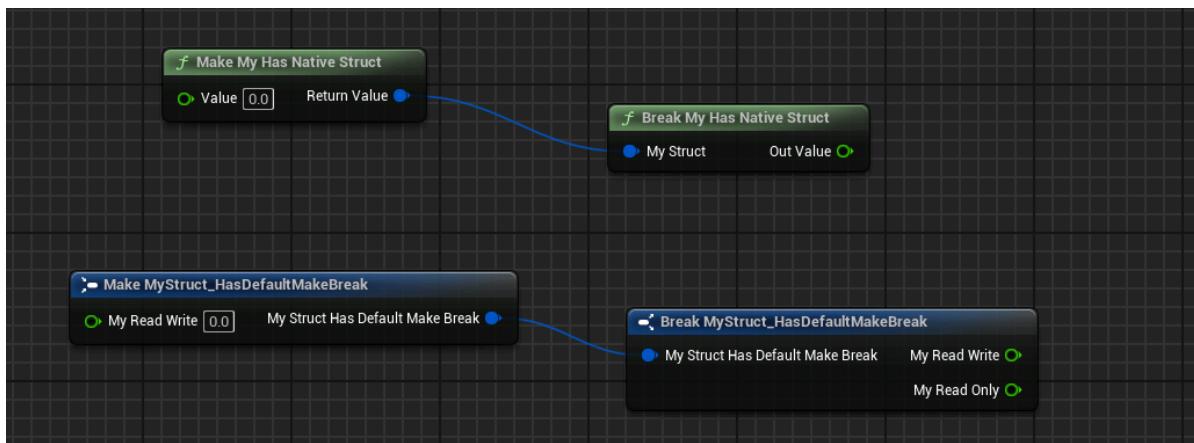
UPROPERTY(BlueprintReadWrite, EditAnywhere)
float MyReadWrite;
UPROPERTY(BlueprintReadOnly, EditAnywhere)
float MyReadOnly;
UPROPERTY(EditAnywhere)
float MyNotBlueprint;
};

UCLASS()
class UMyHasNativeStructHelperLibrary : public UBlueprintFunctionLibrary
{
GENERATED_BODY()
public:
UFUNCTION(BlueprintPure, meta = (BlueprintThreadSafe))
static void BreakMyHasNativeStruct(const FMyStruct_HasNative& myStruct,
float& outValue)
{
    outValue = myStruct.MyReadWrite + myStruct.MyReadOnly +
myStruct.MyNotBlueprint;
}

UFUNCTION(BlueprintPure, meta = (BlueprintThreadSafe))
static FMyStruct_HasNative MakeMyHasNativeStruct(float value)
{
    FMyStruct_HasNative result;
    result.MyReadWrite = value;
    result.MyReadOnly = value;
    result.MyNotBlueprint = value;
    return result;
}
};

```

## Blueprint Node:



## The principle is:

Locate the UFunction by the value configured in the Meta, thus the complete path value provided during configuration must allow the UFunction to be found. The function's signature is automatically reflected and its information extracted to the UK2Node\_CallFunction node, enabling the construction of Make and Break blueprint nodes with various styles.

```
E:\P4V\Engine\Source\Editor\BlueprintGraph\Private\EdGraphSchema_K2.cpp

const FString& MetaData = StructType->GetMetaData(FBlueprintMetadata::MD_NativeMakeFunction);
const UFunction* Function = FindObject<UFunction>(nullptr, *MetaData, true);

UK2Node_CallFunction* CallFunctionNode;

if (Params.bTransient || Params.CompilerContext)
{
    CallFunctionNode = (Params.bTransient ? NewObject<UK2Node_CallFunction>(Graph) : Params.CompilerContext->SpawnIntermediateNode<UK2Node_CallFunction>(GraphNode, Params.SourceGraph));
    CallFunctionNode->SetFromFunction(Function);
    CallFunctionNode->AllocateDefaultPins();
}
else
{
    FGraphNodeCreator<UK2Node_CallFunction> MakeStructCreator(*Graph);
    CallFunctionNode = MakeStructCreator.CreateNode(false);
    CallFunctionNode->SetFromFunction(Function);
    MakeStructCreator.Finalize();
}

SplitPinNode = CallFunctionNode;
```

## HasNativeMake

- **Function description:** Specifies a C++ UFunction as the implementation for the Make node within this structure
- **Usage location:** USTRUCT
- **Metadata type:** string="abc"
- **Associated items:** HasNativeBreak
- **Commonly used:** ★★★★☆

## DataflowFlesh

- **Function Description:** ScriptStruct /Script/DataflowNodes.FloatOverrideDataflowNode
- **Usage Location:** USTRUCT
- **Engine Module:** Struct
- **Metadata Type:** boolean
- **Commonality:** 0

No examples of application were found in the source code

# AllowedTypes

---

- **Function Description:** Specifies allowable asset types for FPrimaryAssetId.
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** strings = "a, b, c"
- **Restriction Type:** FPrimaryAssetId
- **Commonality:** ★★★

Allowed asset types can be specified for FPrimaryAssetId.

## Test Code:

---

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_PrimaryAsset :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "PrimaryAsset")
    FPrimaryAssetId MyPrimaryAsset;

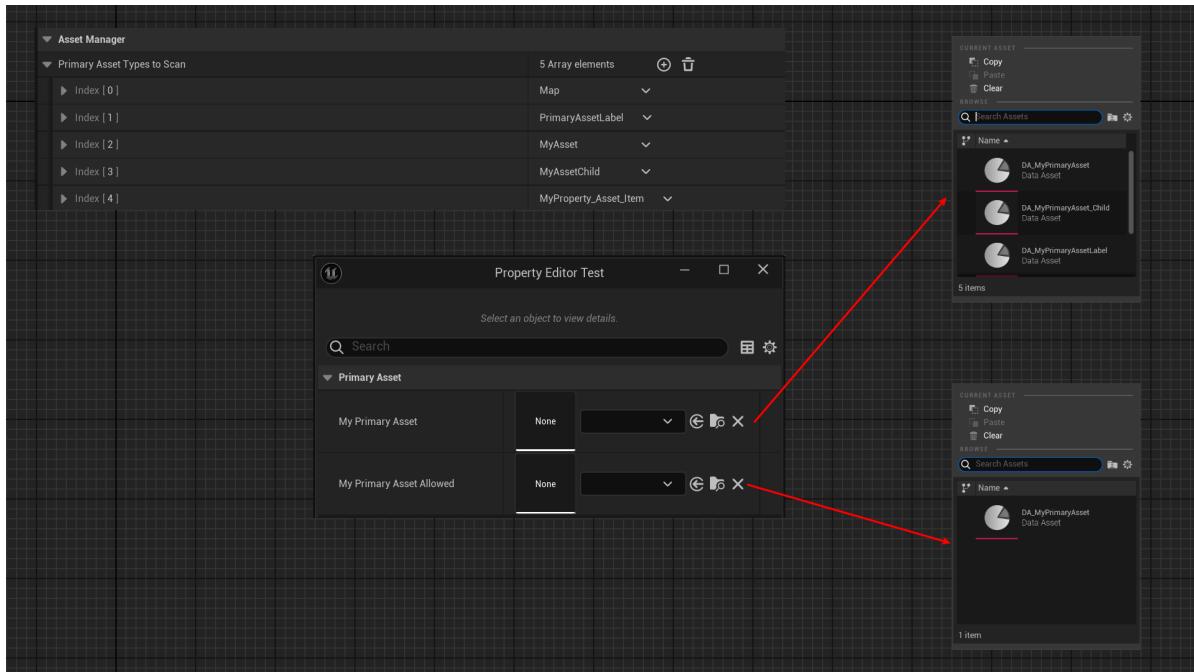
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "PrimaryAsset", meta=
(AccordionGroup="MyAsset"))
    FPrimaryAssetId MyPrimaryAsset_Allowed;
};
```

## Test Results:

---

Several UPrimaryDataAssets have been predefined in the project and are also configured in ProjectSettings. (Refer to other articles for details on how to define them.)

Only one option is available for MyPrimaryAsset\_Allowed, indicating that it is subject to restrictions.



## Principle:

In the customization of `FPrimaryAssetIdCustomization` for `FPrimaryAssetId`, the tag is inspected, the value of `AllowedTypes` is parsed, and it is set to the member variable `AllowedTypes`, ultimately achieving the filtering effect.

```

TArray<FPrimaryAssetType> FPrimaryAssetIdCustomization::AllowedTypes;

void FPrimaryAssetIdCustomization::CustomizeHeader(TSharedRef<class
IPropertyHandle> InStructPropertyHandle, class FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    check(UAssetManager::IsInitialized());

    StructPropertyHandle = InStructPropertyHandle;

    const FString& TypeFilterString = StructPropertyHandle-
>GetMetaData("AllowedTypes");
    if( !TypeFilterString.IsEmpty() )
    {
        TArray< FString > CustomTypeFilterNames;
        TypeFilterString.ParseIntoArray(CustomTypeFilterNames, TEXT(","), true);

        for(auto It = CustomTypeFilterNames.CreateConstIterator(); It; ++It)
        {
            const FString& TypeName = *It;

            AllowedTypes.Add(*TypeName);
        }
    }

    IAssetManagerEditorModule::MakePrimaryAssetIdSelector(
        FOnGetPrimaryAssetDisplayText::CreateSP(this,
        &FPrimaryAssetIdCustomization::GetDisplayText),
        FOnSetPrimaryAssetId::CreateSP(this,
        &FPrimaryAssetIdCustomization::OnIdSelected),

```

```
bAllowClear, AllowedTypes, AllowedClasses, DisallowedClasses)
```

```
}
```

## BaseClass

- **Function Description:** Used exclusively within the StateTree module to restrict the base class type for selection in FStateTreeEditorNode.
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** bool
- **Restriction Type:** FStateTreeEditorNode attribute
- **Commonality:** ★

Only utilized in the StateTree module to limit the base class type that can be selected by FStateTreeEditorNode.

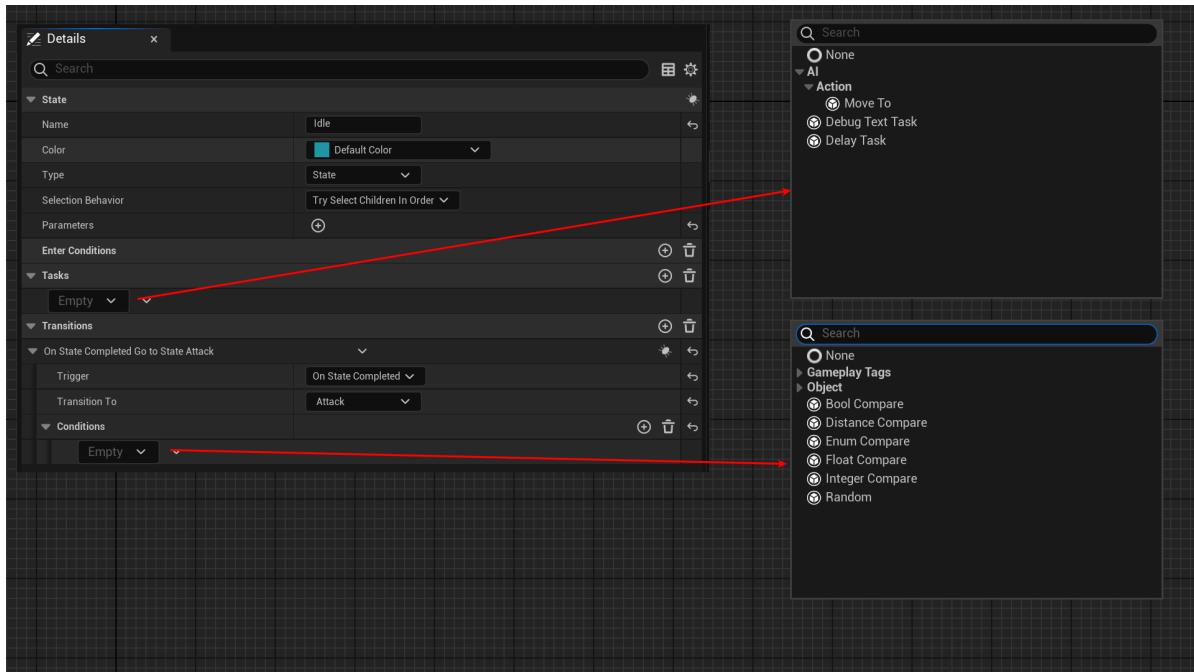
## Source Code Example:

```
USTRUCT()
struct STATETREEEDITORMODULE_API FStateTreeTransition
{
    /** Conditions that must pass so that the transition can be triggered. */
    UPROPERTY(EditDefaultsOnly, Category = "Transition", meta = (BaseStruct =
"/Script/StateTreeModule.StateTreeConditionBase", BaseClass =
"/Script/StateTreeModule.StateTreeConditionBlueprintBase"))
    TArray<FStateTreeEditorNode> Conditions;

    UPROPERTY(EditDefaultsOnly, Category = "Tasks", meta = (BaseStruct =
"/script/StateTreeModule.StateTreeTaskBase", BaseClass =
"/script/StateTreeModule.StateTreeTaskBlueprintBase"))
    TArray<FStateTreeEditorNode> Tasks;
}
```

## Test Results:

It is evident that although both Conditions and Tasks are of type FStateTreeEditorNode, the content in the option list differs. This discrepancy is due to the differing BaseStruct and BaseClass specifications above, which define the base class types for the structure and blueprint class, respectively.



## Principle:

This property is retrieved in the UI customization of `FStateTreeNodeEditorNodeDetails` and then employed to filter the available node types.

```
void FStateTreeNodeEditorNodeDetails::CustomizeHeader(TSharedRef<class IPropertyHandle> StructPropertyHandle, class FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    static const FName BaseClassMetaName(TEXT("Baseclass")); // TODO: move these names into one central place.
    const FString BaseClassName = StructProperty->GetMetaData(BaseClassMetaName);
    BaseClass = UClass::TryFindTypeSlow<UClass>(BaseClassName);
}
```

## AllowedClasses

- **Function description:** Used on class or object selectors to specify that the selected object must belong to certain base class types.
- **Use location:** UPROPERTY
- **Engine module:** TypePicker
- **Metadata type:** strings = "a, b, c"
- **Restriction types:** TSubClassOf, UClass, FSoftClassPath, UObject, FSoftObjectPath, FPrimaryAssetId, FComponentReference,
- **Associated items:** ExactClass, DisallowedClasses, GetAllowedClasses, GetDisallowedClasses
- **Commonly used:** ★★★

Used on class or object selectors to specify that the selected objects must belong to certain base class types.

- The applicable attributes for class selectors are: TSubClassOf, UClass, FSoftClassPath.  
*Attributes that cannot be applied are: UScriptStruct*
- The applicable attributes for object selectors are: UObject\*, FSoftObjectPath, FPrimaryAssetId, FComponentReference
- These selectors often display a long list of object resources for selection, thus AllowedClasses can be used to restrict the types they must belong to.
- AllowedClasses can separate multiple types with commas, allowing simultaneous support for multiple type filters.

## Test code:

---

```

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|TSubclassof")
    TSubclassOf<UObject> MyClass_NoAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|TSubclassof", meta = (AllowedClasses = "MyCommonObject"))
    TSubclassOf<UObject> MyClass_AllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|UClass*")
    UClass* MyClassPtr_NoAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|UClass*", meta = (AllowedClasses = "MyCommonObject"))
    UClass* MyClassPtr_AllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSOFTCLASSPATH")
    FSoftClassPath MySoftClass_NoAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSOFTCLASSPATH", meta = (AllowedClasses = "MyCommonObject"))
    FSoftClassPath MySoftClass_AllowedClasses;

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSOFTOBJECTPATH")
    UObject* MyObject_NoAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSOFTOBJECTPATH", meta = (AllowedClasses =
"/script/Engine.Texture2D"))
    UObject* MyObject_AllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSOFTOBJECTPATH")
    FSoftObjectPath MySoftObject_NoAllowedClasses;

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FSoftObjectPath", meta = (AllowedClasses =
"/script/Engine.Texture2D"))
FSoftObjectPath MySoftObject_AllowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FPrimaryAssetId")
FPrimaryAssetId MyPrimaryAsset_NoAllowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"AllowedClassesTest|FPrimaryAssetId", meta = (AllowedClasses =
"MyPrimaryDataAsset"))
FPrimaryAssetId MyPrimaryAsset_AllowedClasses;

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_Class :public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"AllowedClassesTest|FComponentReference", meta = (UseComponentPicker))
    FComponentReference MyComponentReference_NoAllowedClasses;

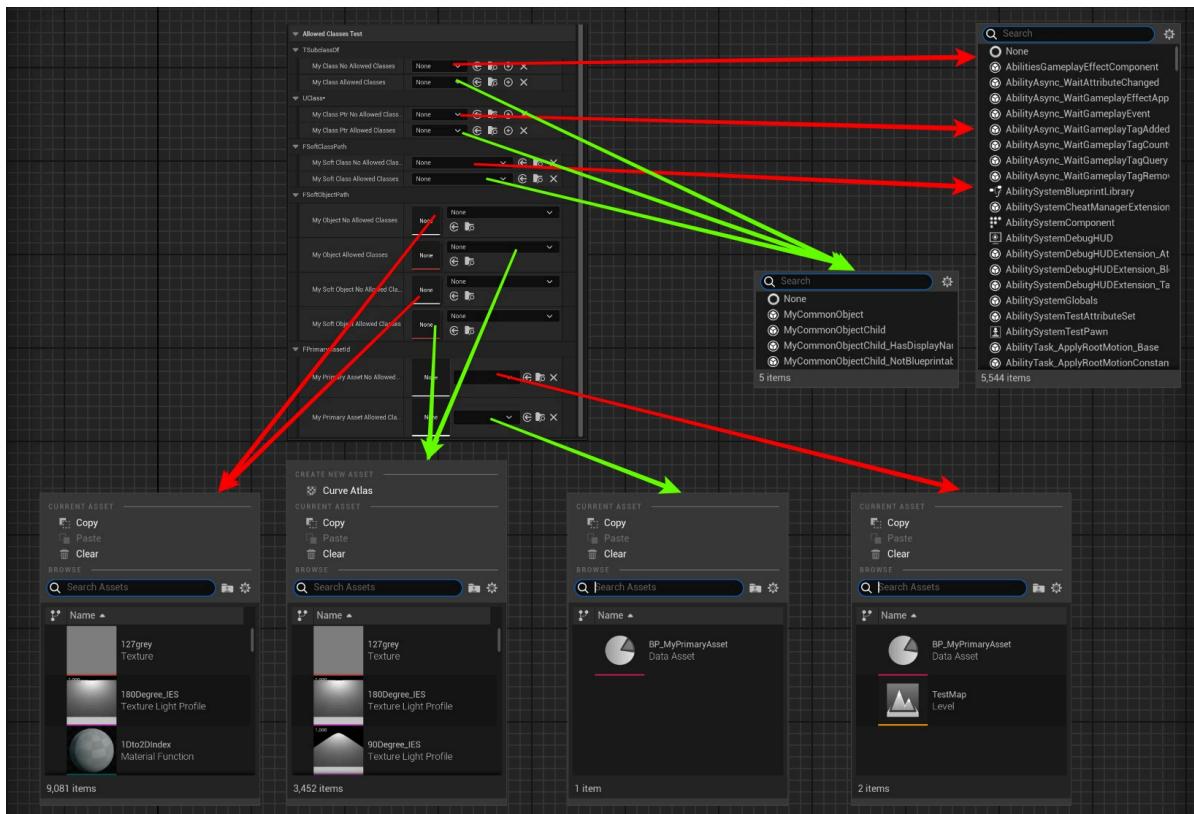
    UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"AllowedClassesTest|FComponentReference", meta =
(UseComponentPicker, AllowedClasses = "MyActorComponent"))
    FComponentReference MyComponentReference_AllowedClasses;
};

UCLASS(BlueprintType)
class INSIDER_API UMyPrimaryDataAsset :public UPrimaryDataAsset
{}

```

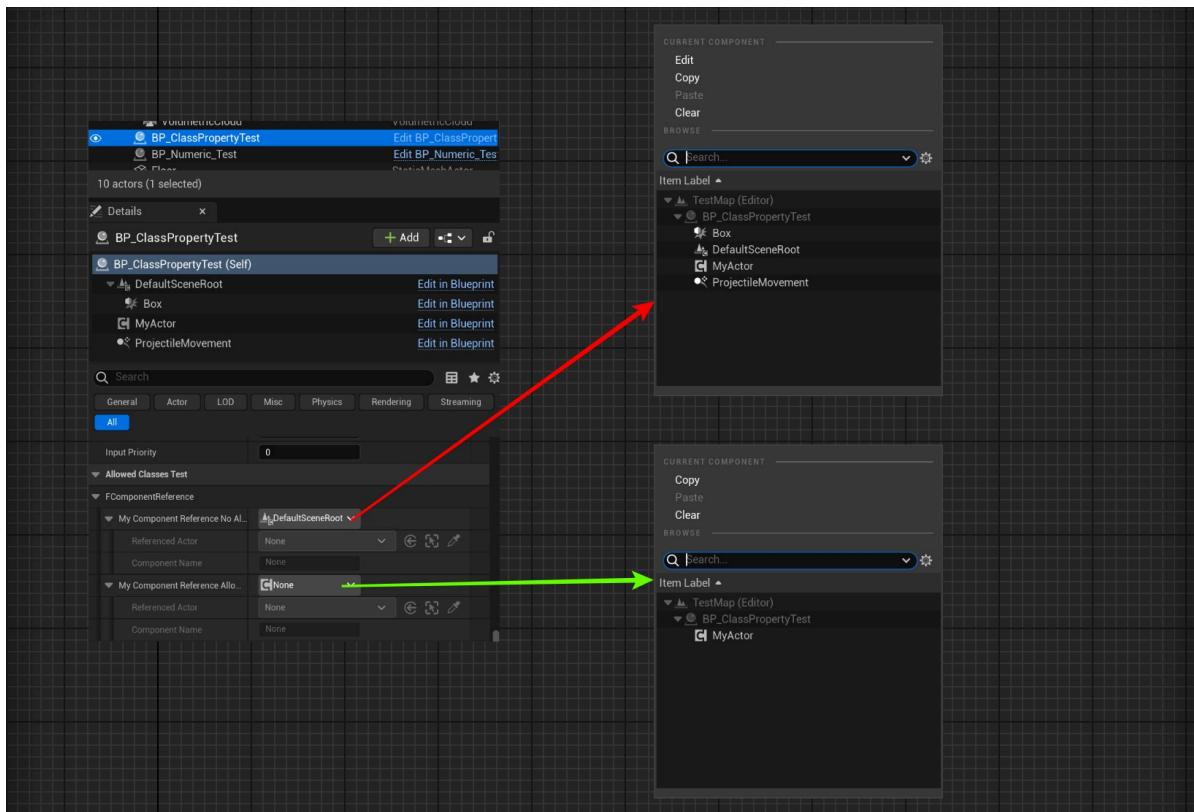
## Test results:

- On the class selector, it is evident that after adding AllowedClasses, the selection is restricted to the subclasses of MyCommonObject.
- After adding AllowedClasses = "/Script/Engine.Texture2D" to the object selector, the type is restricted to textures.
- In the asset filtering for the FPrimaryAssetId attribute, after adding AllowedClasses, it can be restricted to the MyPrimaryDataAsset type, as seen in the BP\_MyPrimaryAsset. Note that UMyPrimaryDataAsset needs to be configured in ProjectSettings.



### Testing the effect of FComponentReference:

With the above code, it is clear that by default, the selection range for FComponentReference is all components under the current Actor. By adding AllowedClasses, the selection range can be limited to the MyActorComponent described in the code.



# Principle:

In the source code, it is often seen that in various Customization or SPropertyEditorXXX types, there is a judgment of AllowedClasses and DisallowedClasses, followed by a type filter using IsChildOf.

```
void FPrimaryAssetIdCustomization::CustomizeHeader(TSharedRef<class  
IPropertyHandle> InStructPropertyHandle, class FDetailWidgetRow& HeaderRow,  
IPropertyTypeCustomizationUtils& StructCustomizationUtils)  
{  
    AllowedClasses =  
PropertyCustomizationHelpers::GetClassesFromMetadataString(StructPropertyHandle-  
>GetMetaData("Allowedclasses"));  
    DisallowedClasses =  
PropertyCustomizationHelpers::GetClassesFromMetadataString(StructPropertyHandle-  
>GetMetaData("Disallowedclasses"));  
}  
void FComponentReferenceCustomization::BuildClassFilters()  
{  
    const FString& AllowedClassesFilterString = PropertyHandle-  
>GetMetaData(NAME_Allowedclasses);  
    ParseClassFilters(AllowedClassesFilterString, AllowedActorClassFilters,  
AllowedComponentClassFilters);  
  
    const FString& DisallowedClassesFilterString = PropertyHandle-  
>GetMetaData(NAME_Disallowedclasses);  
    ParseClassFilters(DisallowedClassesFilterString,  
DisallowedActorClassFilters, DisallowedComponentClassFilters);  
}  
void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPropertyHandle>  
InPropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&  
StructCustomizationUtils)  
{  
    TArray<const UClass*> AllowedClasses =  
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyHandle-  
>GetMetaData("Allowedclasses"));  
    TArray<const UClass*> DisallowedClasses =  
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyHandle-  
>GetMetaData("Disallowedclasses"));  
}  
  
void PropertyEditorUtils::GetAllowedAndDisallowedClasses(const TArray<UObject*>&  
ObjectList, const FProperty& MetadataProperty, TArray<const UClass*>&  
AllowedClasses, TArray<const UClass*>& DisallowedClasses, bool bExactClass, const  
UClass* ObjectClass)  
{  
    AllowedClasses =  
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOw-  
nerProperty()->GetMetaData("Allowedclasses"));  
    DisallowedClasses =  
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOw-  
nerProperty()->GetMetaData("Disallowedclasses"));  
    if (MetadataProperty.GetOwnerProperty()-  
>HasMetaData("GetAllowedclasses"))  
    {
```

```

        const FString GetAllowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()->GetMetaData("GetAllowedClasses");
    }

    if (MetadataProperty.GetOwnerProperty()-
>HasMetaData("GetDisallowedClasses"))
{
    const FString GetDisallowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()->GetMetaData("GetDisallowedClasses");
}

void SPropertyEditorAsset::InitializeClassFilters(const FProperty* Property)
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*MetadataProperty, AllowedClassFilters, DisallowedClassFilters, bExactClass,
ObjectClass);
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPropertyEditor >& InPropertyEditor)
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*Property, AllowedClassFilters, DisallowedClassFilters, false);
}

TSharedRef<swidget> SPropertyEditorEditInline::GenerateClassPicker()
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*Property, AllowedClassFilters, DisallowedClassFilters, false);
}

```

## #ExactClass

- **Function Description:** When both AllowedClasses and GetAllowedClasses are set, ExactClass specifies that only the intersection of types that are exactly the same in both collections will be selected; otherwise, the intersection plus its subclasses will be selected.
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** bool
- **Restricted Types:** FSoftObjectPath, UObject\*
- **Associated Items:** AllowedClasses
- **Commonality:** ★

When both AllowedClasses and GetAllowedClasses are set, ExactClass specifies that only the intersection of types that are exactly the same in both collections will be selected; otherwise, the intersection plus its subclasses will be selected.

- Only applies to FSoftObjectPath and UObject\*, as currently only SPropertyEditorAsset utilizes this Meta.

- The distinction between the two is not easily understood, because if the consistent type happens to be the base class of other types, it does not matter whether its subclasses are included.
- In the test code, an example is constructed where Texture2D and TextureCube are selected from AllowedClasses, and TextureLightProfile and TextureCube are selected from GetAllowedClasses. We know that TextureLightProfile inherits from Texture2D. Therefore, if ExactClass is true, the final filtered type is TextureCube, as the types must be completely identical. If ExactClass is false, the final filtered types are TextureCube and TextureLightProfile, because TextureLightProfile is a subclass of Texture2D, and thus will be selected.

## Test Code:

```

UFUNCTION()
TArray<UClass*> MyGetAllowedClassesFunc()
{
    TArray<UClass*> classes;
    classes.Add(UTextureLightProfile::StaticClass());
    classes.Add(UTextureCube::StaticClass());
    return classes;
}

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ExactClassTest|UObject*", meta = (AllowedClasses =
"/Script/Engine.Texture2D,/Script/Engine.TextureCube", GetAllowedClasses =
"MyGetAllowedClassesFunc"))
UObject* MyObject_NoExactClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ExactClassTest|UObject*", meta = (ExactClass, AllowedClasses =
"/Script/Engine.Texture2D,/Script/Engine.TextureCube", GetAllowedClasses =
"MyGetAllowedClassesFunc"))
UObject* MyObject_ExactClass;

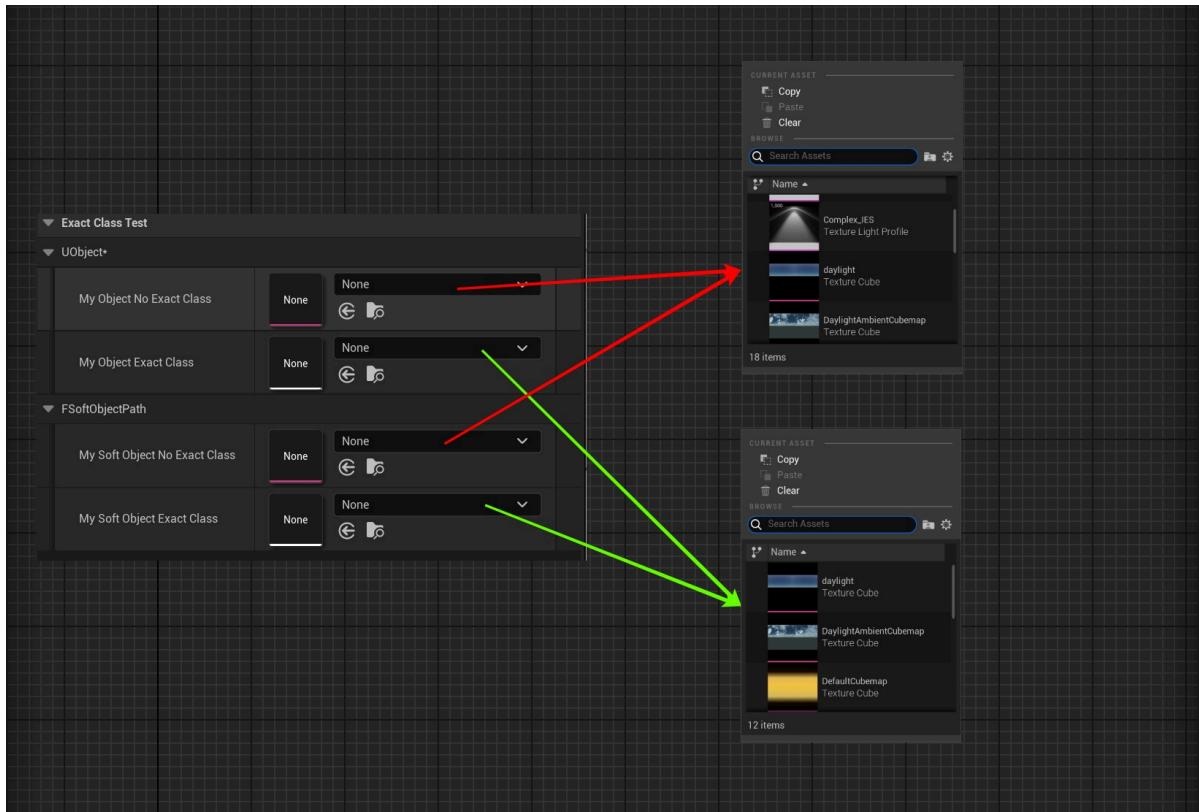
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ExactClassTest|FSoftObjectPath", meta = (AllowedClasses =
"/Script/Engine.Texture2D,/Script/Engine.TextureCube", GetAllowedClasses =
"MyGetAllowedClassesFunc"))
FSoftObjectPath MySoftObject_NoExactClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"ExactClassTest|FSoftObjectPath", meta = (ExactClass, AllowedClasses =
"/Script/Engine.Texture2D,/Script/Engine.TextureCube", GetAllowedClasses =
"MyGetAllowedClassesFunc"))
FSoftObjectPath MySoftObject_ExactClass;

```

## Test Results:

- It can be observed that without ExactClass, the filtered types are TextureCube and TextureLightProfile, with a total of 18 items.
- With ExactClass, the filtered type is only TextureCube, with a total of 12 items.



## Principle:

After testing and reviewing the source code logic, it has been confirmed that `ExactClass` must be used in conjunction with `GetAllowedClasses`, and both must be present in `AllowedClasses`. The "`ExactClass`" attribute is passed to `GetAllowedAndDisallowedClasses`. This is the only place in the entire source code where it is used. Following the logic within `GetAllowedAndDisallowedClasses`, when both `AllowedClasses` and `GetAllowedClasses` have values, the `bExactClass` condition is evaluated.

- If `bExactClass` is false, the values in `AllowedClasses` and `GetAllowedClasses` must be equal or one must be a subclass of the other, meaning they must belong to the same inheritance hierarchy, and the final selection will include subclasses rather than base classes.
- If `bExactClass` is true, only types that are exactly the same in both `AllowedClasses` and `GetAllowedClasses` will be selected.

```
void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<  
FPropertyEditor >& InPropertyEditor)  
{  
    //The default value is false, hence ExactClass is not utilized  
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList, *Property,  
    AllowedClassFilters, DisallowedClassFilters, false);  
}  
void SPropertyEditorAsset::InitializeClassFilters(const FProperty* Property)  
{  
    if (Property == nullptr)  
    {  
        AllowedClassFilters.Add(ObjectClass);  
        return;  
    }  
  
    // Account for the allowed classes specified in the property metadata  
    const FProperty* MetadataProperty = GetActualMetadataProperty(Property);
```

```

bExactClass = GetTagOrBoolMetadata(MetadataProperty, "ExactClass", false);

TArray<UObject*> ObjectList;
if (PropertyEditor && PropertyEditor->GetPropertyHandle()->IsValidHandle())
{
    PropertyEditor->GetPropertyHandle()->GetOuterObjects(ObjectList);
}
else if (PropertyHandle.IsValid())
{
    PropertyHandle->GetOuterObjects(ObjectList);
}

PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*MetadataProperty, AllowedClassFilters, DisallowedClassFilters, bExactClass,
ObjectClass);

if (AllowedClassFilters.Num() == 0)
{
    // always add the object class to the filters
    AllowedClassFilters.Add(ObjectClass);
}
}

void GetAllowedAndDisallowedClasses(const TArray<UObject*>& ObjectList, const
FPropertyParams MetadataProperty, TArray<const UClass*>& AllowedClasses, TArray<const
UClass*>& DisallowedClasses, bool bExactClass, const UClass* ObjectClass)
{
    TArray<const UClass*> CurrentAllowedClassFilters =
MoveTemp(AllowedClasses);
    ensure(AllowedClasses.IsEmpty());
    for (const UClass* MergedClass : MergedClasses)
    {
        // Keep classes that match both allow list
        for (const UClass* CurrentClass : CurrentAllowedClassFilters)
        {
            if (CurrentClass == MergedClass || (!bExactClass &&
CurrentClass->IsChildOf(MergedClass)))
            {
                AllowedClasses.Add(CurrentClass);
                break;
            }
            if (!bExactClass && MergedClass->IsChildOf(CurrentClass))
            {
                AllowedClasses.Add(MergedClass);
                break;
            }
        }
    }
}
}

```

# DisallowedClasses

- **Function description:** Used on class or object selectors to specify the exclusion of certain base class types for the selected objects.
- **Use location:** UPROPERTY
- **Engine module:** TypePicker
- **Metadata type:** strings = "a, b, c"
- **Limit types:** TSubClassOf, UClass\*, FSoftClassPath, FComponentReference
- **Related items:** AllowedClasses
- **Frequency:** ★★

Used on class or object selectors to specify the exclusion of certain base class types for the selected objects.

- The applicable attributes for class selectors are: TSubClassOf, UClass, *FSoftClassPath*.  
*Attributes that cannot be applied are: UScriptStruct*
- The applicable properties for the object selector are: FComponentReference. It will be noticed that compared to AllowedClasses, there are fewer "UObject\*", FSoftObjectPath, FPrimaryAssetId". This is because the UI corresponding to these three is SAssetPicker, which does not implement the exclusion using FARFilter.RecursiveClassPathsExclusionSet.

## Test Code:

```
public:  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"DisallowedClassesTest|TSubclassof")  
    TSubclassof<UObject> MyClass_NoDisallowedClasses;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"DisallowedClassesTest|TSubclassof", meta = (DisallowedClasses =  
"/Script/GameplayAbilities.AbilityAsync"))  
    TSubclassof<UObject> MyClass_DisallowedClasses;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"DisallowedClassesTest|UClass*")  
    UClass* MyClassPtr_NoDisallowedClasses;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"DisallowedClassesTest|UClass*", meta = (DisallowedClasses =  
"/Script/GameplayAbilities.AbilityAsync"))  
    UClass* MyClassPtr_DisallowedClasses;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"DisallowedClassesTest|FSoftClassPath")  
    FSoftClassPath MySoftClass_NoDisallowedClasses;  
  
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"DisallowedClassesTest|FSoftClassPath", meta = (DisallowedClasses =  
"/Script/GameplayAbilities.AbilityAsync"))  
    FSoftClassPath MySoftClass_DisallowedClasses;  
public://Not work
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FSoftObjectPath")
UObject* MyObject_NoDisallowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FSoftObjectPath", meta = (DisallowedClasses =
"/Script/Engine.Texture2D"))
UObject* MyObject_DisallowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FSoftObjectPath")
FSoftObjectPath MySoftObject_NoDisallowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FSoftObjectPath", meta = (DisallowedClasses =
"/Script/Engine.Texture2D"))
FSoftObjectPath MySoftObject_DisallowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FPrimaryAssetId")
FPrimaryAssetId MyPrimaryAsset_NoDisallowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"DisallowedClassesTest|FPrimaryAssetId", meta = (DisallowedClasses =
"MyPrimaryDataAsset"))
FPrimaryAssetId MyPrimaryAsset_DisallowedClasses;

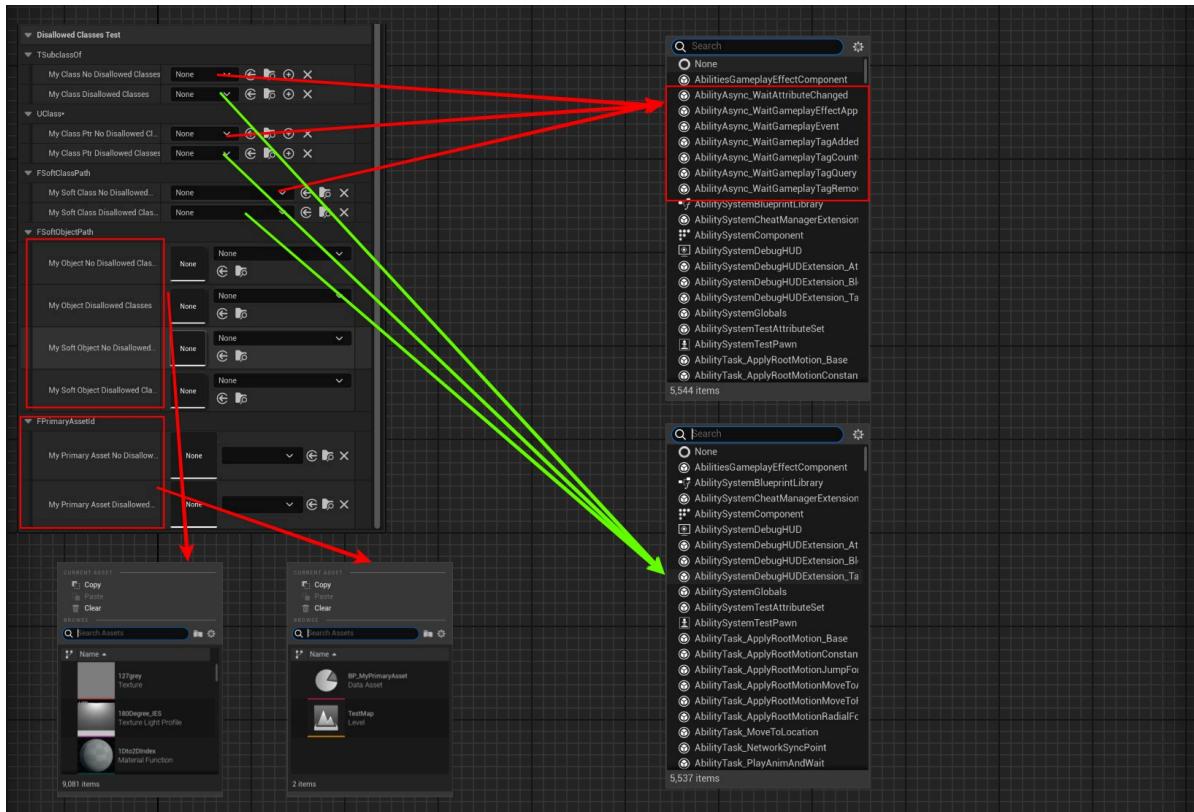
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor_Class :public AActor
{
    GENERATED_BODY()
public:
    UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"DisallowedClassesTest|FComponentReference", meta = (UseComponentPicker))
    FComponentReference MyComponentReference_NoDisallowedClasses;

    UPROPERTY(EditInstanceOnly, BlueprintReadWrite, Category =
"DisallowedClassesTest|FComponentReference", meta = (UseComponentPicker,
DisallowedClasses = "MyActorComponent"))
    FComponentReference MyComponentReference_DisallowedClasses;
};

```

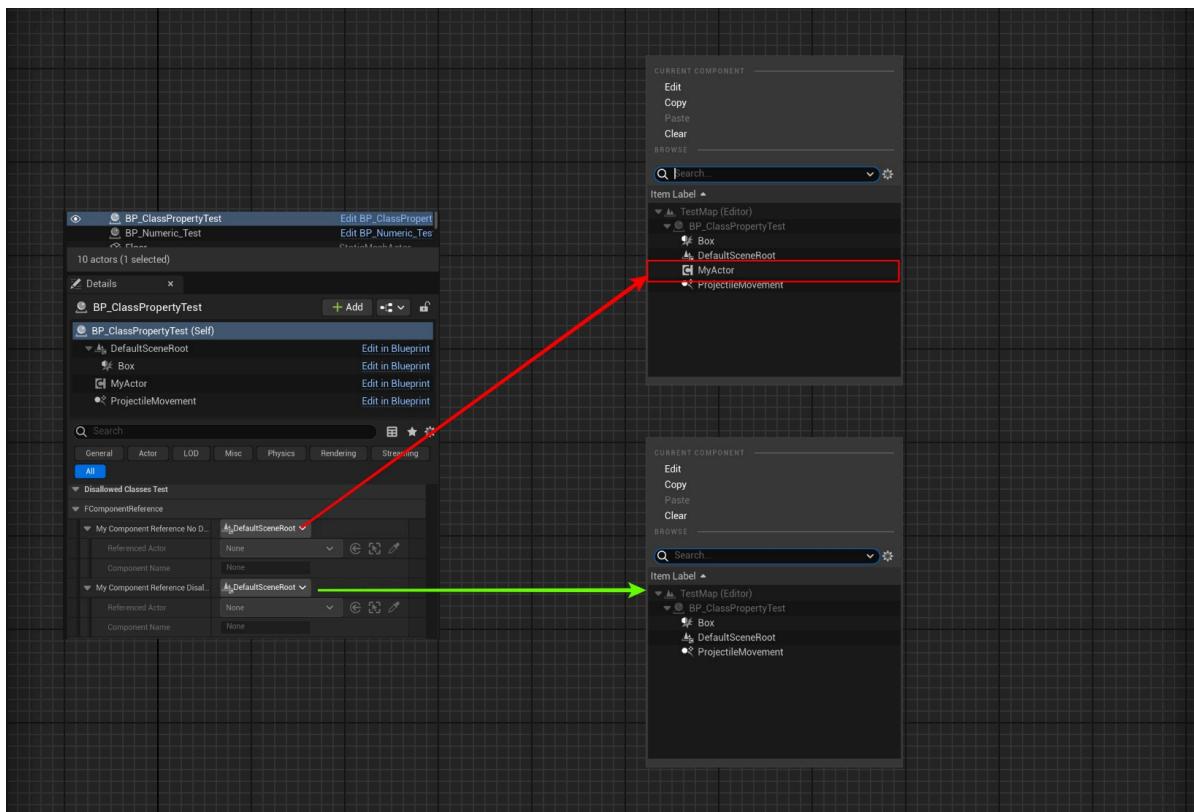
## Test Results:

- In the class selector, it is evident that after adding DisallowedClasses, the AbilityAsync class is excluded.
- However, on the object selector, it does not necessarily take effect. The list of selectable objects remains the same for both. The reason is that SAssetPicker does not actually apply DisallowedClasses.



The test effect on FComponentReference is:

DisallowedClasses can exclude MyActorComponent.



## Principle:

The primary principles have already been demonstrated with AllowedClasses. The data for DisallowedClasses is mainly set in DisallowedClassFilters. Subsequently, when creating ClassViewer, it is set in ClassFilter, and ultimately, it still relies on the IsChildOf judgment. However, it is important to note that not all class and object selectors use DisallowedClasses, as it is not implemented in AssetPicker.

```

TSharedRef<FPropertyEditorClassFilter> PropEdClassFilter =
MakeShared<FPropertyEditorClassFilter>();
PropEdClassFilter->ClassPropertyMetaClass = MetaClass;
PropEdClassFilter->InterfaceThatMustBeImplemented = RequiredInterface;
PropEdClassFilter->bAllowAbstract = bAllowAbstract;
PropEdClassFilter->AllowedClassFilters = AllowedClassFilters;
PropEdClassFilter->DisallowedClassFilters = DisallowedClassFilters;

ClassViewerOptions.ClassFilters.Add(PropEdClassFilter);

ClassFilter = FModuleManager::LoadModuleChecked<FClassViewerModule>
("Classviewer").CreateClassFilter(ClassViewerOptions);

template <typename TClass>
bool FPropertyEditorClassFilter::IsClassAllowedHelper(TClass InClass)
{
    bool bMatchesFlags = !InClass->HasAnyClassFlags(CLASS_Hidden |
CLASS_HideDropDown | CLASS_Deprecated) &&
(bAllowAbstract || !InClass->HasAnyClassFlags(CLASS_Abstract));

    if (bMatchesFlags && InClass->IsChildOf(classPropertyMetaClass)
        && (!InterfaceThatMustBeImplemented || InClass-
>ImplementsInterface(InterfaceThatMustBeImplemented)))
    {
        auto PredicateFn = [InClass](const UClass* Class)
        {
            return InClass->IsChildOf(Class);
        };

        if (DisallowedClassFilters.FindByPredicate(PredicateFn) == nullptr &&
            (AllowedClassFilters.Num() == 0 ||
AllowedClassFilters.FindByPredicate(PredicateFn) != nullptr))
        {
            return true;
        }
    }

    return false;
}

void SAssetPicker::Construct( const FArguments& InArgs )
{
    if (InArgs._AssetPickerConfig.bAddFilterUI)
    {
        // We create available classes here. These are used to hide away the type
        // filters in the filter list that don't match this list of classes
        TArray<UClass*> FilterclassList;
        for(auto Iter = CurrentBackendFilter.ClassPaths.CreateIterator(); Iter;
++Iter)
        {
            FTopLevelAssetPath className = (*Iter);
            UClass* FilterClass = FindObject<UClass>(className);
            if(FilterClass)
            {

```

```

        FilterClassList.AddUnique(FilterClass);
    }
}

```

## GetAllowedClasses

- **Function description:** Used in class or object selectors to specify, through a function, that the selected objects must belong to certain base class types.
- **Usage location:** UPROPERTY
- **Engine module:** TypePicker
- **Metadata type:** string = "abc"
- **Restricted types:** TSubClassOf, UClass, UObject, FSoftObjectPath  
Code: TArray<UClass\*> FuncName() const;
- **Associated items:** AllowedClasses
- **Commonly used:** ★★

AllowedClass specifies the base class by directly referencing the class name as a string. GetAllowedClasses takes it a step further by allowing a function to return the filtered base classes, offering greater flexibility for dynamic and custom selections.

Of course, GetAllowedClasses does not support as many attribute types as AllowedClass, limited to: TSubClassOf, UClass, UObject, FSoftObjectPath

## Test Code:

```

public:
UFUNCTION()
TArray<UClass*> MyGetAllowedClassesFunc()
{
    TArray<UClass*> classes;
    classes.Add(UMyCommonObject::StaticClass());
    classes.Add(UTexture2D::StaticClass());
    classes.Add(UMyPrimaryDataAsset::StaticClass());

    return classes;
}

public:
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetAllowedClassesTest|TSubclassof")
TSubclassOf<UObject> MyClass_NoGetAllowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetAllowedClassesTest|TSubclassof", meta = (GetAllowedClasses =
"MyGetAllowedClassesFunc"))
TSubclassOf<UObject> MyClass_GetAllowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetAllowedClassesTest|UClass*")
UClass* MyClassPtr_NoGetAllowedClasses;

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetAllowedClassesTest|UClass*", meta = (GetAllowedClasses =
"MyGetAllowedClassesFunc"))
    UClass* MyClassPtr_GetAllowedClasses;
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetAllowedClassesTest|FSoftObjectPath")
    UObject* MyObject_NoGetAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetAllowedClassesTest|FSoftObjectPath", meta = (GetAllowedClasses =
"MyGetAllowedClassesFunc"))
    UObject* MyObject_GetAllowedClasses;

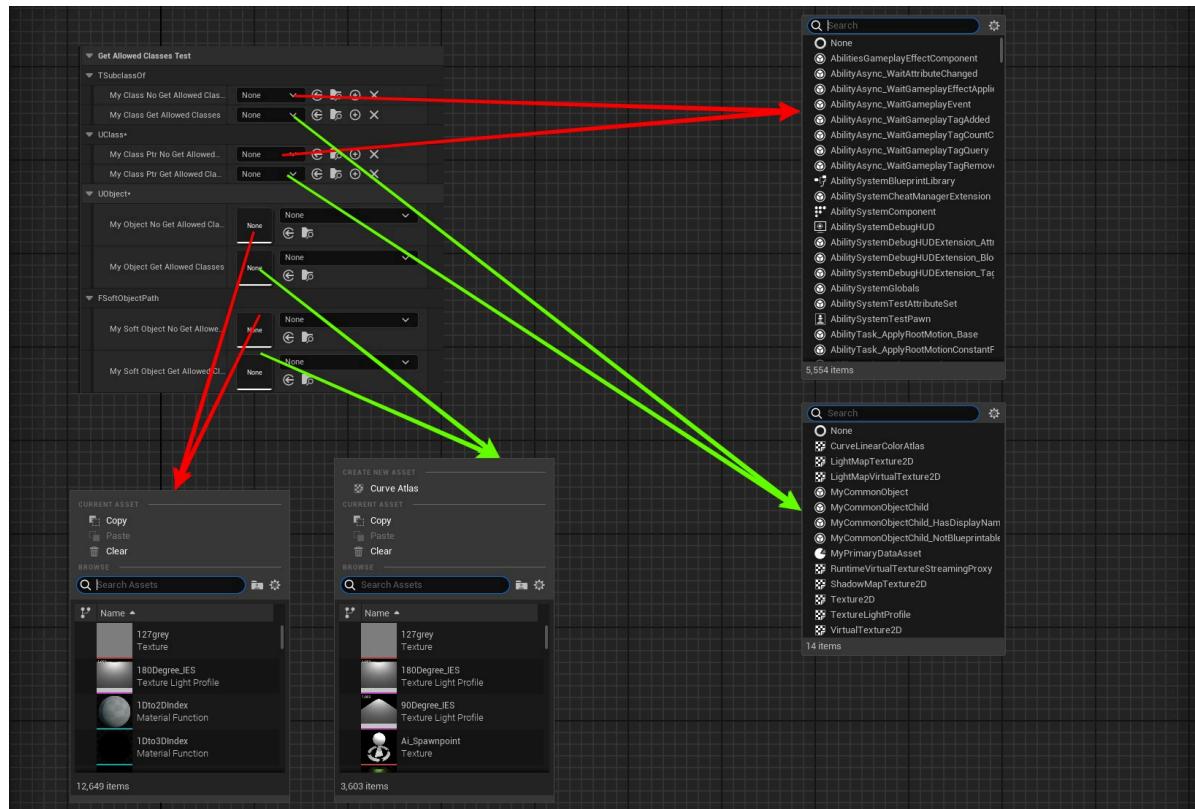
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetAllowedClassesTest|FSoftObjectPath")
    FSoftObjectPath MySoftObject_NoGetAllowedClasses;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetAllowedClassesTest|FSoftObjectPath", meta = (GetAllowedClasses =
"MyGetAllowedClassesFunc"))
    FSoftObjectPath MySoftObject_GetAllowedClasses;

```

## Test Results:

It is evident that the Class selector restricts the selection to the three specified base classes, and similarly, the Object selector limits the objects to these same three base classes.



## Principle:

Investigating the source code reveals that only SPropertyEditorClass and SPropertyEditorAsset properties can be applied, which correspond to the Class type (not FSoftClassPath) and the UObject's Asset type (FSoftObjectPath corresponds to SPropertyEditorAsset), respectively

```
void SPropertyEditorAsset::InitializeClassFilters(const FProperty* Property)
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*MetadataProperty, AllowedClassFilters, DisallowedClassFilters, bExactClass,
ObjectClass);
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPropertyEditor >& InPropertyEditor)
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*Property, AllowedClassFilters, DisallowedClassFilters, false);
}

TSharedRef<SWidget> SPropertyEditorEditInline::GenerateClassPicker()
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*Property, AllowedClassFilters, DisallowedClassFilters, false);
}

void PropertyEditorUtils::GetAllowedAndDisallowedClasses(const TArray<UObject*>&
ObjectList, const FProperty& MetadataProperty, TArray<const UClass*>&
AllowedClasses, TArray<const UClass*>& DisallowedClasses, bool bExactClass, const
UClass* ObjectClass)
{
    AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOw
nerProperty()->GetMetaData("AllowedClasses"));
    DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOw
nerProperty()->GetMetaData("DisallowedClasses"));

    bool bMergeAllowedClasses = !AllowedClasses.IsEmpty();

    if (MetadataProperty.GetOwnerProperty()->HasMetaData("GetAllowedClasses"))
    {
        const FString GetAllowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()->GetMetaData("GetAllowedClasses");
    }

    if (MetadataProperty.GetOwnerProperty()->HasMetaData("GetDisallowedClasses"))
    {
        const FString GetDisallowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()->GetMetaData("GetDisallowedClasses");
        if (!GetDisallowedClassesFunctionName.IsEmpty())
        {
            for (UObject* Object : ObjectList)
            {

```

```
const UFunction* GetDisallowedClassesFunction = Object ? Object->FindFunction(*GetDisallowedClassesFunctionName) : nullptr;
if (GetDisallowedClassesFunction)
{
    DECLARE_DELEGATE_RetVal(TArray<UClass*>, FGetDisallowedClasses);

    DisallowedClasses.Append(FGetDisallowedClasses::CreateUFunction(Object,
        GetDisallowedClassesFunction->GetFName()).Execute());
}
}
```

# GetDisallowedClasses

- **Function Description:** Used on class selectors to specify, via a function, certain base class types to be excluded from the list of selected types.
  - **Usage Location:** UPROPERTY
  - **Engine Module:** TypePicker
  - **Metadata Type:** string="abc"
  - **Restricted Types:** TSubClassOf, UClass\*  
Code: TArray<UClass\*> FuncName() const;
  - **Associated Items:** AllowedClasses
  - **Commonliness:** ★★

Its functionality is similar to GetAllowedClasses, but with the opposite effect.

However, its attribute type is similar to DisallowedClasses, and it can only be applied to class selectors. As such, after testing, it is found to only work with TSubClassOf and UClass\*.

## Test Code:

```
FUNCTION()
TArray<UClass*> MyGetDisallowedClassesFunc()
{
    TArray<UClass*> classes;
    classes.Add(UAbilityAsync::StaticClass());
    classes.Add(UTexture2D::StaticClass());
    return classes;
}

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetDisallowedClassesTest|TSubclassOf")
TSubclassOf<UObject> MyClass_NoGetDisallowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetDisallowedClassesTest|TSubclassOf", meta = (GetDisallowedClasses =
"MyGetDisallowedClassesFunc"))
TSubclassOf<UObject> MyClass_GetDisallowedClasses;
```

```

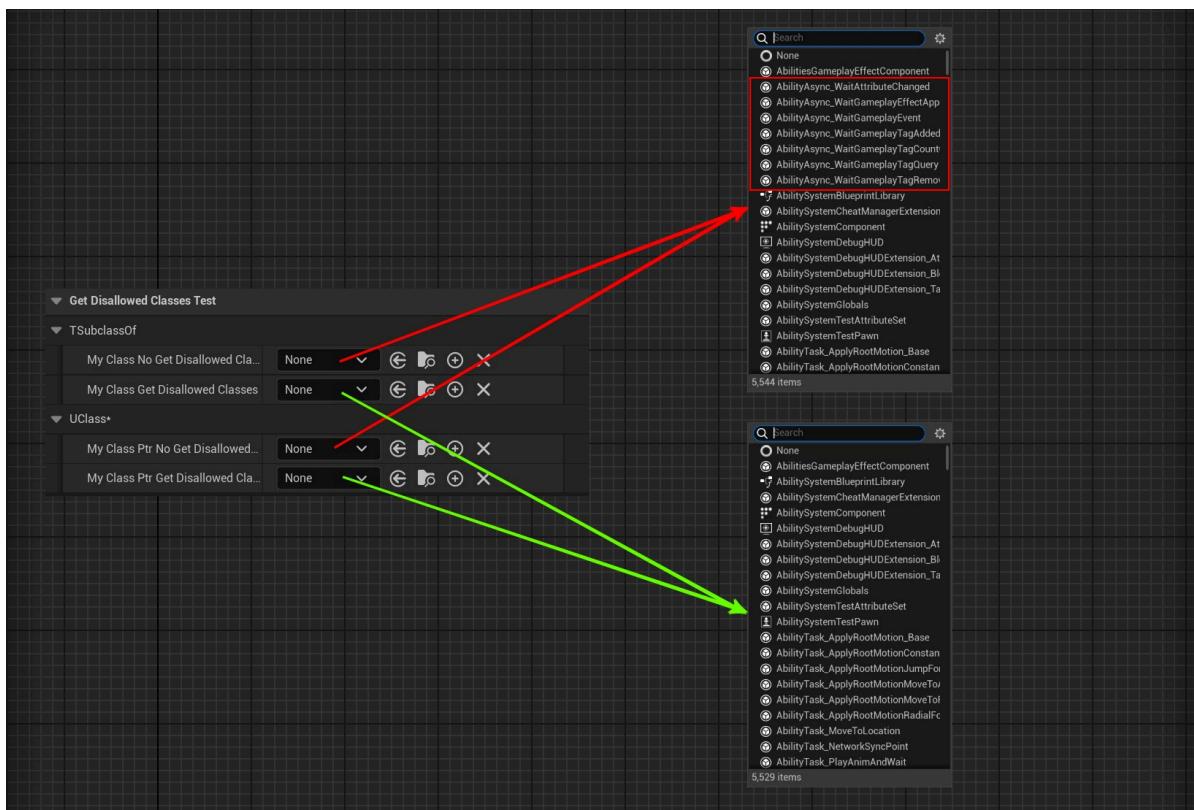
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetDisallowedClassesTest|UClass*")
UClass* MyClassPtr_NoGetDisallowedClasses;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"GetDisallowedClassesTest|UClass*", meta = (GetDisallowedClasses =
"MyGetDisallowedClassesFunc"))
UClass* MyClassPtr_GetDisallowedClasses;

```

## Test Effects:

You can observe that after adding GetDisallowedClasses, some types are missing from the selection list.



## Principle:

While both SPropertyEditorClass and SPropertyEditorAsset utilize GetAllowedAndDisallowedClasses, allowing the use of GetDisallowedClasses, SPropertyEditorAsset subsequently employs SAssetPicker, which does not make use of DisallowedClasses. Therefore, SPropertyEditorAsset does not support GetDisallowedClasses, and as a result, attributes of type UObject\* do not support GetDisallowedClasses.

```

void SPropertyEditorAsset::InitializeClassFilters(const FPropertyParams*PropertyParams)
{
   PropertyParamsEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*MetadataProperty, AllowedClassFilters, DisallowedClassFilters, bExactClass,
ObjectClass);
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPropertyParamsEditor>& InPropertyParams)

```

```

{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*Property, AllowedClassFilters, DisallowedClassFilters, false);
}

TSharedRef<SWidget> SPropertyEditorEditInline::GenerateClassPicker()
{
    PropertyEditorUtils::GetAllowedAndDisallowedClasses(ObjectList,
*Property, AllowedClassFilters, DisallowedClassFilters, false);
}

void PropertyEditorUtils::GetAllowedAndDisallowedClasses(const TArray<UObject*>&
ObjectList, const FProperty& MetadataProperty, TArray<const UClass*>&
AllowedClasses, TArray<const UClass*>& DisallowedClasses, bool bExactClass, const
UClass* objectClass)
{
    AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOwnerProperty()->GetMetaData("AllowedClasses"));

    DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(MetadataProperty.GetOwnerProperty()->GetMetaData("DisallowedClasses"));

    bool bMergeAllowedClasses = !AllowedClasses.IsEmpty();

    if (MetadataProperty.GetOwnerProperty()->HasMetaData("GetAllowedClasses"))
    {
        const FString GetAllowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()->GetMetaData("GetAllowedClasses");
    }

    if (MetadataProperty.GetOwnerProperty()->HasMetaData("GetDisallowedClasses"))
    {
        const FString GetDisallowedClassesFunctionName =
MetadataProperty.GetOwnerProperty()->GetMetaData("GetDisallowedClasses");
        if (!GetDisallowedClassesFunctionName.IsEmpty())
        {
            for (UObject* Object : ObjectList)
            {
                const UFunction* GetDisallowedClassesFunction = Object ? Object-
>FindFunction(*GetDisallowedClassesFunctionName) : nullptr;
                if (GetDisallowedClassesFunction)
                {
                    DECLARE_DELEGATE_RetVal(TArray<UClass*>,
FGetDisallowedClasses);

                    DisallowedClasses.Append(FGetDisallowedClasses::Create(Object,
GetDisallowedClassesFunction->GetFName()).Execute());
                }
            }
        }
    }
}

```

# BaseStruct

- **Function Description:** Specifies that the structure selected in the FInstancedStruct attribute option list must inherit from the structure referred to by BaseStruct.
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** bool
- **Restriction Type:** FInstancedStruct
- **Associated Items:** ExcludeBaseStruct, StructTypeConst
- **Commonality:** ★★★

The structure chosen in the FInstancedStruct attribute option list must inherit from the structure indicated by BaseStruct.

## Test Code:

```
USTRUCT(BlueprintType)
struct INSIDER_API FMyCommonStruct
{
};

USTRUCT(BlueprintType)
struct INSIDER_API FMyCommonStructChild:public FMyCommonStruct
{
    GENERATED_BODY()
};

USTRUCT(BlueprintType, DisplayName = "This is MyCommonStructChild")
struct INSIDER_API FMyCommonStructChild_HasDisplayName :public FMyCommonStruct
{
    GENERATED_BODY()
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Struct :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InstancedStruct")
    FInstancedStruct MyStruct;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InstancedStruct",
meta = (BaseStruct = "/Script/Insider.MyCommonStruct"))
    FInstancedStruct MyStruct_BaseStruct;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InstancedStruct",
meta = (ExcludeBaseStruct, BaseStruct = "/Script/Insider.MyCommonStruct"))
    FInstancedStruct MyStruct_ExcludeBaseStruct;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "InstancedStruct",
meta = (StructTypeConst))

```

```

FInstancedStruct MyStruct_Const;
};

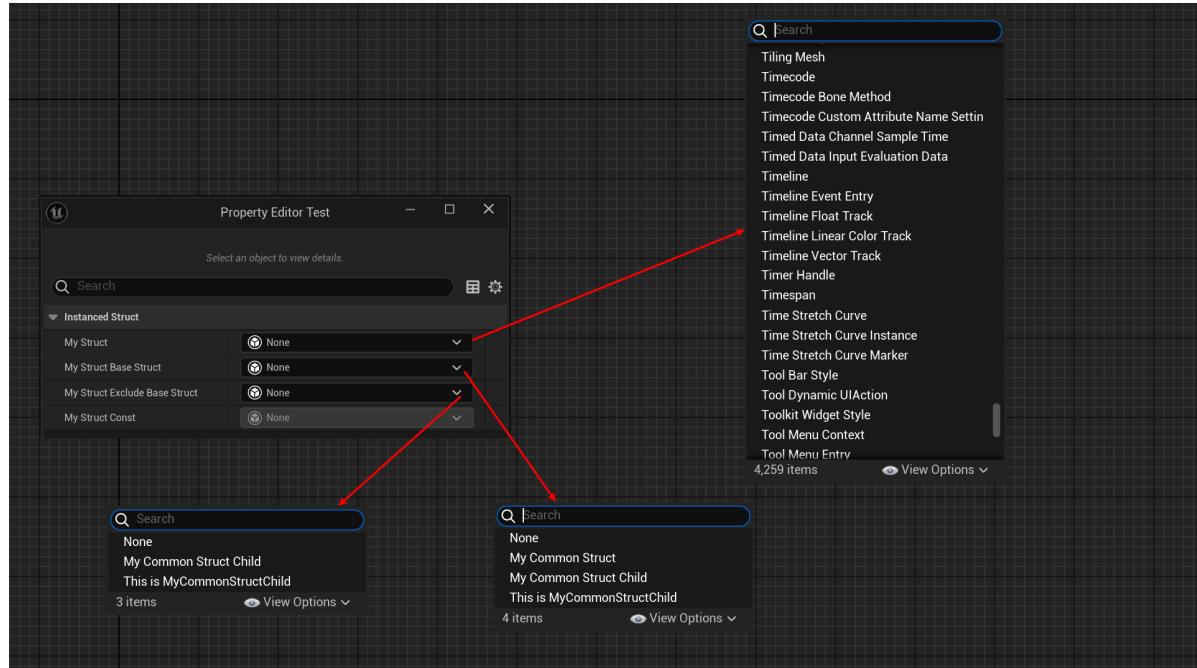
```

## Test Results:

The selection of MyStruct\_BaseStruct is restricted to FMyCommonStruct itself and its subclasses.

If the base class itself is not desired, MyStruct\_ExcludeBaseStruct with ExcludeBaseStruct will not include FMyCommonStruct.

MyStruct\_Const with StructTypeConst cannot be edited.



## Principle:

Extract the metadata of BaseStruct to populate the StructFilter.

```

void FInstancedStructDetails::CustomizeHeader(TSharedRef<class IPropertyHandle>
StructPropertyHandle, class FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    static const FName NAME_BaseStruct = "BaseStruct";
    const FString& BaseStructName = StructProperty->GetMetaData(NAME_BaseStruct);
    if (!BaseStructName.IsEmpty())
    {
        BaseScriptStruct = UClass::TryFindTypeSlow<UScriptStruct>
(BaseStructName);
        if (!BaseScriptStruct)
        {
            BaseScriptStruct = LoadObject<UScriptStruct>(nullptr,
*BaseStructName);
        }
    }
}

TSharedRef<swidget> FInstancedStructDetails::GenerateStructPicker()

```

```

{
    static const FName NAME_ExcludeBaseStruct = "ExcludeBaseStruct";
    static const FName NAME_HideviewOptions = "HideViewOptions";
    static const FName NAME_ShowTreeView = "ShowTreeView";

    const bool bExcludeBaseStruct = StructProperty-
>HasMetaData(NAME_ExcludeBaseStruct);
    const bool bAllowNone = !(StructProperty->GetMetaDataProperty()-
>PropertyFlags & CPF_NoClear);
    const bool bHideViewOptions = StructProperty-
>HasMetaData(NAME_HideviewOptions);
    const bool bShowTreeView = StructProperty->HasMetaData(NAME_ShowTreeView);

    StructFilter->BaseStruct = BaseScriptStruct;
    StructFilter->bAllowBaseStruct = !bExcludeBaseStruct;

}

```

## ExcludeBaseStruct

---

- **Function Description:** Ignore the base class structure pointed to by BaseStruct when using the FInstancedStruct property of BaseStruct.
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** bool
- **Restriction Type:** FInstancedStruct
- **Associated Item:** BaseStruct
- **Commonality:** ★★★

Ignore the base class structure pointed to by BaseStruct when using the FInstancedStruct property.

## StructTypeConst

---

- **Function Description:** Specifies that the type of the FInstancedStruct attribute cannot be selected within the editor.
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** bool
- **Restricted Type:** FInstancedStruct
- **Associated Item:** BaseStruct
- **Commonality:** ★

The type specified for the FInstancedStruct attribute cannot be selected in the editor.

Its use is often for subsequent initialization by the user within the code.

## Principle:

If this attribute is marked, the editing control is disabled.

```
void FInstancedStructDetails::CustomizeHeader(TSharedRef<class IPropertyHandle>
StructPropertyHandle, class FDetailWidgetRow& HeaderRow,
IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    static const FName NAME_StructTypeConst = "StructTypeConst";
    const bool bEnableStructSelection = !StructProperty-
>HasMetaData(NAME_StructTypeConst);

    .IsEnabled(bEnableStructSelection)

}
```

## MetaStruct

- **Function Description:** Specifies the parent structure of the chosen type when set to a UScriptStruct\* attribute.
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** string="abc"
- **Restricted Type:** UScriptStruct\*
- **Commonality:** ★★★

Sets the parent structure of the selected type when assigned to a UScriptStruct\* attribute.

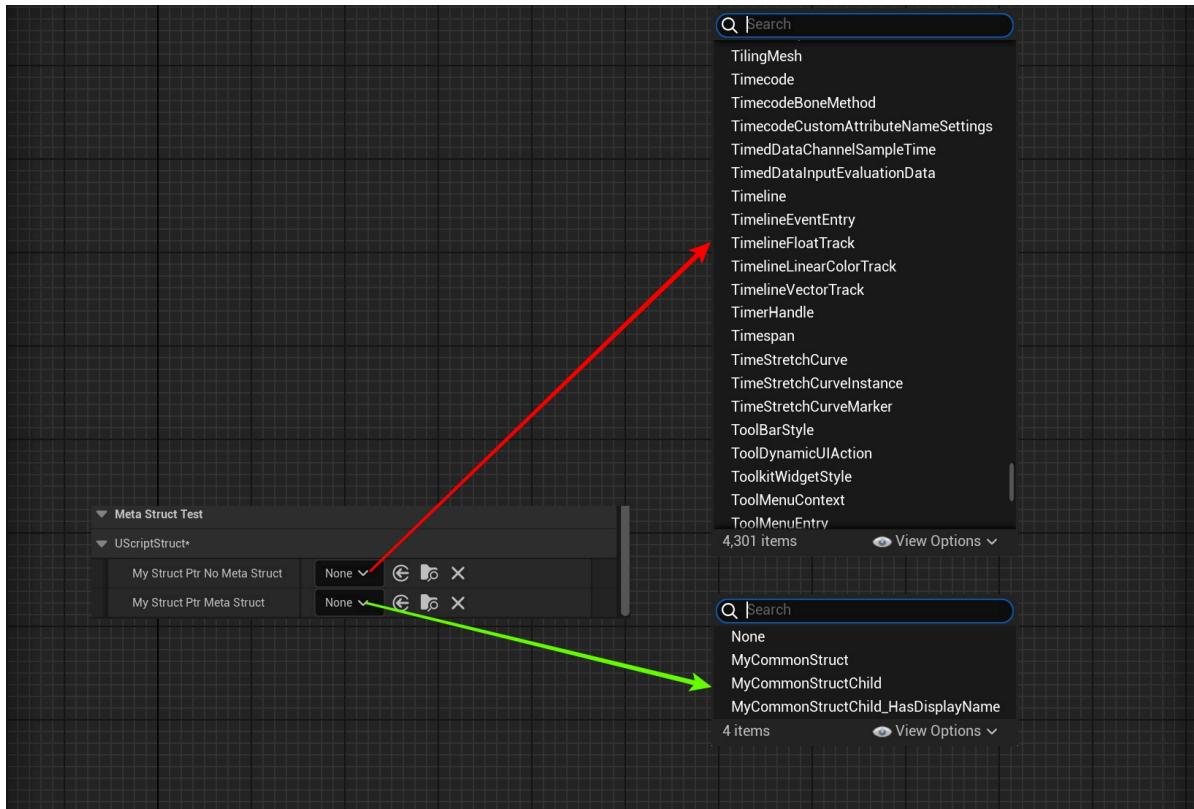
## Test Code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaStructTest|UScriptStruct*", meta = ())
UscriptStruct* MyStructPtr_NoMetaStruct;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaStructTest|UScriptStruct*", meta = (MetaStruct = "MyCommonStruct"))
UscriptStruct* MyStructPtr_MetaData;
```

## Test Results:

Entities with MetaStruct can filter the type list down to subclasses of MyCommonStruct.



## Principle:

Locate the MetaStruct and set it to the MetaStruct on StructFilter for filtering. This defines the base class for the selection of structures.

```

void SPropertyEditorStruct::Construct(const FArguments& InArgs, const TSharedPtr<
class FPropertyEditor >& InPropertyEditor)
{
    const FString& MetaStructName = Property->GetOwnerProperty()-
>GetMetaData(TEXT("MetaStruct"));
    if (!MetaStructName.IsEmpty())
    {
        MetaStruct = UClass::TryFindTypesIn
(MetaStructName, EFindFirstObjectOptions::EnsureIfAmbiguous);
        if (!MetaStruct)
        {
            MetaStruct = LoadObject(nullptr,
*MetaStructName);
        }
    }
}

virtual bool FPropertyEditorStructFilter::IsStructAllowed(const
FStructViewerInitializationOptions& InInitOptions, const UScriptStruct* InStruct,
TSharedRef<FStructViewerFilterFuncs> InFilterFuncs) override
{
    if (InStruct->IsA<UUserDefinedStruct>())
    {
        // User Defined Structs don't support inheritance, so only include them
        if we have don't a MetaStruct set
        return MetaStruct == nullptr;
    }
}

```

```

    // query the native struct to see if it has the correct parent type (if any)
    return !MetaStruct || InStruct->IsChildof(MetaStruct);
}

```

# ShowDisplayNames

- **Function Description:** Specifies that for Class and Struct properties, an alternative display name should be shown in the class selector instead of the class's original name.
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** bool
- **Restricted Types:** TSubClassOf, FSoftClassPath, UClass, UScriptStruct
- **Commonality:** ★

For Class and Struct properties, specifies that an alternative display name should be shown in the class selector rather than the class's original name.

The class display name refers to the name assigned to the DisplayName on UCLASS or USTRUCT, which is typically more user-friendly. The original class name is the class's type name.

## Test Code:

```

UCLASS(BlueprintType, NotBlueprintable, DisplayName="This is MyCommonObjectChild")
class INSIDER_API UMyCommonObjectChild_HasDisplayName :public UMyCommonObject
{
    GENERATED_BODY()
public:
};

USTRUCT(BlueprintType, DisplayName="This is MyCommonStructChild")
struct INSIDER_API FMyCommonStructChild_HasDisplayName:public FMyCommonStruct
{
    GENERATED_BODY()
};

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowDisplayNamesTest|TSubclassOf", meta = ())
TSubclassof<UMyCommonObjectChild_HasDisplayName> MyClass_NotShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowDisplayNamesTest|TSubclassof", meta = (ShowDisplayNames))
TSubclassof<UMyCommonObjectChild_HasDisplayName> MyClass_ShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowDisplayNamesTest|UClass*", meta = (AllowedClasses =
"MyCommonObjectChild_HasDisplayName"))
UClass* MyClassPtr_NotShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowDisplayNamesTest|UClass*", meta = (AllowedClasses =
"MyCommonObjectChild_HasDisplayName", ShowDisplayNames))
UClass* MyClassPtr_ShowDisplayNames;

```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowDisplayNamesTest|FSoftClassPath", meta = (MetaClass =
"MyCommonObjectChild_HasDisplayName"))

FSoftClassPath MySoftClass_NotShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowDisplayNamesTest|FSoftClassPath", meta = (MetaClass =
"MyCommonObjectChild_HasDisplayName", ShowDisplayNames))
FSoftClassPath MySoftClass_ShowDisplayNames;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowDisplayNamesTest|UScriptStruct*", meta = (MetaStruct =
"MyCommonStructChild_HasDisplayName"))
UScriptStruct* MyStructPtr_NotShowDisplayNames;

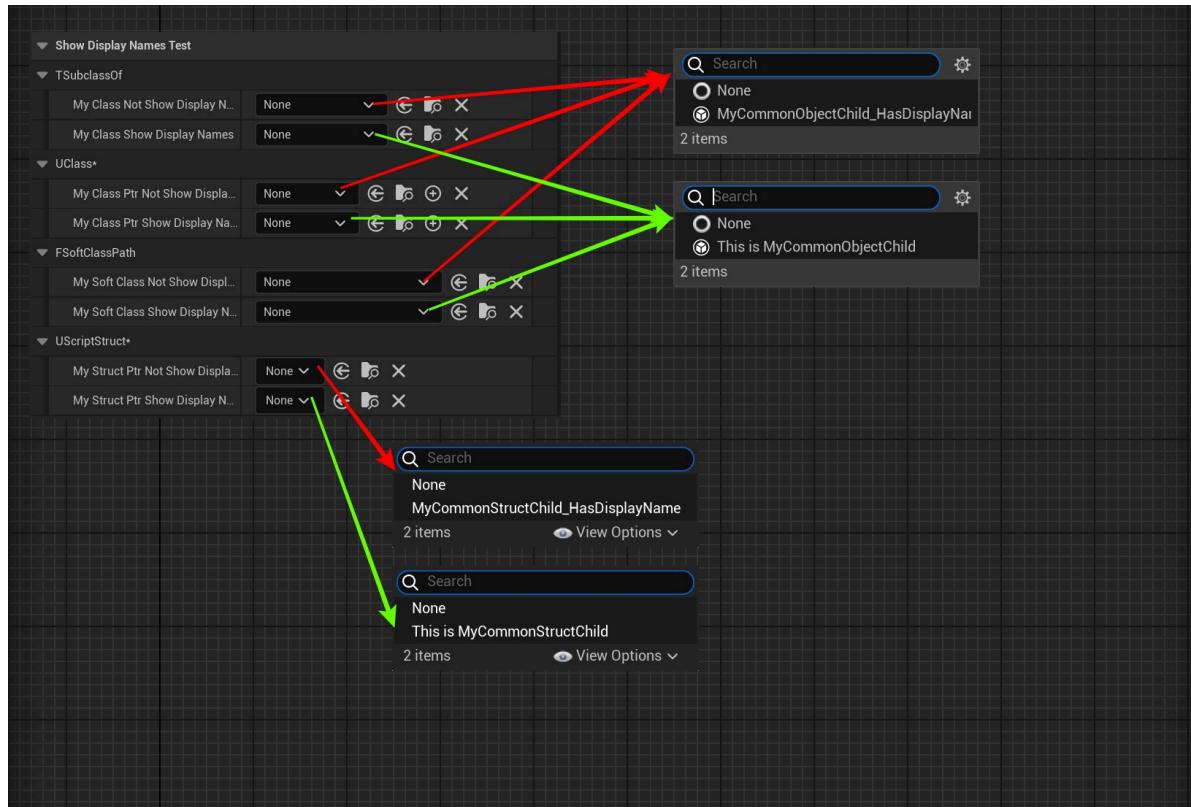
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowDisplayNamesTest|UScriptStruct*", meta = (MetaStruct =
"MyCommonStructChild_HasDisplayName", ShowDisplayNames))
UscriptStruct* MyStructPtr_ShowDisplayNames;

```

## Test Results:

It is evident that with ShowDisplayNames added, the name displayed in the list is a more friendly "This is XXX" rather than the direct class name.

To make the effect more intuitive, the test code also includes MetaClass, MetaStruct, and AllowedClasses to restrict the selection range.



# Principle:

In the source code, it is apparent that if bShowDisplayNames is enabled, the display will be (Class, Struct) → GetDisplayNameText instead of (Class, Struct) → GetName.

Because FInstancedStructDetails does not utilize this Meta, the option is not supported.

```
void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPropertyHandle>
InPropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&
StructCustomizationUtils)
{
    const bool bShowDisplayNames = PropertyHandle-
>HasMetaData("ShowDisplayNames");
    SNew(SClassPropertyEntryBox)
        .ShowDisplayNames(bShowDisplayNames)
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPropertyEditor >& InPropertyEditor)
{
    bShowViewOptions = Property->GetOwnerProperty()->HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowDisplayNames"));
}

void SPropertyEditorStruct::Construct(const FArguments& InArgs, const TSharedPtr<
class FPropertyEditor >& InPropertyEditor)
{
    bShowViewOptions = Property->GetOwnerProperty()->HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowDisplayNames"));
}

static FText GetClassDisplayName(const UObject* Object, bool bShowDisplayNames)
{
    const UClass* Class = Cast<UClass>(Object);
    if (Class != nullptr)
    {
        if (bShowDisplayNames)
        {
            return Class->GetDisplayNameText();
        }

        UBlueprint* BP = UBlueprint::GetBlueprintFromClass(Class);
        if (BP != nullptr)
        {
            return FText::FromString(BP->GetName());
        }
    }
}
```

```

        return (Object) ? FText::Fromstring(Object->GetName()) :
LOCTEXT("Invalidobject", "None");
}

FText SPropertyEditorStruct::GetDisplayValue() const
{
    static bool bIsReentrant = false;

    auto GetStructDisplayName = [this](const Uobject* Inobject) -> FText
    {
        if (const UScriptStruct* Struct = Cast<UScriptStruct>(Inobject))
        {
            return bShowDisplayNames
                ? Struct->GetDisplayNameText()
                : FText::AsCultureInvariant(Struct->GetName());
        }
        return LOCTEXT("None", "None");
    };
}

```

## DisallowedStructs

- **Function Description:** Exclusive to the SmartObject module, this is used to exclude a specific class and its subclasses from the class selector.
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** string="abc"
- **Commonality:** ★

Applied solely within the SmartObject module to exclude a particular class and its subclasses from the class selector.

## Source Code:

```

UPROPERTY(EditDefaultsOnly, Category = "SmartObject", meta=
(DisallowedStructs="/Script/SmartObjectsModule.SmartObjectsSlotAnnotation"))
TArray<FSmartObjectDefinitionDataProxy> DefinitionData;

```

## RowType

- **Function Description:** Specifies the base class for optional row types associated with the FDataTableRowHandle attribute.
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** string="abc"
- **Restricted Type:** FDataTableRowHandle
- **Commonality:** ★★★

Specifies the base class for the optional row types of the FDataTableRowHandle attribute.

## Test Code:

```
USTRUCT(BlueprintType)
struct FMyCommonRow : public FTableRowBase
{
    GENERATED_BODY()

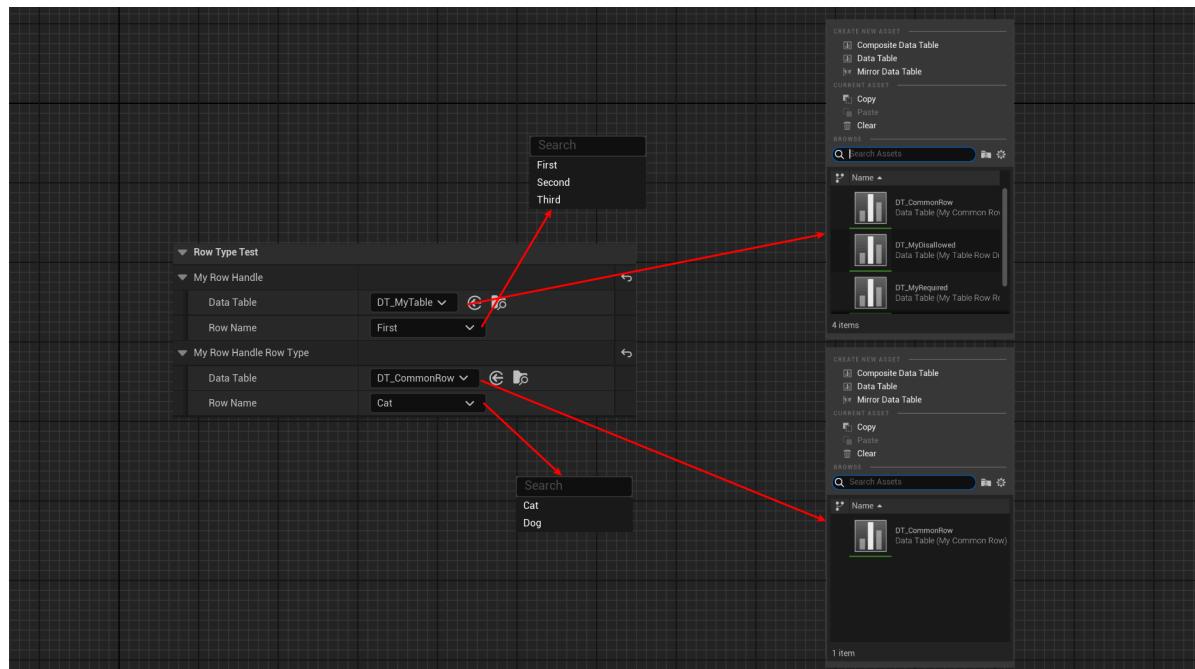
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString MyString;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FVector MyVector;
};

UCLASS(BlueprintType)
class INSIDER_API UMyProperty_RowType :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "RowTypeTest")
    FDataTableRowHandle MyRowHandle;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "RowTypeTest", meta =
    (RowType = "/Script/Insider.MyCommonRow"))
    FDataTableRowHandle MyRowHandle_RowType;
};
```

## Test Results:

Create a DataTable based on FMyCommonRow in the editor, referred to as DT\_MyCommonRow. Of course, there are other DataTables with different RowStructs in the project.

You can observe that the options for MyRowHandle\_RowType are restricted to DT\_MyCommonRow, and the RowName is also displayed correctly.



## Principle:

The UI customization is targeted at the FDataTableRowHandle type. If RowType data is present, it is assigned to RowFilterStruct, thereby facilitating the filtering process.

```
void FDataTableCustomizationLayout::CustomizeHeader(TSharedRef<class IPropertyHandle> InStructPropertyParams, class FDetailWidgetRow& HeaderRow, I.PropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    if (StructPropertyParams->HasMetaData(TEXT("RowType")))
    {
        const FString& RowType = StructPropertyParams->GetMetaData(TEXT("RowType"));
        RowTypeFilter = FName(*RowType);
        RowFilterStruct = UClass::TryFindTypeSlow<UScriptStruct>(RowType);
    }

}

bool FDataTableCustomizationLayout::ShouldFilterAsset(const struct FAssetData& AssetData)
{
    if (!RowTypeFilter.IsNone())
    {
        static const FName RowStructureTagName("RowStructure");
        FString RowStructure;
        if (AssetData.GetTagValue<FString>(RowStructureTagName, RowStructure))
        {
            if (RowStructure == RowTypeFilter.ToString())
            {
                return false;
            }

            // This is slow, but at the moment we don't have an alternative to
            // the short struct name search
            UScriptStruct* RowStruct = UClass::TryFindTypeSlow<UScriptStruct>(RowStructure);
            if (RowStruct && RowFilterStruct && RowStruct->IsChildOf(RowFilterStruct))
            {
                return false;
            }
        }
        return true;
    }
    return false;
}

RegisterCustomPropertyParamsLayout("DataTableRowHandle",
    FOnGetPropertyTypeCustomizationInstance::CreateStatic(&FDataTableCustomizationLayout::MakeInstance));
```

# MustImplement

- **Function Description:** Specifies that the class selected by the TSubClassOf or FSoftClassPath attribute must implement this interface
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** string="abc"
- **Restricted Types:** TSubClassOf, FSoftClassPath, UClass\*
- **Commonality:** ★★★

The class specified by the TSubClassOf or FSoftClassPath attribute must implement this interface.

- TSubClassOf, FSoftClassPath, and the original UClass\* attributes can all be used to locate a UClass, *with the distinction that UClass is a hard reference to a specific class object, while FSoftClassPath is a soft reference to the path of the class object*. Both, however, are general references to UClass\* and do not constrain subtypes. TSubClassOf, on the other hand, although it is also a hard reference to a class object, further restricts the selection of types to subclasses of T, which can be more convenient, especially when you are aware of the range of your subclasses, such as TSubClassOf or TSubClassOf.
- For attributes used to select a class, if there are no restrictions, the engine will search for all classes, allowing you to choose from them, which is not very convenient.
- Therefore, additional filtering mechanisms have been introduced into the engine. MustImplement is used to filter and ensure that a specified class attribute must implement a certain interface.

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyCommonInterfaceChild :public UObject, public
IMyCommonInterface
{
    GENERATED_BODY()
};

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|TSubclassOf")
TSubclassOf<UObject> MyClass_NoMustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|TSubclassOf", meta = (MustImplement = "MyCommonInterface"))
TSubclassOf<UObject> MyClass_MustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|TSubclassOf", meta = (MustImplement =
"/Script/UMG.UserListEntry"))
TSubclassOf<UUserWidget> MyWidgetClass_MustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|FSoftClassPath")
FSoftClassPath MySoftClass_NoMustImplement;
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|FSoftClassPath", meta = (MustImplement = "MyCommonInterface"))
FSoftClassPath MySoftClass_MustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|FSoftClassPath", meta = (MustImplement =
"/Script/UMG.UserListEntry"))
FSoftClassPath MySoftWidgetClass_MustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|UClass*")
UClass* MyClassStar_NoMustImplement;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|UClass*", meta = (MustImplement = "MyCommonInterface"))
UClass* MyClassStar_MustImplement;

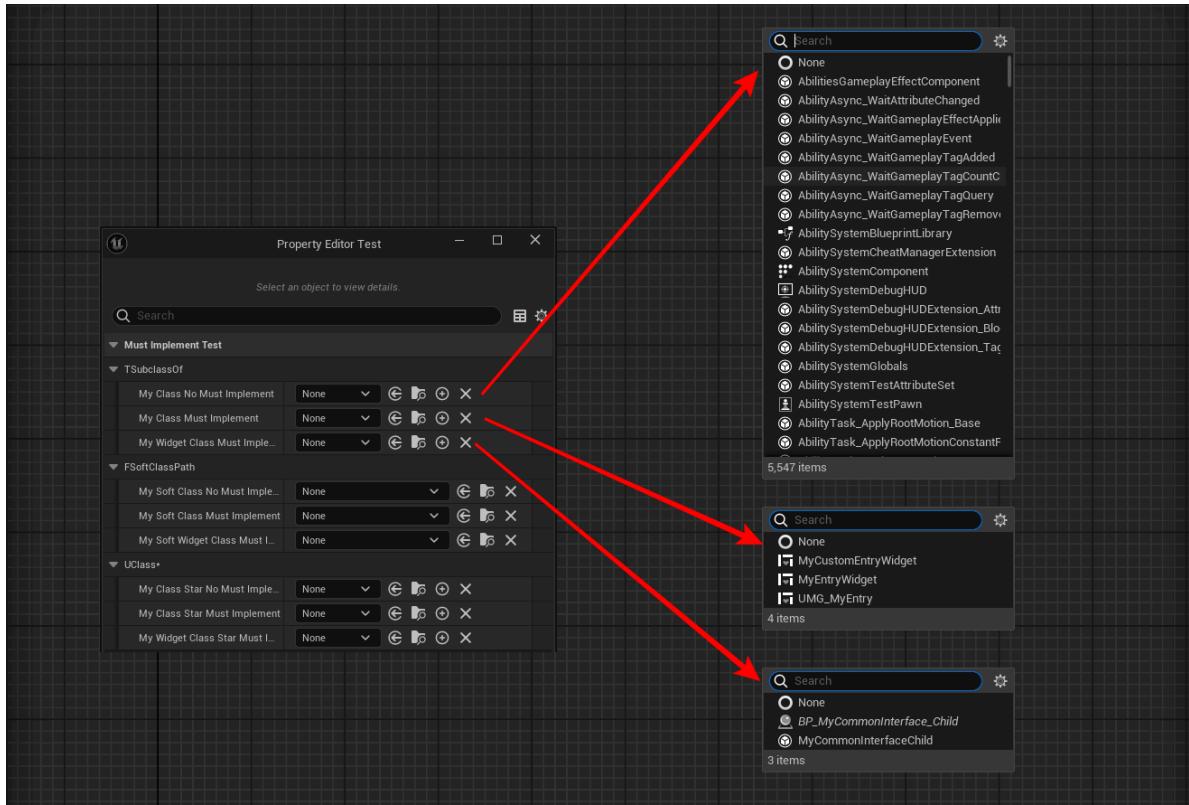
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MustImplementTest|UClass*", meta = (MustImplement =
"/Script/UMG.UserListEntry"))
UClass* MyWidgetClassStar_MustImplement;

UFUNCTION(BlueprintCallable, meta=(Category="MustImplementTest|TSubclassOf"))
static void SetMyClassMustImplement(UPARAM(meta=
(MustImplement="MyCommonInterface")) TSubclassOf<UObject> MNyClass){}

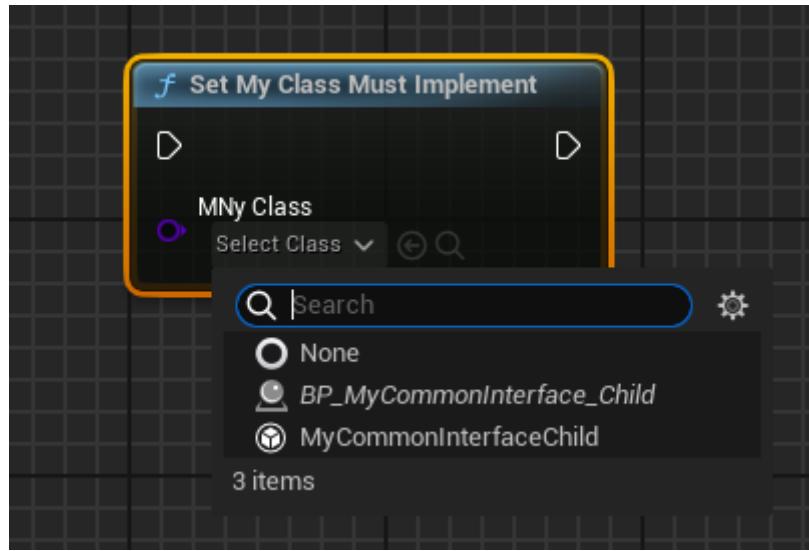
```

## Test Effects:

You can see the results without filtering for the first example, and the filtered results for the second and third examples.



It can also be used as a parameter within a function:



## Principle:

When FPropertyHandleBase generates potential values, you can observe that it applies a series of filters.

```

void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPROPERTYHANDLE>
InPropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&
StructCustomizationUtils)
{
    PropertyHandle = InPropertyHandle;

    const FString& MetaClassName = PropertyHandle->GetMetaData("MetaClass");
    const FString& RequiredInterfaceName = PropertyHandle-
>GetMetaData("RequiredInterface"); // This was the old name, switch to
MustImplement to synchronize with class property
    const FString& MustImplementName = PropertyHandle-
>GetMetaData("MustImplement");
    TArray<const UCLASS*> AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyHandle-
>GetMetaData("AllowedClasses"));
    TArray<const UCLASS*> DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyHandle-
>GetMetaData("DisallowedClasses"));
    const bool bAllowAbstract = PropertyHandle->HasMetaData("AllowAbstract");
    const bool bIsBlueprintBaseOnly = PropertyHandle-
>HasMetaData("IsBlueprintBaseOnly") || PropertyHandle-
>HasMetaData("BlueprintBaseOnly");
    const bool bAllowNone = !(PropertyHandle->GetMetaDataProperty()-
>PropertyFlags & CPF_NoClear);
    const bool bShowTreeView = PropertyHandle->HasMetaData("ShowTreeView");
    const bool bHideViewOptions = PropertyHandle->HasMetaData("HideViewOptions");
    const bool bShowDisplayNames = PropertyHandle-
>HasMetaData("ShowDisplayNames");

    const UCLASS* const MetaClass = !MetaClassName.IsEmpty()
        ? FEditorClassUtils::GetClassFromString(MetaClassName)
        : UObject::StaticClass();
    const UCLASS* const RequiredInterface = !RequiredInterfaceName.IsEmpty()
        ? FEditorClassUtils::GetClassFromString(RequiredInterfaceName)
        : UObject::StaticClass();
}

```

```

        : FEditorClassUtils::GetClassFromString(MustImplementName);
    }

TSharedRef<SWidget> SGraphPinClass::GenerateAssetPicker()
{
    if (UEdGraphNode* ParentNode = GraphPinObj->GetOwningNode())
    {
        FString PossibleInterface = ParentNode->GetPinMetaData(GraphPinObj->PinName, TEXT("MustImplement"));
        if (!PossibleInterface.IsEmpty())
        {
            Filter->RequiredInterface = UClass::TryFindTypeSlow<UClass>(PossibleInterface);
        }
    }
}

bool FPropertyHandleBase::GeneratePossibleValues(TArray< FString>& OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems, TArray< FText *>* OutDisplayNames)
{
    if( Property->IsA(FClassProperty::StaticClass()) || Property->IsA(FSoftClassProperty::StaticClass()) )
    {
        UClass* MetaClass = Property->IsA(FClassProperty::StaticClass())
            ? CastFieldChecked<FClassProperty>(Property)->MetaClass
            : CastFieldChecked<FSoftClassProperty>(Property)->MetaClass;

        FString NoneStr( TEXT("None" ) );
        OutOptionStrings.Add( NoneStr );
        if (OutDisplayNames)
        {
            OutDisplayNames->Add(FText::FromString(NoneStr));
        }

        const bool bAllowAbstract = Property->GetOwnerProperty()->HasMetaData(TEXT("AllowAbstract"));
        const bool bBlueprintBaseOnly = Property->GetOwnerProperty()->HasMetaData(TEXT("BlueprintBaseOnly"));
        const bool bAllowOnlyPlaceable = Property->GetOwnerProperty()->HasMetaData(TEXT("OnlyPlaceable"));
        UClass* InterfaceThatMustBeImplemented = Property->GetOwnerProperty()->GetClassMetaData(TEXT("MustImplement"));

        if (!bAllowOnlyPlaceable || MetaClass->IsChildOf< AActor >())
        {
            for (TObjectIterator< UClass > It; It; ++It)
            {
                if (It->IsChildOf(MetaClass)
                    && PropertyEditorHelpers::IsEditInlineClassAllowed(*It, bAllowAbstract)
                    && (!bBlueprintBaseOnly ||
FKismetEditorUtilities::CanCreateBlueprintOfClass(*It))
                    && (!InterfaceThatMustBeImplemented || It->ImplementsInterface(InterfaceThatMustBeImplemented)))

```

```
&& (!bAllowOnlyPlaceable || !It->  
HasAnyClassFlags(CLASS_Abstract | CLASS_NotPlaceable)))  
{  
    OutOptionStrings.Add(It->GetName());  
    if (OutDisplayNames)  
    {  
        OutDisplayNames->Add(FText::FromString(It->GetName()));  
    }  
}  
}  
}  
}
```

# ShowTreeView

- **Function description:** Used to select attributes of Class or Struct so that they are displayed as a tree rather than a list in the class selector.
  - **Use location:** UPROPERTY
  - **Engine module:** TypePicker
  - **Metadata type:** bool
  - **Limit types:** TSubClassOf, FSoftClassPath, UClass, *UScriptStruct*, FInstancedStruct
  - **Frequency of use:** ★★

Applied to Class or Struct attributes to display them as a tree rather than a list in the class selector.

Applies to attribute types such as TSubClassOf, FSoftClassPath, UClass, *UScriptStruct*, FInstancedStruct, which are used for type selection. It does not function with TSoftObjectPtr or FSoftObjectPath, which are used for object selection.

# Test Code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"ShowTreeViewTest|TSubclassOf")  
TSubclassOf<UObject> MyClass_NotShowTreeView;  
  
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"ShowTreeViewTest|TSubclassOf", meta = (ShowTreeView))  
TSubclassOf<UObject> MyClass_ShowTreeView;  
  
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"ShowTreeViewTest|UClass*")  
UClass* MyClassPtr_NotShowTreeView;  
  
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"ShowTreeViewTest|UClass*", meta = (ShowTreeView))  
UClass* MyClassPtr_ShowTreeView;  
  
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =  
"ShowTreeViewTest|FSoftClassPath")  
FSoftClassPath MySoftClass_NotShowTreeView;
```

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowTreeViewTest|FSoftClassPath", meta = (ShowTreeView))
FSoftClassPath MySoftClass_ShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowTreeViewTest|UScriptStruct*)
UScriptStruct* MyStructPtr_NotShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowTreeViewTest|UScriptStruct*", meta = (ShowTreeView))
UScriptStruct* MyStructPtr_ShowTreeView;

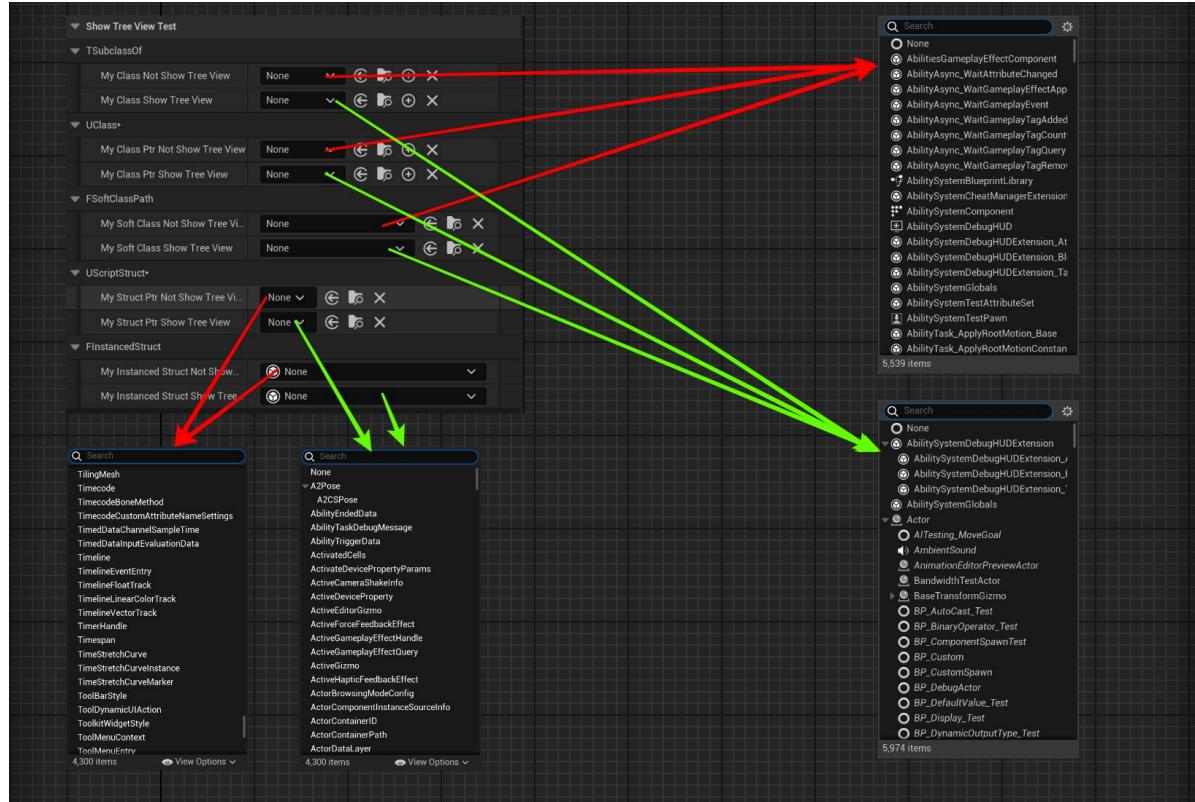
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowTreeViewTest|FInstancedStruct")
FInstancedStruct MyInstancedStruct_NotShowTreeView;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
>ShowTreeViewTest|FInstancedStruct", meta = (ShowTreeView))
FInstancedStruct MyInstancedStruct_ShowTreeView;

```

## Test Results:

It is evident that properties with the ShowTreeView attribute are displayed as a tree in the pop-up selection box, rather than as a list.



## Principle:

In the source code, SPropertyEditorClass is used for TSubClassOf and UClass, FSoftClassPathCustomization for FSoftClassPath, SPropertyEditorStruct for UScriptStruct, and FInstancedStructDetails for FInstancedStruct to customize the UI.

```

void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPROPERTYHandle>
InPROPERTYHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&
StructCustomizationUtils)
{
    const bool bShowTreeView = PROPERTYHandle->HasMetaData("ShowTreeView");
    const bool bHideviewOptions = PROPERTYHandle-
>HasMetaData("HideViewOptions");

    SNew(SClassPropertyEntryBox)
        .ShowTreeView(bShowTreeView)
        .HideViewOptions(bHideviewOptions)
        .ShowDisplayNames(bShowDisplayNames)
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPROPERTYEditor >& InPROPERTYEditor)
{
    bShowViewOptions = PROPERTY->GetOwnerProperty()->HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = PROPERTY->GetOwnerProperty()->HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = PROPERTY->GetOwnerProperty()->HasMetaData(TEXT("ShowDisplayNames"));
}

void SPropertyEditorStruct::Construct(const FArguments& InArgs, const TSharedPtr<
class FPROPERTYEditor >& InPROPERTYEditor)
{
    bShowViewOptions = PROPERTY->GetOwnerProperty()->HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = PROPERTY->GetOwnerProperty()->HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = PROPERTY->GetOwnerProperty()->HasMetaData(TEXT("ShowDisplayNames"));
}

TSharedRef<SWidget> FInstancedStructDetails::GenerateStructPicker()
{
    const bool bExcludeBaseStruct = StructProperty->HasMetaData(NAME_ExcludeBaseStruct);
    const bool bAllowNone = !(StructProperty->GetMetaDataProperty() ->PropertyParams & CPF_NoClear);
    const bool bHideviewOptions = StructProperty->HasMetaData(NAME_HideViewOptions);
    const bool bShowTreeView = StructProperty->HasMetaData(NAME_ShowTreeView);
}

```

## BlueprintBaseOnly

- **Function Description:** Used for class attributes to specify whether only base classes that can generate Blueprint subclasses are accepted
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** bool

- **Restricted Types:** TSubClassOf, FSoftClassPath, UClass\*
- **Commonality:** ★★

This attribute is particularly useful when you want to restrict the class to those that can serve as Blueprint base classes.

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyCommonObject :public UObject
{
    GENERATED_BODY()
};

UCLASS(BlueprintType)
class INSIDER_API UMyCommonObjectChild :public UMyCommonObject
{
    GENERATED_BODY()
};

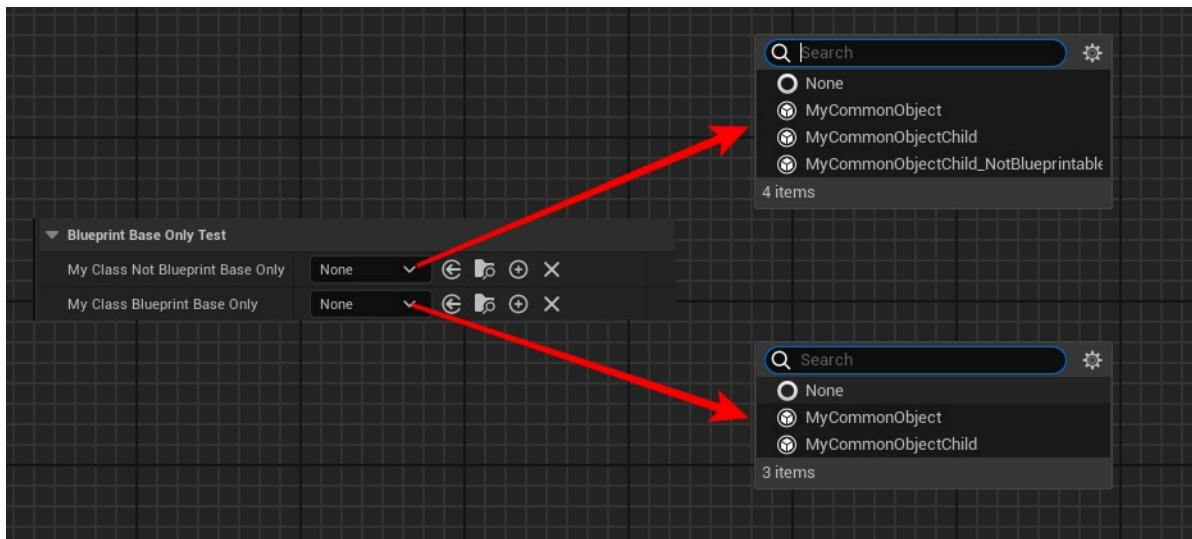
UCLASS(BlueprintType, NotBlueprintable)
class INSIDER_API UMyCommonObjectChild_NotBlueprintable :public UMyCommonObject
{
    GENERATED_BODY()
public:
};

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"BlueprintBaseOnlyTest")
    TSubclassOf<UMyCommonObject> MyClass_NotBlueprintBaseOnly;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"BlueprintBaseOnlyTest", meta = (BlueprintBaseOnly))
    TSubclassOf<UMyCommonObject> MyClass_BlueprintBaseOnly;
```

## Testing Code:

With the BlueprintBaseOnly restriction added, the class UMyCommonObjectChild\_NotBlueprintable cannot be selected due to its NotBlueprintable attribute.



## 测试效果：

If `bBlueprintBaseOnly` is true, further checks are required to determine if `CanCreateBlueprintOfClass` is valid, which assesses whether a class can be used to create Blueprint subclasses.

Typically, if a C++ class does not have the `Blueprintable` attribute defined, it cannot generate Blueprint subclasses and will not be selected by this property.

Classes that are already Blueprints can always be used to create additional Blueprint subclasses.

```

bool FPropertyHandleBase::GeneratePossibleValues(TArray< FString>&
OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems,
TArray< FText *>* OutDisplayNames)
{
    if( Property->IsA(FClassProperty::StaticClass()) || Property-
>IsA(FSoftClassProperty::StaticClass()) )
    {
        UClass* MetaClass = Property->IsA(FClassProperty::StaticClass())
            ? CastFieldChecked< FClassProperty>(Property)->MetaClass
            : CastFieldChecked< FSoftClassProperty>(Property)->MetaClass;

        FString NoneStr( TEXT("None") );
        OutOptionStrings.Add( NoneStr );
        if (OutDisplayNames)
        {
            OutDisplayNames->Add(FText::FromString(Nonestr));
        }

        const bool bAllowAbstract = Property->GetOwnerProperty()->HasMetaData(TEXT("AllowAbstract"));
        const bool bBlueprintBaseOnly = Property->GetOwnerProperty()->HasMetaData(TEXT("BlueprintBaseOnly"));
        const bool bAllowOnlyPlaceable = Property->GetOwnerProperty()->HasMetaData(TEXT("OnlyPlaceable"));
        UClass* InterfaceThatMustBeImplemented = Property->GetOwnerProperty()->GetClassMetaData(TEXT("MustImplement"));

        if (!bAllowOnlyPlaceable || MetaClass->IsChildOf< AActor >())
        {
    
```

```

        for (TObjectIterator<UClass> It; It; ++It)
        {
            if (It->IsChildOf(MetaClass)
                && PropertyEditorHelpers::IsEditInlineClassAllowed(*It,
bAllowAbstract)
                && (!bBlueprintBaseOnly ||
FKismetEditorUtilities::CanCreateBlueprintOfClass(*It))
                && (!InterfaceThatMustBeImplemented || It-
>ImplementsInterface(InterfaceThatMustBeImplemented))
                && (!bAllowOnlyPlaceable || !It-
>HasAnyClassFlags(CLASS_Abstract | CLASS_NotPlaceable)))
            {
                OutOptionStrings.Add(It->GetName());
                if (OutDisplayNames)
                {
                    OutDisplayNames->Add(FText::FromString(It->GetName()));
                }
            }
        }
    }
}

```

## MetaClass

- **Function Description:** Used on soft reference attributes to specify the base class of the objects that can be selected
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** string = "abc"
- **Restricted Types:** FSoftClassPath, FSoftObjectPath
- **Commonality:** ★★

Applied to soft reference attributes to specify the base class of the resources to be selected.

Soft reference attributes refer to FSoftClassPath and FSoftObjectPath, which do not inherently impose type restrictions like TSubClassOf; hence, an additional MetaClass can be used to limit the base class of the objects to be selected.

The value in MetaClass can also be an ObjectPath, such as "/Script/Engine.Actor".

## Test Code:

```

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|TSubclassOf")
TSubclassOf<UObject> MyClass_NotMetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|TSubclassOf", meta = (MetaClass = "MyCommonObject"))
TSubclassOf<UObject> MyClass_MetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftClassPath")

```

```

FSoftClassPath MySoftClass_NotMetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftClassPath", meta = (MetaClass = "MyCommonObject"))
FSoftClassPath MySoftClass_MetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftClassPath", meta = (MetaClass = "MyCommonObject"))
TSoftClassPtr<UObject> MySoftClassPtrT_MetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MetaClassTest|UClass*")
UClass* MyClassPtr_NotMetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MetaClassTest|UClass*", meta = (MetaClass = "MyCommonObject"))
UClass* MyClassPtr_MetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftObjectPath")
FSoftObjectPath MySoftObject_NotMetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftObjectPath", meta = (MetaClass = "MyCustomAsset"))
FSoftObjectPath MySoftObject_MetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|FSoftObjectPath", meta = (MetaClass = "MyCustomAsset"))
TSoftObjectPtr<UObject> MySoftObjectPtrT_MetaClass;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|UScriptStruct*")
UScriptStruct* MyStructPtr_NotMetaClass;

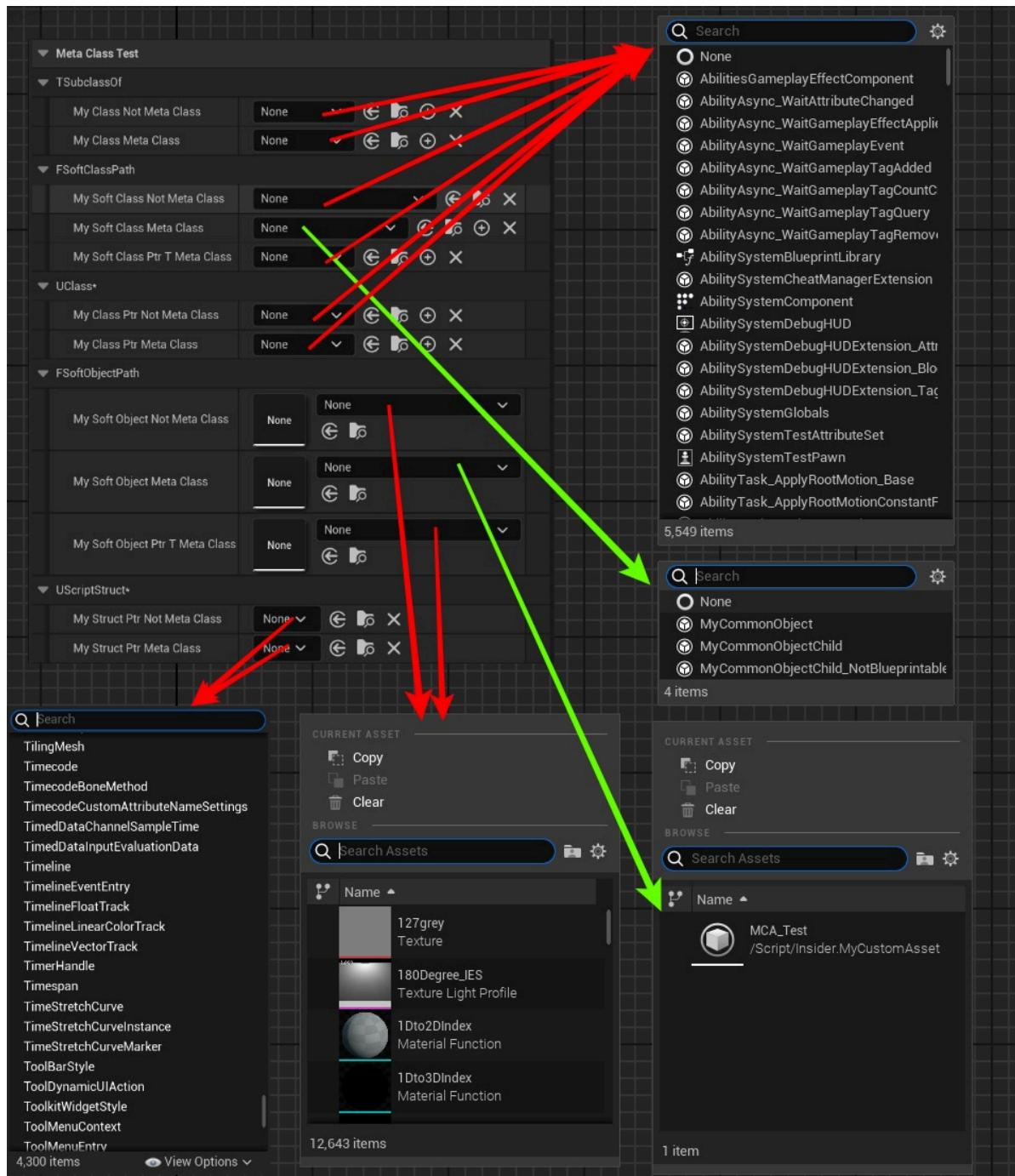
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"MetaClassTest|UScriptStruct*", meta = (MetaClass="MyCommonStruct"))
UScriptStruct* MyStructPtr_MetaClass;

```

## Test Results:

---

The test results show that only the selection lists for MySoftClass\_MetaClass and MySoftObject\_MetaClass have been filtered.



## Principle:

Investigating the source code reveals that both FSoftClassPath and FSoftObjectPath have been customized for type, with their MetaClass and AllowedClass set to SClassPropertyEntryBox and SObjectPropertyEntryBox respectively, based on the MetaClass.

It has been observed that TSoftObjectPtr and TSoftClassPtr have not been customized, thus they do not support this feature. Similarly, UScriptStruct\* does not support this feature, despite also being a selection type.

```
void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPropertyHandle> InPropertyHandle, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils& StructCustomizationUtils)
{
    const FString& MetaClassName = PropertyHandle->GetMetaData("MetaClass");
```

```

        const FString& RequiredInterfaceName = PropertyHandle-
>GetMetaData("RequiredInterface"); // This was the old name, switch to
MustImplement to synchronize with class property
        const FString& MustImplementName = PropertyHandle-
>GetMetaData("MustImplement");
        TArray<const UClass*> AllowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyHandle-
>GetMetaData("AllowedClasses"));
        TArray<const UClass*> DisallowedClasses =
PropertyCustomizationHelpers::GetClassesFromMetadataString(PropertyHandle-
>GetMetaData("DisallowedClasses"));
        const bool bAllowAbstract = PropertyHandle->HasMetaData("AllowAbstract");
        const bool bIsBlueprintBaseOnly = PropertyHandle-
>HasMetaData("IsBlueprintBaseOnly") || PropertyHandle-
>HasMetaData("BlueprintBaseOnly");
        const bool bAllowNone = !(PropertyHandle->GetMetaDataProperty()->
PropertyFlags & CPF_NoClear);
        const bool bShowTreeView = PropertyHandle->HasMetaData("ShowTreeView");
        const bool bHideViewOptions = PropertyHandle->HasMetaData("HideViewOptions");
        const bool bShowDisplayNames = PropertyHandle-
>HasMetaData("ShowDisplayNames");

        const UClass* const MetaClass = !MetaClassName.IsEmpty()
? FEditorClassUtils::GetClassFromString(MetaClassName)
: UObject::StaticClass();

SNew(SClassPropertyParams)
.MetaClass(MetaClass)

}

void FSoftObjectPathCustomization::CustomizeHeader( TSharedRef<IPROPERTYHandle>
InStructPropertyParams, FDetailWidgetRow& HeaderRow,
I.PropertyTypeCustomizationUtils& StructCustomizationUtils )
{
    const FString& MetaClassName = InStructPropertyParams-
>GetMetaData("MetaClass");
    UClass* MetaClass = !MetaClassName.IsEmpty()
? FEditorClassUtils::GetClassFromString(MetaClassName)
: UObject::StaticClass();
    TSharedRef<SOBJECTPropertyParams> ObjectPropertyParams =
SNew(SObjectPropertyParams)
.AllowedClass(MetaClass)
.PropertyHandle(InStructPropertyParams)
.ThumbnailPool(StructCustomizationUtils.GetThumbnailPool());
}

```

## AllowAbstract

- **Function Description:** Used for class attributes to indicate whether abstract classes are accepted.
- **Placement:** UPARAM, UPROPERTY
- **Engine Module:** TypePicker

- **Metadata Type:** bool
- **Restricted Types:** TSubClassOf, FSoftClassPath, UClass\*
- **Commonality:** ★★

## Test Code:

```
UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyCommonObject :public UObject
{
    GENERATED_BODY()
};

UCLASS(BlueprintType)
class INSIDER_API UMyCommonObjectChild :public UMyCommonObject
{
    GENERATED_BODY()
};

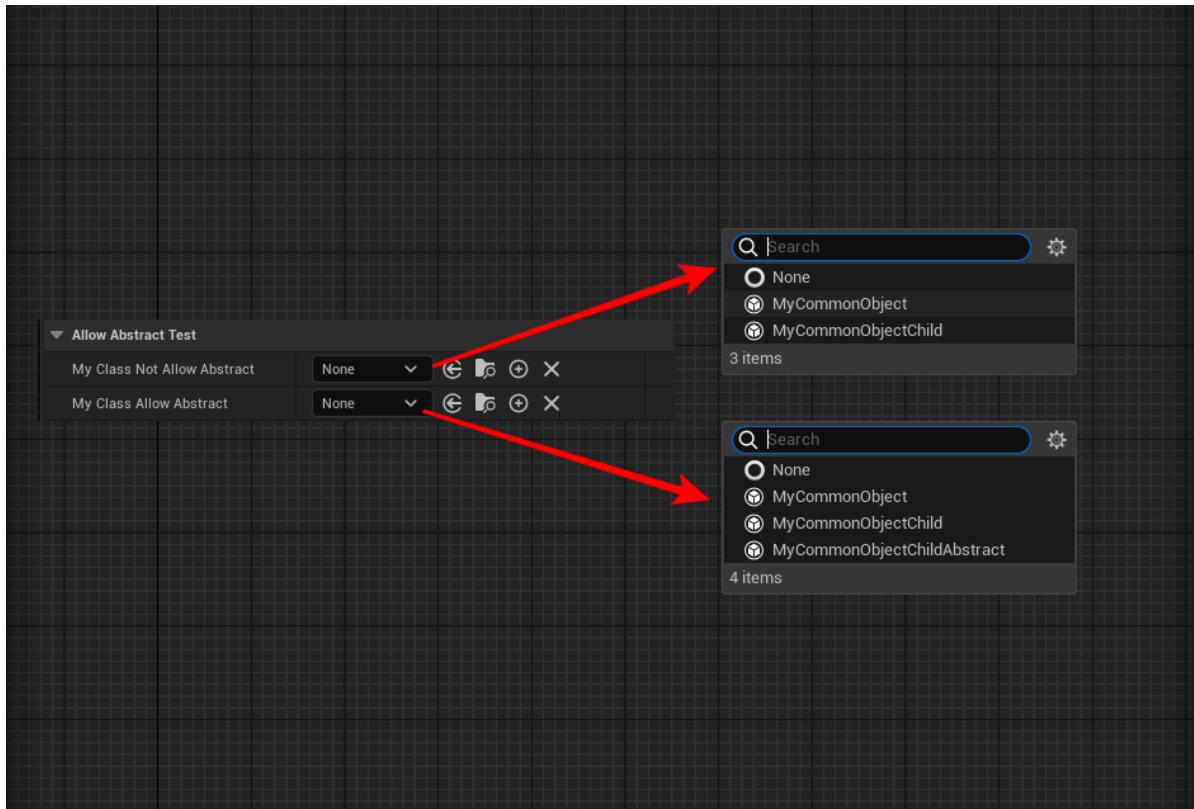
UCLASS(BlueprintType, Abstract)
class INSIDER_API UMyCommonObjectChildAbstract :public UMyCommonObject
{
    GENERATED_BODY()
public:
};

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AllowAbstractTest")
    TSubclassOf<UMyCommonObject> MyClass_NotAllowAbstract;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "AllowAbstractTest",
meta = (AllowAbstract))
    TSubclassOf<UMyCommonObject> MyClass_AllowAbstract;
```

## Test Results:

Visible addition of the abstract class UMyCommonObjectChildAbstract to the class selector with the AllowAbstract attribute.



## Principle:

One of the determinations involves the `IsEditInlineClassAllowed` check, which includes the `bAllowAbstract` flag.

```

bool FPropertyHandleBase::GeneratePossibleValues(TArray< FString>&
OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems,
TArray< FText *>* OutDisplayNames)
{
    if( Property->IsA(FClassProperty::StaticClass()) || Property-
>IsA(FSoftClassProperty::StaticClass()) )
    {
        UClass* MetaClass = Property->IsA(FClassProperty::StaticClass())
            ? CastFieldChecked< FClassProperty >(Property)->MetaClass
            : CastFieldChecked< FSoftClassProperty >(Property)->MetaClass;

        FString NoneStr( TEXT("None") );
        OutOptionStrings.Add( NoneStr );
        if (OutDisplayNames)
        {
            OutDisplayNames->Add(FText::FromString(NoneStr));
        }

        const bool bAllowAbstract = Property->GetOwnerProperty()->HasMetaData(TEXT("AllowAbstract"));
        const bool bBlueprintBaseOnly = Property->GetOwnerProperty()->HasMetaData(TEXT("BlueprintBaseOnly"));
        const bool bAllowOnlyPlaceable = Property->GetOwnerProperty()->HasMetaData(TEXT("OnlyPlaceable"));
        UClass* InterfaceThatMustBeImplemented = Property->GetOwnerProperty()->GetClassMetaData(TEXT("MustImplement"));

        if (!bAllowOnlyPlaceable || MetaClass->IsChildOf< AActor >())
    }
}

```

```

    {
        for (TObjectIterator<UClass> It; It; ++It)
        {
            if (It->IsChildOf(MetaClass)
                && PropertyEditorHelpers::IsEditInlineClassAllowed(*It,
bAllowAbstract)
                && (!bBlueprintBaseOnly ||

FKismetEditorUtilities::CanCreateBlueprintOfClass(*It))
                && (!InterfaceThatMustBeImplemented || It-
>ImplementsInterface(InterfaceThatMustBeImplemented))
                && (!bAllowOnlyPlaceable || !It-
>HasAnyClassFlags(CLASS_Abstract | CLASS_NotPlaceable)))
            {
                OutOptionStrings.Add(It->GetName());
                if (OutDisplayNames)
                {
                    OutDisplayNames->Add(FText::FromString(It->GetName()));
                }
            }
        }
    }

bool IsEditInlineClassAllowed( UClass* CheckClass, bool bAllowAbstract )
{
    return !CheckClass-
>HasAnyClassFlags(CLASS_Hidden|CLASS_HideDropDown|CLASS_Deprecated)
    && (bAllowAbstract || !CheckClass->HasAnyClassFlags(CLASS_Abstract));
}

```

## HideViewOptions

- **Function description:** Used to hide the display options modification function for Class or Struct attributes within the class selector.
- **Use location:** UPROPERTY
- **Engine module:** TypePicker
- **Metadata type:** bool
- **Limit types:** TSubClassOf, FSoftClassPath, UClass, *UScriptStruct*, FInstancedStruct
- **Frequency of use:** ★

Selects attributes of Class or Struct to hide the function for modifying display options in the class selector.

Applies to attribute types such as TSubClassOf, FSoftClassPath, UClass, *UScriptStruct*, and FInstancedStruct, which are used for type selection. It does not work for TSoftObjectPtr or FSoftObjectPath, which are used for object selection.

## Test Code:

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"HideviewOptionsTest|TSubclassOf")
TSubclassOf<UObject> MyClass_NotHideviewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"HideviewOptionsTest|TSubclassOf", meta = (HideViewOptions))
TSubclassOf<UObject> MyClass_HideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"HideviewOptionsTest|UClass*")
UClass* MyClassPtr_NotHideviewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"HideviewOptionsTest|UClass*", meta = (HideViewOptions))
UClass* MyClassPtr_HideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"HideviewOptionsTest|FSoftClassPath")
FSoftClassPath MySoftClass_NotHideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"HideviewOptionsTest|FSoftClassPath", meta = (HideViewOptions))
FSoftClassPath MySoftClass_HideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"HideviewOptionsTest|UScriptStruct*")
UScriptStruct* MyStructPtr_NotHideViewOptions;

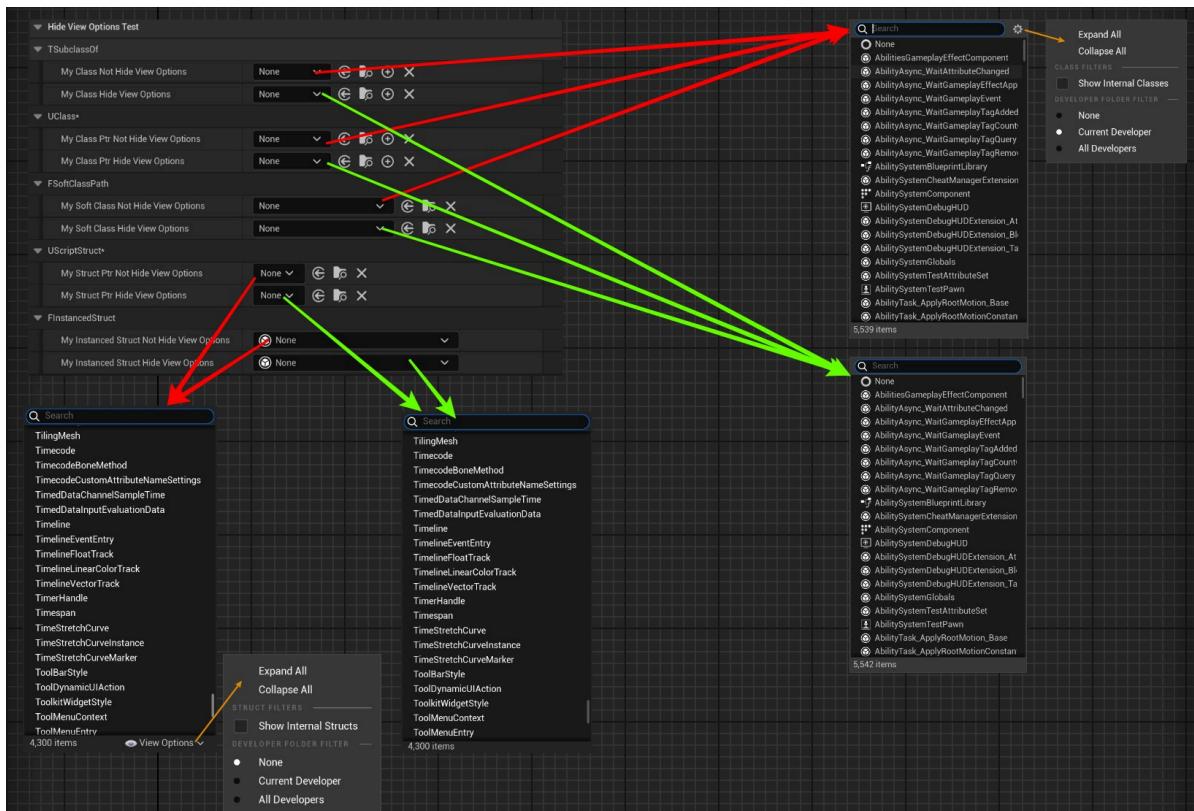
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"HideviewOptionsTest|UScriptStruct*", meta = (HideViewOptions))
UScriptStruct* MyStructPtr_HideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"HideviewOptionsTest|FInstancedStruct")
FInstancedStruct MyInstancedStruct_NotHideViewOptions;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
"HideviewOptionsTest|FInstancedStruct", meta = (HideViewOptions))
FInstancedStruct MyInstancedStruct_HideViewOptions;
```

## Test Code:

It is visible that without HideViewOptions, there is a gear or eye icon in the corner of the pop-up window to modify the display options.



## 测试效果：

In the source code, `SPropertyEditorClass` is used for `TSubClassOf` and `UClass`, `FSoftClassPathCustomization` for `FSoftClassPath`, `SPropertyEditorStruct` for `UScriptStruct`, and `FInstancedStructDetails` for `FInstancedStruct` to customize the UI.

```

void FSoftClassPathCustomization::CustomizeHeader(TSharedRef<IPropertyHandle>
InPropertyParams, FDetailWidgetRow& HeaderRow, IPropertyTypeCustomizationUtils&
StructCustomizationUtils)
{
    const bool bShowTreeView =PropertyParams->HasMetaData("ShowTreeView");
    const bool bHideViewOptions =PropertyParams-
>HasMetaData("HideViewOptions");

    SNew(SClassPropertyEntryBox)
        .ShowTreeView(bShowTreeView)
        .HideViewOptions(bHideViewOptions)
        .ShowDisplayNames(bShowDisplayNames)
}

void SPropertyEditorClass::Construct(const FArguments& InArgs, const TSharedPtr<
FPropertyEditor >& InPropertyEditor)
{
    bShowViewOptions = Property->GetOwnerProperty()->HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = Property->GetOwnerProperty()->HasMetaData(TEXT("ShowDisplayNames"));
}

void SPropertyEditorStruct::Construct(const FArguments& InArgs, const TSharedPtr<
class FPropertyEditor >& InPropertyEditor)

```

```

{
    bShowViewOptions = Property->GetOwnerProperty()-
>HasMetaData(TEXT("HideViewOptions")) ? false : true;
    bShowTree = Property->GetOwnerProperty()-
>HasMetaData(TEXT("ShowTreeView"));
    bShowDisplayNames = Property->GetOwnerProperty()-
>HasMetaData(TEXT("ShowDisplayNames"));
}
TSharedRef<SWidget> FInstancedStructDetails::GenerateStructPicker()
{
    const bool bExcludeBaseStruct = StructProperty-
>HasMetaData(NAME_ExcludeBaseStruct);
    const bool bAllowNone = !(StructProperty->GetMetaDataProperty()-
>PropertyFlags & CPF_NoClear);
    const bool bHideViewOptions = StructProperty-
>HasMetaData(NAME_HideViewOptions);
    const bool bShowTreeView = StructProperty-
>HasMetaData(NAME_ShowTreeView);
}

```

## OnlyPlaceable

- **Function Description:** Used on class properties to indicate whether only Actors that can be placed within a scene are accepted
- **Usage Location:** UPROPERTY
- **Engine Module:** TypePicker
- **Metadata Type:** bool
- **Restricted Types:** TSubClassOf, FSoftClassPath, UClass\*
- **Commonality:** ★★

Ability to exclude certain Actor classes, such as AInfo, which cannot be placed in a scene.

## Test Code:

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActor :public AActor
{
GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API AMyActorChild_Placeable :public AMyActor
{
GENERATED_BODY()
};

UCLASS(Blueprintable, BlueprintType, NotPlaceable)
class INSIDER_API AMyActorChild_NotPlaceable :public AMyActor
{
GENERATED_BODY()
};

```

```

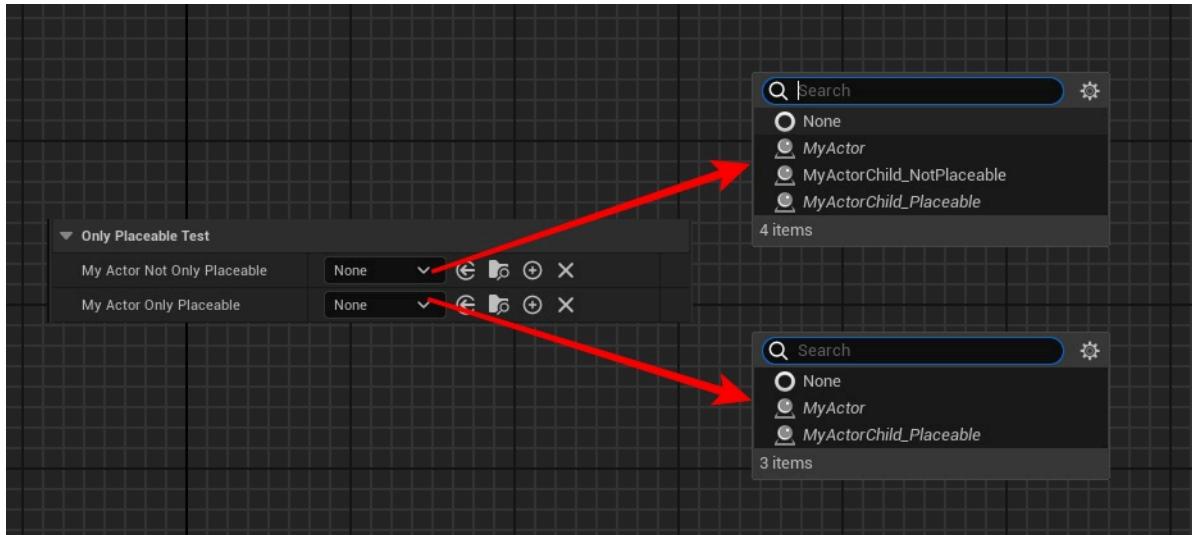
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "OnlyPlaceableTest")
    TSubclassOf<AMyActor> MyActor_NotOnlyPlaceable;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "OnlyPlaceableTest",
meta = (OnlyPlaceable))
    TSubclassOf<AMyActor> MyActor_OnlyPlaceable;

```

## Test Effects:

Visible AMyActorChild\_NotPlaceable class cannot be selected by the MyActor\_OnlyPlaceable property due to the NotPlaceable tag.



## Principle:

```

bool FPropertyHandleBase::GeneratePossibleValues(TArray< FString>&
OutOptionStrings, TArray< FText >& OutToolTips, TArray< bool >& OutRestrictedItems,
TArray< FText *> OutDisplayNames)
{
    if( Property->IsA(FClassProperty::StaticClass()) || Property-
>IsA(FSoftClassProperty::StaticClass()) )
    {
        UClass* MetaClass = Property->IsA(FClassProperty::StaticClass())
            ? CastFieldChecked< FClassProperty >(Property)->MetaClass
            : CastFieldChecked< FSoftClassProperty >(Property)->MetaClass;

        FString NoneStr( TEXT("None") );
        OutOptionStrings.Add( NoneStr );
        if (OutDisplayNames)
        {
            OutDisplayNames->Add(FText::FromString(NoneStr));
        }

        const bool bAllowAbstract = Property->GetOwnerProperty()->HasMetaData(TEXT("AllowAbstract"));
        const bool bBlueprintBaseOnly = Property->GetOwnerProperty()->HasMetaData(TEXT("BlueprintBaseOnly"));
        const bool bAllowOnlyPlaceable = Property->GetOwnerProperty()->HasMetaData(TEXT("OnlyPlaceable"));
    }
}

```

```

UClass* InterfaceThatMustBeImplemented = Property->GetOwnerProperty()->GetClassMetaData(TEXT("MustImplement"));

    if (!bAllowOnlyPlaceable || MetaClass->IsChildOf<AAActor>())
    {
        for (TObjectIterator<UClass> It; It; ++It)
        {
            if (It->IsChildOf(MetaClass)
                && PropertyEditorHelpers::IsEditInlineClassAllowed(*It,
bAllowAbstract)
                && (!bBlueprintBaseOnly ||
FKismetEditorUtilities::CanCreateBlueprintOfClass(*It))
                && (!InterfaceThatMustBeImplemented || It-
>ImplementsInterface(InterfaceThatMustBeImplemented))
                && (!bAllowOnlyPlaceable || !It-
>HasAnyClassFlags(CLASS_Abstract | CLASS_NotPlaceable)))
            {
                OutOptionStrings.Add(It->GetName());
                if (OutDisplayNames)
                {
                    OutDisplayNames->Add(FText::FromString(It->GetName()));
                }
            }
        }
    }
}

```

## Documentation Policy

- **Function Description:** Specifies the rules for document validation, which can currently only be set to Strict
- **Usage Location:** Anywhere
- **Engine Module:** UHT
- **Metadata Type:** string="abc"
- **Commonality:** ★

Within the ValidateDocumentationPolicy function of UHT, this value primarily determines if a Comment or Tooltip is provided for a type or field, or if a Float variable has corresponding "UIMin / UIMax" settings. This information is then extracted to generate the corresponding documentation.

Currently, the only available configuration is Strict, which enables all checks by default. Therefore, configuring DocumentationPolicy=Strict in the C++ source code implies a desire for the engine to enforce documentation standards.

```

.documentationPolicies["Strict"] = new()
{
    ClassOrStructCommentRequired = true,
    FunctionToolTipsRequired = true,
    MemberToolTipsRequired = true,
    ParameterToolTipsRequired = true,
    FloatRangesRequired = true,
};

```

```

protected override void validateDocumentationPolicy(UhtDocumentationPolicy
policy)
{
    if (policy.ClassOrStructCommentRequired)
    {
        string classTooltip = MetaData.GetValueOrDefault(UhtNames.ToolTip);
        if (classTooltip.Length == 0 || classTooltip.Equals(EngineName,
StringComparison.OrdinalIgnoreCase))
        {
            this.LogError($"{EngineType.CapitalizedText()} '{SourceName}' 
does not provide a tooltip / comment (DocumentationPolicy).");
        }
    }
//...
}

```

## Similar examples can be found in the source code:

```

USTRUCT(meta=(DisplayName="Set Transform", Category="Transforms", TemplateName =
"Set Transform", DocumentationPolicy = "Strict",
Keywords="SetBoneTransform,SetControlTransform,SetInitialTransform,SetSpaceTransf
orm", NodeColor="0, 0.364706, 1.0", Varying))
struct CONTROLRIG_API FRigUnit_SetTransform : public FRigUnitMutable
{
}

```

## My own test code:

```

UCLASS(BlueprintType, meta = (DocumentationPolicy=Strict))
class INSIDER_API UMyClass_DocumentationPolicy :public UObject
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;

    //This is a float
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (UIMin = "0.0", UIMax =
"100.0"))
    float MyFloat_withValidate;

UFUNCTION(meta = (DocumentationPolicy=Strict))
void MyFunc() {}

/**
 * Test Func for validate param
 * @param keyOtherName The name of Key
 * @param keyValue
 */
UFUNCTION(BlueprintCallable, meta = (DocumentationPolicy=Strict))

```

```

        int MyFunc_ValidateParamFailed(FString keyName,int keyValue){return
0;}//Parameter annotation verification is only activated when there is at least
one "@param" tag

/**
 *  Test Func for validate param
 *
 *  @param keyName The name of key
 *  @param keyValue The value of key
 *  @return Return operation result
 */
UFUNCTION(meta = (DocumentationPolicy=Strict))
int MyFunc_ValidateParam(FString keyName,int keyValue){return 0;
};

USTRUCT(BlueprintType, meta = (DocumentationPolicy=Strict))
struct INSIDER_API FMyStruct_DocumentationPolicy
{
    GENERATED_BODY()
public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    float MyFloat;
    UPROPERTY(BlueprintReadWrite, EditAnywhere)
    FString MyString;
};

UENUM(BlueprintType, meta = (DocumentationPolicy=Strict))
enum class EMyEnum_DocumentationPolicy :uint8
{
    First,
    Second,
    Third,
};

// This a tooltip / comment
UCLASS(BlueprintType, meta = (DocumentationPolicy = Strict))
class INSIDER_API UMyClass_DocumentationPolicy_TypeA :public UObject
{
    GENERATED_BODY()
};

/**
 *  This a tooltip / comment
 *
 */
UCLASS(BlueprintType, meta = (DocumentationPolicy = Strict))
class INSIDER_API UMyClass_DocumentationPolicy_TypeB :public UObject
{
    GENERATED_BODY()
};

UCLASS(BlueprintType, meta = (DocumentationPolicy = Strict,ToolTip="This a
tooltip")) //Cannot use ShortToolTip
class INSIDER_API UMyClass_DocumentationPolicy_TypeC :public UObject
{
    GENERATED_BODY()
};

```

```
};
```

## The resulting UHT compilation error:

```
error : Class 'UMyClass_DocumentationPolicy' does not provide a tooltip /  
comment(DocumentationPolicy).  
error : Property 'UMyClass_DocumentationPolicy::MyFloat' does not provide a  
tooltip / comment(DocumentationPolicy).  
error : Property 'UMyClass_DocumentationPolicy::MyString' does not provide a  
tooltip / comment(DocumentationPolicy).  
error : Property 'UMyClass_DocumentationPolicy::MyFloat' does not provide a valid  
UIMin / UIMax(DocumentationPolicy).  
error : Function 'UMyClass_DocumentationPolicy::MyFunc' does not provide a  
tooltip / comment(DocumentationPolicy).  
error : Function 'UMyClass_DocumentationPolicy::MyFunc' does not provide a  
comment(DocumentationPolicy).  
error : Function 'UMyClass_DocumentationPolicy::MyFunc_ValidateParamFailed'  
doesn't provide a tooltip for parameter 'keyName' (DocumentationPolicy).  
error : Function 'UMyClass_DocumentationPolicy::MyFunc_ValidateParamFailed'  
doesn't provide a tooltip for parameter 'keyValue' (DocumentationPolicy).  
error : Function 'UMyClass_DocumentationPolicy::MyFunc_ValidateParamFailed'  
provides a tooltip for an unknown parameter 'keyOtherName'  
error : Struct 'FMyStruct_DocumentationPolicy' does not provide a tooltip /  
comment(DocumentationPolicy).  
error : Property 'FMyStruct_DocumentationPolicy::MyFloat' does not provide a  
tooltip / comment(DocumentationPolicy).  
error : Property 'FMyStruct_DocumentationPolicy::MyString' does not provide a  
tooltip / comment(DocumentationPolicy).  
error : Property 'FMyStruct_DocumentationPolicy::MyFloat' does not provide a  
valid UIMin / UIMax(DocumentationPolicy).  
error : Enum 'EMyEnum_DocumentationPolicy' does not provide a tooltip /  
comment(DocumentationPolicy)  
error : Enum entry  
'EMyEnum_DocumentationPolicy::EMyEnum_DocumentationPolicy::First' does not  
provide a tooltip / comment(DocumentationPolicy)  
error : Enum entry  
'EMyEnum_DocumentationPolicy::EMyEnum_DocumentationPolicy::Second' does not  
provide a tooltip / comment(DocumentationPolicy)  
error: Enum entry  
'EMyEnum_DocumentationPolicy::EMyEnum_DocumentationPolicy::Third' does not  
provide a tooltip / comment(DocumentationPolicy)
```

#GetByRef

- **Function Description:** Instructs UHT to generate C++ code that returns a reference for this attribute
- **Usage Location:** UPROPERTY
- **Engine Module:** UHT
- **Metadata Type:** bool
- **Restriction:** Applicable only to attributes within structures specified by SparseClassDataTypes.
- **Associated Items:** SparseClassDataTypes

Indicates that UHT should generate C++ code that returns a reference for this attribute.

Applicable only to attributes within structures specified by SparseClassDataTypes.

## Code Example:

```
USTRUCT(BlueprintType)
struct FMySparseClassData
{
    GENERATED_BODY()

    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly)
    FString MyString_EditDefault = TEXT("MyName");
    //FString GetMyString_EditDefault() const { return
    GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
    >MyString_EditDefault; } \
    // "GetByRef" means that Blueprint graphs access a const ref instead of a
    copy.
    UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, meta = (GetByRef))
    FString MyString_EditDefault_ReadOnly = TEXT("MyName");
    //const FString& GetMyString_EditDefault_ReadOnly() const { return
    GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
    >MyString_EditDefault_ReadOnly; }
};

UCLASS(Blueprintable, BlueprintType, SparseClassDataTypes = MySparseClassData)
class INSIDER_API AMyActor_SparseClassDataTypes :public AActor
{
    GENERATED_BODY()
}
```

## Generated Code:

It can be seen that the return value generated by the latter is const FString& instead of FString .

```
#define
FID_Hello_Source_Insider_Class_Trait_MyClass_SparseClassDataTypes_h_36_SPARSE_DAT
A_PROPERTY_ACCESSORS \
FString GetMyString_EditDefault() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
>MyString_EditDefault; } \
const FString& GetMyString_EditDefault_ReadOnly() const { return
GetMySparseClassData(EGetSparseClassDataMethod::ArchetypeIfNull)-
>MyString_EditDefault_ReadOnly; }
```

## Principle:

When generating code for SparseDataType in UHT, GetByRef will be judged to generate different format codes respectively.

```

private StringBuilder AppendSparseDeclarations(StringBuilder builder, uhtClass
classObj, IEnumerable<UhtScriptStruct> sparseScriptStructs,
uhtUsedDefineScopes<UhtProperty> sparseProperties)
{
    if (property.MetaData.ContainsKey(UhtNames.GetByRef))
    {
        builder.Append("const ").AppendSparse(property).Append("&
Get").Append(cleanPropertyName).Append("( ) const");
    }
    else
    {
        builder.AppendSparse(property).Append(""
Get").Append(cleanPropertyName).Append("( ) const");
    }
}

```

## CustomThunk

---

- **Function Description:** Instructs UHT not to generate an auxiliary function for blueprint invocation for this function, requiring the user to define it customly.
- **Usage Location:** UFUNCTION
- **Engine Module:** UHT
- **Metadata Type:** bool
- **Associated Items:**  
UFUNCTION: Service Request, CustomThunk
- **Commonality:** ★★★★☆

## NativeConstTemplateArg

---

- **Function description:** Specify that this attribute is a const template parameter.
- **Usage Location:** UPROPERTY
- **Engine Module:** UHT
- **Metadata Type:** bool
- **Commonality:** 0

Specifies that this attribute is a const template parameter.

No usage is found in the source code; it is only utilized within UHT.

Within UHT, it is primarily used in UhtArrayProperty, UhtObjectPropertyBase, and UhtOptionalProperty.

## CppFromBpEvent

---

- **Usage Location:** To be determined
- **Engine Module:** UHT
- **Metadata Type:** boolean
- **Commonality:** 0

Indicates that this is a Blueprint event defined within C++.

Former versions of UHT utilized this metadata, but it is no longer used in the current engine versions.

## Original Code:

```
public static class UhtFunctionParser
{
    private static UhtParseResult ParseUFunction(UhtParsingScope parentscope,
UhtToken token)
    {
        if (function.MetaData.Containskey(UhtNames.CppFromBpEvent))
        {
            function.FunctionFlags |= EFunctionFlags.Event;
        }
    }
}
```

## Include Path

- **Function Description:** Records the reference path for a UClass
- **Usage Location:** UClass
- **Engine Module:** UHT
- **Metadata Type:** string="abc"
- **Restriction Type:** Information related to UClass
- **Commonality:** 0

Records the reference path for a UClass.

Developers typically do not need to concern themselves with this value.

One of its uses is to facilitate the inclusion of the class header file during the generation of .gen.cpp by UHT in the header file section.

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Template :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    int32 MyFunc(FString str){return 0;}
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
};
```

# Its Type Information:

```
[class MyProperty_Template  Class->Struct->Field->Object
/Script/Insider.MyProperty_Template]
(BlueprintType = true, IncludePath = Property/MyProperty_Template.h,
ModuleRelativePath = Property/MyProperty_Template.h)
ObjectFlags: RF_Public | RF_Standalone | RF_Transient
Outer: Package /Script/Insider
ClassHierarchy: MyProperty_Template:Object
ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI |
CLASS_TokenStreamAssembled | CLASS_Intrinsic | CLASS_Constructed
Size: 56
Within: Object
ClassConfigName: Engine
{
    (Category = MyProperty_Template, ModuleRelativePath =
Property/MyProperty_Template.h)
    48-[4] int32 MyProperty;
        PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash | |
CPF_NativeAccessSpecifierPublic
        ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
        Outer: Class /Script/Insider.MyProperty_Template
        Path: IntProperty /Script/Insider.MyProperty_Template:MyProperty
    [func MyFunc    Function->Struct->Field->Object
/Script/Insider.MyProperty_Template:MyFunc]
        (ModuleRelativePath = Property/MyProperty_Template.h)
        ObjectFlags: RF_Public | RF_Transient
        Outer: Class /Script/Insider.MyProperty_Template
        FunctionFlags: FUNC_Final | FUNC_Native | FUNC_Public | |
FUNC_BlueprintCallable
        NumParms: 2
        ParmSize: 20
        ReturnValueOffset: 16
        RPCId: 0
        RPCResponseId: 0
    public int32 MyFunc(FString str)final;
{
    0-[16] FString str;
        PropertyFlags: CPF_Parm | CPF_ZeroConstructor | |
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
        ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
        Outer: Function /Script/Insider.MyProperty_Template:MyFunc
        Path: StrProperty /Script/Insider.MyProperty_Template:MyFunc:str
    16-[4] int32 ReturnValue;
        PropertyFlags: CPF_Parm | CPF_OutParm | CPF_ZeroConstructor | |
CPF_ReturnParm | CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
| CPF_NativeAccessSpecifierPublic
        ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
        Outer: Function /Script/Insider.MyProperty_Template:MyFunc
        Path: IntProperty
/Script/Insider.MyProperty_Template:MyFunc:ReturnValue
};
};
```

## Principle:

The value is also added after analysis in UHT. For the specific logic, refer to the principle code section within ModuleRelativePath.

```
protected override void UhtClass::ResolveSuper(UhtResolvePhase resolvePhase)
{
    switch (classType)
    {
        case UhtClassType::Class:
        {
            MetaData.Add(UhtNames::IncludePath,
HeaderFile.IncludeFilePath);
        }
    }
}
```

## ModuleRelativePath

- **Function Description:** Records the header file path for type definitions, representing the relative path within the module.
- **Usage Location:** Any
- **Engine Module:** UHT
- **Metadata Type:** string="abc"
- **Commonality Level:** 0

Records the header file path for the current metadata type definition, which is the relative path within the module.

Generally not managed by developers, but the engine editor uses it to locate where a type is defined in a .h file, allowing the corresponding header file to be opened in Visual Studio when the type is double-clicked. The specific logic can be found in FSourceCodeNavigation.

The difference from IncludePath is that ModuleRelativePath is present on various type information, while IncludePath is used only for UCLASS. Additionally, the value of ModuleRelativePath can begin with "Classes/Public/Internal/Private", and we typically recommend organizing .h and .cpp files into these four directories. The value of IncludeFilePath, however, will exclude this prefix.

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Template :public UObject
{
    GENERATED_BODY()
public:
    UFUNCTION(BlueprintCallable)
    int32 MyFunc(FString str){return 0;}
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    int32 MyProperty = 123;
};
```

## Metadata type information printout:

It can be observed that ModuleRelativePath is present on classes, properties, and functions.

IncludePath is only present on UCLASS.

```
[class MyProperty_Template Class->Struct->Field->Object
/Script/Insider.MyProperty_Template]
(BlueprintType = true, IncludePath = Property/MyProperty_Template.h,
ModuleRelativePath = Property/MyProperty_Template.h)
ObjectFlags: RF_Public | RF_Standalone | RF_Transient
Outer: Package /Script/Insider
ClassHierarchy: MyProperty_Template:Object
ClassFlags: CLASS_MatchedSerializers | CLASS_Native | CLASS_RequiredAPI |
CLASS_TokenStreamAssembled | CLASS_Intrinsic | CLASS_Constructed
Size: 56
Within: Object
ClassConfigName: Engine
{
    (Category = MyProperty_Template, ModuleRelativePath =
Property/MyProperty_Template.h)
    48-[4] int32 MyProperty;
    PropertyFlags: CPF_Edit | CPF_BlueprintVisible | CPF_ZeroConstructor |
CPF_IsPlainOldData | CPF_NoDestructor | CPF_HasGetValueTypeHash |
CPF_NativeAccessSpecifierPublic
    ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
    Outer: Class /Script/Insider.MyProperty_Template
    Path: IntProperty /Script/Insider.MyProperty_Template:MyProperty
    [func MyFunc Function->Struct->Field->Object
/Script/Insider.MyProperty_Template:MyFunc]
    (ModuleRelativePath = Property/MyProperty_Template.h)
    ObjectFlags: RF_Public | RF_Transient
    Outer: Class /Script/Insider.MyProperty_Template
    FunctionFlags: FUNC_Final | FUNC_Native | FUNC_Public |
FUNC_BlueprintCallable
    NumParms: 2
   ParmsSize: 20
    ReturnValueOffset: 16
    RPCId: 0
    RPCResponseId: 0
    public int32 MyFunc(FString str)final;
```

```

{
    0-[16] FString str;
    PropertyFlags: CPF_Parm | CPF_ZeroConstructor | 
CPF_HasGetValueTypeHash | CPF_NativeAccessSpecifierPublic
    ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
    Outer: Function /Script/Insider.MyProperty_Template:MyFunc
    Path: StrProperty /Script/Insider.MyProperty_Template:MyFunc:str
    16-[4] int32 ReturnValue;
    PropertyFlags: CPF_Parm | CPF_OutParm | CPF_ZeroConstructor | 
CPF_ReturnParm | CPF_IsPlainOldDataData | CPF_NoDestructor | CPF_HasGetValueTypeHash
| CPF_NativeAccessSpecifierPublic
    ObjectFlags: RF_Public | RF_MarkAsNative | RF_Transient
    Outer: Function /Script/Insider.MyProperty_Template:MyFunc
    Path: IntProperty
/Script/Insider.MyProperty_Template:MyFunc:ReturnValue
};

}

```

## Principle:

---

During UHT analysis, the type is automatically annotated with the header file path information.

As seen in the source code logic, the value of ModuleRelativePath can begin with "Classes/Public/Internal/Private", and we generally suggest organizing .h and .cpp files into these four folders. The value of IncludeFilePath, on the other hand, will not include this prefix.

```

public enum UhtHeaderFileType
{
    /// <summary>
    /// Classes folder
    /// </summary>
    Classes,

    /// <summary>
    /// Public folder
    /// </summary>
    Public,

    /// <summary>
    /// Internal folder
    /// </summary>
    Internal,

    /// <summary>
    /// Private folder
    /// </summary>
    Private,
}

public static void AddModuleRelativePathToMetaData(UhtMetaData metaData,
UhtHeaderFile headerFile)
{

```

```

        metaData.Add(uhtNames.ModuleRelativePath, headerFile.ModuleRelativeFilePath);
    }

//Analyze file path
private void StepPrepareHeaders(UhtPackage package, IEnumerable<string>
headerFiles, UhtHeaderFileType headerFileType)
{
    string typeDirectory = headerFileType.ToString() + '/';

    headerFile.ModuleRelativeFilePath = normalizedFullPath[stripLength..];
    if (normalizedFullPath[stripLength..].StartsWith(typeDirectory, true,
null))
    {
        stripLength += typeDirectory.Length;
    }
    headerFile.IncludeFilePath = normalizedFullPath[stripLength..];
}

```

## DisableNativeTick

- **Function description:** Disable the NativeTick for this UserWidget.
- **Usage location:** UCLASS
- **Engine module:** Widget Property
- **Metadata type:** bool
- **Restriction type:** Subclass of UserWidget
- **Commonly used:** ★★★

Disable the NativeTick for this UserWidget.

Will not function if there is only a C++ class, as a pure C++ Widget lacks a WidgetBPClass.

Moreover, subclasses of BP must remove the Tick blueprint node.

## Test code:

```

UCLASS(BlueprintType, meta=())
class INSIDER_API UMyWidget_WithNativeTick :public UUserWidget
{
    GENERATED_BODY()
public:
    virtual void NativeTick(const FGeometry& MyGeometry, float InDeltaTime)
override
{
    Super::NativeTick(MyGeometry, InDeltaTime);
    UKismetSystemLibrary::PrintString(nullptr, TEXT("WithNativeTick"), true);
}
};

UCLASS(BlueprintType,meta=(DisableNativeTick))
class INSIDER_API UMyWidget_DisableNativeTick :public UUserWidget
{
    GENERATED_BODY()

```

```

public:
    virtual void NativeTick(const FGeometry& MyGeometry, float InDeltaTime)
override
{
    Super::NativeTick(MyGeometry, InDeltaTime);
    UKismetSystemLibrary::PrintString(nullptr, TEXT("DisableNativeTick")),
true);
}
};

```

## Test results:

Create the UMG\_WithTick and UMG\_DisableTick subclasses of UMyWidget\_WithNativeTick and UMyWidget\_DisableNativeTick in the blueprint, respectively. Then, add them to a UMG, place them on the screen, and observe the invocation of NativeTick.

Only UMG\_WithTick will be invoked.



## Principle:

Marking the UCLASS will set the UMG blueprint's bClassRequiresNativeTick to false. This is then evaluated in the UpdateCanTick function of UUserWidget. If WidgetBPClass is not null (indicating a Blueprint subclass) and ClassRequiresNativeTick is false, bCanTick will initially be set to false. Next, checking bHasScriptImplementedTick requires that there is no EventTick in the blueprint (which is created by default and must be manually deleted). Furthermore, it must be confirmed that there are no delayed blueprint nodes and no animations. In summary, if the Widget truly has no need for a Tick, bCanTick can finally be set to false.

```

void UwidgetBlueprint::UpdateTickabilityStats(bool& OutHasLatentActions, bool&
OutHasAnimations, bool& OutClassRequiresNativeTick)
{
    static const FName DisableNativeTickMetaTag("DisableNativeTick");
    const bool bClassRequiresNativeTick = !NativeParent-
>HasMetaData(DisableNativeTickMetaTag);
    OutClassRequiresNativeTick = bClassRequiresNativeTick;
}

```

```

}

void FWidgetBlueprintCompilerContext::CopyTermDefaultsToDefaultObject(UObject* DefaultObject)
{
    WidgetBP->UpdateTickabilityStats(bClassOrParentsHaveLatentActions,
bClassOrParentsHaveAnimations, bClassRequiresNativeTick);
    WidgetClass->SetClassRequiresNativeTick(bClassRequiresNativeTick);
}

void UUserWidget::UpdateCanTick()
{
    UWidgetBlueprintGeneratedClass* WidgetBPClass =
Cast<UWidgetBlueprintGeneratedClass>(GetClass());
    bCanTick |= !WidgetBPClass || WidgetBPClass->ClassRequiresNativeTick();
    bCanTick |= bHasScriptImplementedTick;
    bCanTick |= World->GetLatentActionManager().GetNumActionsForObject(this)
!= 0;
    bCanTick |= ActiveSequencePlayers.Num() > 0;
    bCanTick |= QueuedWidgetAnimationTransitions.Num() > 0;
    SafeGCWidget->SetCanTick(bCanTick);
}

```

## ViewmodelBlueprintWidgetExtension

---

- **Function Description:** This function verifies if the Object type of InListItems matches the ViewModelProperty bound to EntryWidgetClass via MVVM.
- **Usage Location:** UFUNCTION
- **Engine Module:** Widget Property
- **Metadata Type:** string="abc"
- **Frequency of Use:** 0

This function verifies if the Object type of InListItems conforms to the ViewModelProperty bound to EntryWidgetClass in an MVVM context.

Currently, this function is only used within the ListView.

## Principle:

```
UCLASS(meta = (EntryInterface = "/Script/UMG.UserObjectListEntry"), MinimalAPI)
class UListView : public UListViewBase, public ITypedUMGListView<UObject*>
{
    UFUNCTION(BlueprintCallable, Category = ListView, meta = (AllowPrivateAccess
= true, DisplayName = "Set List Items", ViewmodelBlueprintWidgetExtension =
"EntryViewModel"))
    UMG_API void BP_SetListItems(const TArray<UObject*>& InListItems);
}

void
UMVVMViewBlueprintListViewBaseExtension::Precompile(UE::MVVM::Compiler::IMVVMBlue
printViewPrecompile* Compiler, UWidgetBlueprintGeneratedClass* Class)
{
}
```

## DesignerRebuild

- **Function description:** Specifies that the UMG preview interface should be refreshed when a certain attribute within a Widget is modified.
- **Use location:** UPROPERTY
- **Engine module:** Widget Property
- **Metadata type:** bool
- **Restriction type:** Attributes in UWidget subclasses
- **Frequency of use:** ★

Specifies that the UMG preview interface should be refreshed after a certain attribute value in the Widget is changed.

The first question that comes to mind is, which attributes require the use of the DesignerRebuild tag?

This attribute is seldom needed. Typically, when a Widget's attribute is updated, it only needs to refresh its own display, not the entire interface, such as font size. There are likely two scenarios where its use is necessary:

1. Changes to some attributes significantly alter the control's style. While it's possible to refine the update to only redraw the control itself, it's often simpler to refresh the entire preview interface, especially within the editor environment. For instance, the bSimpleTextMode of UTextBlock and EntryWidgetClass under UListViewBase result in substantial changes.
2. When certain attributes impact other elements across the entire interface, it's also practical to refresh everything. Although I can't provide a specific example, if a user's custom control requires this, the tag can be applied.

## Source code examples include:

```
UCLASS(DisplayName="Text", MinimalAPI)
class UTextBlock : public UTextLayoutWidget
{
```

```

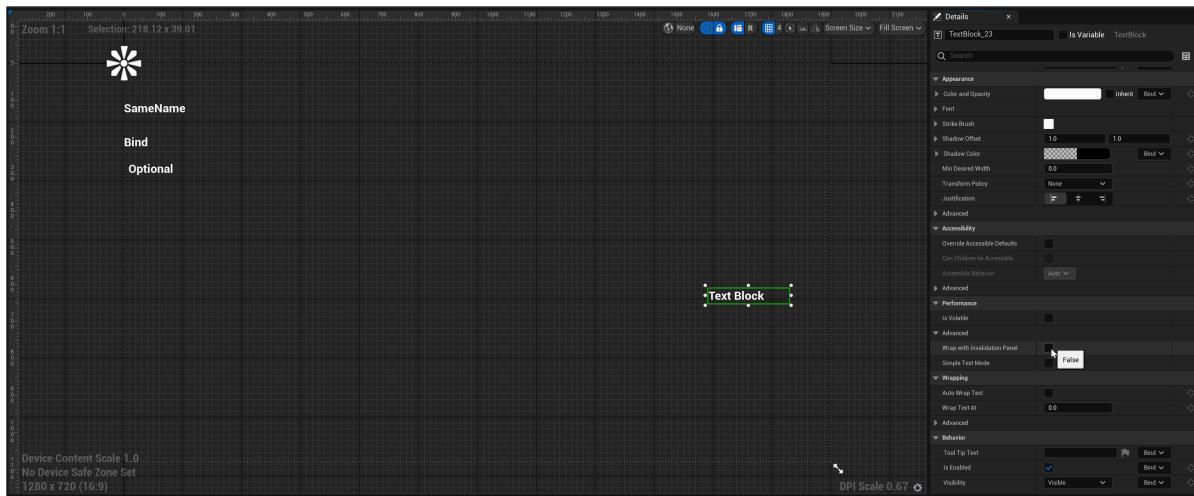
    /**
     * If this is enabled, text shaping, wrapping, justification are disabled in
     favor of much faster text layout and measurement.
     * This feature is only suitable for "simple" text (ie, text containing only
     numbers or basic ASCII) as it disables the complex text rendering support
     required for certain languages (such as Arabic and Thai).
     * It is significantly faster for text that can take advantage of it
     (particularly if that text changes frequently), but shouldn't be used for
     localized user-facing text.
    */
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category=Performance,
AdvancedDisplay, meta=(AllowPrivateAccess = "true", DesignerRebuild))
    bool bSimpleTextMode;
}

UCLASS(Abstract, NotBlueprintable, hidedropdown, meta = (EntryInterface =
UserListEntry), MinimalAPI)
class UListViewBase : public UWidget
{
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = ListEntries, meta =
(DesignerRebuild, AllowPrivateAccess = true, MustImplement =
"/Script/UMG.UserListEntry"))
    TSubclassOf<UUserWidget> EntryWidgetClass;
}

```

## The test effect for UTextBlock:

You can observe that when bSimpleTextMode is changed, the left-hand preview interface flickers and refreshes. This does not happen when other buttons are clicked.



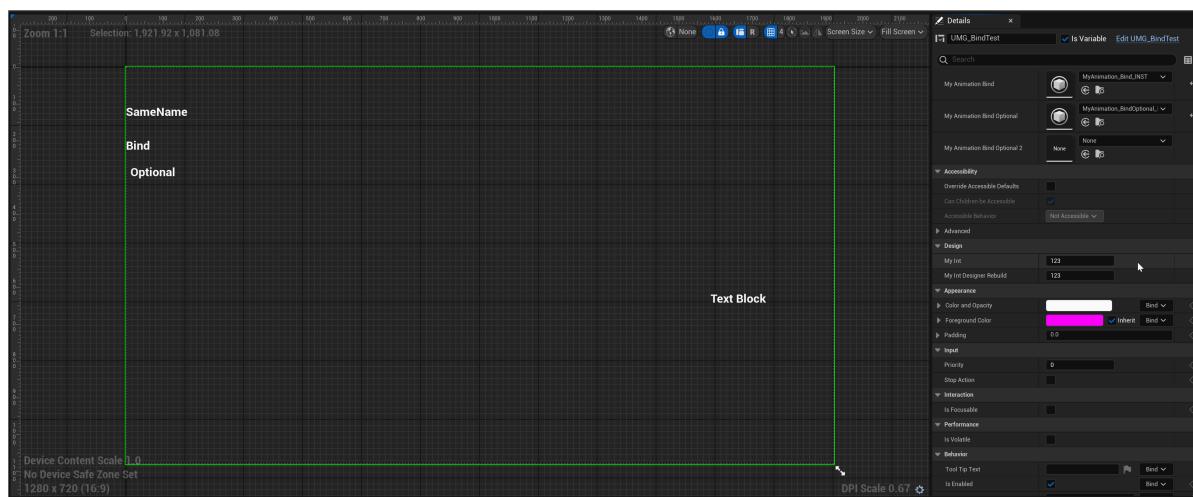
## Test code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_BindWidget :public UUserWidget
{
public:
    UPROPERTY(EditAnywhere, Category = Design)
    int32 MyInt = 123;

    UPROPERTY(EditAnywhere, Category = Design, meta = (DesignerRebuild))
    int32 MyInt_DesignerRebuild = 123;
}
```

## Test effect:

It is evident that when the ordinary attribute MyInt is changed, the interface does not refresh. However, when MyInt\_DesignerRebuild is altered, the number in the upper left corner of the interface flickers (though the interface itself undergoes no substantial change).



## Principle:

When a property with the DesignerRebuild tag in a Widget is modified, InvalidatePreview is notified to update the preview window within the editor.

```
void SWidgetDetailsView::NotifyPostChange(const FPropertyChangedEvent& PropertyChangedEvent, FEditPropertyChain* PropertyThatChanged)
{
    const static FName DesignerRebuildName("DesignerRebuild");

    //...
    // If the property that changed is marked as "DesignerRebuild" we invalidate
    // the preview.
    if (PropertyChangedEvent.Property->HasMetaData(DesignerRebuildName) ||
        PropertyThatChanged->GetActiveMemberNode()->GetValue()-
        >HasMetaData(DesignerRebuildName) )
    {
        const bool bViewOnly = true;
        BlueprintEditor.Pin()->InvalidatePreview(bViewOnly);
    }
}
```

```
}
```

# DefaultGraphNode

- **Function Description:** Identifies the default blueprint nodes created by the engine.
- **Usage Location:** UCLASS
- **Engine Module:** Widget Property
- **Metadata Type:** boolean
- **Commonality:** 0

Indicates blueprint nodes that are created by default by the engine.

This enables filtering out the nodes automatically generated by the engine when assessing whether there are any nodes manually created by the user within the blueprint.

For internal use only; users are not required to utilize this themselves.

## Principle:

```
static bool BlueprintEditorImpl::GraphHasUserPlacedNodes(UEdGraph const* InGraph)
{
    bool bHasUserPlacedNodes = false;

    for (UEdGraphNode const* Node : InGraph->Nodes)
    {
        if (Node == nullptr)
        {
            continue;
        }

        if (!Node->GetOutermost()->GetMetaData()->HasValue(Node,
FNodeMetadata::DefaultGraphNode))
        {
            bHasUserPlacedNodes = true;
            break;
        }
    }

    return bHasUserPlacedNodes;
}
```

# BindWidget

- **Function Description:** Specifies that a Widget attribute in a C++ class must be bound to a corresponding control with the same name in UMG.
- **Usage Location:** UPROPERTY
- **Engine Module:** Widget Property
- **Metadata Type:** bool
- **Restriction Type:** Attributes of UUserWidget subclasses

- **Related Items:** BindWidgetOptional, OptionalWidget

- **Commonality:** ★★★★☆

Specifies that the Widget attribute in the C++ class must be bound to a control with the same name in UMG.

A common programming practice is to define a UUserWidget subclass in C++, and then inherit from this C++ class in UMG. This allows for the implementation of some logic in C++ while arranging controls in UMG. Often, there is a need to reference specific controls defined in UMG using attributes in C++.

- The usual approach in C++ is to use WidgetTree->FindWidget to search for a control by name. However, if there are dozens of controls defined in the class, this process can be very tedious.
- Therefore, a more convenient method is to define control properties with the same name in C++. This will automatically establish the association, and after the UMG blueprint object is created, the engine will automatically assign the C++ Widget property to the control with the same name.
- It is important to note that BindWidget serves only as an editing and compilation hint for the UMG editor, reminding you to ensure that the names are properly associated. For the property defined in C++, remember to also create a control with the same name in UMG. When creating or modifying a control name in UMG, if you know there is a corresponding property in C++ to bind to, no error will be reported. Otherwise, an error indicating a name conflict will be prompted.
- To summarize, BindWidget serves two purposes: First, it reminds you to create a corresponding control with the same name in UMG to avoid compilation errors. Second, it prevents errors in UMG when defining a control with the same name as a C++ property.
- The recommended practice is to explicitly add BindWidget to all properties you wish to bind with the same name, rather than relying on the ambiguous default automatic binding mechanism with the same name.

## Testing Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Bindwidget :public UUserWidget
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    class UTextBlock* MyTextBlock_NotFound;

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    class UTextBlock* MyTextBlock_SameName;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (BindWidget))
    class UTextBlock* MyTextBlock_Bind;
};

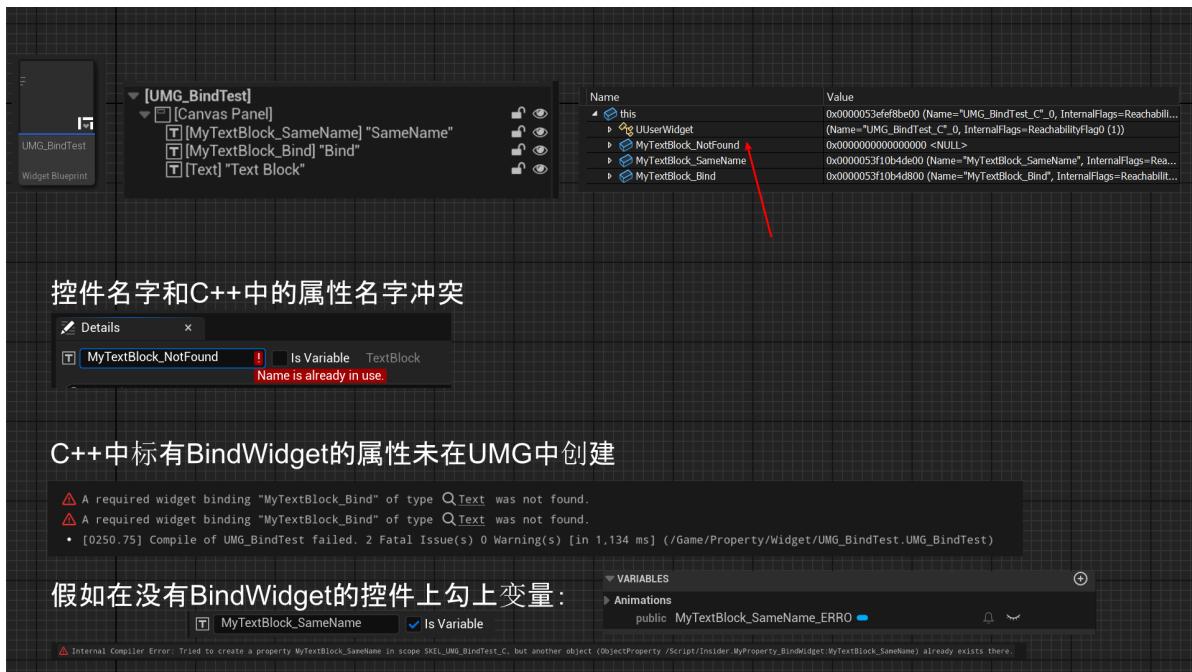
void UMyProperty_Bindwidget::RunTest()
{
    //Locating Widgets in C++: A Comprehensive Guide
    UTextBlock* bindwidget= WidgetTree->FindWidget(TEXT("MyTextBlock_Bind"));
    check(bindwidget==MyTextBlock_Bind);
```

```
}
```

## Test Outcomes:

The test involves defining a UUserWidget base class in C++ and then creating a blueprint subclass in UMG. The list of controls is shown in the figure below.

- For comparative verification, three controls are defined in both C++ and Blueprint, with some having the same name and others not. After creating the widget, these three attribute values are then inspected in C++ during debugging.
- It can be observed that both MyTextBlock\_Bind and MyTextBlock\_SameName are automatically associated with values. The logic for associating attribute values does not depend on whether BindWidget is marked. However, if the variable is checked in MyTextBlock\_SameName, a name conflict error is reported. This is because checking the variable creates an attribute in the Blueprint, which conflicts with the one in C++. When the variable is not checked, MyTextBlock\_SameName is essentially just an object under the WidgetTree. The editor can either prompt for a name conflict (first defined in C++, then in UMG) or choose not to prompt (the actual effect of BindWidget). However, if a variable named MyTextBlock\_SameName is also created in the Blueprint, the conflict is inevitable. Without BindWidget, the engine will consider these as two separate and distinct properties (if you did not explicitly write BindWidget in C++, but the engine automatically adds it, this could lead to more inexplicable errors). Only by explicitly adding BindWidget will the engine know that a C++ property already exists, eliminating the need to create another Blueprint attribute (which would not appear in the BP panel).
- MyTextBlock\_NotFound has no value, which is logical since we did not define the control in UMG. However, it is worth noting that if we attempt to define a control with this name in UMG, an error will be reported indicating that the name is already taken. This is normal, as it is similar to defining a member variable in a subclass of a C++ class; conflicts are not allowed. However, if we define MyTextBlock\_Bind, no "name occupied" error will be reported because the engine knows that a property with the same name is intended to reference the control. This is the precise meaning of BindWidget, serving only as a hint. At this point, someone might ask how MyTextBlock\_SameName was created in UMG without causing an error. The answer is to define it in UMG first, and then in C++, so no error will be reported.
- If MyTextBlock\_Bind is not defined in UMG at the end, the UMG compiler will report an error stating that the control you want to bind cannot be found, reminding you that you intended to bind a widget but did not create it.



## Operational Principle:

The function used to determine whether a property is a BindWidget is `IsBindWidgetProperty`.

The operation used to determine whether to generate an error prompt when the control is renamed or compiled is in `FinishCompilingClass`. The general logic is to determine whether the control wants to be bound based on `IsBindWidgetProperty` and then generate a prompt based on the current situation.

The logic for automatically associating values due to the same name is in `UWidgetBlueprintGeneratedClass::InitializeWidgetStatic`. The logic essentially involves traversing the controls under `WidgetTree`, searching for them by name in C++, and automatically assigning values if found.

```

void UWidgetBlueprintGeneratedClass::InitializeWidgetStatic()
{
    // Find property with the same name as the template and assign the new widget
    // to it.
    if (FObjectPropertyBase** PropPtr = ObjectPropertiesMap.Find(Widget-
>GetFName()))
    {
        FObjectPropertyBase* Prop = *PropPtr;
        check(Prop);
        Prop->SetObjectPropertyValue_InContainer(UserWidget, Widget);
        UObject* Value = Prop->GetObjectPropertyValue_InContainer(UserWidget);
        check(Value == Widget);
    }
}

void FWidgetBlueprintCompilerContext::FinishCompilingClass(UClass* Class)
{
    // Check that all BindWidget properties are present and of the appropriate
    // type
    for (TFOBJECTPROPERTYBASE<UWidget*>* WidgetProperty :
        TFieldRange<TFOBJECTPROPERTYBASE<UWidget*>>(ParentClass))
    
```

```

    {
        bool bIsOptional = false;

        if (FWidgetBlueprintEditorUtils::IsBindWidgetProperty(WidgetProperty,
bIsOptional))
            {}
    }

}

bool FWidgetBlueprintEditorUtils::IsBindWidgetProperty(const FProperty* InProperty, bool& bIsOptional)
{
    if (InProperty)
    {
        bool bIsBindWidget = InProperty->HasMetaData("BindWidget") || InProperty-
>HasMetaData("BindWidgetOptional");
        bIsOptional = InProperty->HasMetaData("BindWidgetOptional") || (
InProperty->HasMetaData("OptionalWidget") || InProperty-
>GetBoolMetaData("OptionalWidget"));
    }

    return bIsBindWidget;
}

return false;
}

```

## BindWidgetOptional

---

- **Function Description:** Specifies that a Widget attribute in a C++ class can optionally be bound to a corresponding control in UMG with the same name, or it may remain unbound.
- **Usage Location:** UPROPERTY
- **Engine Module:** Widget Property
- **Metadata Type:** bool
- **Restriction Type:** Attributes of UWidget subclasses
- **Related Items:** BindWidget
- **Commonality:** ★★★

Specifies that the Widget attribute in the C++ class can optionally be bound to a control with the same name in UMG, or it can be left unbound.

Its function is similar to BindWidget, with the distinction being:

- As the name implies, BindWidgetOptional is optional, meaning that if the control is not defined in UMG, the compilation will not result in an error. However, a warning will be issued regarding the missing control.
- 
- The difference compared to an attribute of a control without BindWidgetOptional is that the former will not produce an error when a control with the same name is defined in UMG, whereas the latter will trigger an error due to a name conflict.

There are two ways to express BindWidgetOptional:

BindWidgetOptional can be considered a combined version of BindWidget and OptionalWidget.

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Bindwidget :public UUserWidget
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    class UTextBlock* MyTextBlock_SameName;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (BindWidgetOptional))
    class UTextBlock* MyTextBlock_Optional1;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (BindWidget,
OptionalWidget))
    class UTextBlock* MyTextBlock_Optional2;
};
```

## Test Results:

- An optional widget binding "MyTextBlock\_Optional2" of type `Q_Text` is available.

## Principle:

```
bool FwidgetBlueprintEditorUtils::IsBindWidgetProperty(const FProperty* InProperty, bool& bIsOptional)
{
    if ( InProperty )
    {
        bool bIsBindWidget = InProperty->HasMetaData("Bindwidget") || InProperty->HasMetaData("BindwidgetOptional");
        bIsOptional = InProperty->HasMetaData("BindWidgetOptional") || ( InProperty->HasMetaData("OptionalWidget") || InProperty->GetBoolMetaData("OptionalWidget") );

        return bIsBindWidget;
    }

    return false;
}
```

## OptionalWidget

- Function Description:** Specifies that a Widget attribute within a C++ class can optionally be bound to a corresponding UMG control with the same name, or it may remain unbound.
- Usage Location:** UPROPERTY
- Engine Module:** Widget Property
- Metadata Type:** bool
- Restriction Type:** Attributes of UWidget subclasses
- Associated Items:** BindWidget

- **Commonality:** ★★★

Must be used in conjunction with BindWidget.

BindWidget+OptionalWidget=BindWidgetOptional

#IsBindableEvent

- **Function description:** Exposes a dynamic unicast delegate to UMG blueprints for event binding.
- **Use location:** UPROPERTY
- **Engine module:** Widget Property
- **Metadata type:** bool
- **Restriction type:** Dynamic unicast properties in UWidget subclasses
- **Commonality:** ★★★

Expose a dynamic unicast delegate to UMG blueprints to bind corresponding events.

Points to consider:

- It must be a dynamic delegate, specifically those prefixed with DYNAMIC, to enable serialization in blueprints.
- Dynamic multicast delegates (DECLARE\_DYNAMIC\_MULTICAST\_DELEGATE) are bindable in UMG by default, so there's no need to add IsBindableEvent. They often include BlueprintAssignable as well, allowing manual binding in blueprints.
- Dynamic unicast delegates (DECLARE\_DYNAMIC\_DELEGATE) are not exposed in UMG by default. However, adding IsBindableEvent allows binding in the instance's details panel.
- Why are there both multicast and unicast events in UMG? Apart from the difference in the number of recipients, the key distinction is that multicast events do not return a value. For example, compare the OnClicked multicast event in UButton with the OnMouseButtonDownEvent unicast delegate in UImage. The former is a click event, already a "conclusive" action that may be handled by multiple recipients, hence designed as multicast. The latter is a mouse press event, where the logic to determine whether the event should continue routing depends on the return value FEventReply, thus requiring unicast.

## Source Code Example:

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FOnButtonClickedEvent);
class UButton : public UContentWidget
{
    UPROPERTY(BlueprintAssignable, Category="Button|Event")
    FOnButtonClickedEvent onclicked;
}

DECLARE_DYNAMIC_DELEGATE_RetVal_TwoParams(FEventReply, FOnPointerEvent,
FGeometry, MyGeometry, const FPointerEvent&, MouseEvent);
class UImage : public UWidget
{
    UPROPERTY(EditAnywhere, Category=Events, meta=( IsBindableEvent="True" ))
    FOnPointerEvent OnMouseButtonDownEvent;
}
```

## Test Code:

```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Bindwidget :public UUserWidget
{
public:
    DECLARE_DYNAMIC_MULTICAST_DELEGATE(FOnMyClickedMulticastDelegate);

    UPROPERTY(EditAnywhere, BlueprintAssignable, Category = MyEvent)
    FOnMyClickedMulticastDelegate MyClickedMulticastDelegate;

public:

DECLARE_DYNAMIC_DELEGATE_RetVal_OneParam(FString, FOnMyClickedDelegate, int32, MyValue);

    UPROPERTY(EditAnywhere, Category = MyEvent)
    FOnMyClickedDelegate MyClickedDelegate_Default;

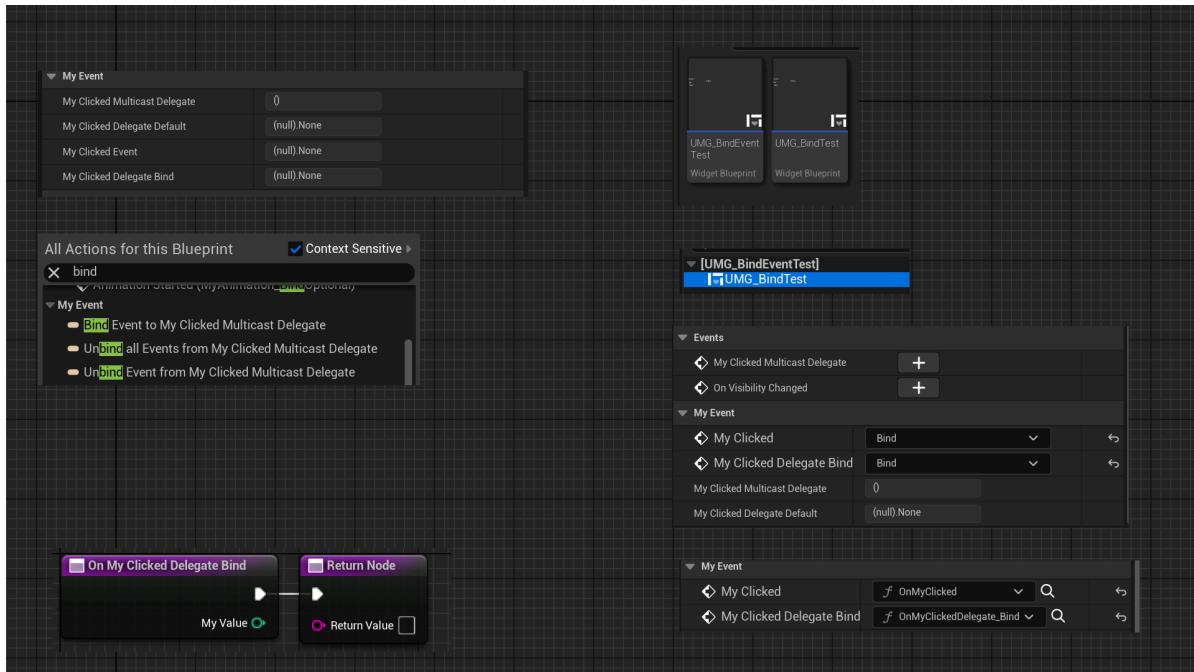
    UPROPERTY(EditAnywhere, Category = MyEvent)
    FOnMyClickedDelegate MyClickedEvent;

    UPROPERTY(EditAnywhere, Category = MyEvent, meta = (IsBindableEvent =
"True"))
    FOnMyClickedDelegate MyClickedDelegate_Bind;
}
```

## Test Results:

The steps involve creating another UMG outside UMG\_BindTest, making UMG\_BindTest a child widget, and observing the event bindings on its instance, as shown in the figure on the right.

- It can be observed that dynamic multicast delegates automatically appear with a customizable bind button, such as MyClickedMulticastDelegate.
- When a dynamic multicast delegate is marked with BlueprintAssignable (not applicable to unicast delegates), it can be bound in blueprints, as shown in the lower left image.
- With IsBindableEvent added to MyClickedDelegate\_Bind, a dropdown button for binding appears. After binding, the function name is displayed and can be cleared.
- Without IsBindableEvent, MyClickedDelegate\_Default does not appear in the bindable buttons, and you must bind it manually in C++.
- MyClickedEvent without IsBindableEvent also shows a bindable button because its name ends with Event, which can be considered a current workaround. The source code comments suggest this will be removed in the future.
- Although I've added EditAnywhere to these delegates, it's known that they are not actually editable.



## Principle:

For the widget's details panel, the engine defines various customizations, including FBlueprintWidgetCustomization. The code for the binding part is as follows.

The code is straightforward: dynamic multicast delegates are bound by default, while dynamic unicast delegates are bound if IsBindableEvent is added or the name ends with Event.

```

PropertyView->RegisterInstancedCustomPropertyParams(UWidget::StaticClass(),
FOnGetDetailCustomizationInstance::CreateStatic(&FBlueprintWidgetCustomization::MakeInstance,
BlueprintEditorRef, BlueprintEditorRef->GetBlueprintObj()));

void
FBlueprintWidgetCustomization::PerformBindingCustomization(IDetailLayoutBuilder&
DetailLayout, const TArrayView<UWidget*> Widgets)
{
    static const FName IsBindableEventName(TEXT("IsBindableEvent"));

    bCreateMulticastEventCustomizationErrorAdded = false;
    if (Widgets.Num() == 1)
    {
        UWidget* Widget = Widgets[0];
        UClass* PropertyClass = Widget->GetClass();

        for (TFieldIterator<FProperty> PropertyIt(PropertyClass,
EFieldIteratorFlags::IncludeSuper); PropertyIt; ++PropertyIt)
        {
            FProperty* Property = *PropertyIt;

            if (FDelegateProperty* DelegateProperty =
CastField<FDelegateProperty>(*PropertyIt))
            {
                //TODO Remove the code to use ones that end with "Event". Prefer
                metadata flag.
                if (DelegateProperty->HasMetaData(IsBindableEventName) ||
                    DelegateProperty->GetName().EndsWith(TEXT("Event")))
            }
        }
    }
}

```

```

        {
            CreateEventCustomization(DetailLayout, DelegateProperty,
        widget);
    }
}
else if ( FMulticastDelegateProperty* MulticastDelegateProperty =
CastField<FMulticastDelegateProperty>(Property) )
{
    CreateMulticastEventCustomization(DetailLayout, widget-
>GetFName(), PropertyClass, MulticastDelegateProperty);
}
}
}
}
}

```

## EntryInterface

---

- **Function Description:** Defines the interface that optional classes for the EntryWidgetClass attribute must implement, used in both DynamicEntryBox and ListView widgets.
- **Usage Locations:** UCLASS, UPROPERTY
- **Engine Module:** Widget Property
- **Metadata Type:** string="abc"
- **Restriction Type:** Subclass of UWidget
- **Associated Items:** EntryClass
- **Commonality:** ★★★

Defines the interface that optional classes for the EntryWidgetClass attribute must implement, used in both DynamicEntryBox and ListView widgets.

For example, in ListView, the term "Entry" refers to the sub-controls displayed in the list, while "Item" refers to the data elements behind the list. For instance, a list backpack may have 1000 items (Items), but only 10 controls (Entry) can be displayed on the interface simultaneously.

Therefore, EntryInterface and EntryClass, as their names imply, refer to the interface to be implemented on EntryWidget and its base class.

For demonstration of usage, the following examples use ListView, and the same applies to DynamicBox.

```

//1. ListView is an attribute of another widget, so metadata extraction and
validation will occur on the property.
//The property must be a BindWidget to automatically bind to the control in UMG
and to be enumerated as a C++ property.
class UMyUserWidget : public UUserWidget
{
    UPROPERTY(BindWidget, meta = (EntryClass =
MyListEntryWidget,EntryInterface = MyUserListEntry ))
        UListViewBase* MyListView;
}

//2. If the metadata is not found on the property, it will also attempt to search
directly on the widget class

```

```

UCLASS(meta = (EntryClass = MyListEntryWidget, EntryInterface =
"/Script/UMG.UserObjectListEntry"))
class UMyListView : public UListViewBase, public ITypedUMGListView<UObject*>
{};

//3. During ClassPicker selection, EntryClass specifies the parent class, and
EntryInterface specifies the interface that the class must implement

```

## Usage in Source Code:

```

UCLASS(Abstract, NotBlueprintable, hidedropdown, meta = (EntryInterface =
UserListEntry), MinimalAPI)
class UListViewBase : public UWidget
{
    UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = ListEntries, meta =
(DesignerRebuild, AllowPrivateAccess = true, MustImplement =
"/Script/UMG.UserListEntry"))
    TSubclassOf<UUserWidget> EntrywidgetClass;
}

UCLASS(meta = (EntryInterface = "/Script/UMG.UserObjectListEntry"), MinimalAPI)
class UListView : public UListViewBase, public ITypedUMGListView<UObject*>
{};

//The UserObjectListEntry interface inherits from UserListEntry, and all Entry
widgets must inherit from this interface.
UINTERFACE(MinimalAPI)
class UUserObjectListEntry : public UUserListEntry
{};

SNew(SClassPropertyEntryBox)
.AllowNone(false)
.IsBlueprintBaseOnly(true)
.RequiredInterface(RequiredEntryInterface)
.MetaClass(EntryBaseClass ? EntryBaseClass : UUserWidget::StaticClass())
.SelectedClass(this, &FDynamicEntryWidgetDetailsBase::GetSelectedEntryClass)
.OnSetClass(this, &FDynamicEntryWidgetDetailsBase::HandleNewEntryClassSelected)

```

In FDynamicEntryWidgetDetailsBase, the EntryInterface and EntryClass are determined, and then the ClassPicker options for the property details panel in SClassPropertyEntryBox are restricted. FDynamicEntryWidgetDetailsBase is the base class for FListViewBaseDetails and FDynamicEntryBoxDetails, thus customizing the property details panels for ListView and DynamicBox.

## Test Code:

```

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyEntryWidget :public UUserWidget, public IUserObjectListEntry
{
    GENERATED_BODY()
public:
    virtual void NativeOnListItemObjectSet(UObject* ListItemObject) override;
public:

```

```

UPROPERTY(meta = (BindWidget))
class UTextBlock* ValueTextBlock;
};

//////////////////////////////



UINTERFACE(MinimalAPI)
class UMyCustomListEntry : public UUserObjectListEntry
{
    GENERATED_UINTERFACE_BODY()
};

class IMyCustomListEntry : public IUserObjectListEntry
{
    GENERATED_IINTERFACE_BODY()
};

UCLASS(Blueprintable, BlueprintType)
class INSIDER_API UMyCustomEntryWidget :public UUserWidget, public
IMyCustomListEntry
{
    GENERATED_BODY()

public:
    virtual void NativeOnListItemObjectSet(UObject* ListItemObject) override;
public:
    UPROPERTY(meta = (BindWidget))
    class UTextBlock* ValueTextBlock;
};

//////////////////////////////



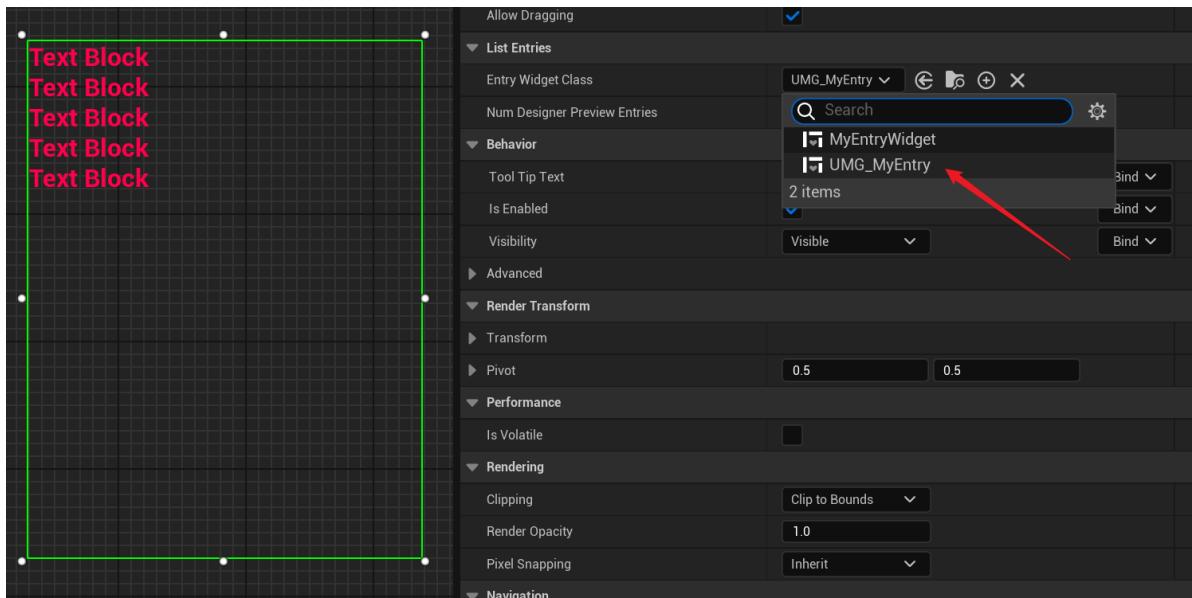
UCLASS()
class INSIDER_API UMyListContainerWidget :public UUserWidget
{
    GENERATED_BODY()

public:
    UPROPERTY(BlueprintReadWrite, EditAnywhere, meta = (BindWidget, EntryClass =
MyCustomEntryWidget, EntryInterface = MyCustomListEntry))
    class UListView* MyListView;
};

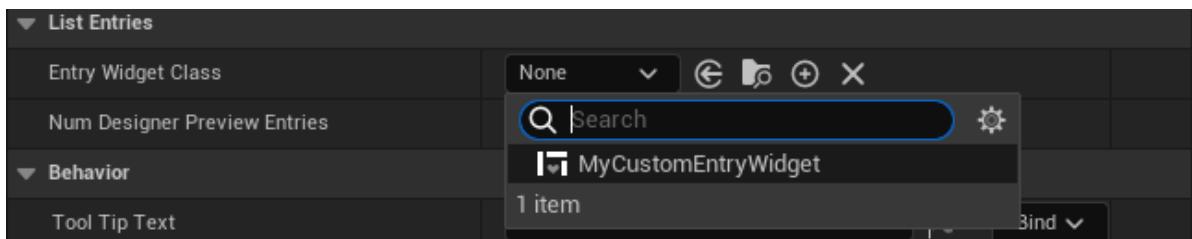
```

## Blueprint Effect:

If no EntryClass or EntryInterface is specified on ListView, then the blueprint-created UMG\_MyEntry (inherited from C++'s UMyEntryWidget) can be selected as the class for the ListView's EntryWidgetClass property.



If, as shown in the code above, a new interface `MyCustomListEntry` is created, along with a new `MyCustomEntryWidget`, and `EntryClass` or `EntryInterface` is specified (either together or individually) on the `MyListView` property, then the selectable classes for the `ListView`'s `EntryWidgetClass` property are restricted.



Another approach is to customize a `ListView` by inheriting from `ListViewBase`, and then directly specifying `EntryClass` or `EntryInterface` on this subclass, achieving the same effect as shown in the figure above.

```
UCLASS(meta = (EntryClass = MyCustomEntryWidget, EntryInterface =
MyCustomListEntry))
class UMyListView : public UListViewBase, public ITypedUMGListView<UObject*>
{}
```

## EntryClass

- Function description:** Defines the base class that the optional classes for the `EntryWidgetClass` attribute must inherit from, utilized in both `DynamicEntryBox` and `ListView` widgets.
- Usage location:** `UCLASS`, `UPROPERTY`
- Metadata type:** `string="abc"`
- Restriction type:** Subclass of `UWidget`
- Related items:** `EntryInterface`
- Frequency:** ★★★

# BindWidgetAnim

- **Function description:** Specifies that the UWidgetAnimation attribute in the C++ class must be bound to a specific animation within UMG
- **Use location:** UPROPERTY
- **Engine module:** Widget Property
- **Metadata type:** bool
- **Restriction type:** UWidgetAnimation attribute within a UWidget subclass
- **Related items:** BindWidgetAnimOptional
- **Commonly used:** ★★★★☆

Specifies that the UWidgetAnimation attribute in the C++ class must be bound to a specific animation within UMG.

Functions similarly to BindWidget, both used to bind C++ properties to controls or animations in BP, but with distinct differences:

- UWidgetAnimation is different from Widget. Widget's properties and controls can be automatically bound as long as they have the same name. However, UWidgetAnimation does not allow the same name without adding BindWidgetAnim, otherwise a name conflict error will be reported. This is because the Widget created in UMG does not create BP variables by default. The sub-control is just an object under WidgetTree, but the animation will create BP variables by default. Therefore, even if the animation is defined first in UMG, and then the property of the same name is defined in C++, it will not pass compilation.
- The UWidgetAnimation attribute must be Transient, or an error will occur. This is likely because UWidgetAnimation is naturally serialized as a sub-object in BP and does not need to be accessed during C++ serialization, hence the enforced Transient to prevent accidental serialization. Additionally, since UWidgetAnimation is solely for presentation, it is also automatically marked with CPF\_RepSkip to skip network replication.

## Test Code:

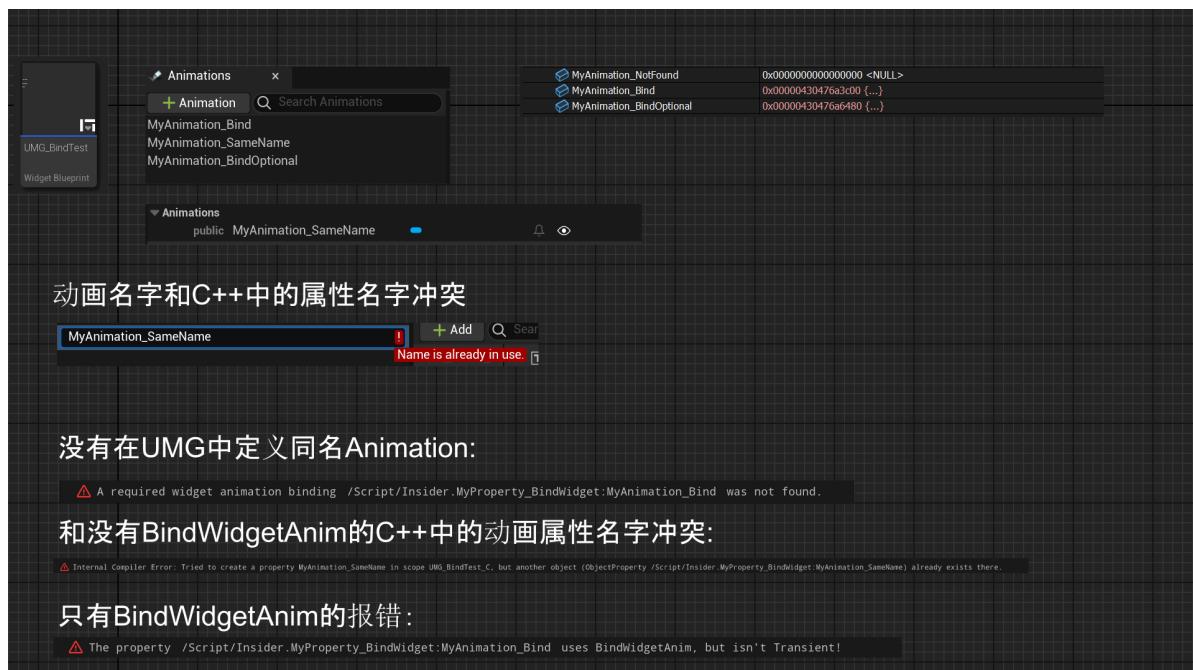
```
UCLASS(BlueprintType)
class INSIDER_API UMyProperty_Bindwidget :public UUserWidget
{
    GENERATED_BODY()
    UMyProperty_Bindwidget(const FObjectInitializer& ObjectInitializer);

public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
        class UWidgetAnimation* MyAnimation_NotFound;
    //UPROPERTY(EditAnywhere, BlueprintReadWrite)
    //class UWidgetAnimation* MyAnimation_SameName;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient, meta =
        (BindWidgetAnim))
        class UWidgetAnimation* MyAnimation_Bind;
    UPROPERTY(EditAnywhere, BlueprintReadWrite, Transient, meta =
        (BindWidgetAnimOptional))
        class UWidgetAnimation* MyAnimation_BindOptional;
};
```

# Test Results:

The testing process is similar to that of BindWidget, involving the definition of different property types and animation objects in both C++ and UMG. It can be observed in the actual object values in VS that:

- MyAnimation\_Bind and MyAnimation\_BindOptional are both automatically bound to the correct animation objects.
- MyAnimation\_SameName, without BindWidgetAnim, must be commented out to prevent a name conflict with the MyAnimation\_SameName in UMG.
- Moreover, it should be noted that one cannot define an animation in UMG first, as one might with a Widget, and then define a property with the same name in C++. This is because WidgetAnimation inherently creates BP variables, which is a crucial distinction.



## Principle:

The logic is roughly similar to that of BindWidget, involving checks for whether a property has BindWidgetAnim set. Subsequent actions are taken during compilation and renaming accordingly.

The logic for setting PropertyFlags for animation variables can be found in CreateClassVariablesFromBlueprint, where four properties are added, explicitly indicating that the property should not be serialized.

The logic for automatically binding and assigning values to UWidgetAnimation\* attributes is located in BindAnimationsStatic, and it is straightforward to understand.

```
bool FWidgetBlueprintEditorUtils::IsBindWidgetAnimProperty(const FProperty* InProperty, bool& bIsOptional)
{
    if (InProperty)
    {
        bool bIsBindWidgetAnim = InProperty->HasMetaData("BindWidgetAnim") ||
        InProperty->HasMetaData("BindWidgetAnimOptional");
        bIsOptional = InProperty->HasMetaData("BindWidgetAnimOptional");
```

```

        return bIsBindWidgetAnim;
    }

    return false;
}

void FWidgetBlueprintCompilerContext::CreateClassVariablesFromBlueprint()
{
    for (UWidgetAnimation* Animation : WidgetBP->Animations)
    {
        FEdGraphPinType WidgetPinType(UEdGraphSchema_K2::PC_Object, NAME_None,
Animation->GetClass(), EPinContainerType::None, true, FEdGraphTerminalType());
        FProperty* AnimationProperty = CreateVariable(Animation->GetFName(),
WidgetPinType);

        if (AnimationProperty != nullptr)
        {
            const FString DisplayName = Animation->GetDisplayName().ToString();
            AnimationProperty->SetMetaData(TEXT("DisplayName"), *DisplayName);

            AnimationProperty->SetMetaData(TEXT("Category"), TEXT("Animations"));

            AnimationProperty->SetPropertyParams(CPF_Transient);
            AnimationProperty->SetPropertyParams(CPF_BlueprintVisible);
            AnimationProperty->SetPropertyParams(CPF_BlueprintReadOnly);
            AnimationProperty->SetPropertyParams(CPF_Repskip);

            WidgetAnimToMemberVariableMap.Add(Animation, AnimationProperty);
        }
    }
}

void FWidgetBlueprintCompilerContext::FinishCompilingClass(UClass* Class)
{
    if (!WidgetAnimProperty->HasAnyPropertyParams(CPF_Transient))
    {
        const FText BindWidgetAnimTransientError =
LOCTEXT("BindWidgetAnimTransient", "The property @@ uses BindWidgetAnim, but
isn't Transient!");
        MessageLog.Error(*BindWidgetAnimTransientError.ToString(),
WidgetAnimProperty);
    }
}

void UWidgetBlueprintGeneratedClass::BindAnimationsStatic(UUserWidget* Instance,
const TArrayView<UWidgetAnimation*> InAnimations, const TMap< FName,
FObjectPropertyBase*>& InPropertyMap)
{
    // Note: It's not safe to assume here that the UserWidget class type is a
UWidgetBlueprintGeneratedClass!
    // - @see InitializeWidgetStatic()

    for (UWidgetAnimation* Animation : InAnimations)
    {
        if (Animation->GetMovieScene())
        {

```

```

        // Find property with the same name as the animation and assign the
        animation to it.
        if (FObjectPropertyBase* const* PropPtr =
InPropertyParams.Find(Animation->GetMovieScene()->GetFName()))
{
    check(*PropPtr);
    (*PropPtr)->SetObjectPropertyValue_InContainer(Instance,
Animation);
}
}
}
}

```

## BindWidgetAnimOptional

- **Function Description:** Specifies that the UWidgetAnimation attribute in a C++ class may optionally be bound to an animation within UMG, or may remain unbound.
- **Usage Location:** UPROPERTY
- **Engine Module:** Widget Property
- **Metadata Type:** bool
- **Restriction Type:** UWidgetAnimation attribute in a subclass of UWidget
- **Associated Items:** BindWidgetAnim
- **Commonality:** ★★★

Like BindWidgetOptional, it also provides a similar function, where there will be a prompt in the compilation output if it is not bound, rather than enforcing an error as with BindWidget.

- An optional widget animation binding /Script/Insider.MyProperty\_BindWidget:MyAnimation\_BindOptional2 is available.

It is also inherently understood that it does not automatically bind by default without the BindWidget, as is the case with Widget.

Therefore, the usage requires either adding BindWidgetAnim or BindWidgetAnimOptional.