Table of Contents

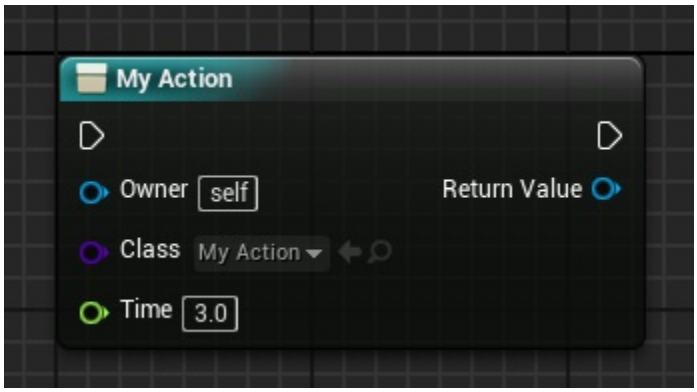
Actions Extension

Actions Extension is a plugin that adds **blueprintable async tasks** called actions. It can be used for a lot of things but some examples are AI or API Rest.

If you like our plugins, consider becoming a Patron. It will go a long way in helping me create more awesome tech!

BECOME A PATRON

What is an Action?



Actions are quite similar to async task nodes (*like Delays or AIMoves*) in their concept, but have some extra features that make them widely useful.

An Action is a blueprint (or c++ class) that executes inside another object to encapsulate logic.

Where can I use an Action?

Any object with world context can have an action and its usage goes from AI behaviors to API Rest calls.

We have tried both options extensively and the results are a lot more simple than normal code. You get better parallel programming, quality of code. At the end it just becomes easier to deal with complex logic.

This system is also heavily focused on the usage of Actions inside Actions, creating a **tree of dependencies**. This is specially useful for **AI**.

Quick Start

Check Quick Start to see how to setup and configure the plugin.

Installation

Manually

This are the general steps for installing the plugin into your project:

1. Download the last release from here
Make sure you download the same version that your project uses

2. Extract the folder “ActionsExtension” into the **Plugins folder** of your existing project (e.g "MyProject/Plugins")

2. Done! You can now open the project

From Marketplace

Install from the launcher: AVAILABLE HERE

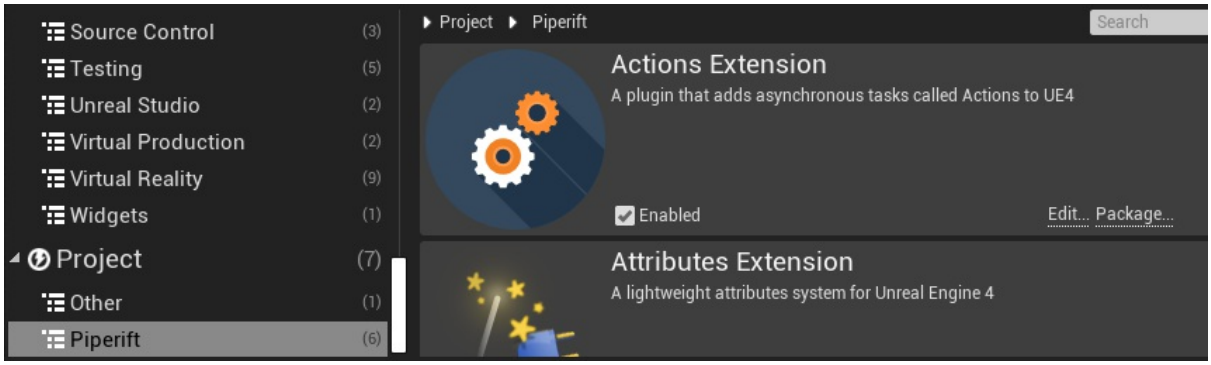
Quick Start

Quick Start will show the basic steps to follow to setup the plugin and start using it at a base level.

Setting Up the Project

We can start by creating an empty project (How to create UE4 projects) or instead using your own. Then installing the plugin from marketplace or inside Plugins folder (See Installation).

If everything went right, we should see the plugin enabled under *Edit->Plugins->Piperift*

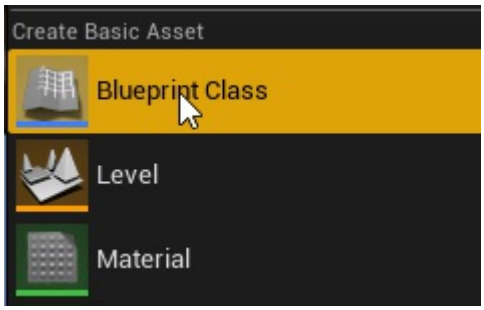


Actions Extension doesn't require anything else to work.

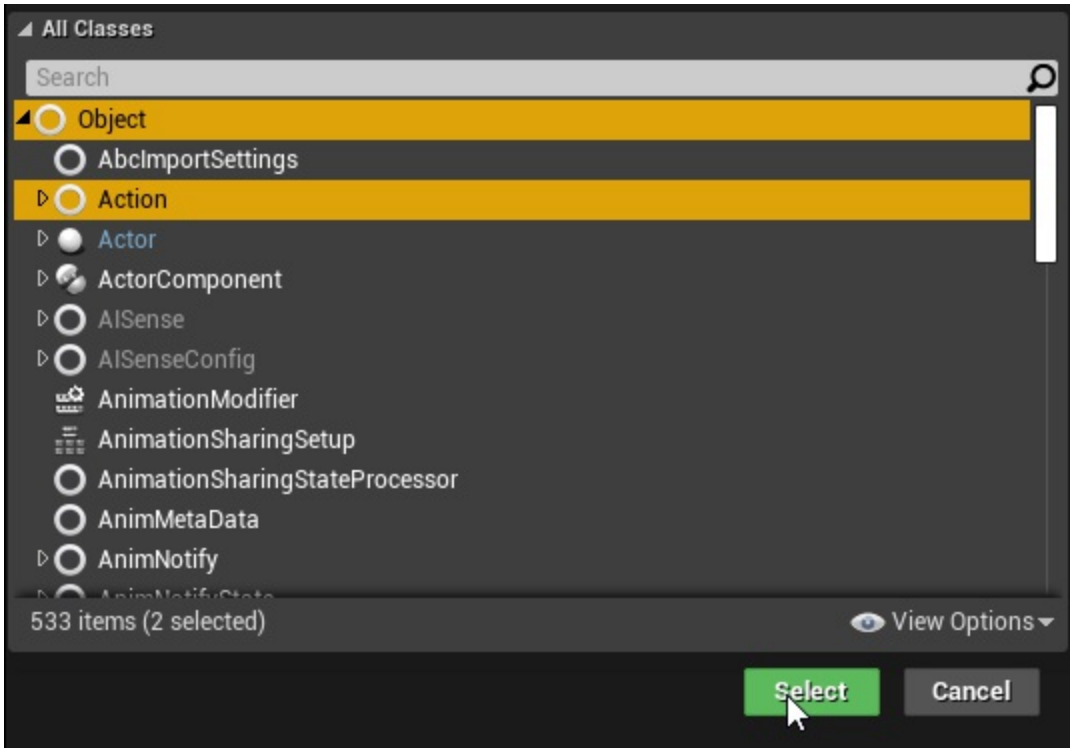
Creating The action

Lets start by creating a very simple action.

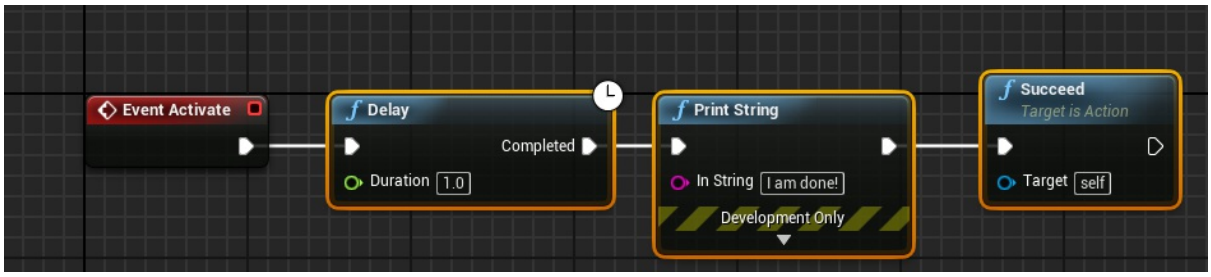
First we go to the **content browser, right click, Blueprint Class**



Then we **select Action class** (or any other child class of Action)



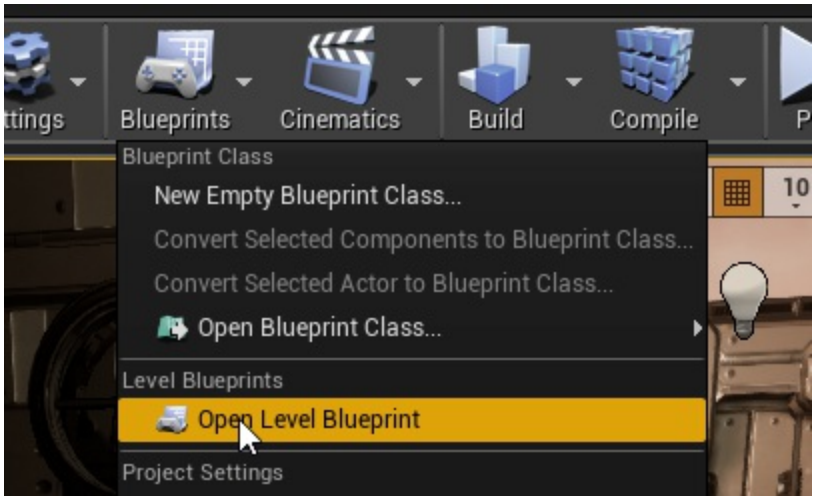
Then we **open the blueprint** we created and add the following functions on Activate. This will be called when the action starts its execution, then wait 1 second, and finish.



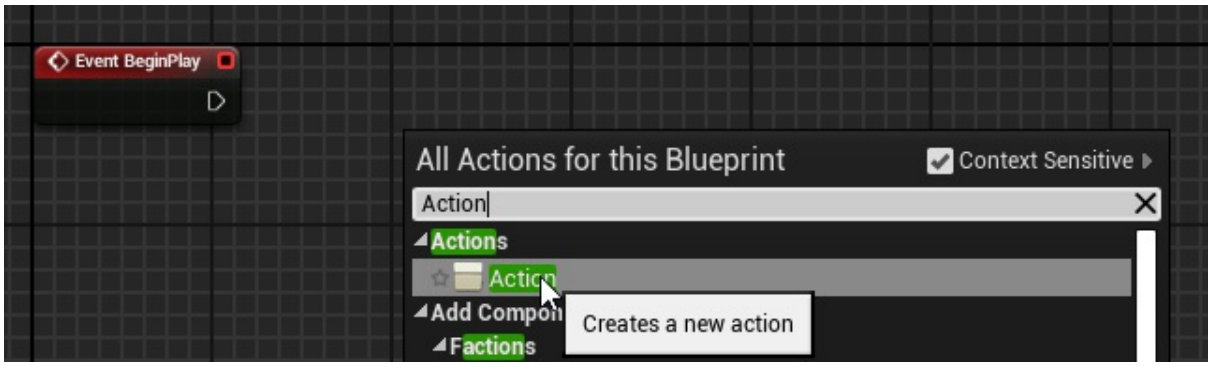
> Make sure your actions call **Succeed** or **Fail**. Otherwise the action will run until its owner is destroyed or the game closes.

Calling The action

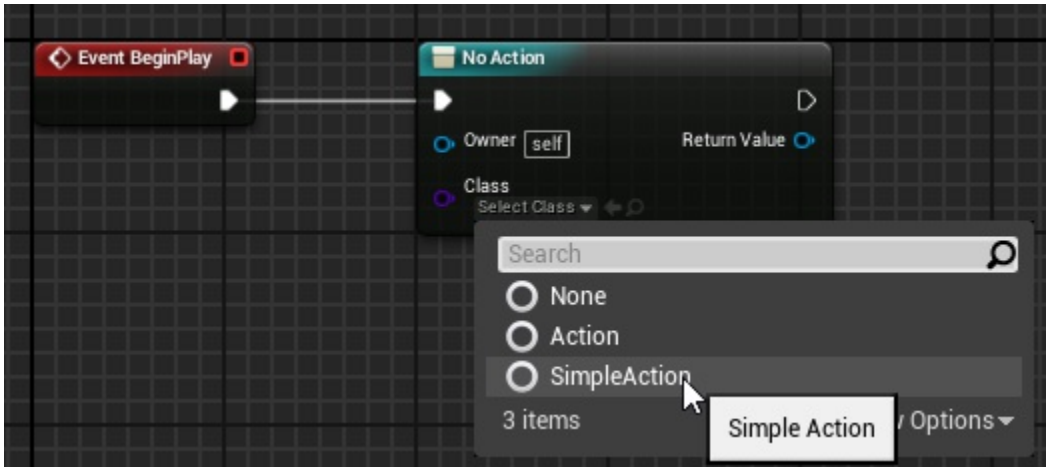
Now that we have our action ready, we have to execute it. For that we will go to our level blueprint.



Then from our BeginPlay we **add the node "Action"**

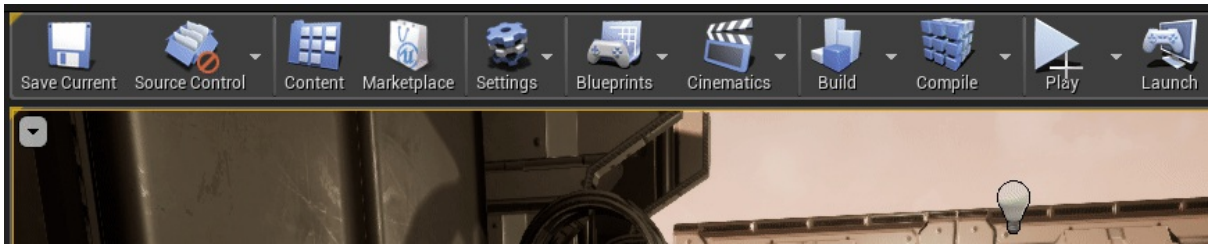


Finally, we assign the action we created previously to the class pin



The Result

After all the previous steps we will see that the message prints exactly 1 second after we hit play.

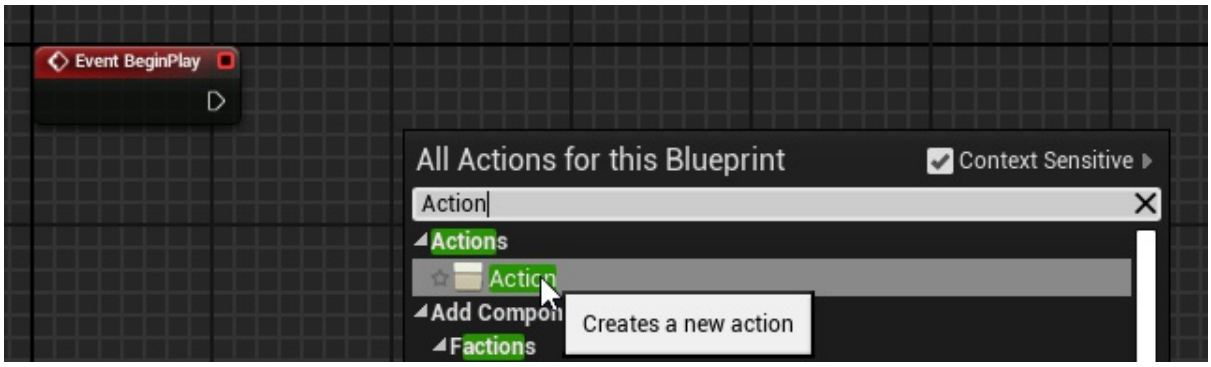

Save Current
Source Control
Content
Marketplace
Settings
Blueprints
Cinematics
Build
Compile
Play
Launch

Usage in Blueprints

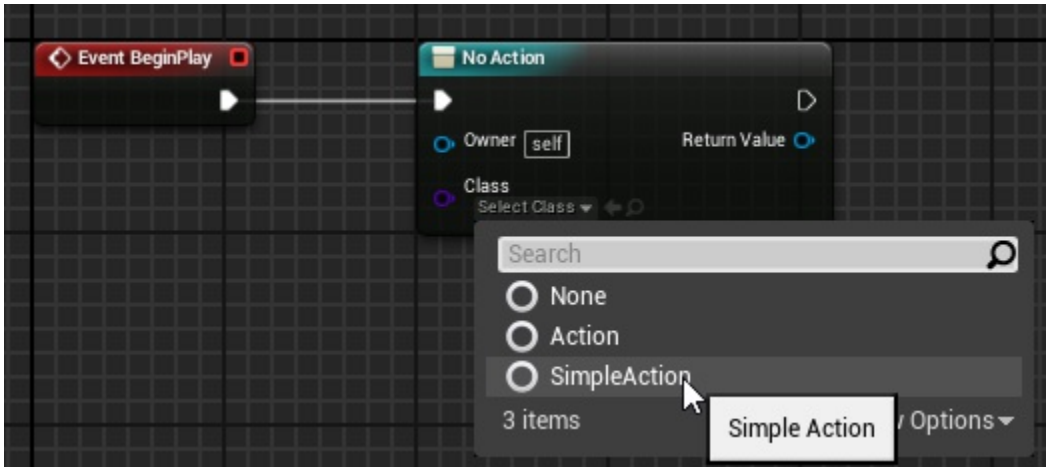
Call an Action

To execute an action we have to use the Action node.

You can find it by right clicking on a graph and searching for **"Action"**:



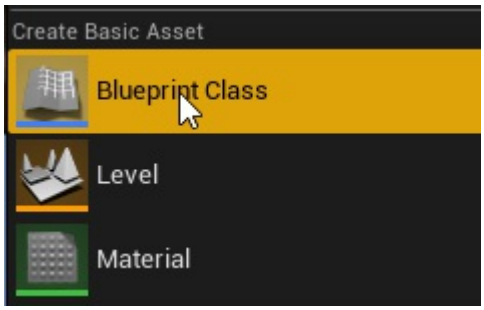
Then we have to assign the action class we want to use.



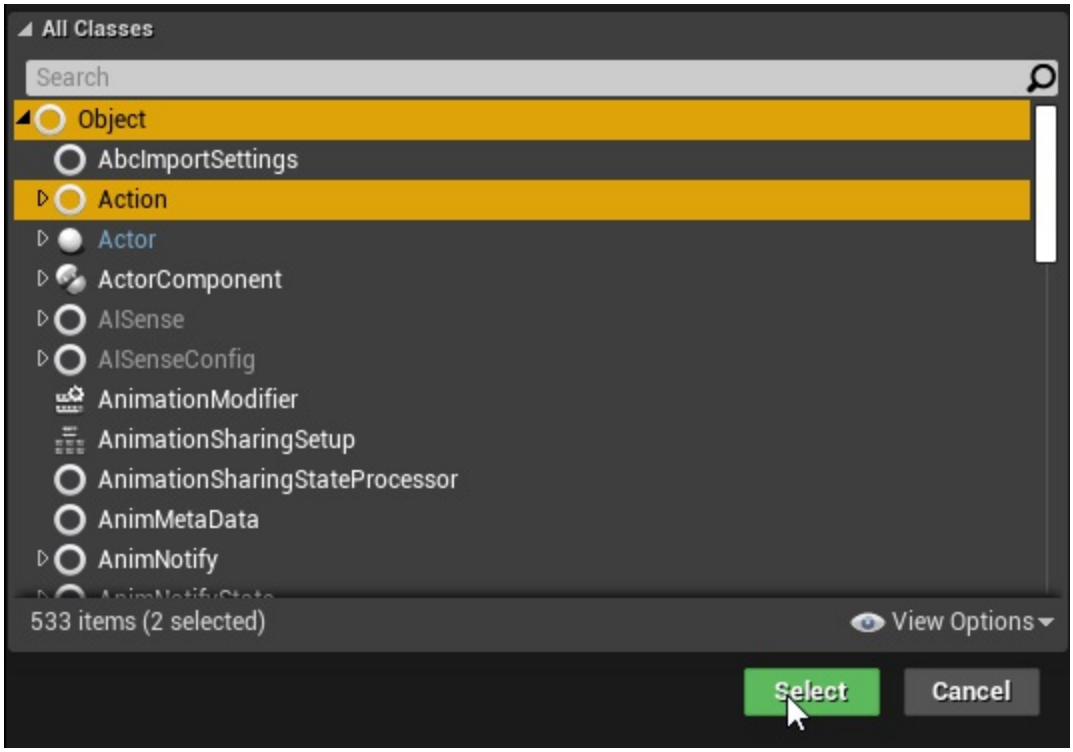
After this, **all its variables and delegates will show up** for you to use and the action is ready to execute.

Create an Action

To create an action, we have to go to
content browser -> right click -> Blueprint Class



Then we **select "Action" class** or one of Action's children



Then we **open the blueprint** we created.

All actions have 3 main events:

- **Activate**: When it gets created
- **Tick**: When it ticks, if it is enabled. TickRate is applied.
- **Finish**: When the action finished and why (*Success, Fail or Cancel*)

Make sure your actions call **Succeed** or **Fail**. Otherwise the action will run until its owner is destroyed or the game closes.

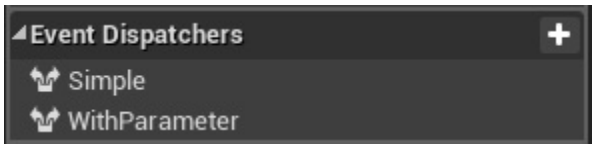
Exposing Delegates

Actions can expose Event Dispatchers to action calls.

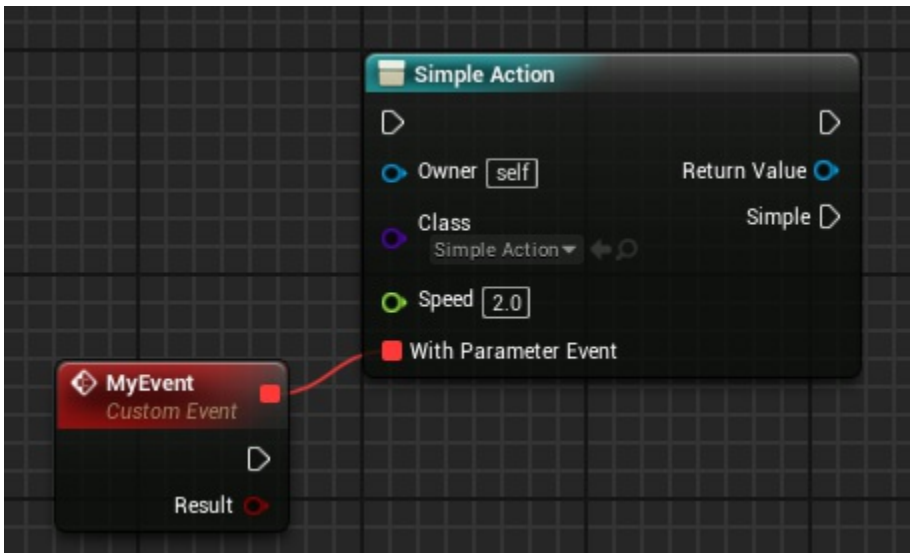
- If the event dispatcher has any parameter, it will show up as an event pin
- If it has no parameters, an execution pin will show up

So as an example, if we have the following two events:

- **SimpleEvent** with no parameters
- **EventWithParameter** with a boolean



Our Action will look like this:



If an event doesn't appear on an action node, **right click -> Refresh Node** to refresh it

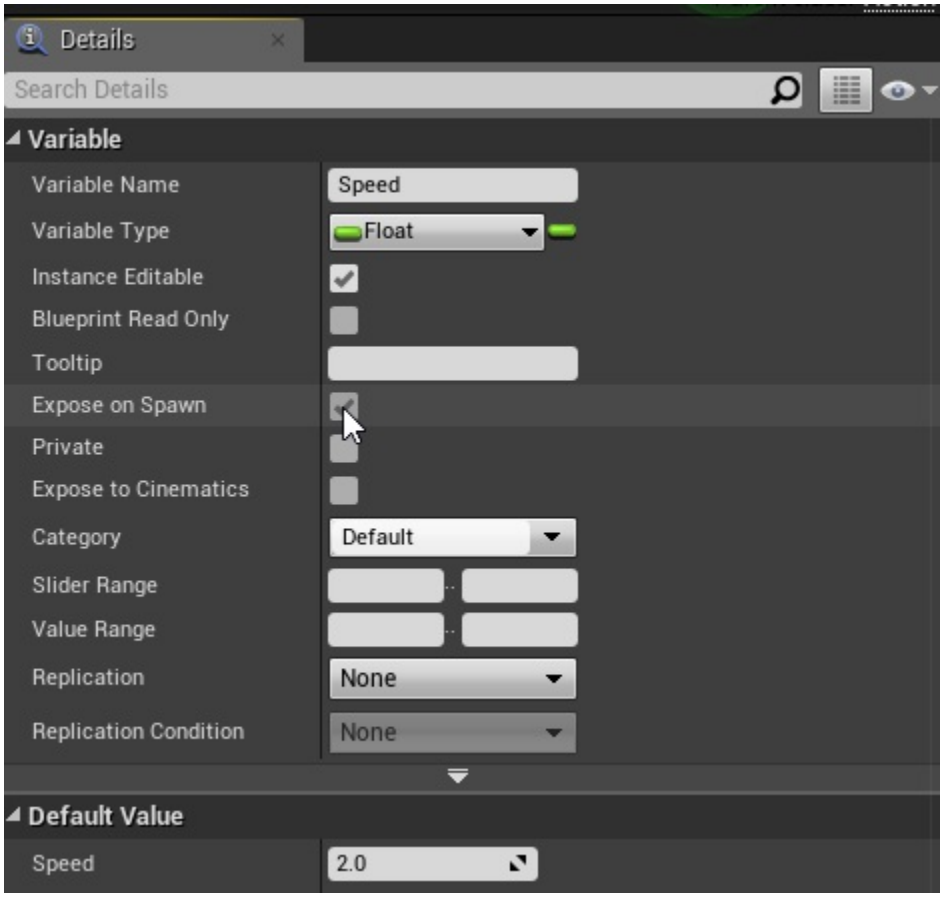
Exposing Variables

Most of the times, when we have an action, we want to feed it with variables to customize its behavior, and as you will see, it's very easy to do so.

First open your Action blueprint and add a new variable of any type

Mark the variable as Editable

Finally, check "**Expose on Spawn**" as true



This variable will now show up on all action nodes.

> If a variable doesn't appear on an action node, **right click -> Refresh Node** to refresh it

Usage in C++

All functions are well commented on code. Feel free to give them a look for detailed information.

Call an Action

There are multiple ways of creating an action, here are some:

```
// Templated version
UAction* MyActionA = UAction::Create<UMyAction>(this, true);

// Class version
UAction* MyActionB = UAction::Create(this, UMyAction::StaticClass(), true);
```

By default actions will **auto activate**, but if we want to do some kind of setup, we can leave activation for later:

```
UMyAction* MyActionC = UAction::Create<UMyAction>(this, false);
MyActionC->bMyBool = true;
MyActionC->Activate();
```

Note that Actions require an owner with access to world. Otherwise, activation will fail.

Create an Action

You can simply create a child class of UAction (or any other action class).

```
UCLASS()
class UMyAction : public UAction
{
    GENERATED_BODY()

public:

    bool bMyBool = false;

    // Optional
    virtual void OnActivation() override;

    // Optional
    virtual void OnFinish(const EActionState Reason) override;

    // Optional
    virtual void Tick(float DeltaTime) override;
};
```

Make sure your actions call **Succeed** or **Fail**. Otherwise the action will run until its owner is destroyed or the game closes.

Exposing Delegates

Actions can expose Multicast delegates to action calls.

- If the delegate has any parameter, it will show up as an event pin
- If it has no parameters, an execution pin will show up

So as an example, if we have the following two events:

- **NoParameter** with no parameters
- **WithParameter** with a boolean

We first have to declare our delegates

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FNoParamDelegate);
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FWithParamDelegate, bool, bResult);
```

Then we add their variables as BlueprintAssignable, and that's it.

```
UPROPERTY(BlueprintAssignable)
FNoParamDelegate NoParameter;

UPROPERTY(BlueprintAssignable)
FWithParamDelegate WithParameter;
```

Ignore those errors, it's just Intellisense doing its usual magic. Everything is fine, I promise

Exposing Variables

Most of the times, when we have an action, we want to feed it with variables to customize its behavior, and as you will see, it's very easy to do so.

Member variables marked as **BlueprintReadWrite** and with metadata **ExposeOnSpawn="true"** will show on Blueprint Action nodes.

```
UPROPERTY(BlueprintReadWrite, meta = (ExposeOnSpawn = "true"), EditAnywhere,
bool bMyBool = false;
```

With C++ in particular, any public variable can be edited as usual before Activation.

Debugging

Gameplay Debugger

Actions can be debugged in-game with gameplay debugger.

Actions will display for the focused actor, its controller and the current player controller (the first local player). It is displayed as a tree of sub-actions.

Blueprint Debugger

Like any other normal Blueprint, blueprint debugging is supported such as instance selection or visualization of graphs.