FlowCanvas Offline Documentation

http://flowcanvas.paradoxnotion.com

1. Getting Started 3
- 1.1. The FlowScript Controller 3
- 1.2. Understanding the Flow 4
- 1.3. Adding nodes 5
- 1.4. Connecting Nodes 7
- 1.5. The "Self" Parameter 8
- 1.6. Visual Debugging 9

2. Working with Types 10
- 2.1. Generic Type Nodes 10
- 2.2. Connecting Assignable Types 11

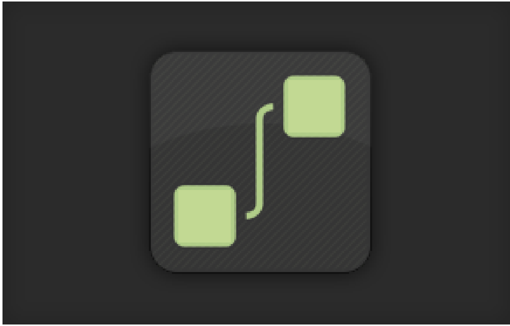
3. Node Categories 12
- 3.1. Events 12
- 3.2. Flow Controllers 12
- 3.3. Actions 13
- 3.4. Functions 13
- 3.5. Constructors & Extractors 14

4. Variables 15
- 4.1. Blackboard Variables 16
- 4.2. Using Variables in FlowScript 16
- 4.3. Property Binding 19
- 4.4. Network Sync 19

5. Macros 20
- 5.1. Creating Macros 20
- 5.2. Using Macros 22

6. Creating Custom Nodes 23
- 6.1. Creating Simplex Nodes 24
- 6.2. Creating Full Nodes 25
- 6.3. Creating Event Nodes 26

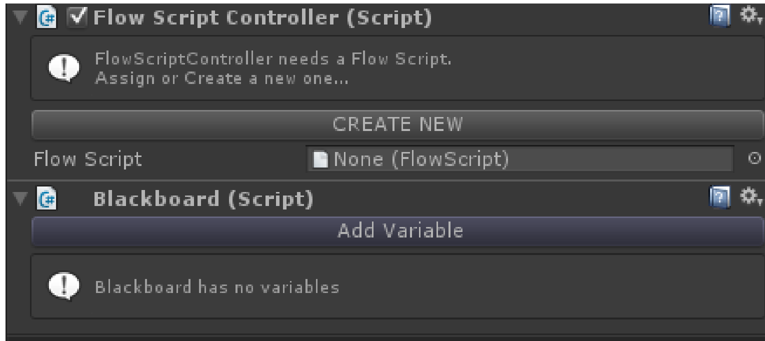
1. Getting Started



Hello and welcome to the FlowCanvas documentation!
While reading, don't forget to leave your feedback with the buttons bellow!

1.1. The FlowScript Controller

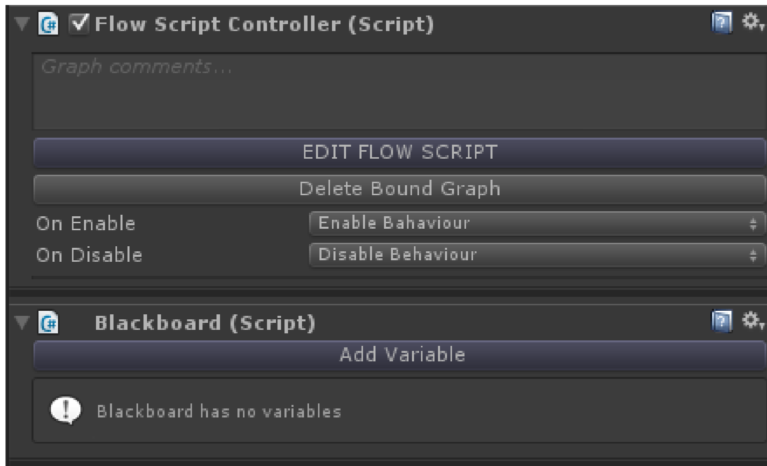
The most important component of FlowCanvas, is the **FlowScriptController** component. As the name suggests, this component is responsible for controlling a flowscript and you have to add it on a game object that you wish to be controlled by a flowscript. Although this is not really mandatory, it's the easiest and the most suggested way to get started quickly. Once you have added the FlowScriptController component on a game object, you will notice that another component named "Blackboard" is also added. This is the place you will be creating variables, but more on these variables will be explained later on.



The FlowScriptController needs to be assigned a FlowScript, or alternatively, you can click the "Create New" button to create and assign one now. An option will pop up, asking you if you want to create a Bound or an Asset graph.

- A **Bound** graph, is one that is bound to the gameobject on which the FlowScriptController is attached on. As such, this allows you to have and assign scene object references within the various nodes of the flowscript directly.
- An **Asset** graph, is one saved as a .asset file within your project. The benefit of doing this is that it can be used and assigned amongst any number of different FlowScriptControllers if you so require, but the downside is that because it is an asset, you can't have scene objet references within the nodes of the flowscript.

Creating a Bound graph is the recommended type in most case, but don't worry about it too much because you are able to convert between the two at any time you wish.



Once you have created a flowscript through the button, the editor will show up immediately.

1.2. Understanding the Flow

Possibly the most important aspect you will need to understand about FlowCanvas, is how the flow and data is transfered through the connections from node to node and why. To begin with, we will have to explain the two different kind of ports that exist.

Flow Ports

Flow ports are those through which an *execution signal* is travelling one node after the other and which "makes things happen". A Flow port is the equivalent of calling methods/functions one after the other and in order, very similar to how it is done in actual coding. The flow signal is always travelling from left to right, equivalent to how methods are called from top to bottom in code. So, connecting a flow port to another input flow port, is like *calling* that port.

In the following example, the event node will begin a flow signal once the button "Fire1" is pressed Down. As soon as this happens, the Log Text node will be called.

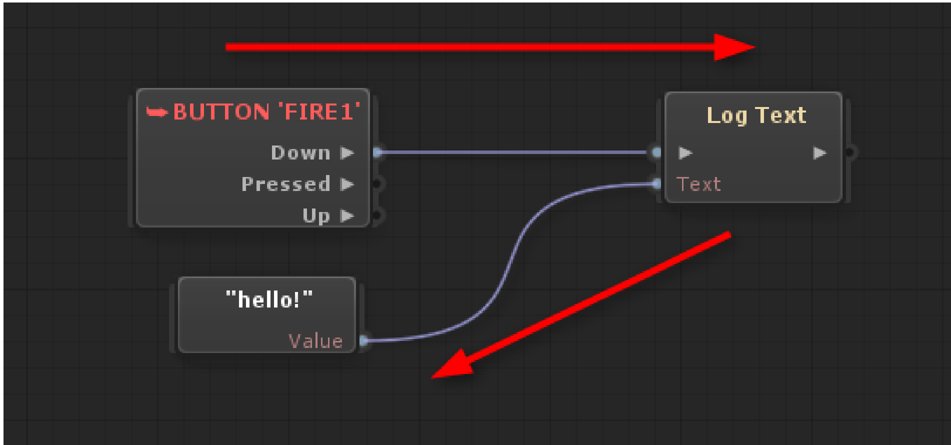
- Flow outputs can only have one outgoing connection *(if you want to split the flow, you can use the Split node).*
- Flow inputs can have any number of incomming connections.
- Flow ports can only be connected to other Flow ports.

Value Ports

Value Ports are those through which actuial data values are transfered, like numbers, strings, booleans, game

objects and prety much any type of data. Value ports are the equivalent of *getting* a value in code and *feeding*this value as a parameter to something. Unlike Flow signal, this is done in a backwards fashion, from right to left. So, value ports are mostly the equivalent of *get*.

In the following example, when the Log Text is called as described before, and only then, the “Text” parameter will request and get the value connected to it and as such the console will now log “hello!”.



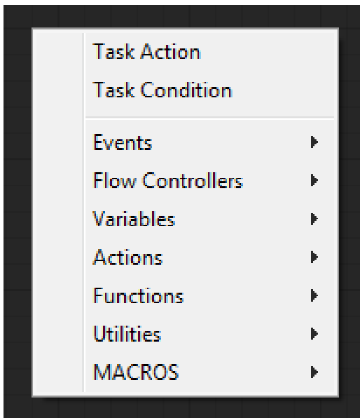
- Value outputs can have any number of outgoing connections.
- Value inputs can have only one incomming connections.
- Value ports can only be connected to other value ports of the same or of *assignable* type. What assignable type means, will be explained more in depth later on.

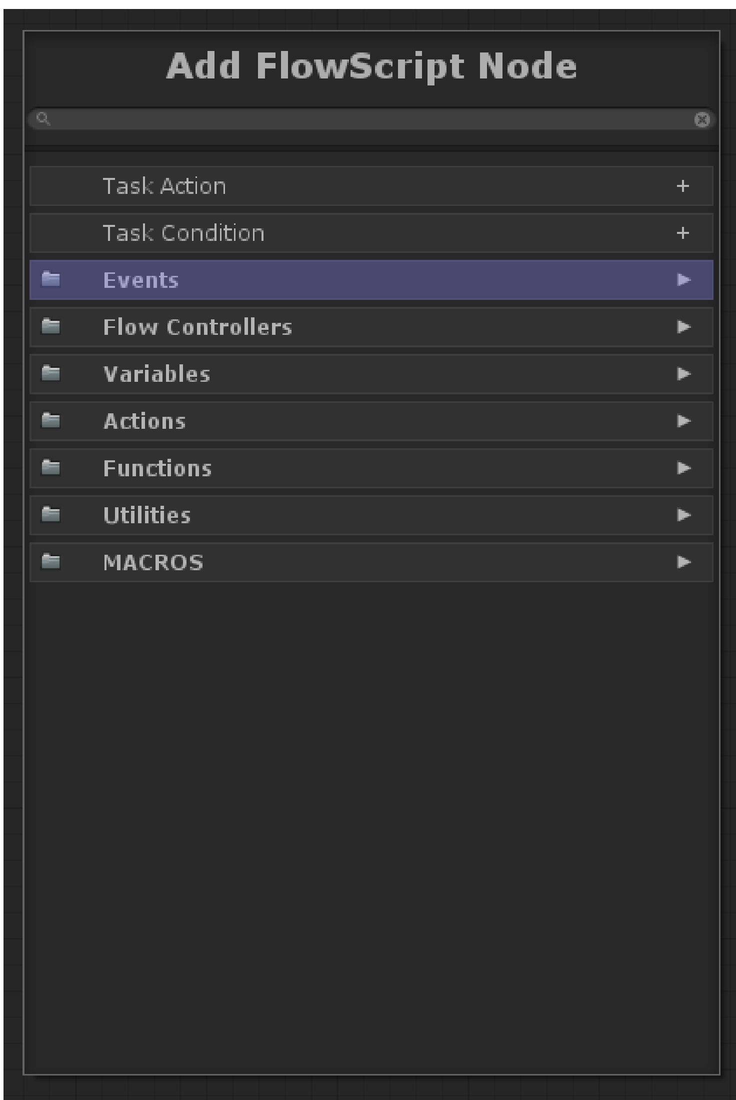
1.3. Adding Nodes

There are many ways by which you are able to add a node into the flowscript. Each way is described in detail bellow.

Context Menu & Browser

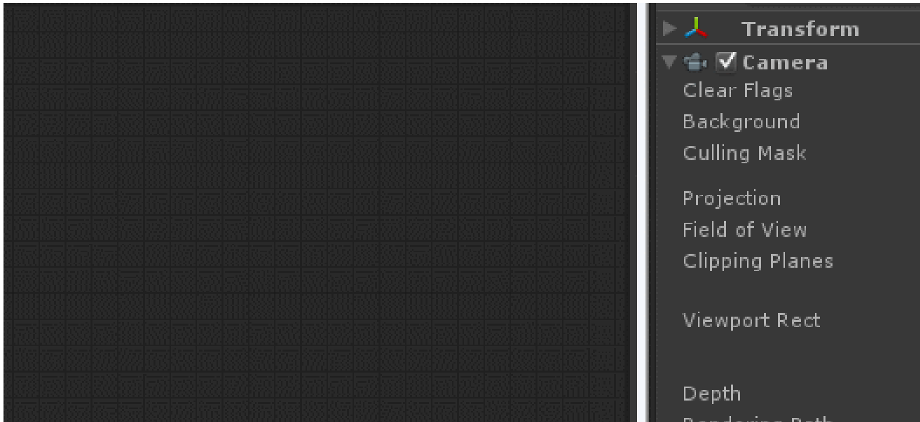
The most immediate ways to add a new node, is either by right clicking in the canvas to bring up the context menu, or by hitting the “Space” key while in the editor to bring up the complete browser. In both cases a menu with the nodes categorized will popup, but in the case of the complete browser, there will also be a search bar, icons, as well as the ability to navigate with arrow keys. Feel free to use either.





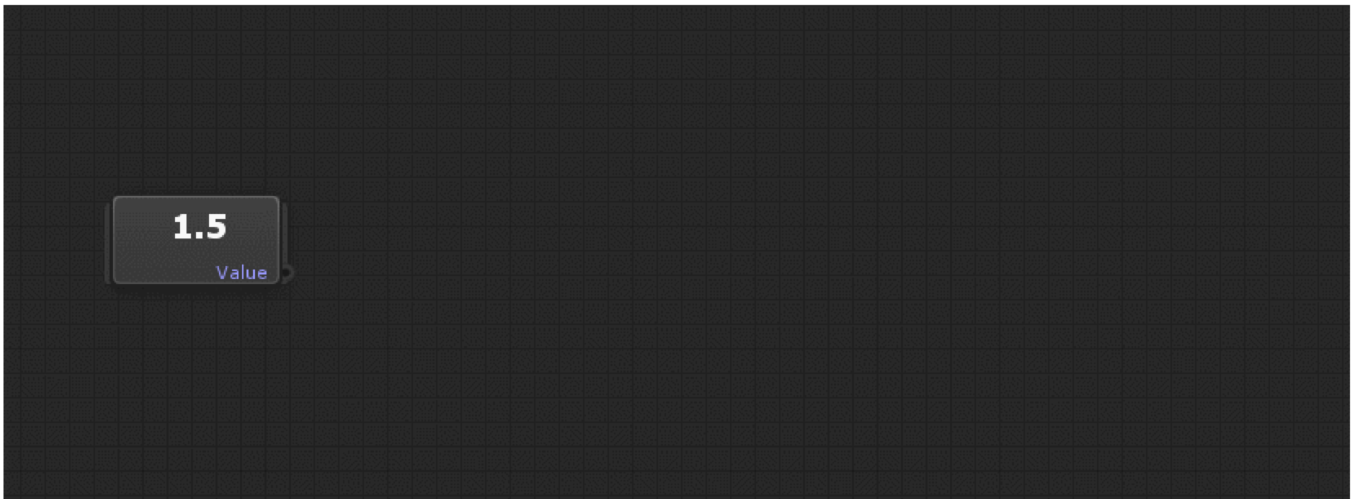
Drag & Drop Objects

Another way to add a node, is to drag and drop any object reference from the scene or the project into the flowscript canvas. Depending on what is dragged, a different context menu will popup. This menu will allow you to either add the object as a variable node, or even call, get/set one of it's methods or properties. In the following example a Camera component is dragged, but you can really drag and drop anything.



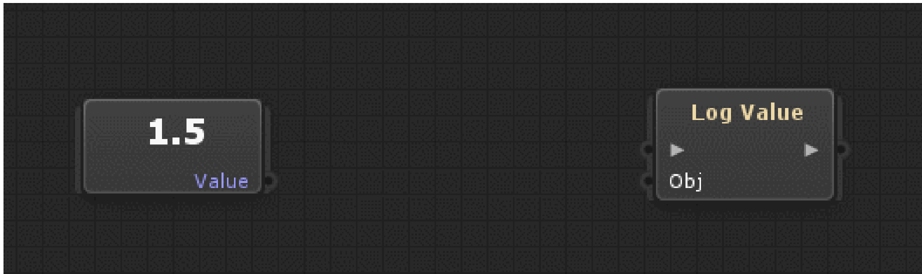
Drag & Drop Port Connections

A final way to add a new node is to drag a port connection into the empty canvas. This will show a context sensitive menu with relevant nodes to the type of the port dragged and also automatically connect the two nodes together as illustrated bellow. This is a time saver and it allows you to explore what can be done with the draged port type although it doesn't cover everything that can be done.

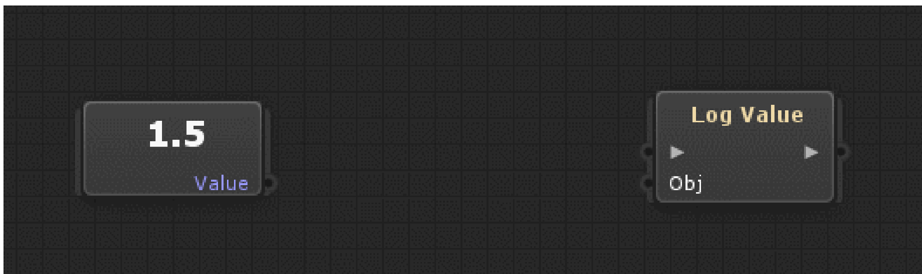


1.4. Connecting Nodes

Connecting node ports together is simple and straight forward and there are even some extra features to help you along. The most basic way of connecting two ports together is to click drag and drop one port to another. This can be done both in a forward as well as a backwards fashion *(meaning connecting output to input, or input to output).*

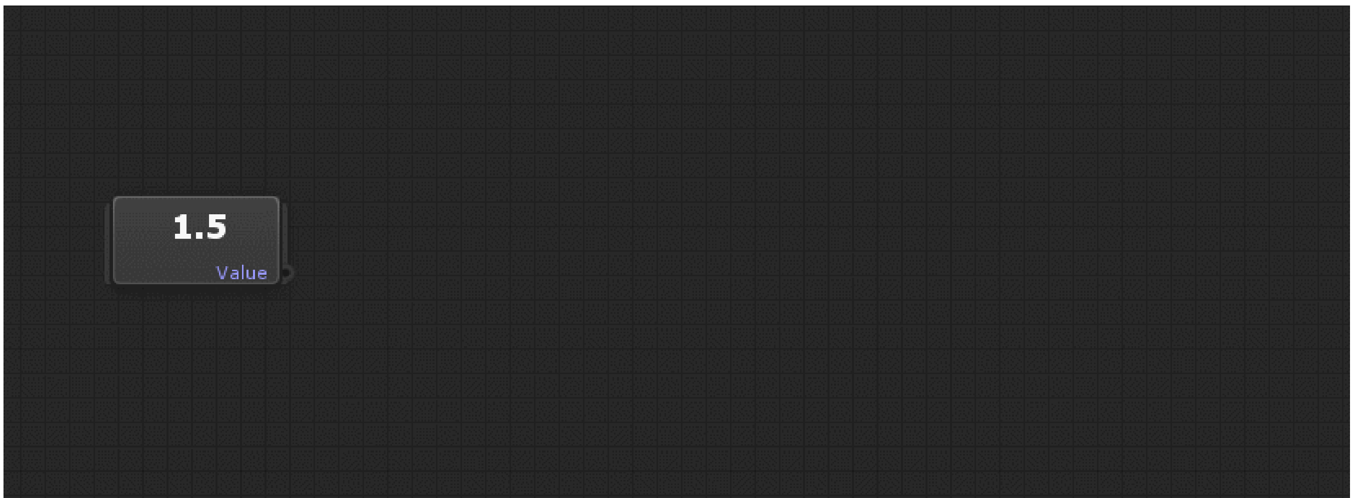


Alternatively you can also click drag and drop a port on to the whole target node. In this case, a sensitive context menu will popup, listing the possible ports that the dragged port can be connected to. This helps speed up your workflow instead of trying to target the right spot of a port. This can also be done in a forward of a backwards fashion.



Deleting a connection can be done by right clicking on a port. If the port has multiple connections, all of them will be deleted.

If you drag a port connection into the empty canvas, a type sensitive context menu will pop up with available nodes relevant to the type of the port draged. Selecting a node in that menu, will automaticaly add it and connect it as well.

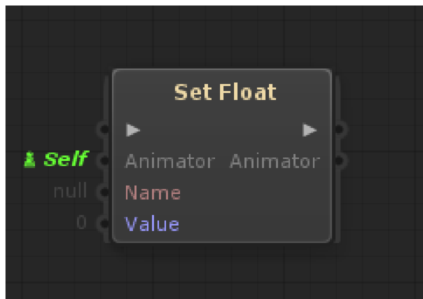


1.5. The "Self" Parameter

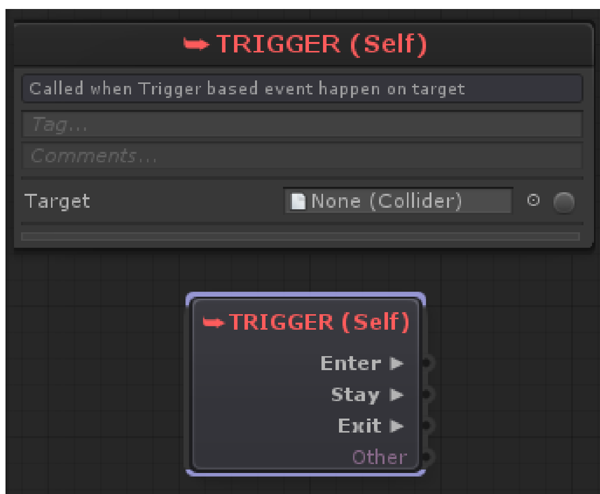
Within FlowCanvas, the term “Self” represents the owner of the flowscript for whom the flowscript is executed for. Or in other words, the game object that the FlowScriptController is attached to. The term “Self” can usualy be seen on the instance port of a node, if that port is of type GameObject or any derived Component.

In the following example, the parameter Animator of the Animator.SetFloat node, is marked as “Self” because it is

the instance on which Set Float will be called. If left un-connected and un-assigned, the parameter will resolve at runtime by getting the Animator component from the gameobject where the FlowScriptController is attached to. If there is none of course, the parameter will resolve to null.

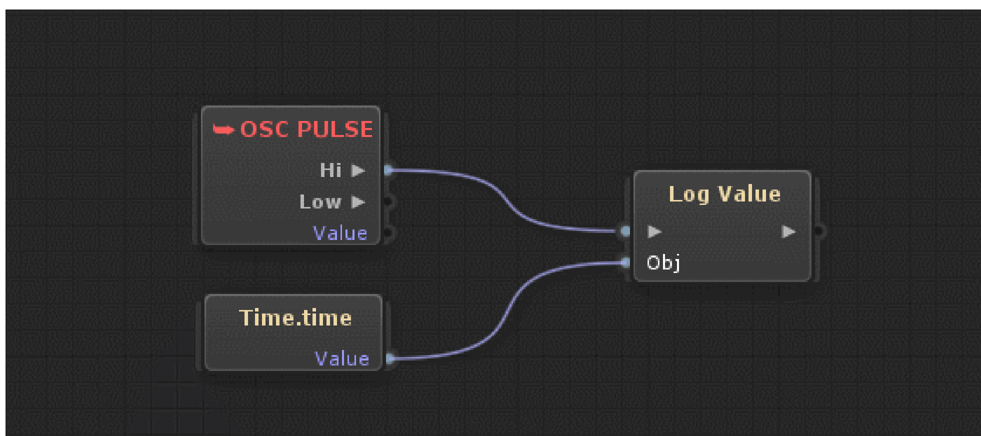


Another place where the “Self” term can be found is at specific Event nodes. In the following example, the event will be checked against the collider attached to the “Self” gameobject by default, but you are free to assign or *link* another collider if you so require. *(More about events and linking variables will be explained later on).*



1.6. Visual Debugging

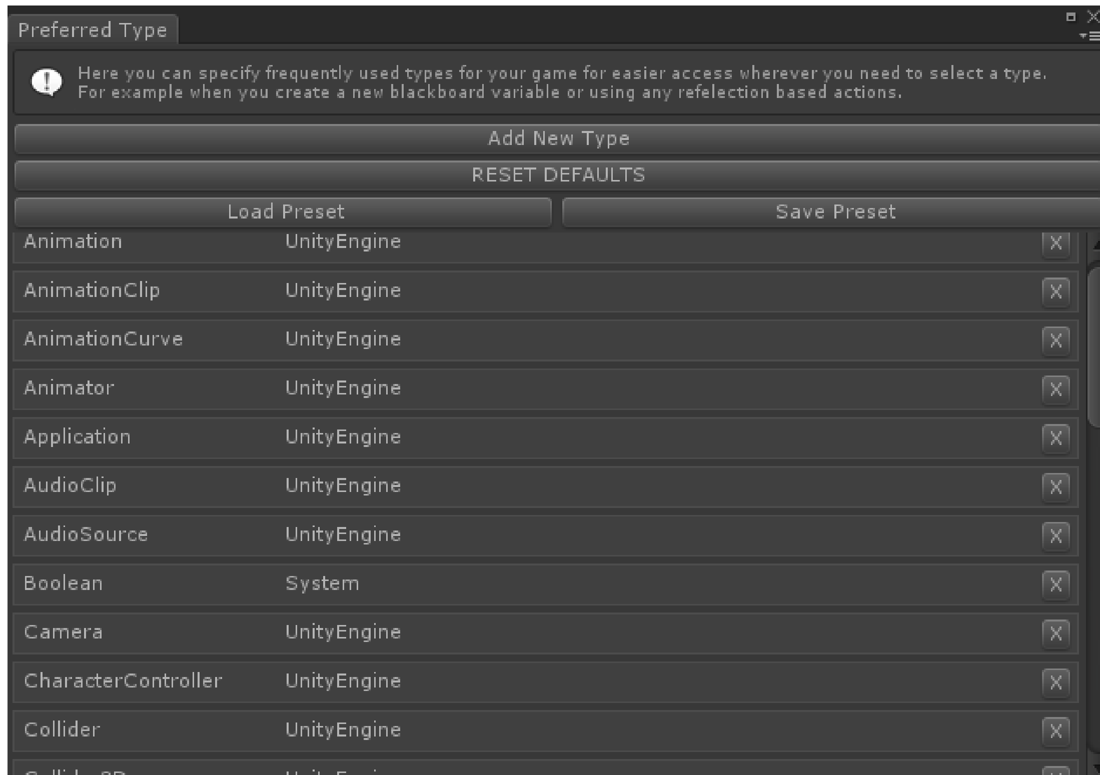
FlowCanvas comes with a very informative visual debugger. At runtime, you are able to watch all flow signals and data transfers within the editor window. The connections will blink in yellow accordingly and the value port connections will display the value that is currently transfered through it and to the target node!



2. Working with Types

The goal of FlowCanvas is to be able to work with any type you may need, from any assembly, which basicaly means that custom types are also supported if you so require. To keep the editor clean and not cluttered with the thousands of types available though, there exists a simply editor window, within which you are able to define which Types you want to work with within your project. By default, a handfull of common types have been added for you, but you are free to add, remove and modify the list as you require.

To open the Preferred Types editor, go to **"Window/NodeCanvas/Preferred Types Editor".**
This editor is inherited from the nodecanvas framework upon which flowcanvas is built upon.



The types listed in this editor will be responsible for a number of things.

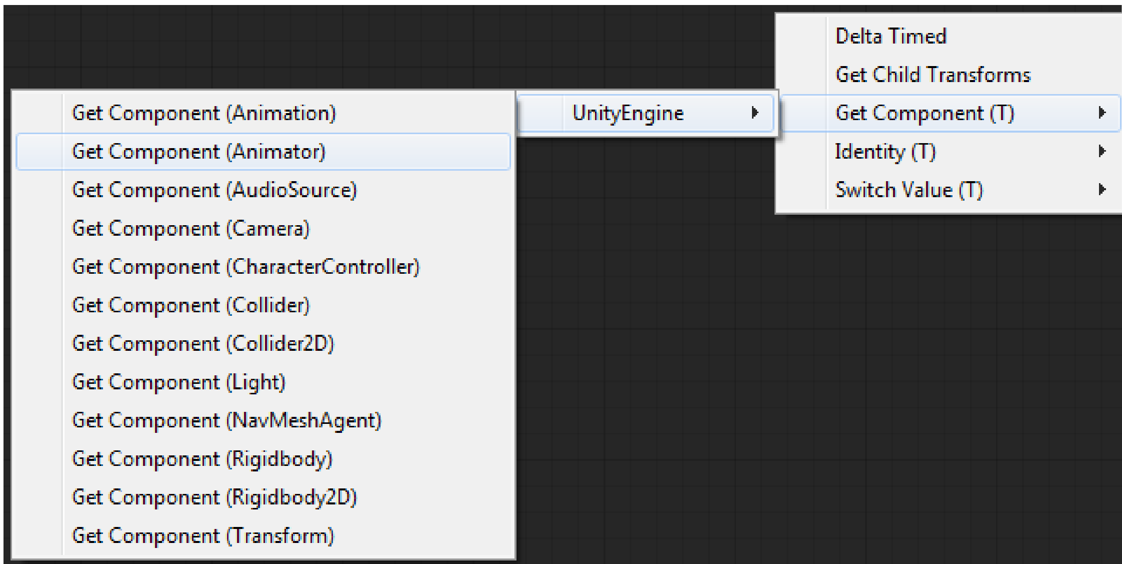
- The available blackboard variable types for you to add.
- The automaticaly generated reflection nodes.
- All generic centric nodes and menus that come with a (T) by the end.

This is a very important editor window!

2.1. Generic Type Nodes

When you are to select a node to add through the "Add Node" context menu or similar menus, there are a lot of nodes that come with a (T) suffix. These nodes are generic nodes. What this means is that they can work with any relevant type you want. Such nodes, will show as a category, under which the several types available will list.

As such, you will not find nodes like GetTransform, or GetAnimator. Instead you will encounter nodes like GetComponent(T), which will suffice to get any component type that you so require.

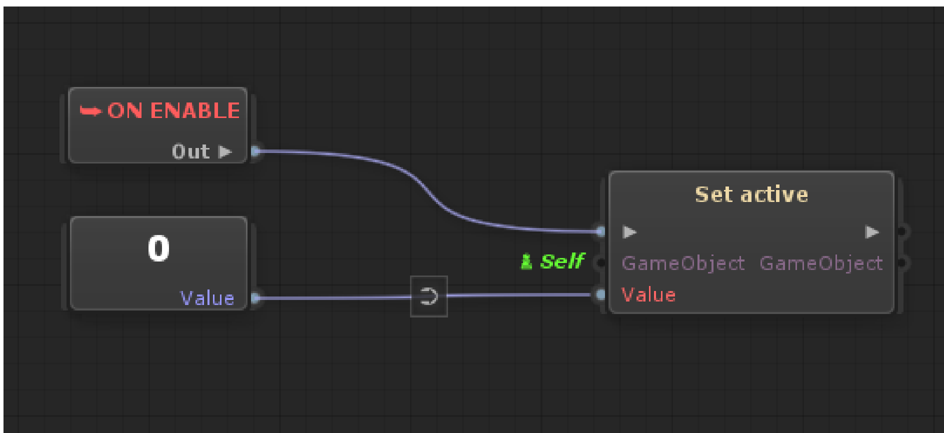


Now, the most important thing about these nodes, is that the type that will be listed as available as directly affected by the **Preferred Types editor** and the types listed there. As such it is very important to tailor your Preferred Types list to your project if the default types do not suffice.

2.2. Connecting Assignable Types

As explained earlier, *"value ports can only be connected to other value ports of the same or of assignable types"*.

What this basicaly means, is that you are able to connect certain value type ports together even though they are not of the exact same type, if they can be converted or casted to that type. This is handled automaticaly for you and saves you a lot of time. In practise, if the editor allows you to connect 2 ports together then it means it's possible and you have nothing more to worry about. When such a conversion is taking place, the connection will display an icon indicating the fact. In the following example, a float has been connected to a boolean. As a result, this conversion will result to false since the value is 0 *(and true if the value was 1+).*



For your convenience, here are the supported convertions by FlowCanvas and how they are done.

From	To	Through
Any Primitive	Any Primitive	.ConvertTo()
GameObject	Any Component	.GetComponent
GameObject	Transform	.transform
Any Component	GameObject	.gameObject

From	To	Through
GameObject	Vector3	.transform.position
AnyComponent	Vector3	.transform.position
A Child Type	Base Type	upcasting
A Base Type	Child Type	downcasting
Anything	String	.ToString()

3. Node Categories

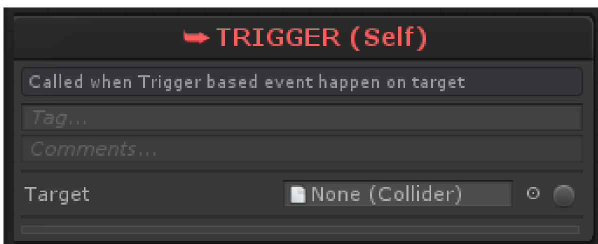
All available nodes in FlowCanvas can really be categorized into four seperated types, those being…

- Events
- Flow Controllers
- Actions
- Functions

3.1. Events

Event Nodes are responsible for beginning and triggering any *Flow Execution.* Without an Event node nothing will really happen in your flowscript. You add Event nodes to respond to world or object events and changes like Input, Triggers, Collissions etc. and finally connect their flow outputs to other nodes to designate what will happen when the event is triggered.

Some Event Nodes require an object reference on which the event will take place. By default the target object will be “Self”, which coresponds to the gameobject on which the FlowScriptController component is attached to. You are though free to change the target object through the inspector by either assigning a new reference or linking a blackboard variable.

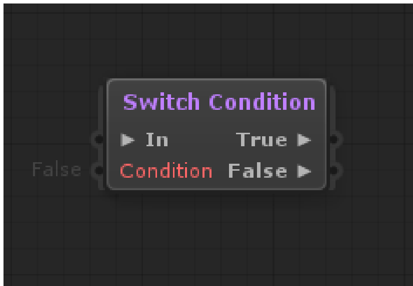


All Event nodes have a red title name, no inputs and any number of outputs in a mix of Flow and Value ports. Value ports of Event nodes, usually contain information about the event that has just been triggered.

3.2. Flow Controllers

Flow Controller Nodes are responsible for manipulating the *Flow Execution,* like branching the flow in a number of

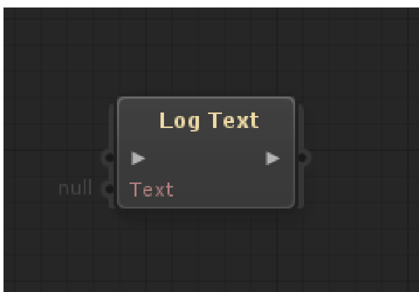
different ways, filtering the flow, or even merging the flow. They are the means by which you create logic into your flowscript and correspond to the various scripting commands like if-then-else, while, for-each etc.



Flow Cotnrollers have a blue-ish title color and always have at least 1 Flow input and 1 Flow output port.

3.3. Actions

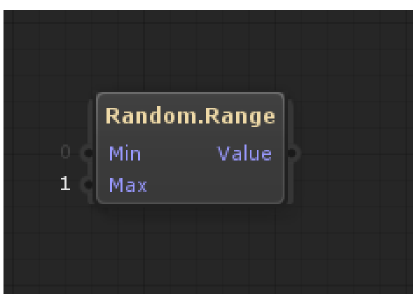
Action Nodes are responsible for “making stuff happen”, like moving objects, setting values, destroying object, applying physics forces etc. Action nodes are the equivalent of *methods* in a scripting language. 99% of the times, Action nodes have no Value outputs, but a number of different Value inputs for setting the parameters of the action.



Action nodes, always have 1 Flow input, which simply refers to “Execute” the action!

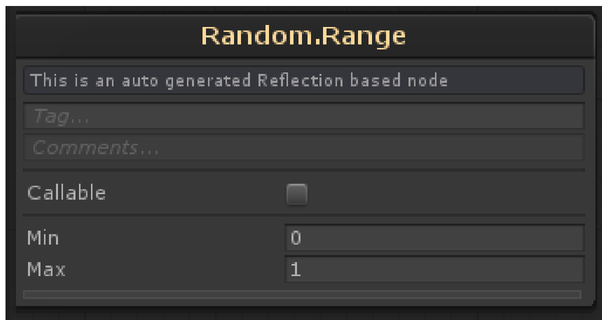
3.4. Functions

Function nodes are the equivalent to *functions* in a scripting language. The big difference in comparison to Action nodes, is that they **always** return a value in the form of a Value output port. Function nodes are responsible for manipulating data, math operations getting data, as well as feeding these data into other nodes.

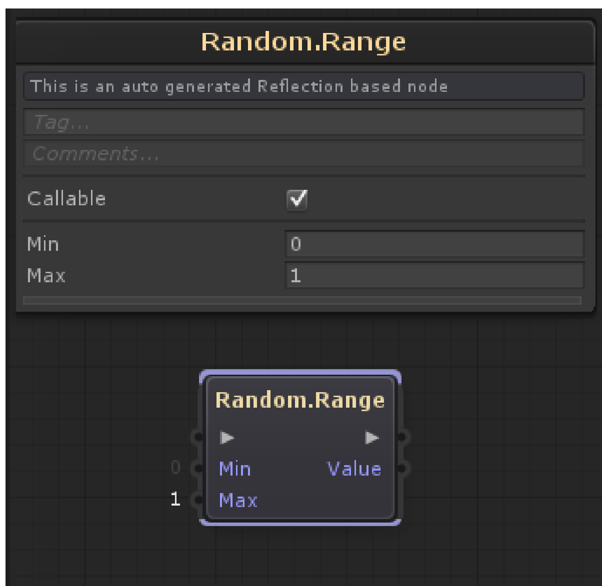


By default, a Function node does not require Flow execution to work and as such they have no Flow input or output ports! You simply connect the Value output to some other node you wish and that’s it. Having said that, there are some rare cases in which you would prefer to have control on when the node is called, by explicitely connecting a Flow into the node. In such cases, you can make the node “Callable” by ticking the respective

toggle on the node inspector.

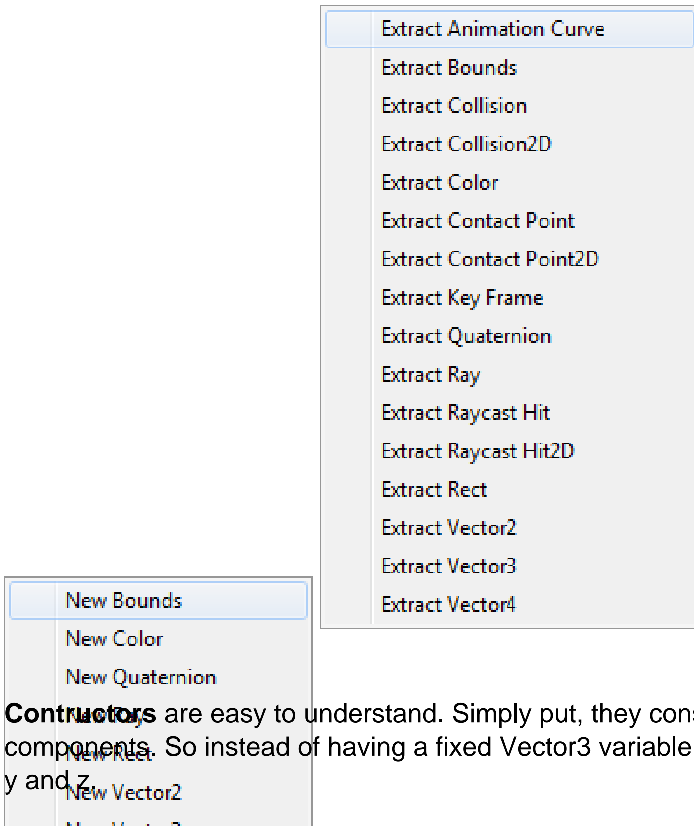


Having done that, the node will be converted to an Callable Function as shown bellow, where 1 Flow input and 1 Flow output will appear. As such, the Value output of the node will only be set whenever you explicitely call the node with a Flow signal.



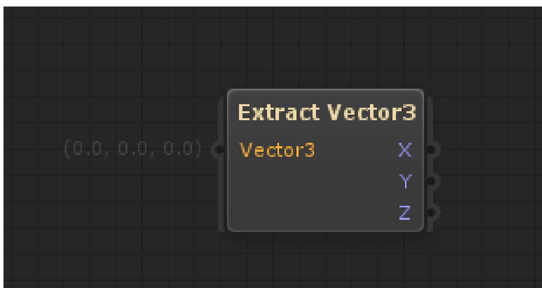
3.5. Constructors & Extractors

While technicaly Constructor and Exctractor nodes are Function nodes, they are well worth mentioning seperately. Both of these can be found under the “Utilities” category.



Contructors are easy to understand. Simply put, they construct a new object *(mostly a struct)* out of it's several components. So instead of having a fixed Vector3 variable, you can construct one out of it's three components x, y and z.

Extractors on the other hand, are quite the opposite. They take an object *(again mostly a struct)*, and expose their components for you to use.

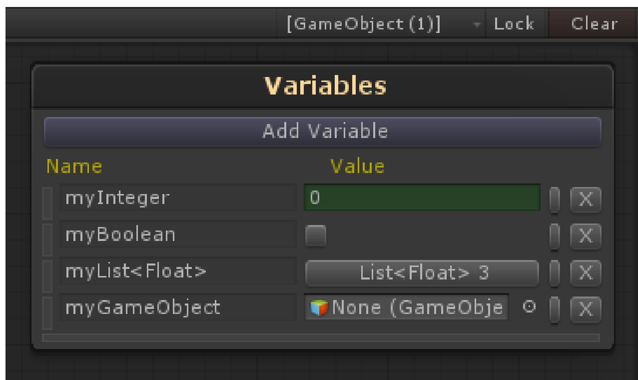


4. Variables

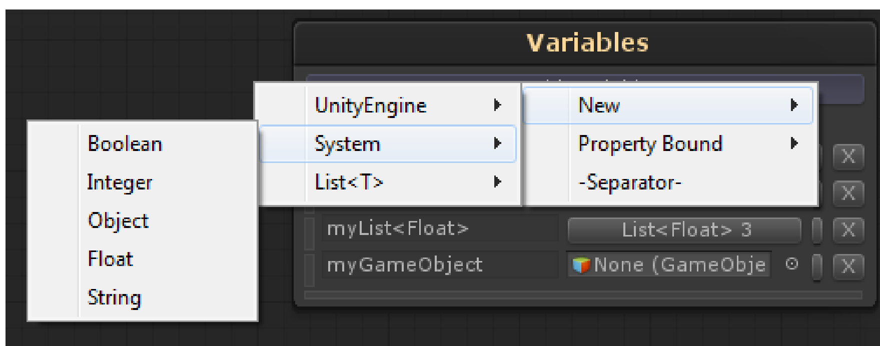
Variables are a very important aspect of any program, script or visual script. FlowCanvas has superior support for all and any type of variables out-of-the box, along with some important features like Property Binding, or network sync through UNET!

4.1. Blackboard Variables

The Blackboard is a component which is automaticaly added for you on the same game object where you have attached any FlowScriptController component and is the main place where you create, store and retrieve variables to be used within any flowscript of that game object. The blackboard inspector is also displayed withing the flowscript editor at the top right.



To create a blackboard variable you simply have to click on the “Add Variable” button as seen above. A context menu with all available types will show up for you to add a variable of. Remember that all types are supported, as long as you add them in the Preferred Types list as explained earlier!

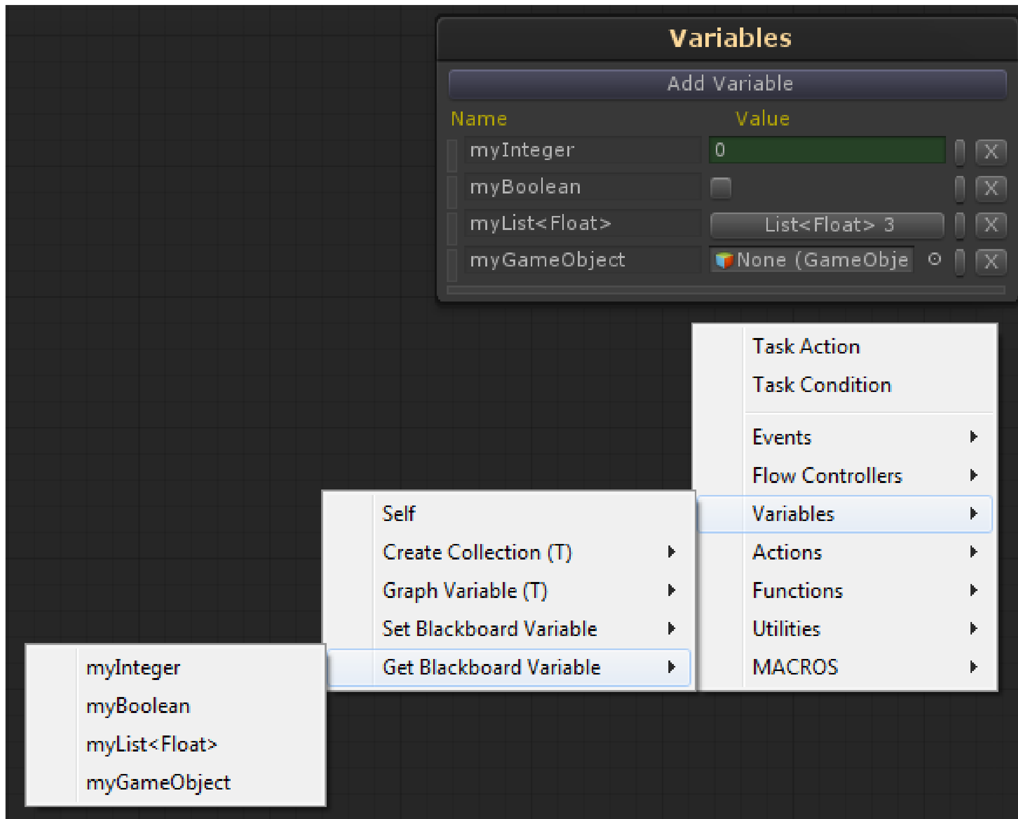


Once you have created a blackboard variable, you are able to use it within any flowscript of the gameobject either getting it’s value or setting it’s value as you require.

4.2. Using Variables in FlowScript

Using Blackboard Variables

To make use of a Blackboard Variable created, either getting or setting it’s value, you can do so through the “Add Node” context menu, under the “Variables” category and either through “Get Blackboard Variable” or “Set Blackboard Variable” sub-categories, selecting the variable name you wish to get or set.



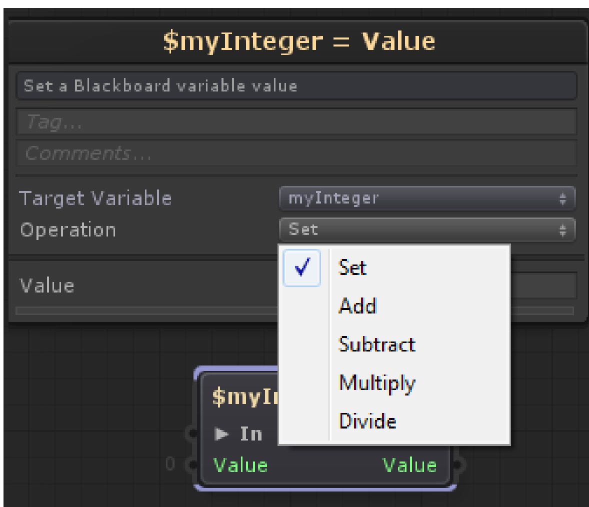
Get Value

Selecting to get a blackboard variable, will create a variable node, *linked* to that blackboard variable. The**"$"** token is always used before a name, to indicate that it is a linked blackboard variable name. In this example "$myInteger". The single Value output port of the node, will return the value of that *linked* variable.

Set Value

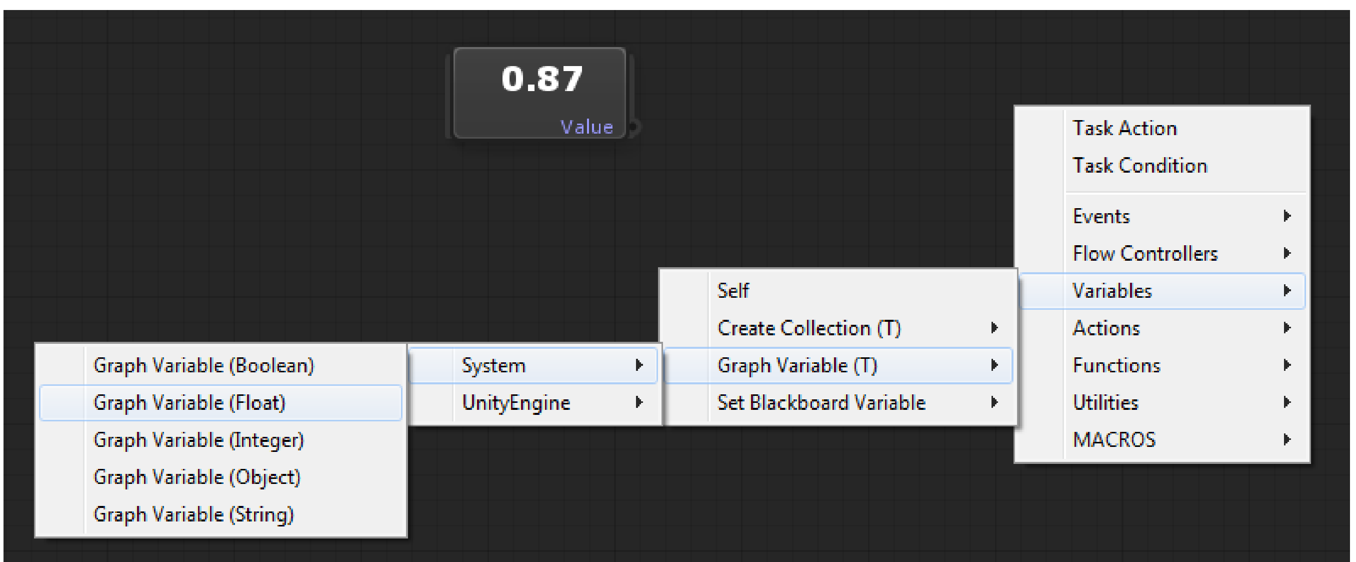
Selecting to set a blackboard variable on the other hand will create a different type of node as seen bellow. An action node. As such, the value will only be set when the node is called with a Flow signal and by default will be set to be equal to the value input parameter connected.

In the node inspector of the node above, you are also able to specify an *operation* to be used when setting the variable. Instead of simply setting the variable to be equal to the value parameter, you can choose to add, subtract, multiply, or divide the value parameter instead.



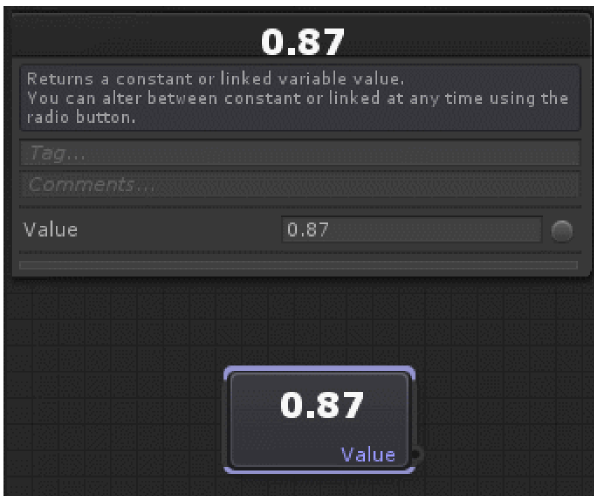
Using Constant Variable Nodes

You don't always have to use Blackboard Variables though. In some cases you may simply want a constant variable used in some place and nowhere else. In these cases, it's best to simply create a GraphVariable node as illustrated bellow and set it's value directly through the node inspector.



Later on, if you change your mind and want to *link* this node to a Blackboard Variable instead, you can simply do

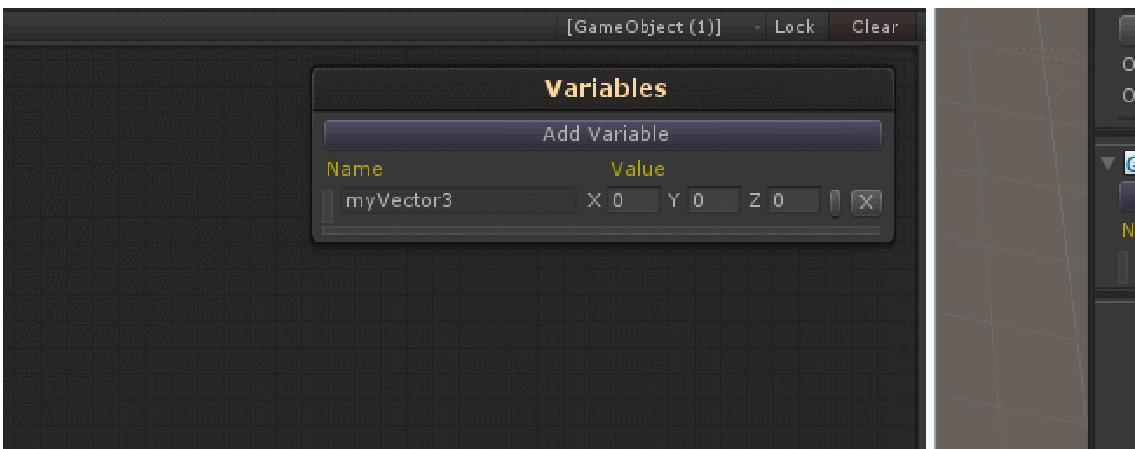
so through the inspector by clicking on the radio button!



4.3. Property Binding

Binding variables to properties is a great way to speed up your workflow. In practise, a variable bound to a property means that the variable represents that property and acts as a “bridge” to it. As such, whenever you get the variable value you will instead get the bound property value and when you set the variable value you will instead set the bound property value.

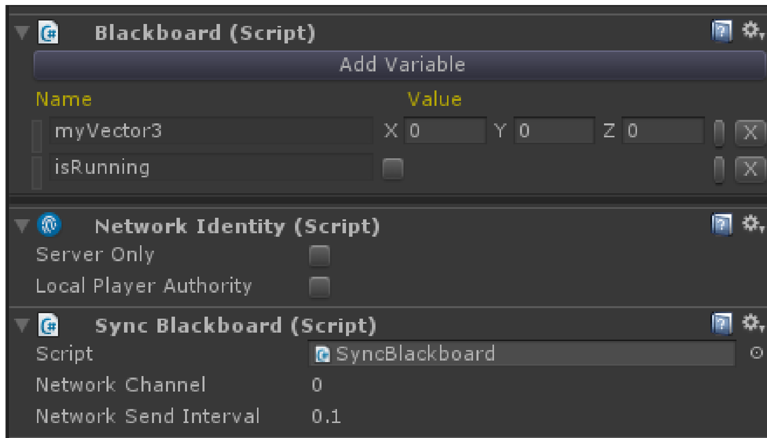
A blackbord variable can be bound to any property of any component on the same gameobject where the blackboard component is attached on. To bind a variable to a property, you simply have to click the variable*options button*, and select the property you want to bound it to!



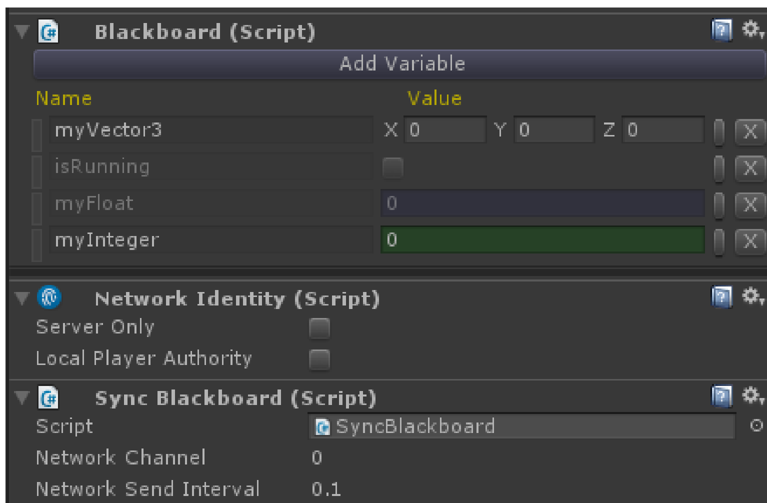
Alternatevily, if you don’t already have a created variable, you can directly create a bound variable through the “Add Variable” button menu.

4.4. Network Sync

Blackboard Variables are possible to be synced across the network using the new Unity 5 system, UNET with ease. All you have to do, is to add the SyncBlackboard component on the same gameobject that has the blackboard you want to sync variables.



Once that is done, all variable of the blackboard that are supported by the UNET system and that are not marked as *protected*, will be synced across the network. There is no extra steps needed at all. If you want to ignore a specific variable from being synced, you have to mark it as *protected*. This can simply be done so through the variables *options button*. Protected variables are also shown in transparent as a reminder. In the following example, “isRunning” and “myFloat” are marked as protected.



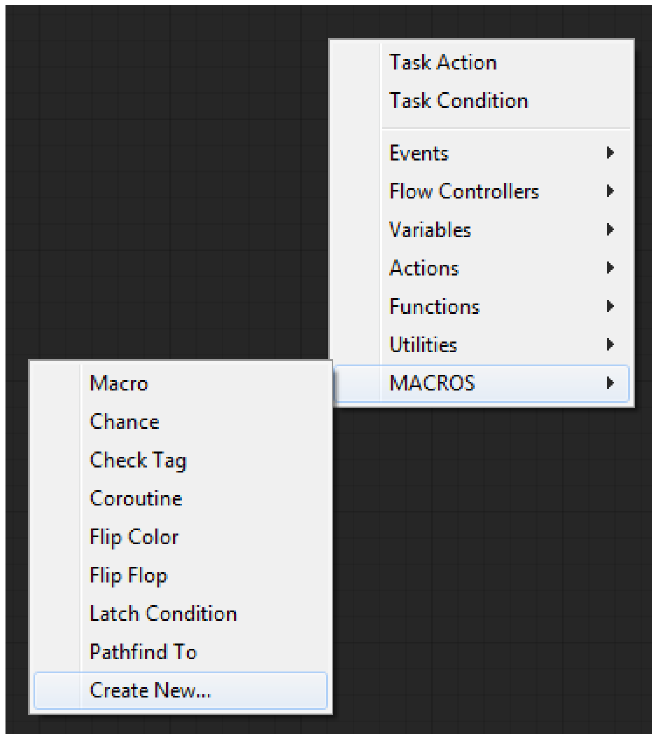
5. Macros

Macros are a bunch of nodes compounded into a single reusable flowscript, that can be used in any other flowscript as a whole…a macro and are always saved as a .asset file in the project.

Macros are an extremely powerfull feature to organize and make reusable visual scripts for your projects, even share them with other people with ease!

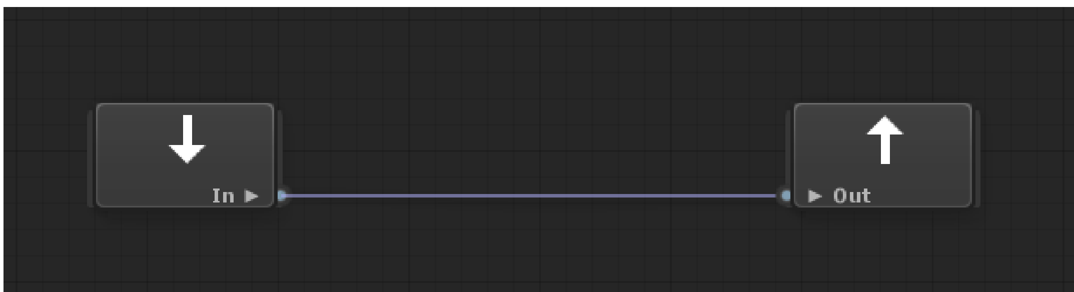
5.1. Creating Macros

Creating a Macro can be done in two ways. Either through the “Add Node” context menu and under the “MACROS/Create New…” option, or by right clicking in the Unity project panel and selecting “Create/New/Macro”. In either case, a “Save As” window will pop up for you to select where to save the new macro asset file in your project.

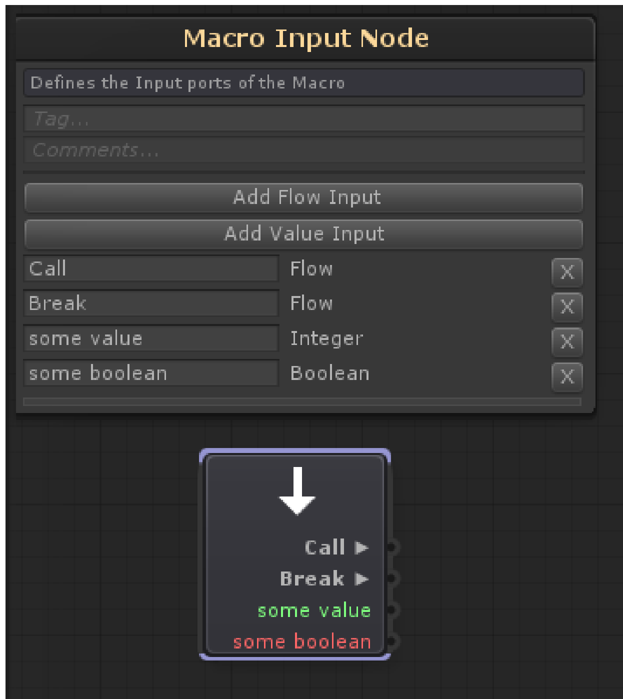


Editing the newly created Macro can be done by either selecting it in the project panel, or simply double clicking the macro added in the flowscript, in which case it will open up inline to your flowscript.

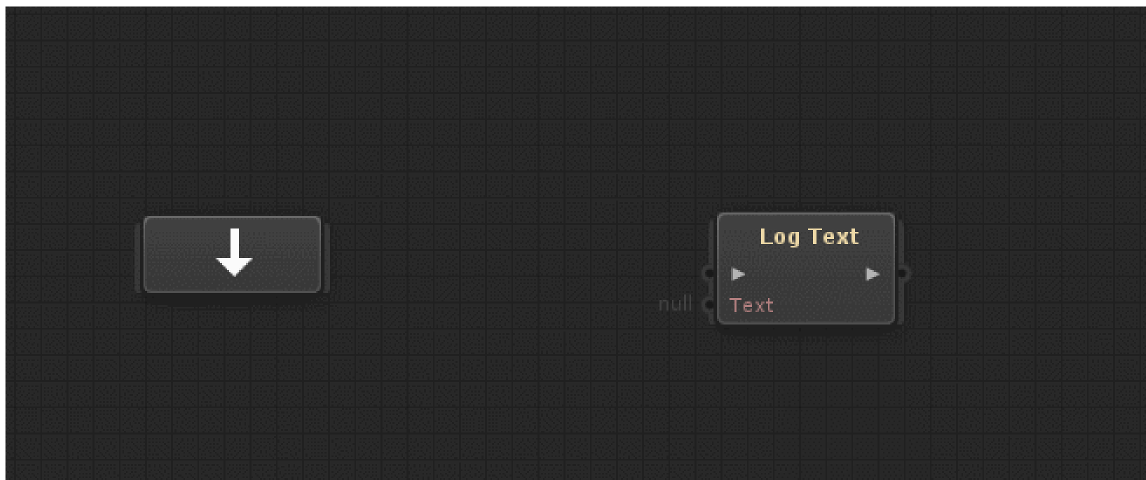
Every newly created Macro, will come with 2 very important nodes already created within, those being the Macro Inputs and the Macro Outputs.



As they suggest, those two nodes are there to define all the input and outputs that the Macro has. Clicking on the Macro Input node, you are able through it's node inspector to define the various input ports of the macro and similar it can also be done by clicking on the Macro Output node. In the following example, there are 4 different input ports defined. 2 Flow inputs called "Call" and "Break", as well as 2 Value inputs one being of an integer and the other of a boolean type.



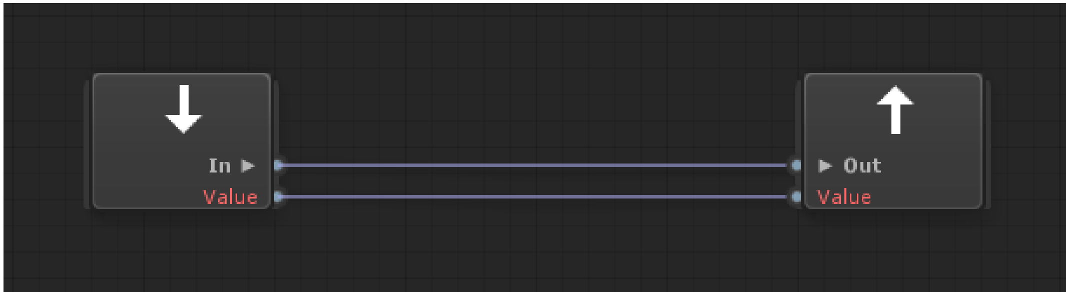
An alternative way of defining new macro ports, is to drag and drop another node's port on top of either the Macro Input or Macro Output nodes, as illustrated bellow. You can do the same for output ports as well and you are free to rename the ports at a later time of course.



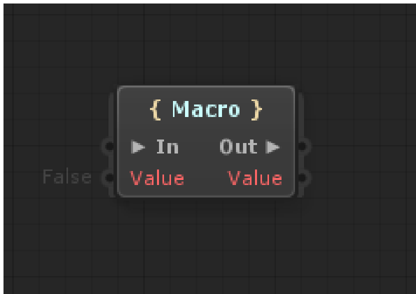
5.2. Using Macros

After a Macro has been created, it can be added within any flowscript to be used. Adding an existing macro into a flowscript can be done either though the "Add Node" context menu and under the "MACROS" category, or by drag and dropping a Macro file into the flowscript canvas.

Once a Macro has been placed within a flowscript, all of it's defined inputs and outputs (as described before), will be exposed for you to use and connect. For example, the following Macro *(even though it doesn't really do much)…,*

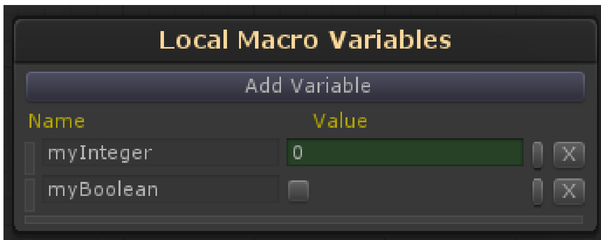


…will look like this when added within a flowscript.



Once again, double clicking the Macro node, will open it up for you to edit.

Macros also have their own **local variables,** which can be used for internal data keeping within the scope of the macro only. Of course you are free to set these variables through input ports or expose them through outputs ports if so required.



Finaly, here are some important things to keep in mind when working with Macros.

- Because Macros are always .asset files, they can't have scene object reference assigned within the nodes of the Macro, but you are free to have scene object references in the Macro node inputs used in the flowscript itself.
- Generaly speaking, it is wise to not have event nodes within a Macro. While it will work fine, it's not suggested as a workflow.
- A Macro does not need to have Flow inputs and outputs. You can remove the ones automaticaly added when creating a new macro if you don't need them.

6. Creating Custom Nodes

There are three possible ways you are able to create nodes in FlowCanvas.

- Automaticaly through reflection.
- Coding Simplex Nodes.
- Coding Full Nodes.

6.1. Creating Simplex Nodes

Creating Simplex Nodes is the most easy and direct way of creating custom nodes for FlowCanvas. To do so, you need to derive from one of the available Simplex node base types.

CallableActionNode

Callable Action Nodes, are ment to create actions requiring a flow signal to be called. They have 1 Flow input, 1 Flow output, up to 5 Value input parameters and no Value outputs at all. Simply put, you derive the generic CallableActionNode class where the generic arguments represent the value input parameters of your node and then you override the Invoke void method, which has a number of parameters equal and of same type as the generic arguments you declared. For example.

```
[Category("Actions/Utility")]
public class LogValue : CallableActionNode<object>{
 public override void Invoke(object obj){
  Debug.Log(obj);
 }
}
```

CallableFunctionNode

Callable Function Nodes, are similar to the Callable Action above, but they always return 1 Value output. The proccess is very similar to before, but in this case the first generic argument represents the type of the return value and the Invoke method that you will override will need to return that type declared. For example.

```
 [Category("Examples")]
 public class FindTransform : CallableFunctionNode<Transform, Transform, string>{
  public override Transform Invoke(Transform root, string name){
   return root != null? root.Find(name) : null;
  }
 }
```

LatentActionNode

Latent Action Nodes are similar to CallableAction Nodes before, but in this case the Invoke method is a IEnumerator and is called as a coroutine. As such, you need to treat it as a coroutine. For example.

```
[Category("Actions/Utility")]
public class Wait : LatentActionNode<float>{
 public override IEnumerator Invoke(float time){
  var timer = time;
  while (timer > 0){
   timer -= Time.deltaTime;
   yield return null;
  }
 }
}
```

PureFunctionNode

Pure Function Nodes, are ment for nodes that do not need any flow signal to use, but instead they simply return a value when they are requested through the single Value output they have. The first generic argument is the return type of the node and the Invoke method you need to override, while the rest of the generic arguments are the value input parameters of the function to be used. For example.

```
[Category("Functions/Math/Floats")]
[Name("!=")]
public class FloatNotEqual : PureFunctionNode<bool, float, float>{
 public override bool Invoke(float a, float b){
  return a != b;
 }
}
```

6.2. Creating Full Nodes

Creating full flow nodes can be a bit more hard to understand, but you get full access to everything and can create any type of node you want, like for example new Events, Flow Controllers or anything else, although I personaly recommend going this way for Events and Flow Controllers only.

Let’s take an example of re-creating a FlowController node, the “SwitchBool”, which practicaly is an if-then-else statement. To do so, create a class and derive from “FlowControlNode”, then you must override the RegisterPorts method and register the ports within.

Registering the ports is done by using the following methods:

FlowInput : AddFlowInput(string name, Action pointer)
name: The name of the port.
pointer: A delegate that will be called once this flow input is called.

FlowOutput: AddFlowOutput(string name)
name: The name of the port.
To execute the port, you have to call, Call() on the FlowOutput object returned, along with the current Flow object received.

ValueInput<T> : AddValueInput<T>(string name)
name: The name of the port.
To get the value connected to the port, you simply do so by calling the .value property of the object returned.

ValueOutput<T> : AddValueOutput<T>(string name, Func<T> getter)
name: The name of the port.
getter: A delegate that will be called to get the value of type T.

In practice it’s easier than it sounds…

```
using ParadoxNotion.Design;

namespace FlowCanvas.Nodes{

 [Name("Switch Condition")]
 [Category("Flow Controllers/Switchers")]
 [Description("Branch the Flow based on a conditional boolean value")]
 public class SwitchBool : FlowControlNode {
  protected override void RegisterPorts(){
   var condition = AddValueInput<bool>("Condition");
   var trueOut = AddFlowOutput("True");
   var falseOut = AddFlowOutput("False");

   AddFlowInput("In", (f)=>
   {
    if (condition.value){
     trueOut.Call(f);
    } else {
     falseOut.Call(f);
    }
   });
  }
 }
}
```

As you see above, there are a number of attributes you can use for specifying the name if it needs to be different than the class name, as well as attributes for the node's category and help description.

6.3. Creating Event Nodes

Sometimes you may want to create a custom event node to listen to some c# event raised. While you can use the CodeEvent node to subscribe to a c# event through reflection, it may not always suffice since with that node it's only possible to subsribe to events of delegate type System.Action and System.Action<T>. As such, you can create a custom event node to handle everything you require. Here is an example for an event node making use of a static c# event:

```
namespace FlowCanvas.Nodes{

 public class CustomEventNodeExample : EventNode {

  private FlowOutput raised;

  public override void OnGraphStarted(){
   //Subscribe to the event here. For example:
   MyClass.MyEvent += EventRaised;
  }

  public override void OnGraphStoped(){
   //Unsubscribe here. For example:
   MyClass.MyEvent -= EventRaised;
```

```
  }

  //Register the output flow port or any other port
  protected override void RegisterPorts(){
   raised = AddFlowOutput("Out");
  }

  //Fire output flow
  void EventRaised(){
   raised.Call(new Flow());
  }
 }
}
```

If the c# event is not static and rather an instance member on a MonoBehaviour, you can derive from the generic version of EventNode. In this case the T argument is the type required and you get a reference to the component, through the *target.value* inherited property. When you create such an Event Node and within the node's inspector panel, as with all other event nodes, you will be able to either leave the component reference blank to automaically get it from “Self” -meaning from the gameobject the FlowScriptController is attached-, or you can assign a reference directly. Here is an example of such a node:

```
namespace FlowCanvas.Nodes{

 public class CustomEventNodeExample : EventNode<MyComponent> {

  private FlowOutput raised;

  public override void OnGraphStarted(){
   base.OnGraphStarted(); //make sure to call base.
   //Subscribe to the event here. For example:
   target.value.MyEvent += EventRaised;
  }

  public override void OnGraphStoped(){
   base.OnGraphStoped(); //make sure to call base.
   //Unsubscribe here. For example:
   target.value.MyEvent -= EventRaised;
  }

  //Register the output flow port or any other port
  protected override void RegisterPorts(){
   raised = AddFlowOutput("Out");
  }

  //Fire output flow
  void EventRaised(){
   raised.Call(new Flow());
  }
 }
}
```