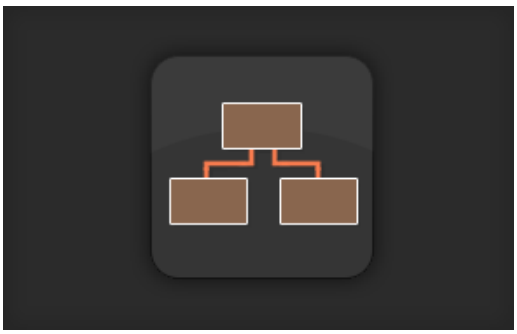
NodeCanvas Offline Documentation

http://nodecanvas.paradoxnotion.com

1. Getting Started



Welcome to the NodeCanvas documentation!
While reading, don't forget to leave your feedback with the buttons bellow!

1.1. Framework Concepts

In this section, the core concepts of the NodeCanvas framework will be explained in as much detail as needed to get help you understand how things work behind the scenes. This information holds true for all systems created on the NC framework.

Agents

Agent is a core term used to describe the 'object' that something is executed for, like for example 'who' an Action will execute for, or 'who' a Condition will be cheked against. In NodeCanvas, Agent is a Component type reference rather than a GameObject reference which allows for some flexibility. An Agent reference is propagated to the *Graph* when it excutes.

Blackboards

Blackboard is a object in which variables can be stored or be retrieved from. It is used to communicate data within different *Nodes* or *Tasks* that have no knowledge of one another what so ever. Blackboards can also be used outside of the context of NodeCanvas with a very simple API. A Blackboard reference is propagated to the Graph when it executes. You will use Blackboards to parametrize the Graph assigned to a GraphOwner.

GraphOwners

The GraphOwner is a simple component that wraps the execution of a graph that is assigned to it. It acts as a front-end "interface" through which the assigned graph gets executed and can be controlled, or information abouts it's state can be retrieved. Using GraphOwners is the easiest way to make an object behave based on a NodeCanvas graph system, but it's not mandatory.

Gameobjects with at least one GraphOwner attached to them will show a ? icon in the scene hierarchy within unity.

Graphs

Graphs hold the actual functionality of a system, what it does and how it does it, as well as all the nodes within it. When a graph executes, it always does so for an *Agent* and using a specific *Blackboard.* A graph is able to run only for one Agent at a time, thus a seperate instance for each agent required is created. This is of course automatically handled when you are using a GraphOwner.

Graphs can be Bound to an Agent or not (Asset). Bound Graphs are local and saved along with the gameobject that the GraphOwner is attached to, while Asset graphs are saved as a .asset file in your assets folder and a such they can be assigned to any number of different GraphOwners. The benefit of using Bound graphs though, is that because they are saved with the gameobject, if that game object is also in the scene, you can have scene object references assigned to that graph.

Nodes

Nodes live within a Graph and depending on the type of the Graph, different Nodes can live within each Graph type. In NodeCanvas the 'what' and 'if' happens in most cases is not contained within a Node's functionality though, but rather a Node wraps the functionality of Tasks which are assigned on nodes that require it. So in other words, a Node doesn't contain the Action. It is rather assigned an Action to use if needed. This allows for non-destructive design by decoupling the tree design from the task implementation.

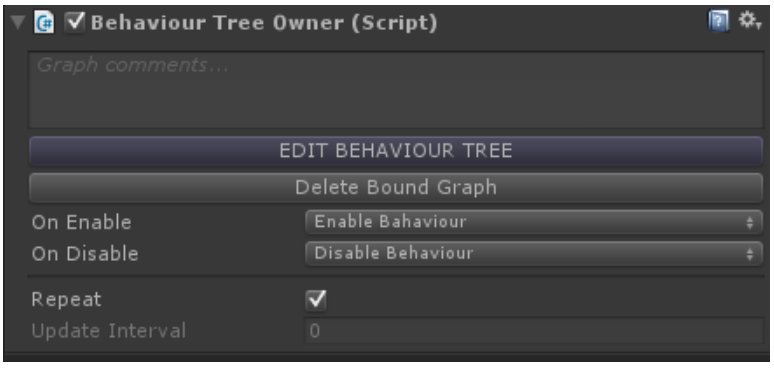
Tasks

Tasks are the Actions and Conditions. They are assigned on Nodes that require them and are the final destination of the tick. When a Task is executed or checked, it is done so for an Agent and a Blackboard. By default, this Agent and Blackboard are the ones that the Graph is using, although an Agent can be overriden if you want for example an Action to be performed by another Agent rather than the default one that the Graph has assigned. Creating custom Tasks is what you most probably going to do for your game if you want so.

1.2. The GraphOwner Component

The GraphOwner components are responsible for making an object behave based on a Behaviour Tree or an FSM, by either using the **BehaviourTreeOwner**, or the **FSMOwner** derived components. These components do not contain the graph itself, but they are rather assigned a graph.

- To make a gameobject behave with a Behaviour Tree, you have to attach the BehaviourTreeOwner component to that gameobject.
- To make a gameobject behaviour with an FSM, you have to attach the FSMOwner component.
- You can attach both type if you want and any number of those on the same game object.



Once you have attached your GraphOwner component, you will need to assign it an actual graph. An assigned graph to an owner can either be Bound or an Asset.

- **A Bound Graph** is local to the owner and is saved along with the gameobject. As such it is possible to have scene object references assigned within the graph if the owner is also within the scene.
- **An Asset Graph** is a reference to a .asset file saved in your assets folder. The benefit of this, is that such a graph can be assigned to any number of different owner, but the downside is that you can't have scene

object references assigned within it *(this is a typical Unity restriction)*.

You always have the option to *Bind an Asset* graph, or to *Save a Bound* graph as an Asset file at any time through the editor GUI, so you don't have to worry about this too much from the start.

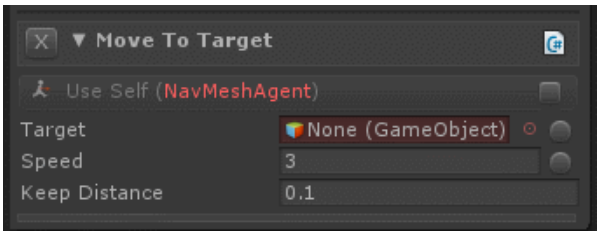
All GraphOwner components, have two common options for you to set in the inspector, those being the OnEnable and the OnDisable action, which respectively represent what will happen when the gameobject is enabled and is disabled. By default the options are the Enable Behaviour and Disable Behaviour respectively, but you are free to change these if you so require.

Finaly, whenever you attach a GraphOwner component, a Blackboard component is also attached automaticaly if there is none attached already. The Blackboard is used to store variables that can be used within any of the graphs attached on the same gameobject, but more on Blackboards and variables can be read in Using the Blackboard section later on. Only one Blackboard should be used per game object.

1.3. The "Self" Parameter

By default, a Task will execute for the the same agent as the one that the behaviour is executed for, or in other words the game object that the GraphOwner is attached to. You are though able to override this through the task inspector and instead specify another agent for this task to work with. You have two options when overriding the agent.

- Assign a reference directly.
- Select a Blackboard variable, either of the type required or of GameObject type.

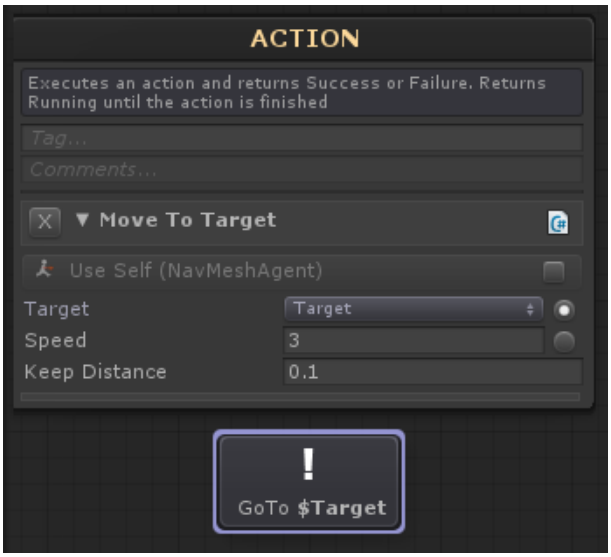


On the task inspector, the type in the parenthesis state the type that the agent needs to have in order for the task to work correctly and without errors. If that type of component does not exist on the current agent, it will show in red.

1.4. BBParameters

In most places where you are able to set a value in NodeCanvas, like for example within Tasks or Nodes, you will encounter what they are called **BBParameters**. BBParameters allow you instead setting a value directly, to link a Blackboard variable instead. This is a very important feature to remember.

This is done by clicking on the radio button on the right of the parameter and a dropdown list will appear.



There are three possible different choice you can have within this dropdown.

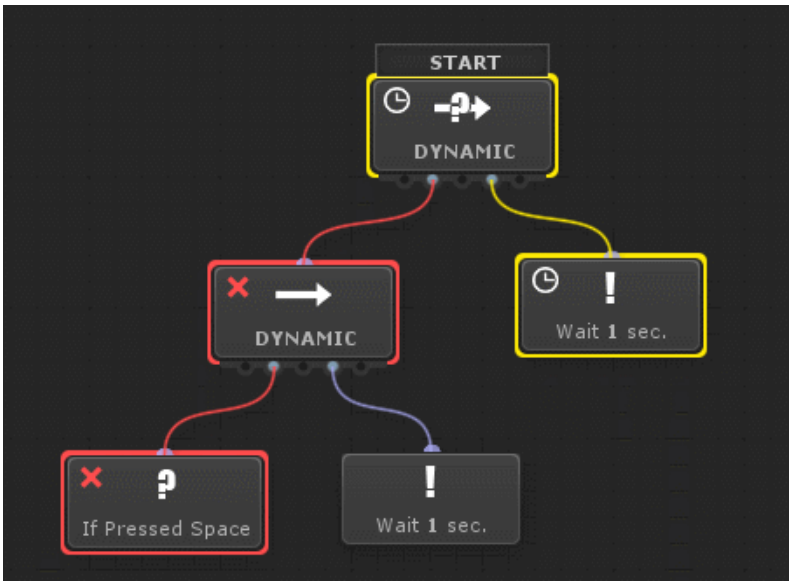
- Select a Blackboard variable to link to. This is the most obvious option.
- Specify a “Dynamic Variable” which allows you to create a variable at runtime.
- Select a Global Blackboard variable if you have a global blackboard in your scene.

When you have selected a variable this way, you will notice that the information written relevant to that task and that variable will come with a “$” token in front of the name of the variable. This is always added to indicate that this name refers to a variable and thus not to be confused with it being just a string.

1.5. Visual Debugging

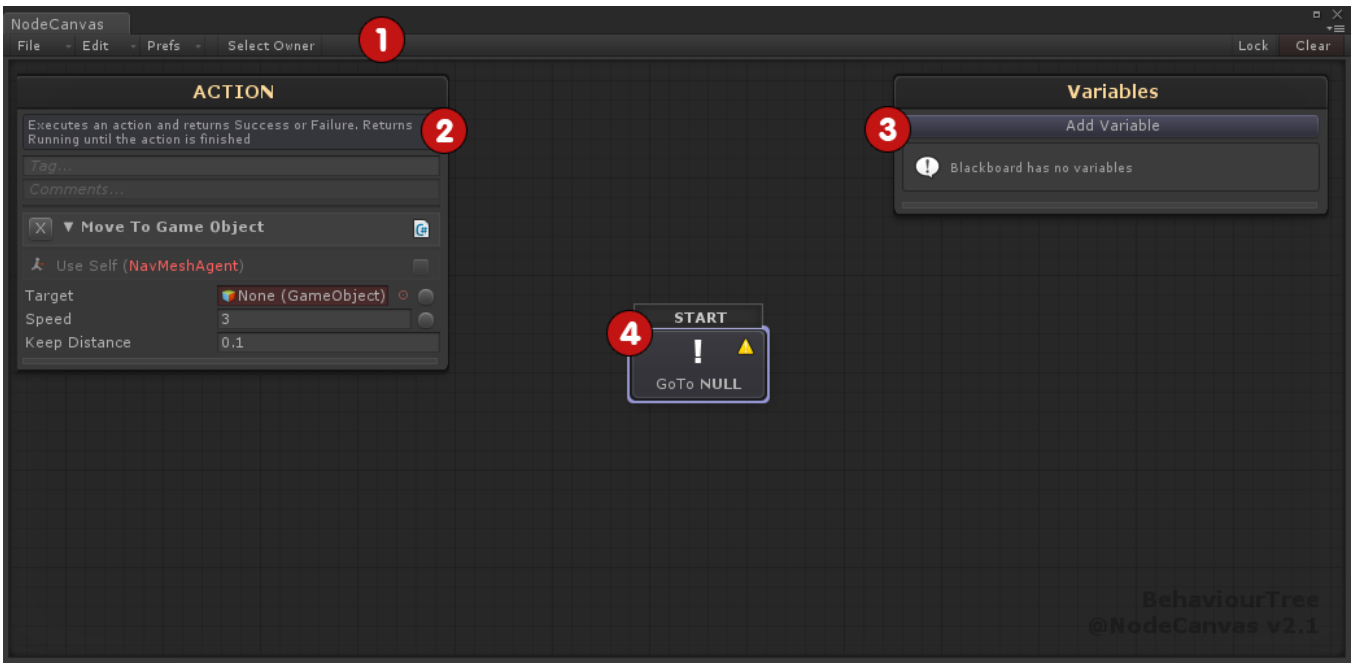
NodeCanvas has a rather informative and advanced runtime visual debugger. In play mode and while a behaviour is running, you are able to see exactly what’s going on within the graph editor window, where nodes and their connections will get highlighted with specific colors to indicate their status. In the context of behaviour trees, these have a more specific meaning as follows.

- **Blue** is for Resting.
- **Yellow** is for Running, along with a clock icon.
- **Green** is for Success, along with a checkmark icon.
- **Red** is for Failure, along with a cross icon.



Also remember, that you can Live Edit the graph while it's running as much as you wish. Adding nodes, connecting disconnecting etc.

2. The Editor



1.Toolbar

File

- Import JSON: Let's you import a graph exported previously in json format.
- Export JSON: Export the graph in a json file format.

Edit

- Bind To Owner: Binds the current graph to the GraphOwner.
- Save To Asset: Saves the current graph to an .asset file.
- Create Defined Variables: Fills the blackboard will all defined variables found in the current graph.

Prefs

- Icon Mode: Toggles between icon or text only modes for the nodes
- Show Summary Info: If true, will show the full summary info of what a task assigned on a node will do, otherwise only the name of the assigned task will be shown.
- Show Node Help: Will toggle the node help text on and off.
- Show Comments: Will toggle node and graph comments on and off.
- Grid Snap: Toggles grid snapping for the nodes.
- Automatic Hierarchy Move: If true, when moving a node, all of it's children will move as well. (This is also possible with the SHIFT key pressed down when moving a node)
- Curve Mode: A preference between smooth curves or stepped lines for connections between nodes.

2.Node Inspector Panel

The selected node's options/settings and parameters will show up into this popup panel if a node is selected. This panel is not shown if no node is selected. At the top, the node help will show if the relevant preference is turned on, as well as common options for the node's name, tag and comments. Clicking on the header of the panel will minimize/maximize it.

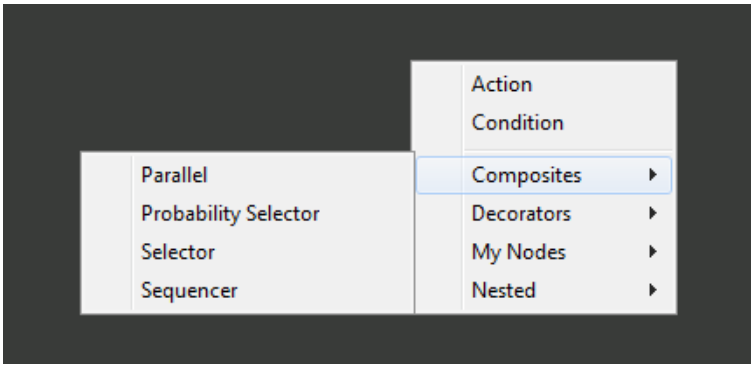
If a node has a task assigned, the task inspector is also shown here.

3.Blackboard Variables Panel

The variables of the blackboard will show here. Clicking Add Variable will open up the relevant context menu for adding a new variable. Clicking on the header of the panel will minimize/maximize it.

4.Canvas

This is where everything takes shape. Right clicking in the canvas will show a popup for adding new nodes. Either behaviour tree nodes of FSM nodes, depending on what you are working with now.



All nodes have a title name or icon and show information as of what they will finaly do. One node is always marked as "START". This is the first node to execute. You can change the start node by right clicking on the node and select "Make Start".

2.1. Controls & Shortcuts

Here are the node/canvas controls and shortcuts:

- **Right Click** on the canvas to add a new node.
- **Click & Drag** to pan the node.
- **Middle Click & Drag** to pan the canvas.
- **ALT + Click & Drag** to pan the canvas. This is usefull in laptops without a middle mouse click!
- **Shift + Click & Drag** a node to pan it and all of it's children together; panning a branch.
- **Click & Drag** from a port over a target node to connect. Right click a connected port to disconnect.
- **Click & Drag** from a port into empty canvas space, will show context menu to automaticaly create and connect a new node.
- **Click** a connected port or connection itself to select the connection.
- **Delete** the selected node or connection with Delete Key, or through the right click context menu.
- Duplicate a node with **Control+D** or through the right click context menu.
- **Double Clicking** a node will open it's script in the IDE, unless it's a Nested Node in which case it will open the nested tree.
- **Relink** a connection by click and dragging it over a new node.
- Hit **"F"** will focus the canvas on the center of all your nodes within it.
- **Mouse Scroll Wheel** will Zoom In/Out the canvas.
- **Left Click and Drag** in the canvas, will allow you to create a selection rectangle.
- **Holding Control** while making a rectangle selection will create a Canvas Group.

On right click on a node or connection link, a context menu will show up. Depending on the node or connection type there might be different extra options. Following are the common operations:

- **Delete**: Deletes the Node or Connection. *(available in multiselection as well).*
- **Copy Node**: Copies the Node possible to Paste later. *(available in multiselection as well)*
- **Copy Assigned Task**: Will copy the task that has been assigned to the Node or Connection.
- **Paste Assigned Task**: Will paste the currently copied Task to the Node or Connection and assign it.

Copy pasting of assigned Tasks can be done even across different graphs!

2.2. Assigning Tasks

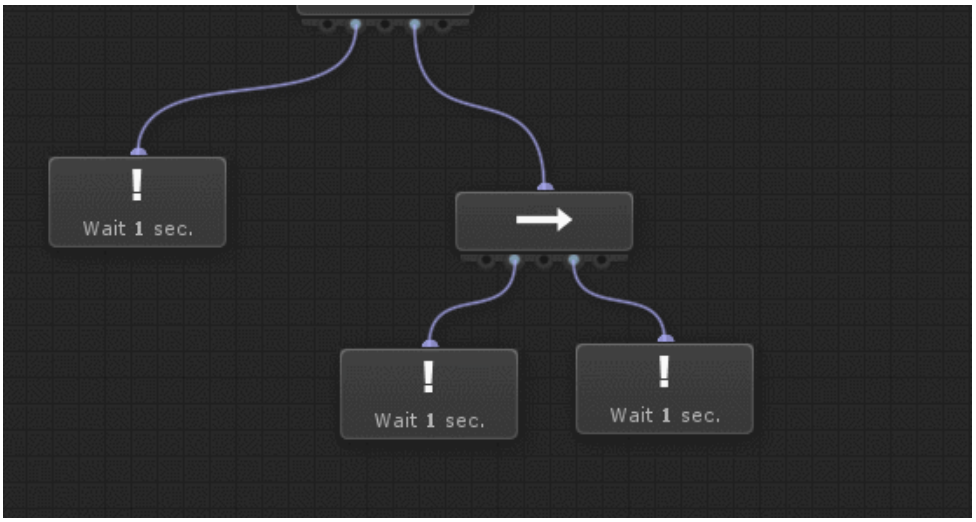
All nodes that are to perform an Action or check a Condition can be assigned an Action or Condition collectively known as Tasks. If a node is *task assignable*, the relevant button will show in it's inspector.

When that it done the inspector panel will now show that assigned action or condition inspector as well.

2.3. Canvas Groups

Canvas Groups is an editor feature that allows to group a bunch of nodes together for organization and commenting reasons.

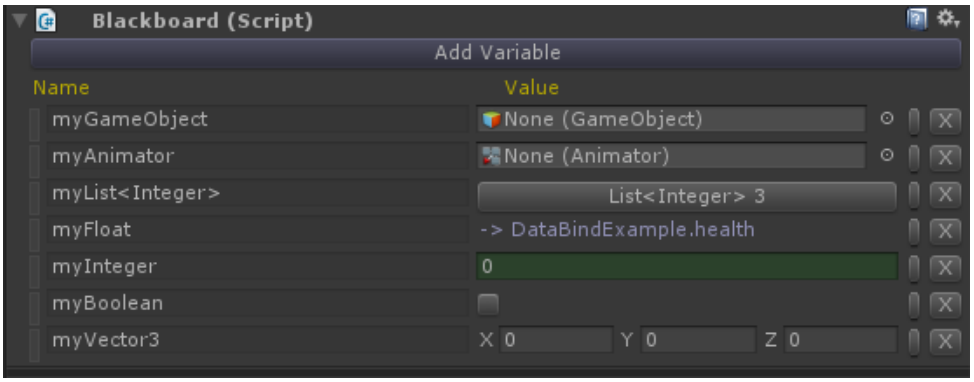
To create a Canvas Group, simply click and drag in the canvas like when doing a multi-selection and then before release the mouse button, hold down Control.



You can then right click on the title to rename the group, delete it or select all of the nodes inside. You **can** have nested canvas groups within other canvas groups if you like.

If you want to move the canvas group without moving the nodes within for re-organization reasons, you can **hold down shift** while dragging the canvas group.

3. The Blackboard



The Blackboard is a place in which you store variables into and then read/write to those variables for any purpose. When you attach any GraphOwner on a gameobject, a Blackboard is also attached for you automaticaly. This represents the Blackboard of the GraphOwner and as such any graph that will be used by that owner, will be using this blackboard to exchange variables between nodes and tasks, but you are also free to read/write to those variable directly with an easy API.

Blackboards can store **any type of variable!** In editor and when you click the “Add Variable” button though, it will seem that only a specific list of types will show up. To modify this list of types, all you have to do is open up the Preferred Types Editor and modify the list of types there. This is simply done for editor convenience and to not clutter the menus.

Blackboards are also able to **Save and Load** their variables state, which can prove usefull for creating persistant states between differrent gaming sessions or even complete save systems.

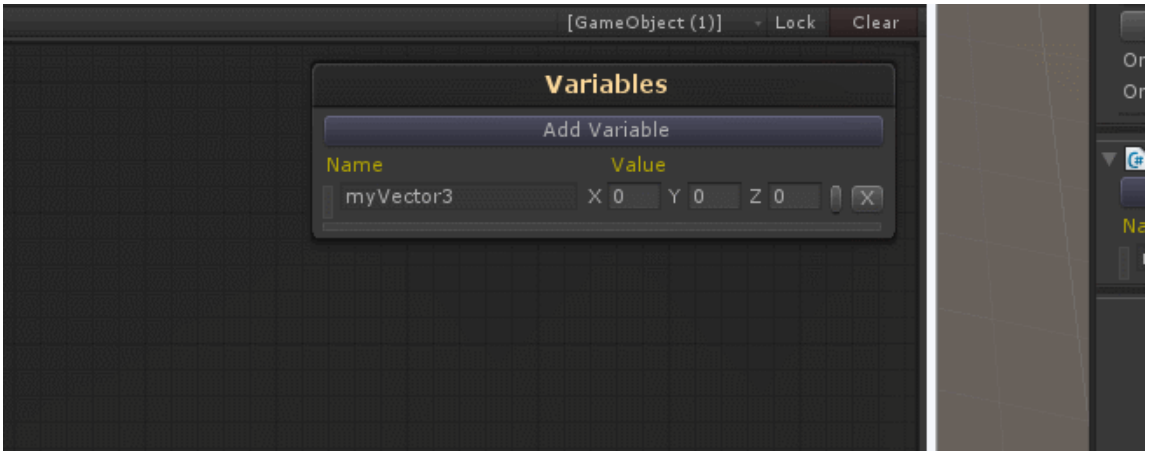
Variables are also able to be **Data Bound** to properties on any component attached on the same game object that the blackboard is attached to. This is extremelt usefull to create a direct bridge of a variable to a property both

back and forth.

3.1. Data Binding Variables

Variables added in a Blackboard, can be data binded to public properties of components that exist on the same game object as the Blackboard. Data binding can be one-way or even two-ways, depending on whether the property can read and can write. Data binding a variable to a property, means that whenever you are getting the value of the variable, you will be getting the value of the property instead, and whenever you are setting the variable value, you will be setting the property value instead. This is very powerful because it allows you to use Blackboard variables as a direct bridge to your own code, without having to set the variable before using it.

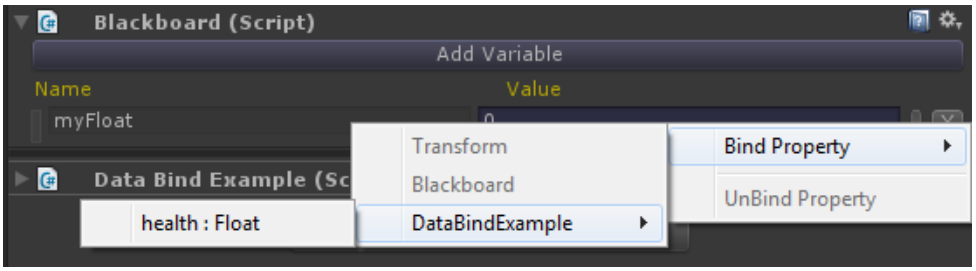
To bind a variable to a property, you simply have to click the variable *options button*, and select the property you want to bound it to!



As a more practical example, lets suppose you have this simple script attached to the same game object as a Blackboard.

```
using UnityEngine;

public class DataBindExample : MonoBehaviour {

 [SerializeField]
 private float _health;

 public float health{
  get {return _health;}
  set {_health = value;}
 }
}
```

And on that Blackboard, you have a float type variable. Clicking the variable options button, you will get a context menu to bind the variable with any float type property existing on any component on the game object. In this case, our DataBindExample component.



3.2. Save & Load

A Blackboard can Save and Load it's variables state in PlayerPrefs with ease. This means that you can possibly save the whole state of a Behaviour, or use it for any other purposes, like for example creating a save system. All the variables of a blackboard are able to save and load.

This can be done in code with these two simple methods.

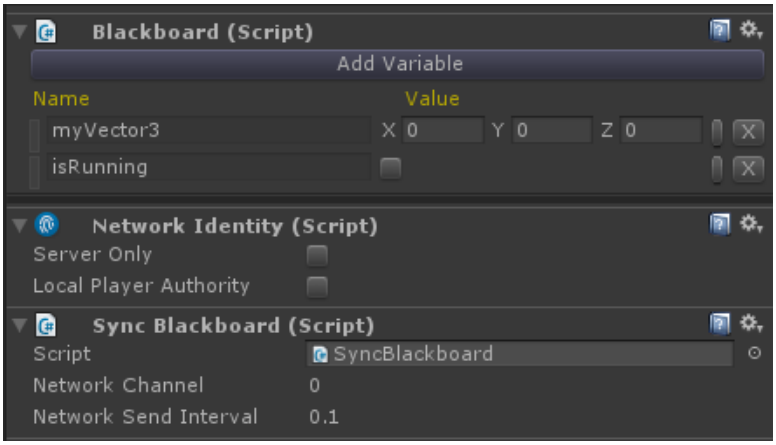
string Save(string savekey)
Save the blackboard state in PlayerPrefs at the gived saveKey. Returns the json string serialized.

bool Load(string saveKey)
Load the blackboard state previously saved in PlayerPrefs of the provided saveKey. Returns whether or not Load was successfull.

Alternatively, this can also be done with the included actions found under the "Utility" category.

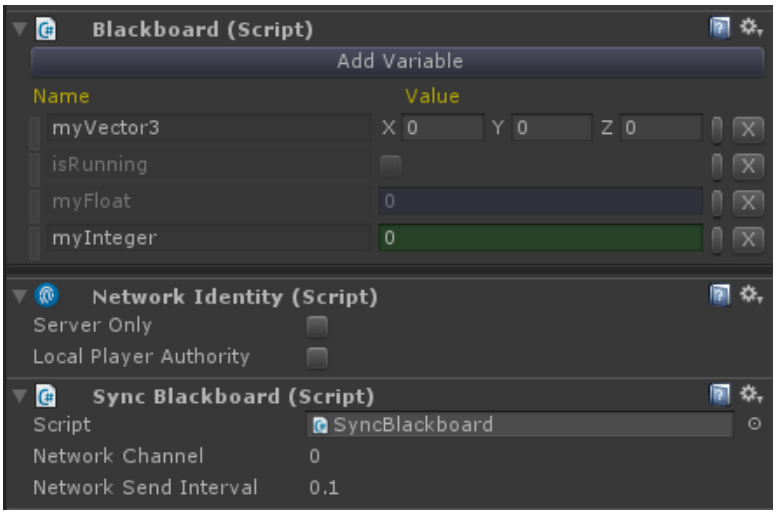
3.3. Network Sync

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

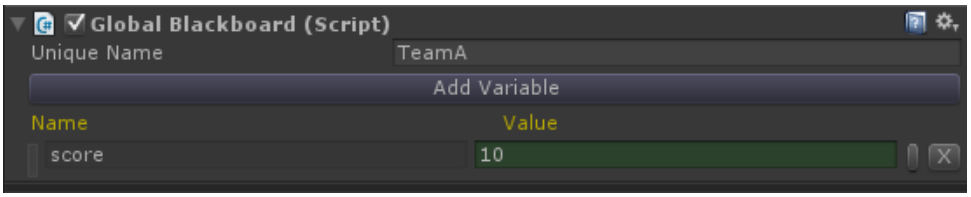
specific variable from being synced, you have to mark it as *protected*. This can simply be done so through the variables *options button*. Protected variables are also shown in transparent as a reminder. In the following example, “isRunning” and “myFloat” are marked as protected.



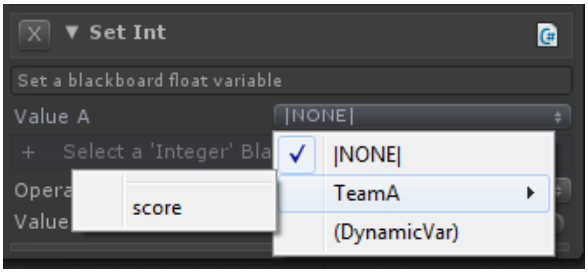
3.4. Global Variables

You can have global variables by creating Global Blackboards through the “Window/NodeCanvas/Create/GlobalBlackboard” menu. Global Blackboards are added in the scene and as such you can have scene object references assigned in their variables. Global Blackboards are persistant between loading scenes (DontDestroyOnLoad) and you can have as many as you wish.

For each GlobalBlackboard in your scene though, you **must** specifiy a completely unique name in it’s inspector.



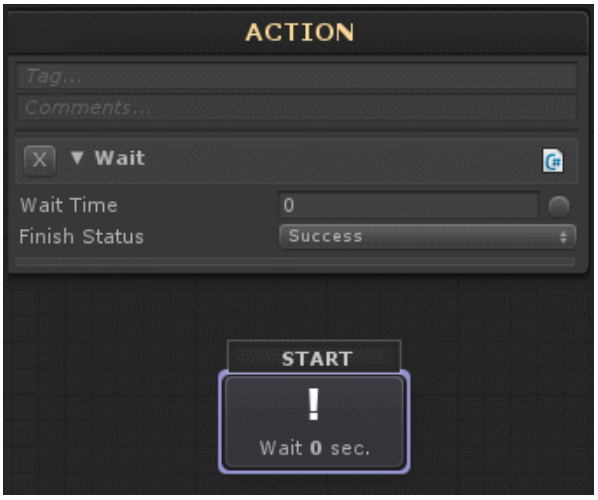
Later on, whenever you encounter a BBParameter for variable selection, you will see the global variables categorized underneath the GlobalBlackboard’s name as illustrated bellow.



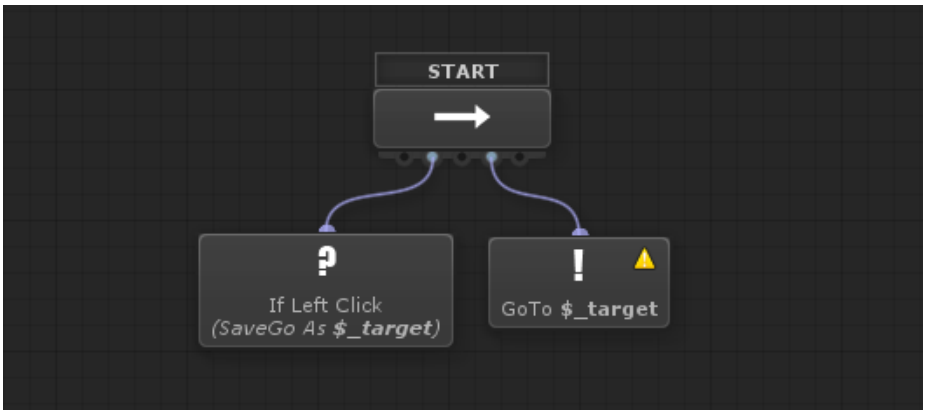
3.5. Dynamic Variables

You will notice that when you are to select a variable from the dropdown there is an option for *(DynamicVar).*You

will often want to use a variable to communicate some data between two tasks in a small scope and it would be tedious as well as unorganized to create a variable in the blackboard for such temporary use cases.



Using a Dynamic Variable, allows you to directly input the name of a variable which will be created at runtime when it's needed, without having one predefined in the blackboard. This is a feature that requires good naming organization on your part, but it's useful nontheless.



4. Tasks (Actions & Conditions)

Tasks are the actual Actions and Conditions; the what and if happens. In NodeCanvas, Tasks are assigned to nodes instead of a node itself containing the functionality. This allows for a great modularity and flexibility in design, since it decouples the design from the actual implementation as well as allows the same Task to be used for a variety of different reasons, since the functionality is encapsulated within it. This is a core concept of NodeCanvas.

There are a variety of Tasks included with the package, but of course it is also very easy to create your own Tasks, very customizable using an easy API.

With NodeCanvas, you have two broad options for implementing your own functionality, meaning mostly Actions and Conditions:

- You can create your own Tasks.
- You can keep your functionality in your own MonoBehaviours and use the *Script Control Tasks* to interface with it.

4.1. Creating Custom Tasks

NodeCanvas is build around the concept that the nodes of the graph are responsible only for the flow and not what and if it happens. Tasks are responisble for that and thus there are two kinds of Tasks those being Action Task and Condition Task. Tasks are assigned to nodes that require them and thus Tasks can be reusable between different types of systems. This concept allows to decouple the graph design from the actual implementation of actions or conditions thus allow for a non-destructive workflow.

Action Tasks

In short, to create an Action Task you must derive from the ActionTask base class and override the virtual methods as needed. When the Action is complete you must call EndAction(bool) to end the Action either in success or failure depending on the bool argument passed and that’t it!

Following is the core API that you can use to create your own action tasks.

```
//This is the agent for whom the action will take place.
public Component agent {get;}

//This is the blackboard that you can use to read/write variables manualy if needed.
public IBlackboard blackboard {get;}

//This is the time in seconds the action is running.
public float elapsedTime {get;}

//Called only the first time the action is executed and before anything else.
virtual protected string OnInit()

//Called once when the action is executed.
virtual protected void OnExecute()

//Called every frame while the action is running.
virtual protected void OnUpdate()

//Called when the action stops for any reason.
//Either because you called EndAction or cause the action was interrupted.
virtual protected void OnStop()

//Called when the action is paused comonly when it was still running while the behav
iour graph gets paused.
virtual protected void OnPause()

//Send an event to the behaviour graph. Use this along with the CheckEvent condition
.
protected void SendEvent(string)

//Similar to above, but sends a value along with the event.
protected void SendEvent<T>(string, T)

//You can use coroutines from within action task as normal.
protected Coroutine StartCoroutine(IEnumerator)
```

```
//You must call this to end the action. You can call this from wherever you want,
//although typicaly is done in either OnExecute or OnUpdate
public void EndAction(bool)
```

Examples

A simple action would be like this:

```
using UnityEngine;
using NodeCanvas.Framework;

public class SimpleAction : ActionTask{

 protected override void OnExecute(){
  Debug.Log("My agent is " + agent.name);
  EndAction(true);
 }
}
```

A simple delay would be like this:

```
using UnityEngine;
using NodeCanvas.Framework;

public class WaitAction : ActionTask {

 public float timeToWait;

 protected override void OnUpdate(){
  if (elapsedTime > timeToWait)
   EndAction(true);
 }
}
```

Condition Tasks

In short, to create a Condition Task you must derive from the ConditionTask base class, override the virtual OnCheck method and return the result.

Following is the core API for creating custom condition tasks:

```
//This is the agent for whom the action will take place.
public Component agent {get;}

//This is the blackboard that you can use to read/write variables manualy if needed.
public IBlackboard blackboard {get;}

//Same as action task.
```

```
virtual protected string OnInit()

//This is called when the condition is checked. Override and return true or false.
virtual protected bool OnCheck()

//This is a helper method to make the condition return true or false from outside th
e OnCheck method.
protected void YieldReturn(bool)
```

Examples

A simple condition example.

```
using UnityEngine;
using NodeCanvas.Framework;

public class EmptyCondition : ConditionTask{

 protected override bool OnCheck(){
  return true;
 }
}
```

4.2. Using Task Attributes

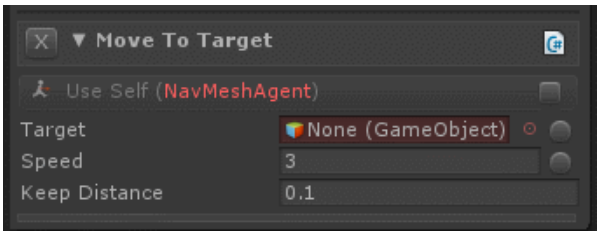
To make life easier and faster, both Action and Condition Tasks can make use of some attributes for both runtime as well as editor convenience. More specificaly:

Runtime Attributes

[AgentType(System.Type)]
You can use this attribute on the Action or Condition Task class. This specifies the Component type which is required for the agent game object to have. If no such Component exists, the Task Initialization will fail. If such a Component exists, then the agent property explained earlier, is certain to be of that type, so for example you can cast it if you want. If you don’t care for a specific type of Component, simply do not use this attribute.

Using this attribute will also show the agent field inspector which allow you to override the agent to be used from within the Editor Inspector. There are three options for which agent is going to perform the Task as follows.

- The owner agent (self), which is the default agent propagated to the Task on execution.
- An override agent reference which is directly assign.
- An override agent selected from a GameObject blackboard variable or a variable whose type is the same as the type defined with the AgentType attribute.



Notice: It is possible to define an interface type for AgentType.
Notice 2: Instead of using this attribute you can alternatively derive from the generic version of ActionTask or ConditionTask where T would be the type of component required.

[EventReceiver(params string[])]
Use this attribute also on the Task class if you want to use Unity's messages like OnTriggerEnter, OnTriggerExit etc. Specify The messages you want to listen to and then you can use them in the task as you normaly would. Here are the event messages that can be listened and used right now:

- OnTriggerEnter
- OnTriggerExit
- OnTriggerStay
- OnCollisionEnter
- OnCollisionExit
- OnCollisionStay
- OnTriggerEnter2D
- OnTriggerExit2D
- OnTriggerStay2D
- OnCollisionEnter2D
- OnCollisionExit2D
- OnCollisionStay2D
- OnMouseDown
- OnMouseDrag
- OnMouseEnter
- OnMouseExit
- OnMouseOver
- OnMouseUp
- OnBecameVisible
- OnBecameInvisible
- OnAnimatorIK

Notice: There already exist Condition Tasks for triggers, collisions and mouse events.

[GetFromAgent]
Use this attribute over a Component type field. Whenever a new agent is set (OnInit of the Task), the field will be set by GetComponent from the agent. If no such component exists, the Task Initialization will fail. This is a helper attribute to save you from doing this manually.

[RequiredField]
Use this over a nullable public field to make it show red if it is indeed null or empty string in case of a string. Furthermore, if such a field is null when the Task gets executed the Initialization will fail. This saves you from doing null checks all the time.

Design Attributes

All **public** fields of both Action and Condition Tasks will automaticaly be shown within the NodeCanvas Editor Inspector for all types. There are also some attributes you can use to customize the shown cotrols as follows.

[SliderField(float min, float max)]
Use this over a public float/int or BBParameter<float/int> field to show a slider instead.

[TextAreaField(float height)]
Use this on top of public string or BBParameter<string> field to show it as a text area control instead.

[TagField]
Use this over a public string r BBparameter<string> field to show a TagField control instead.

[LayerField]
Use this over a public int r BBParameter<int> field to show a LayerField control instead.

Notice 1: All public inherited fields will show up for inspection as well.
Notice 2: If you really need a very special editor you can override the virtual protected void OnTaskInspectorGUI() and wrap it within an #if UNITY_EDITOR

Meta-Information

[Category(string)]
Use this on the Task class to specify a category to be shown in. Notice that you can use subcategories by using "/"

[Name(string)]
If your class name is too weird for some reason or you just want to show by a specific name, use this attribute over the Task class.

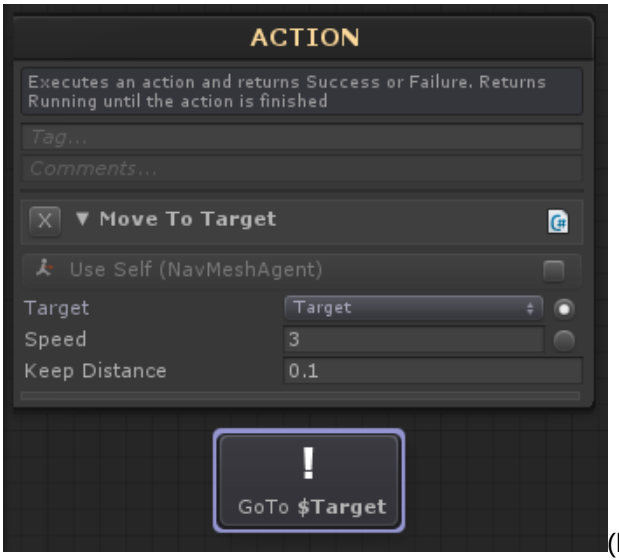
To specify the information shown when assigned on a node, you will have to override the 'virtual string **info'** property and return the string you would like to show, meaning what the Task is going to finaly do or check. This is very useful for getting a good overview of what will happen directly, when looking at the behaviour in the editor. If the property is not overriden then the Task name will simply show instead.

4.3. Using BBParameters in Tasks

BBParameters are a great way of making your tasks be more dynamic by having the ability to use blackboard variables in place of simple value fields within your task. In short, when declaring a field of some type, you would instead use BBParameter<T>, which will allow you to either select a Blackboard variable or define a value directly. Here is the same delay example as before, but using a BBParameter<float> instead.

```
using UnityEngine;
using NodeCanvas.Framework;

public class WaitAction : ActionTask {

 public BBParameter<float> timeToWait;

 protected override void OnUpdate(){
  if (elapsedTime > timeToWait.value)
   EndAction(true);
 }
}
```

The following “Move To Target” action task for example, has two BBParameters defined, those being “Target” and “Speed”. Here, Target has been linked to a Blackboard variable, while Speed is set directly to 3.



(http://paradoxnotiongames.com/wp-content/uploads/BBParameter.png)

Remember that you also have access to the Blackboard directly, through the inherited .blackboard property if you so require to read/write manualy to it’s variables.

4.4. Creating Generic Tasks

NodeCanvas uses a lot of generics and you also have the ability to create and use generic action and condition Tasks, which make things much more reusable without the need to write counteless of tasks doing the same thing.

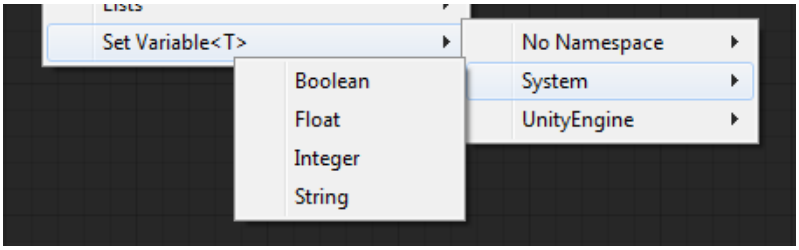
This works well with the fact that BBParameter is also a generic type definition. With that in mind, following is the code of the included SetVariable<T> action task.

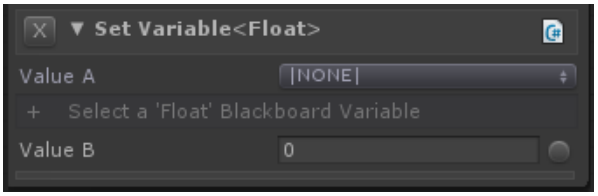
```
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.Tasks.Actions{

 [Category("? Blackboard")]
 public class SetVariable<T> : ActionTask {

  [BlackboardOnly] [RequiredField]
  public BBParameter<T> valueA;
  public BBParameter<T> valueB;

  protected override string info{
   get {return valueA + " = " + valueB;}
  }

  protected override void OnExecute(){
   valueA.value = valueB.value;
   EndAction();
```

```
  }
 }
}
```

Then, in the editor when you are to select an ActionTask, the generic task will be shown as a category where you will be able to select the type of T. Some things to keep in mind:

- The types shown in the editor will be those that are set in the Prefered Types Editor Window.
- The types shown will be constrained in case T has a constrain.
- Only one generic parameter is allowed.





Finaly here is another example code from the included AddElementToList<T> action task.

```
using System.Collections.Generic;
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.Tasks.Actions{

 [Category("? Blackboard/Lists")]
 public class AddElementToList<T> : ActionTask{

  [RequiredField] [BlackboardOnly]
  public BBParameter<List<T>> targetList;
  public BBParameter<T> targetElement;

  protected override void OnExecute(){
   targetList.value.Add(targetElement.value);
   EndAction();
  }
 }
}
```

4.5. The Script Control Tasks

NodeCanvas comes with various Tasks that allow you to make use of existing code and components on game objects, that are well worth mentioning.
By using these Tasks a whole lot of different actions can be performed that at the first glance you might think they are missing. Here are a few things that can be done for example by using these Tasks:

- Translate an object
- Change an object's name
- Check if a CharacterController isGrounded
- Check if a RigidBody isKinematic
- Send a Message to a game object
- Stop an AudioSource
- Set an AudioSource's volume
- and so on…

Furthermore you can keep all your implementations in your own scripts instead of creating custom action and condition tasks, and use these Script Control Tasks to interact with them in an efficient way. Following are the Script Control tasks included.

Actions

Execute Function

This Action will allow you to Execute a Function that takes none or up to three parameters on any Component/Script on the selected Agent and also able to save the return value of that function to a blackboard variable. The value to be used as the parameters can either be directly assigned or taken from blackboard variable.

This Action ends immediately in Success unless the function selected is an IEnumerator is which case it will be executed as a **Coroutine**! In this case, the action will be running and only return Success after the Coroutine is done.

Implemented Action

With the Implemented Action Task, you are able to totaly control for how long the action will run and what Status it will return. In essence it forwards the execution of the Task for a function on a script to control. That function must have a signature of public Status and take one or no parameters. For example:

```
using NodeCanvas.Framework;

public class Example : MonoBehaviour {

 public Status SomeAction(string text){

  Debug.Log(text);
  return Status.Success;
 }
}
```

Get Property

This Action will allow you to Get a property from any of the selected Agent's components and store that property as a blackboard variable.

Set Property

This Action will allow you to set a property on any of the selected Agent's Components.

Get Field

This Action will allow you to Get a field from script on the Agent and store it as blackboard variable.

Set Field

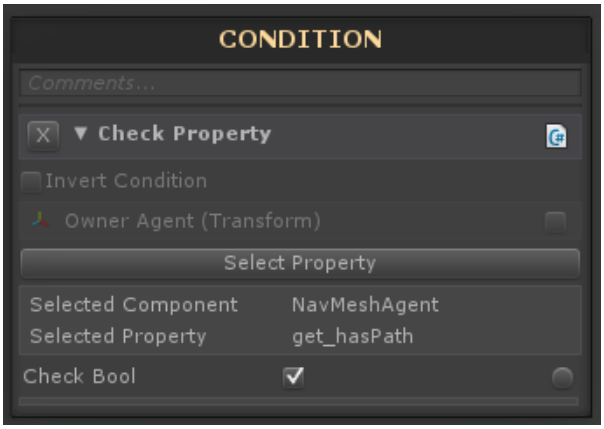
This Action will allow to Set a field on any of the selected Agent's components.

Conditions

Check Function

This Condition will allow you to call a boolean function with no parameters and check it directly against a value

Check Property



This Condition will allow you to check a boolean property directly against a value, saving you from having to first Get it and then Check it.

Check CSharp Event

This Condition can be used to subscribe to an event of EventHandler type or custom handler with 0 arguments and return type of void. As soon as the event is raised, this condition will return true for that specific frame.

5. Behaviour Trees

Overview

Behavior Trees are commonly used in modern games and applications, to create dynamic behaviours by using small building blocks (nodes) that perform a certain task and control the flow of the behaviour for one or even more AI agent. Behaviour Trees can be though as a simplified programming language, that though contains the most common down to the core of the operations needed such as conditions, actions, loops, sequencer and selections to name a few. NodeCanvas allows you to make such Behaviour Trees visualy, without the hassle or the need to understand the whole low level underline logic, but rather only the top high level one.

On each update (called a Tick) of the Behaviour Tree, it starts by executing the first node. That node will then perform a small task and return it's **status,** either **Succeeded, Failed**, or not yet decided, meaning **Running.** Depending on the type of node that decision may stem either from itself or be determined based on one or more of it's child nodes. As such there exist three disnstictive types of nodes a Behaviour Tree can have, those being:

Leaf Nodes

Leaf nodes have no child nodes and they are the final destination of a Behaviour Tree tick, hense their name. Most commonly leaf nodes will either check a Condition; a state of the game, or perform an Action, altering the state of the game.

Composite Nodes

Composite nodes mostly exist to determine the flow of the Behaviour, controling which leaf nodes are going to be executed or in what order for example. As such, Composite nodes may have any number of child nodes, which will "ask" on how to proceed. The most basic of Composite nodes are the Sequencer, which executes all it's child nodes in order until one Fails, and the Selector which executes all it's child nodes until one Succeeds. The most reasonable way to think of those two, would be an AND and an OR in case you are familiar with boolean logic. If you are not, that's still fine and you don't have to be to use Behaviour Trees.

Decorator Nodes

Decorator nodes are special in what each do, but generaly speaking they exists to somehow alter the functionality of their one and only child node that they can have. Common Decorators include Looping the child node for a number of times, Limiting access to a child node or Interrupting a child node's execution to name a few. NodeCanvas comes with a handfull of Decorator nodes to allow faster and more flexible behaviour creation than usual.

Further Reading

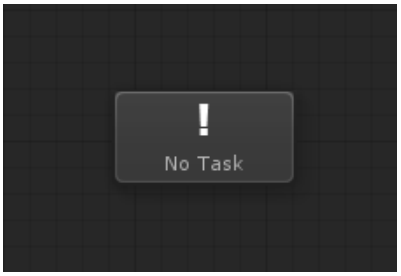
If you want to further understand the logic behind Behaviour Trees, here are a few very interesting links:
ChrisSimpson (Gamasutra)
(http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php)
AIGameDev
(http://aigamedev.com/open/article/behavior-trees-part1/)
GaveDev.StackExchange
(http://gamedev.stackexchange.com/questions/51693/decision-tree-vs-behavior-tree)
ITU.dk
(https://blog.itu.dk/MAIG-E2013/files/2013/09/2commontechniques.pdf)
cse.scu.edu(http://www.cse.scu.edu/~tschwarz/COEN196_13/PPT/Decision%20Making.pdf)
(far more than Behaviour Trees)

5.1. BT Nodes Reference

Leafs

Leafs have no children at all and they reside by the end of a branch.

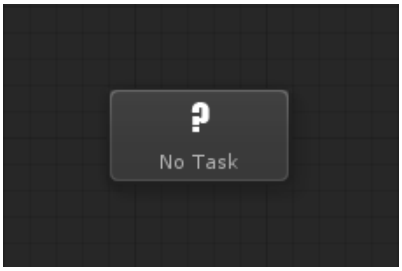
Action



The Action Node will execute an Action Task assigned. The Action node will return Running until the Action Task is finished at which point it will return Success or Failure based on the Action Task assigned. *NC comes with a variaty of different Action Tasks to use.*

- **Success**: When assigned Action Task finish in Success
- **Failure**: When assigned Action Task finish in Failure
- **Running**: While assigned Action Task has not finished.

Condition



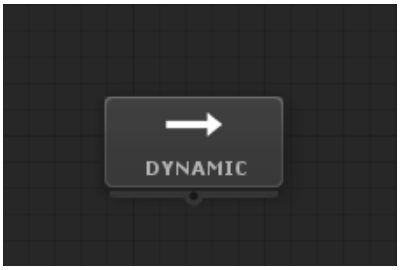
The Condition Node will execute a Condition Task and return Success or Failure based on that Condition Task. *NC comes with a variaty of different Condition Tasks to use.*

- **Success**: If assigned Condition Task is true
- **Failure**: If assigned Condition Task is false
- **Running**: Never

Composites

Composites work with multiple chid nodes and execute them in some order based on the composite functionality.

Sequencer



The Sequencer executes it's child nodes in order from highest to lowest priority (left to right). It will return Failure as soon as any child returns Failure. It will return Success if all children return Success.

A **Dynamic** Sequencer will revaluate higher priority children Status changes. So if a higher priority child status returns Failure, the Sequencer will Interrupt the currently Running child and return Failure as well. This is extremely useful to create dynamic and immediate responses to higher priority status changes like for example conditions.

A **Random** Sequencer will shuffle it's children on each reset and then start executing them in the new order.

- **Success**: When all children finish in Success
- **Failure**: When any child returns Failure
- **Running**: When current executing child is Running

Selector

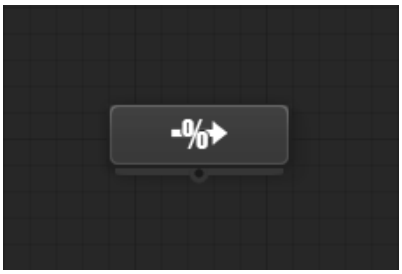
The Selector executes it's child nodes in order from highest to lowest priority (left to right). It will return Success as soon as any child returns Success. It will return Failure if all children return in Failure.

A **Dynamic** Selector will revaluate higher priority children Status changes. So it a higher priority child status returns Success, the Selector will interrup the currently Running child and return Success as well. Like the sequencer, this is extremely usefull to respond immediatly on status changes of higher priority children like for example conditions.

A **Random** Sequencer will shuffle it's children on each reset and then start executing them in the new order.

- **Success**: When any child returns Success.
- **Failure**: When all children return Failure.
- **Running**: When current executing child is Running.

Probability Selector



The Probability Selector will select and execute a child node based on it's chance to be selected. If that selected child returns Succes, the Probability Selector will also returns Success. If it returns Failure though, a new child will be picked. The Probability Selector will return Failure if no child returned Success, or may immediately return Failure if a 'Failure Chance' is introduced.

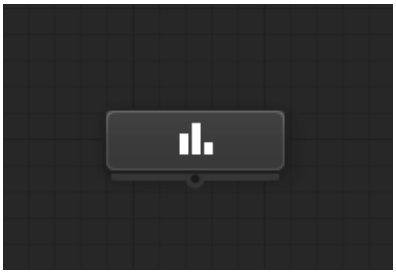
As soon as you connect a child node, a probability value will show up in the inspector for that child and the node information will read the final chances calculated based on all probability values and the direct failure chance.

- **Success:** When the selected child node returns Success.
- **Failure:** When the selected child node returns Failure, or no child is selected because of a 'Failure Chance'.
- **Running:** When the selected child node returns Running.

Note: The probability values can be taken from blackboard variables to dynamicaly alter the probabilities.

Priority Selector

Priority Selector

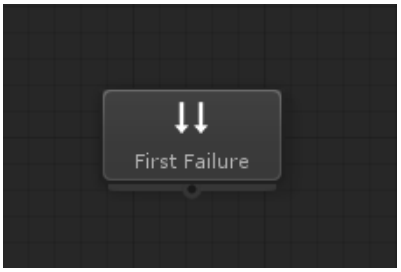


The Priority Selector is similar to a normal Selector but will pick the order of child execution based on the priority each child has. As soon as you connect child nodes, Priority weights will show up which you can alter either directly or through Blackboard variables.

- **Success:** When the first child returns Success.
- **Failure:** When all children return Failure.

- **Running:** When the current child is Running.

Parallel



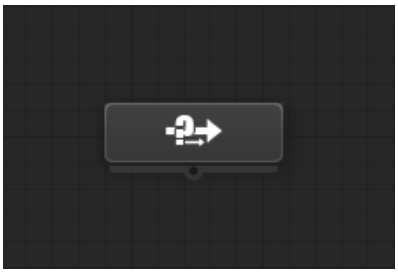
The Parallel will execute all it's children simultaneously. It can be set to have the policy of a Sequencer or a Selector like so:

If set to **First Failure**, then as soon as any child returns Failure, the parallel will reset all current Running children and return Failure as well. Else it will return Success when all children have returned Success.

If set to **First Success**, then as soon as any child returns Success, the parallel will reset all current Running children and return Success as well. Else it will return Failure when all children have returned Failure.

- **Success**: If set to 'First Success', when any child returns Success. If set to 'First Failure', when all children finish in Success.
- **Failure**: If set to 'First Failure', when any child returns Failure. If set to 'First Success', when all children finish in Failure.
- **Running**: When any child is Running.

Flip Selector



The Flip Selector works like a normal Selector, but once a child node returns Success, it is moved to the end (right). As a result previously Failed children will always be checked first and recently Successfull children last.

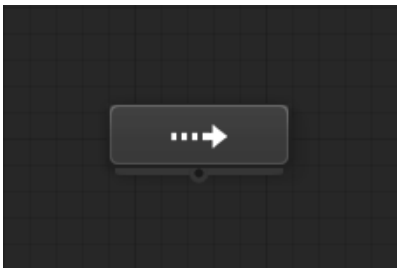
- **Success:** When a child returns Success.
- **Failure:** When all children return Failure.
- **Running:** When the current child is Running.

Switch

The Switch node can switch either an Enum or an Integer value. Depending on what the current Enum or integer value is, it will execute the respective child node. If another child node was previously Running, it will be interrupted. Once you connect child nodes, you the connections will read the enum or integer values.

- **Success:** When the current child node returns Success.
- **Failure:** When the current child node returns Failure, or if the enum or integer value is out of range in respects to the number of child nodes connected.
- **Running:** When the current child node returns Running.

Step Iterator



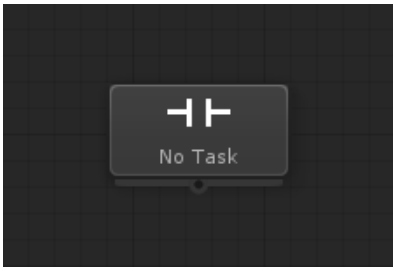
The Step Iterator is like a Sequencer with the important difference being that it does not execute all it's children in one go, but instead, once a child node returns Success, the Step Iterator will return Success as well. The next time the Step Iterator is executed it will execute the next child in order. If a child returns Failure, it will try the next child immediately. If all return Failure it will return Failure as well.

- **Success**: When a child node returns Success.
- **Failure**: When all children return Failure.
- **Running**: When the current child is Running.

Decorators

Decorators always have one child node. They usualy give extra functionality to that child, filter it or modify it in some way.

Interruptor



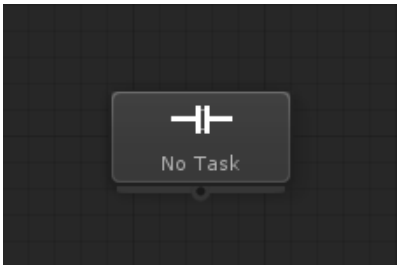
Interuptor gets assigned a Condition Task. If the condition is or becomes true, the child node will be interrupted if Running and the Interruptor will return Failure. Otherwise the Interruptor will return whatever the child node returns.

- **Success**: When the child node returns Success.
- **Failure**: When child node returns Failure, or the condition assigned is or becomes true even while the child node is Running. (Interruption made).
- **Running**: When child node returns Running.

Note: The fact that the Interruptor can be assigned any condition, makes possible for interruptions to happen due

to any reason.

Conditional

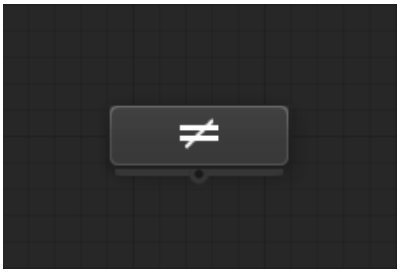


Conditional will execute it's child node only if the condition assigned is true and then it will return whatever the child node returns. It will return Failure if the condition is false but the child node is not already Running. So in other words, if the condition succeeds even for one frame, the child node will execute and not be interrupted even if the condition no longer holds true.

This node is extermely useful for creating state-like behaviours and switching between them.

- **Success**: When the child node returns Success.
- **Failure**: When the child node returns Failure, or the condition is false and the child node is not already Running.
- **Running**: When the child node returns Running.

Inverter



Inverter will remap it's child node's Success and Failure return status, to the opposite. **Success**: When the child node is remaped to Success.

- **Success**: When the child node return Failure.
- **Failure**: When the child node returns Success.
- **Running**: When the child node is Running.

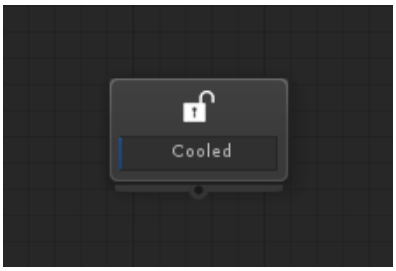
Repeater

Repeater will repeat it's child either a **Number of Times**, or **Until** it returns the specified status, or **Forever**.

- **Success**: If 'Repeat Times', when the child node returns Success on it's last repeat. If 'Repeat Until', when the child node returns Success and the required status is Success.
- **Failure**: If 'Repeat Times', when the child node returns Failure on it's last repeat. If 'Repeat Until', when the child node returns Failure and the required status is Failure.
- **Running**: For as long as it's repeating.

Note: The times to reapeat can be assigned from a blackboard variable to make this more dynamic.

Filter



Filters the access of it's child node either a specific **Number of Times**, or every specific amount of time (like a **Cooldown**). By default this node will be treated as *Inactive* in regards to it's parent node if it is filtered. Uncecking this option will instead return Failure.

- **Success**: When the child node is accessed and returns Success.
- **Failure**: When the child node is accessed and returns Failure, or 'Inactive when limited' is turned off.
- **Running**: When the child node is accessed and returns Running.

Note: The number of times and the cool down time values can be set to be taken from the Blackboard to make this node more dynamic.

Iterator

Iterator will iterate a List taken from the Blackboard. On each iteration the current iterated element will be saved on that same blackboard with the name provided and the child node will be executed.

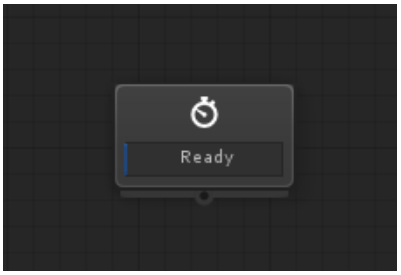
The Iterator can optionaly be set to terminate iteration as soon as the decorated node returns either Success or Failure. If no termination condition is set *(NONE)* or if the list is iterated and the termination condition is not met, then the Iterator will return whatever the last iteration child execution returned.

If Reset Index is checked, then the Iterator will reset the current iterated index to zero whenever it resets, else the index will remain unless it's the last index of the list. Think of it as a 'for each'.

- **Success**: When the list is iterated and the child node returns Success, or if set to 'First Success', the first time that the child node returns Success.
- **Failure**: When the list is iterated and the child node returns Failure, or if set to 'First Failure', the first time that the child node returns Failure.

- **Running**: For as long as it's iterating.

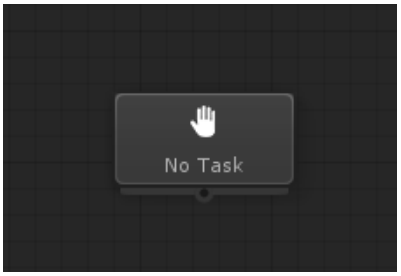
Timeout



The Timeout decorator will interrupt the child node Running if it is Running for more time than the one specified in seconds. Otherwise it will return whatever the child node returns.

- **Success**: When the child returns Success.
- **Failure**: When the child returns Failure or has Timed Out.
- **Running**: When the child returns Running.

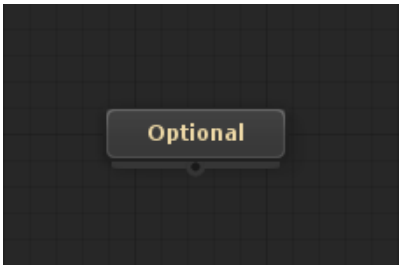
Wait Until



Wait Until will return Running until the assigned Condition Task becomes true. If the condition becomes false after the child has been ticked, it will not interrupt it. The condition is check only when the child is not already Running.

- **Success**: When the child returns Success.
- **Failure**: When the child returns Failure.
- **Running**: When the child returns Running OR the condition is false.

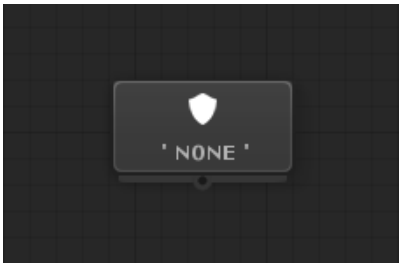
Optional



The Optional Decorator executes the decorated child without taking into acount its Success or Failure return status, thus making it optional to the parent node in regards to status expected.

- **Success**: Never.
- **Failure**: Never.
- **Running**: When the child node is Running.

Guard



Protectes the decorated child from Running if another Guard with the same specified token is already guarding (Running) that token. Guarding is global for all of the Behaviour Trees running for the same agent.

When protected, it can be set to either return Failure or Running.

- **Success**: When the child returns Success.
- **Failure**: When the child returns Failure or protection is active and is set to return Failure when protected.
- **Running**: When the child returns Running or protected is active and is set to return Running when protected.

Override Agent

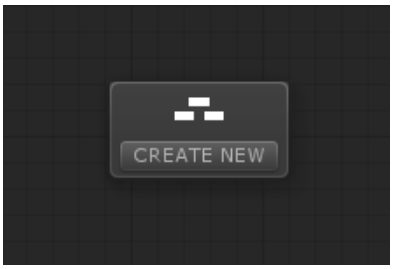
Override Agent will set another Agent for the rest of the Behaviour Tree from it's point and on from the game object directly selected or taken from a Blackboard variable. What this means is that every node underneath the this decorator will now be Ticked for that new agent. This Decorator is one of the ways that NodeCanvas allows you to control more than one agents from a single "master" Behaviour Tree, or to dynamicaly change the agent.

- **Success**: When the child node returns Success.
- **Failure**: When the child node returns Failure.
- **Running**: When the child node returns Running.

Sub-Behaviours

Sub-Behaviours are references to other whole graphs, usualy Behaviour Trees and are used for organization and behaviour modularity.

SubTree

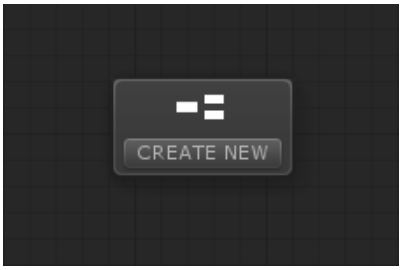


A SubTree is an entire other Behaviour Tree. The SubTree node, will return whatever the assigned Behaviour Tree's root node ("Start") returns. The agent and the blackboard of the root Behaviour Tree will be passed down to the SubTree, thus all available blackboard variables will be the same and available to the SubTree as well.

- **Success**: When SubTree's root node return Success.
- **Failure**: When SubTree's root node return Failure.
- **Running**: When SubTree's root node returns Running.

Note: You can create a new SubTree out of a whole branch! To do this, right click a composite node and select **"Convert To SubTree"**. This will create a new Behaviour Tree and a new SubTree Node. The selected node as well as all of it's children concurently will be moved to that new Behaviour Tree! This is extremely useful to organize the behaviours without knowing before hand how that organization should be.

NestedFSM



Nested FSM can be assigned an entire FSM. When executed, the FSM will Start. The NestedFSM node will return Running for as long as the FSM is running. You can specify one state of the FSM for Success and another for Failure. As soon as the nested FSM enters any of those states, the FSM will stop and this node will return Success or Failure acordingly. Otherwise it will return Success when the nested FSM is finished somehow. The Agent and the Blackboard of this Behaviour Tree will be passed to the Nested FSM, thus all available variables of the root Behaviour Tree will also be the same and available to the Nested FSM.

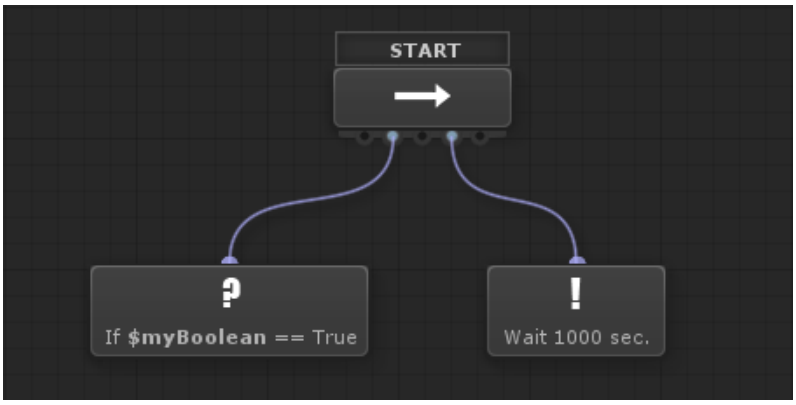
- **Success**: When the Nested FSM enters the selected Success State, or the Nested FSM is finished.
- **Failure**: When the Nested FSM enters the selected Failure State.
- **Running**: For as long as the Nested FSM is running.

5.2. Reactive Evaluation

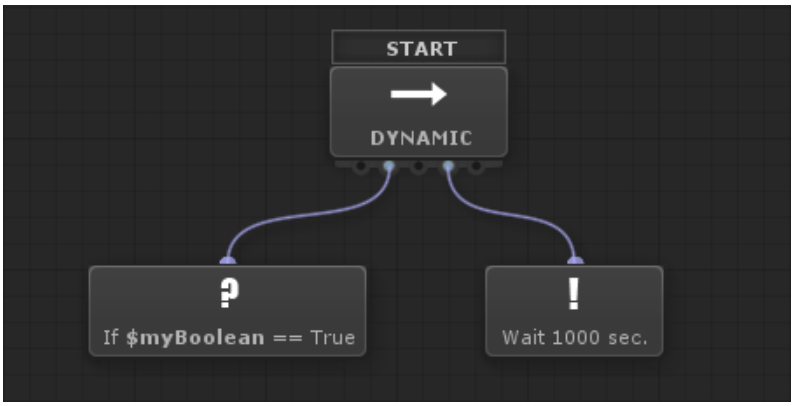
Behaviour Trees in NodeCanvas are designed with the ability to react to immediate changes which is a requirement for modern fast paced games.

Many Behaviour Tree nodes, have a special option named **“Dynamic”.** With this option enabled the node will react immediately to child node status changes. Let’s take an example for convenience:

Suppose we have this simple behaviour tree where if “myBoolean” variable is true, then we will wait 1000 seconds (exagerated on purpose), which resemples an action that can possibly take that much time. If**while** that action is Running the condition no longer holds true (myBoolean is false), then nothing will happen. The condition will not be evaluated until the next tree traversal.

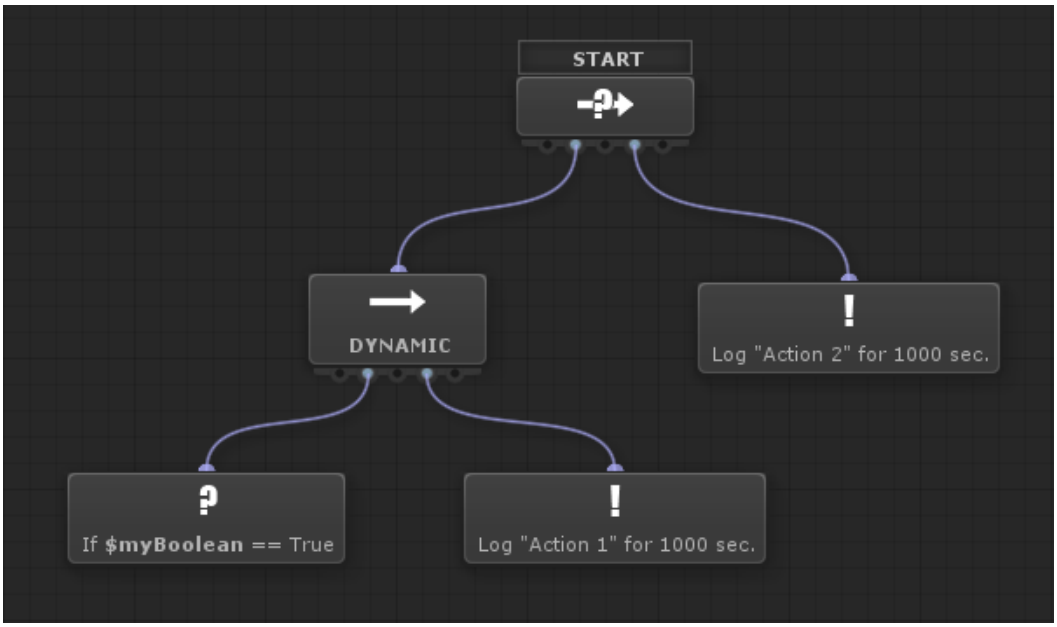


To make this Sequencer Reactive, all we have to do is to enable it’s Dynamic option. As such, **even while** the action Wait 1000 seconds is running, if the condition **becomes** false, the action will be interrupted and the Sequencer will be revaluated, which will result in it return Failure as it should.

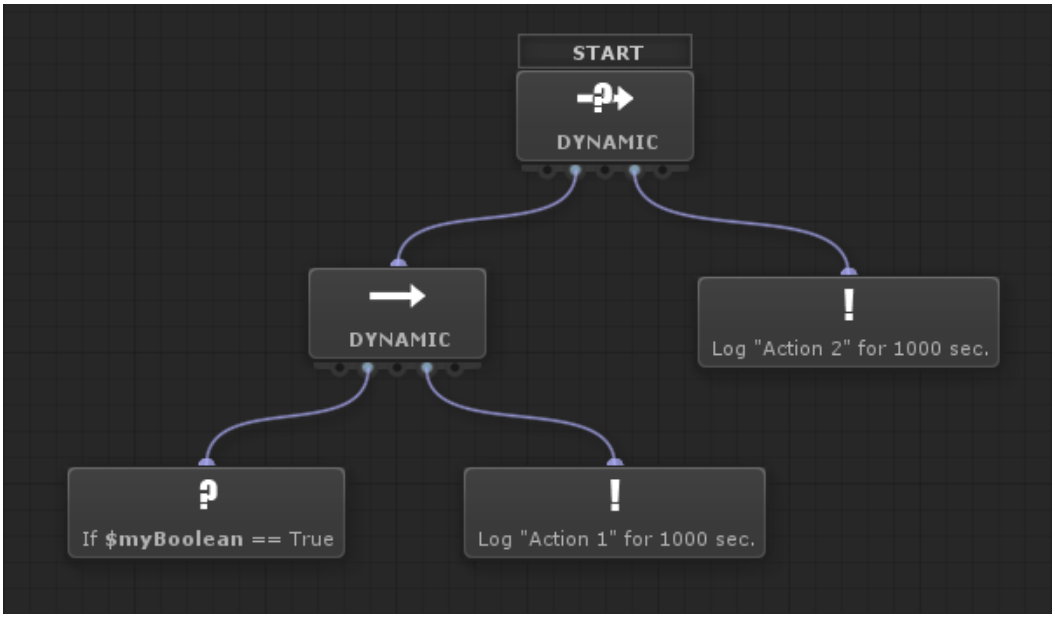


Recursive Reactive Evaluation

The dynamic option works only in the scope of the node being Dynamic. So if you had the following behaviour tree, if while “Action 2” is running, “myBoolean” becomes true, the Selector won’t really care about this change and Action2 will continue to run.



Making the Selector Dynamic as well, will solve this correctly. So in the following tree, if Action2 is running and “myBoolean” becomes true, Action2 will be interrupted and the Sequencer will take over. As a consequence, since the condition is true, Action1 will execute. If the condition later becomes false, the Sequencer being Dynamic, will interrupt Action1 and return Failure. Then the Sequencer will execute Action 2.



If you are making a game that needs reactive evaluation like for example an action or a realtime strategy game. The Dynamic option is certain to help you.

The Dynamic option can been found in the following Behaviour Tree Nodes:

- Sequencer
- Selector
- Parallel
- Conditional Decorator

5.3. Creating Custom BT Nodes

While the recommended way of working with NodeCanvas is to create Tasks and use them instead so that the design can be modular, you are also able to create your own custom Behaviour Tree nodes, either Composites, Decorators or Leafs. In either case, you have to derive from the respective base type, override the required methods and use the inherited properties.

Following is a template with most methods that you can optionaly override and are common to all three behaviour tree node types.

```
using UnityEngine;
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.BehaviourTrees{

 [Category("My Nodes")]
 [Icon("SomeIcon")]
 [Description("Some description")]
 public class NodeName : BTNode {

  //When the BT starts
  public override void OnGraphStarted(){

  }

  //When the BT stops
  public override void OnGraphStoped(){

  }

  //When the BT pauses
  public override void OnGraphPaused(){

  }

  //When the node is Ticked
  protected override Status OnExecute(Component agent, Blackboard blackboard){

   return Status.Success;
  }

  //When the node resets (start of graph, interrupted, new tree traversal).
  protected override void OnReset(){

  }

  #if UNITY_EDITOR

  //This GUI is shown IN the node IF you want
  protected override void OnNodeGUI(){

  }

  //This GUI is shown when the node is selected IF you want
  protected override void OnNodeInspectorGUI(){
```

```
   DrawDefaultInspector(); //this is done when you dont override this method anyway.
  }

  #endif
 }
}
```

Here are some important common inherited properties:

Status status
The current status of the node. You can also use this to temporary store a status as you'll see later on.

Component graphAgent
The executive agent of the Behaviour Tree

Blackboard graphBlackboard
The blackboard of the Bheaviour Tree

List<Connection> outConnections
In NodeCanvas, connections are object by themselves. You get to access child nodes *through* their connections. Connections also have an Execute method which returns a Status as you will see later on.

Creating a Leaf Node

Behaviour Tree leaf nodes, have no child nodes. To create a leaf node derive from BTNode. Following is a simple delay Behaviour Tree action node.

```
using UnityEngine;
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.BehaviourTrees{

 [Category("My Nodes")]
 [Icon("SomeIcon")]
 [Description("Wait for an ammount of seconds then return Success")]
 public class SimpleDelay : BTNode {

  public BBParameter<float> waitTime;
  private float timer;

  protected override Status OnExecute(){

   timer += Time.deltaTime;
   if (timer > waitTime.value)
    return Status.Success;

   return Status.Running;
  }

  protected override void OnReset(){

   timer = 0;
```

```
  }
 }
}
```

*Again, it is **very** recommended to use the existing Action and Condition Behaviour Tree nodes and create Action and Condition Tasks(http://nodecanvas.com/documentation/tasks/creating-custom-tasks/) respectively to use with them instead of creating leaf action/condition nodes this way.*

Creating a Decorator Node

To create a Decorator you should derive from BTDecorator base type. Decorator nodes can only have one child node. You access that child through the connection object. Here is an Inverter Decorator for example.

```
using UnityEngine;
using NodeCanvas.Framework;
using ParadoxNotion.Design;

namespace NodeCanvas.BehaviourTrees{

 [Category("Decorators")]
 [Icon("SomeIcon")]
 [Description("Invert Success to Failure and Failure to Success")]
 public class Inverter : BTDecorator{

  protected override Status OnExecute(){

   status = decoratedConnection.Execute();

   if (status == Status.Success)
    return Status.Failure;

   if (status == Status.Failure)
    return Status.Success;

   return status;
  }
 }
}
```

You get to access and execute the decorated child node through the connection object. In NodeCanvas, connections are objects by themselves and they also have an Execute method that returns a Status.

Creating a Composite Node

To create a Composite, you derive from BTComposite base type. Composite nodes can have any number of connected child nodes. Similar to decorators, you get to access and execute child nodes through the connections. Here is an example of a simple sequencer.

```
using UnityEngine;
using NodeCanvas.Framework;
using ParadoxNotion.Design;
```

```
namespace NodeCanvas.BehaviourTrees{

    [Category("Composites")]
    [Icon("Sequencer")]
    [Description("Execute the child nodes in order from left to right")]
    public class SimpleSequencer : BTComposite {

        private int lastRunningNodeIndex;

        protected override Status OnExecute(){

            for (int i = lastRunningNodeIndex; i < outConnections.Count; i++){

                status = outConnections[i].Execute();

                if (status == Status.Running){
                    lastRunningNodeIndex = i;
                    return Status.Running;
                }

                if (status == Status.Failure)
                    return Status.Failure;
            }

            return Status.Success;
        }

        protected override void OnReset(){

            lastRunningNodeIndex = 0;
        }
    }
}
```

That’s it. Remember that double clicking on a node, opens it up in your IDE.

6. State Machines

States

Every node but a few of the FSM is a State. A State will do some action as soon as it is Entered, while it is Running and as soon as it is Exited. States can have Transitions to other States. Transitions are condition based and as such when a Transition’s Condition is true, the Transition will take place. When a Transition takes place the current State will Exit and the new one will Enter. When a State’s job is done, it will check for any Transition without a Condition. If one exists, the Transition will take place.

Transitions

Every node of an FSM *(except the Concurrent State)* can have Transitions. A Transition holds a Condition Task. While a State is active, all of it's Transitions are checked. If any of them is true, the Transition will take place and the current State will Exit while the target state will Enter and become current. A Transition without a condition will take place as soon as the parent State is finished. Notice that while you can have many Transition without a Condition, only the first one will take place.

You can prioritize the order in which the transitions will be checked through the State's Inspector.

FSM's in NodeCanvas are Conditional based instead of event based like some other tools are. This also allows them to check for Events as well thus making them very flexible.

Further Reading

Here are some interesting links on the net regarding State Machines:
TutsPlus(http://gamedevelopment.tutsplus.com...ines-theory-and-implementation--gamedev-11867/)

ITU.dk(https://blog.itu.dk/MAIG-E2013/files/2013/09/2commontechniques.pdf)

6.1. FSM Node Reference

Action State

The Action State node itself holds an Action List. On Enter, the assigned actions will execute either in parallel or in order one after the other. This can be set in the inspector. The State is considered finished as soon as all actions are finished if at all. In Play mode the running actions will be marked with a play icon in front of them.

- **OnEnter**: All Actions listed will Execute (*OnExecute*)
- **OnExit**: All Actions listed will Stop (*OnStop*)
- **Finished**: When all Actions listed have Ended (*EndAction*)

You can select wether or not you want listed actions to Repeat by checking the "Repeat State Actions".
You can also set if you want to listed actions to Run In Sequence or Run In Parallel by the relevant GUI control underneath the action list.

Any State

The Any State node is used to transition from Any State to any other based on a Condition. It can have any number of Transitions that are constantely checked. If any Transition's Condition becomes true the current State of the FSM (whatever that is) will Exit and the target one of the AnyState transition will Enter.

You can mark "Don't Reenter Active States", to prevent the Any State from transitioning to a state that is already active.

Concurrent State

The Concurrent State node will execute as soon as the FSM is started and in parallel to any other state. This is not a state per-se and as such it has neither incomming nor outgoing transitions. This state's actions will stop as soon as the FSM is finished.

- **OnEnter**: All Actions listed will Execute (*OnExecute*)
- **OnExit**: All Actions listed will Stop (*OnStop*)
- **Finished**: When all Actions listed have Ended (*EndAction*)

Nested Behaviour Tree

The Nested Behaviour Tree node is assigned a whole Behaviour Tree. On Enter, the Behaviour Tree will be executed either once or forever based on the chosen setting in the inspector. If set to **Run Once** this state will finish as soon as one Behaviour Tree cycle is done.

You can optionaly specify two events to be send, one for when the nested Behaviour Tree's root node status returns Success and another for when it returns Failure. As soon as either happen, the event will be send. Use the Check Event Condition on this node's transitions to make use of that event.

Nested BehaviourTree States, are very powerful, since you can have the broad states at a top level of the behaviour and at each state go deeper to define the behaviour of the state using Behaviour Trees!

- **OnEnter**: The Behaviour Tree will Start
- **OnExit**: The Behaviour Tree will Stop
- **Finished**: When the Behaviour Tree is done

When this FSM is paused the Nested Behaviour Tree will also get paused if this node is the current FSM state.

The target Behaviour Tree of the Nested Behaviour Tree node, can be linked to a blackboard variable!

Nested FSM

The Nested FSM is assigned another entire FSM. On Enter, that FSM will execute. This State will be finished when and if the nested FSM is finished. Of course if a Transition's Condition is true, the Transition will take place and this state will Exit as normal. Thus with Nested FSM state you are able to create the so called Hierarchical FSMs.

- **OnEnter**: The FSM will Start
- **OnExit**: The FSM will Stop
- **Finished**: Never unless the nested FSM stops somehow

When this FSM is paused the Nested FSM will also get's paused if this node is the current FSM state.

The target FSM of the Nested FSM node, can be linked to a blackboard variable!

6.2. FSM Callbacks

When an FSM state is Entered, Updated or Exited, you can receive callbacks on your MonoBehaviours attached to the same game object as the FSMOwner. The callbacks that can be used are the following:

- OnStateEnter(IState)
- OnStateUpdate(IState)
- OnStateExit(IState)

Here is a code example for your convenience.

```
using UnityEngine;
```

```
public class StateCallbacksExample : MonoBehaviour {

 public void OnStateEnter(IState state){

 }

 public void OnStateUpdate(IState state){

 }

 public void OnStateExit(IState state){

 }
}
```

IState is an interface having information about the current state the callback is called for.

```
 public interface IState{

  //The name of the state
  string name{get;}

  //The tag of the state
  string tag{get;}

  //The elapsed time of the state
  float elapsedTime{get;}

  //The FSM this state belongs to
  FSM FSM{get;}

  //An array of the state's transition connections
  FSMConnection[] GetTransitions();

  //Evaluates the state's transitions and returns true if a transition has been perf
ormed
  bool CheckTransitions();
 }
```

These callbacks are not ment to be used for modeling a whole state, but rather to get notified and perform some extra actions you might need to do.

6.3. Creating Custom FSM Nodes

While the recomended way of working with FSMs and NodeCanvas in general, is to create Action & Condition Tasks and use those on an Action State for example, it is very possible to create your own FSM States Nodes as an alternative.

To do so you simply have to create a class deriving from FSMState and override the methods as needed, then call Finish() when the state has finished. As soon as you create that class, it will show up to be added in the FSM canvas. You can even specify a custom Icon if you want so. Here is a simple delay State:

```
using UnityEngine;
using NodeCanvas.Framework;

namespace NodeCanvas.StateMachines{

 [Category("My States")]
 [Icon("SomeIconName")] //Icon must be in a Resources folder
 public class SampleState : FSMState {

  public BBParameter<float> timeout;
  private float timer;

  //When the FSM itself starts
  protected override void Init(){
   Debug.Log("Init");
  }

  //When the state is entered. Not when it is resumed though
  protected override void Enter(){
   Debug.Log("Enter");
  }

  //As long as the state is active
  protected override void Stay(){
   timer += Time.deltaTime;
   if (timer >= timeout.value)
    Finish();
  }

  //When the state was active and another state entered thus this exits. Also when t
he whole FSM stops.
  protected override void Exit(){
   Debug.Log("Exit");
   timer = 0;
  }

  //When the state was active and FSM paused
  protected override void Pause(){
   Debug.Log("Pause");
  }
 }
}
```

That’s it, you now have a custom delay state. Remember that double clicking on a node, opens it up in your IDE!

Finaly, here are some important inherited properties and methods from the FSMState class.

FSM FSM
The parent FSM graph of this state.

Component graphAgent
The agent of the FSM.

Blackboard graphBlackboard
The blackboard of the FSM.

float elapsedTime
The time in seconds this state is running.

void Finish()
Call to finish the state.

void SendEvent(string name)
Call to send an event to the FSM graph.

7. Dialogue Trees

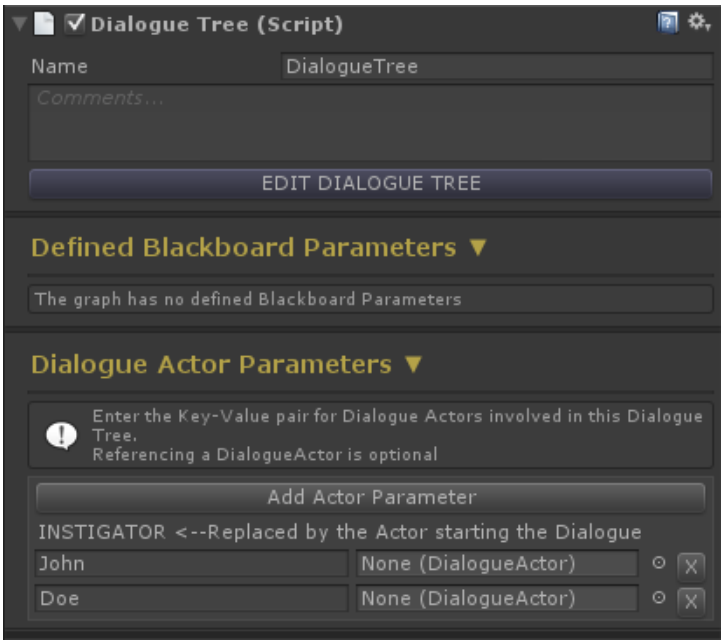
With the bonus Dialogue Trees module included in NodeCanvas, you are able to create non-linear, multi-actor dialogue trees with the same familiar node editor as the rest of the NodeCanvas modules.

1 – Create the Dialogue Tree

Dialogue Trees work a bit different setup-wise than the rest of the modules, in the sense that there is no GraphOwner component needed for them to work. Instead, you just need to create a Dialogue Tree from the top Unity menu *“Window/NodeCanvas/Create/Graph/Dialogue Tree”*. When that is done, a new game object with the Dialogue Tree component attached will be created in your scene hierarchy.

2 – Define the Dialogue Actor Parameters

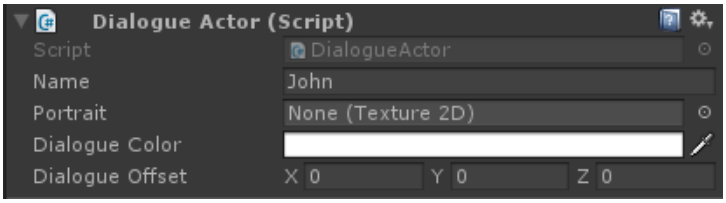
Within the inspector of the Dialogue Tree, you will need to define the different actor parameters that will be used by this Dialogue Tree. In the example bellow, there are two actors defined, those being “John” and “Doe”.



You will notice that on the right side of each actor parameter, there is a DialogueActor field as well. If you want your defined actors above, to also have a visual representation within the game world, you will need to add the *"DialogueActor"* component on the game object that represents them, and then assign that gameobject to the respective DialogueActor field.

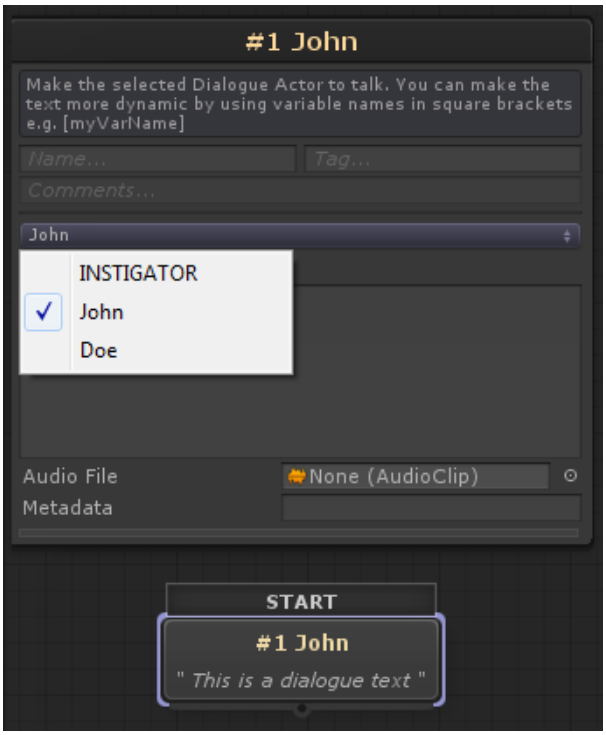
So for example, if we have a gameobject that we would like to use for "John", we would need to add the DialogueActor component on that gameobject and then drag&drop that gameobject in the field next to the "John" actor parameter in the Dialogue Tree inspector.

As you will see, the DialogueActor component, has only but a few self-explanatory settings relevant to the actor's appearence in the UI.



3 – Create the Nodes

All Dialogue Tree nodes, have one dropdown setting in common at the top of their inspector, that being "which actor this node is for". Within this dropdown, all previously defined actor parameters will be listed. The actor selected here, is the actor that will perform this node. In the following example, the node has been set to be used by "John". As such, John is the one that will say "This is a dialogue text".



The Instigator

By now, you might be wondering what the “Instigator” option shown in the actor selection dropdown is. Simply put, the “Instigator” is the default actor. When starting a Dialogue Tree from code, you have the option to set it’s default actor. As such, wherever the Instigator is selected, that default actor will be used instead, which can prove to be usefull for creating reusable Dialogue Trees.

Dynamic Text through Variables

Within the “Say” or “Multiple Choice” nodes of the Dialogue Tree, you are also able to have dymamic dialogue text displayed by the use of the local dialogue tree variables, by using the variable’s name within square brackets within the text. For example, if you had a integer variable named “apples”, you can write the dialogue text like so:

- *“I have [apples] apples in my pocket”.*

In game, “[apples]” will be replaced by the variable’s value ToString and for example show this instead:

- *“I have 3 apples in my pocket”.*

Needless to say, this is very usefull and something to be remembered if you are after creating some dymamic dialogue trees!

4 – Add the UI

You will obviously need a UI to display the dialogue. While you are able to create your own, there already exists one which you can use to get started quickly. All you have to do, is to add the prefab named “@DialogueUGUI” in the scene and you are good to go.

5 – Start the Dialogue

Starting the dialogue in game is a matter of a simple function call. For example:

```
using UnityEngine;
using NodeCanvas.DialogueTrees;

public class Example : MonoBehaviour {

 public DialogueTree dialogue;

 void Update(){
  if (Input.GetKeyDown(KeyCode.Space)){
   dialogue.StartDialogue();
  }
 }
}
```

If you are working with NodeCanvas FSMs or Behaviour Trees, there also exists an ActionTask called "StartDialogue" which you can use instead of doing this in code.

Finaly, here are the various overload methods that the StartDialogue comes with:

```
///Start the DialogueTree without an Instigator
public void StartDialogue();

///Start the DialogueTree with provided actor as Instigator
public void StartDialogue(IDialogueActor instigator)

///Start the DialogueTree with a callback for when it's finished
public void StartDialogue(Action<bool> callback)

///Start the DialogueTree with provided actor as instigator and provided callback fo
r when dialogue is finished
public void StartDialogue(IDialogueActor instigator, Action<bool> callback)
```

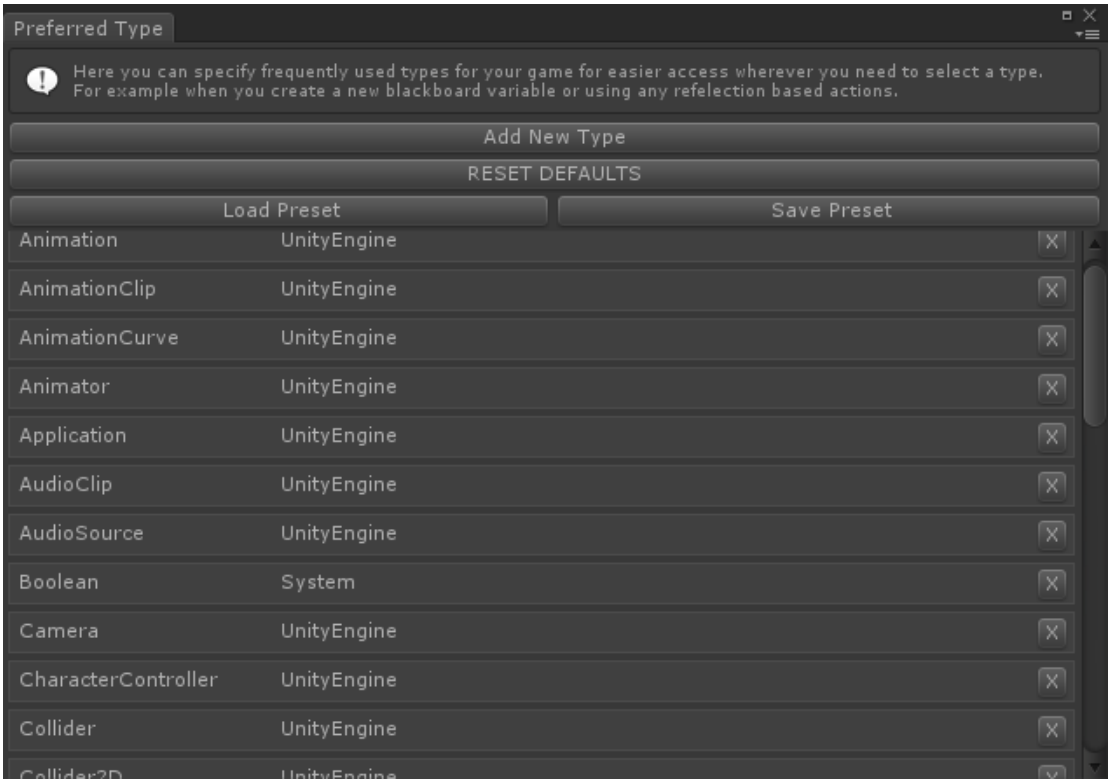
8. Working with Custom Types

NodeCanvas is built to work with any type you might need. To depict this within the editor though, keeping it clean and not cluttered by the thousands of possible types available, a custom editor window called Preferred Types Editor exists, within which you can restrict the types that will be shown within various menus when working with NodeCanvas. Specificaly, the cases where this list will come into use, include:

- Creating a Blackboard variable.
- Working with generic nodes and tasks (T).
- Working with the Script Control tasks when the agent is unknown.

Opening the Preferred Types Editor can be done through the "Window/NodeCanvas/Preferred Types Editor"

unity top menu. As you will find out, there are a number of types already in the list by deafult, but feel free to modify the list however you see fit to your project.



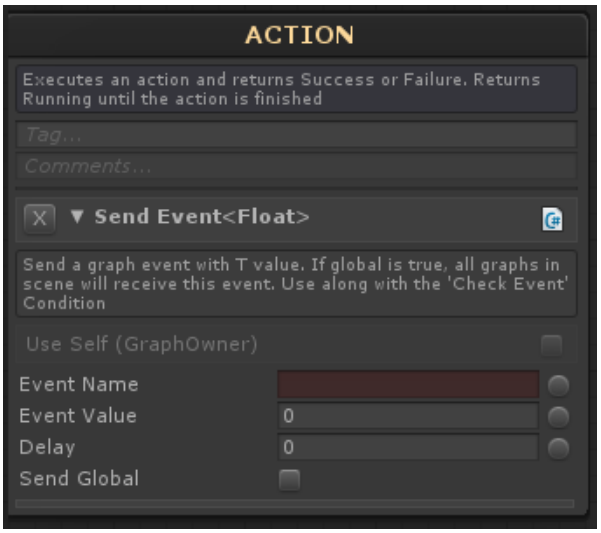
This editor and it's use, is strictly an editor thing for faster workflow and has nothing to do with runtime.

9. Using Graph Events

Graph Events are a great way to signal the graph that something happened and react accordingly. NodeCanvas has an internal Event system which can be used in any type of graph. Events can also have any type of variable included and thus possible to also communicate variables with them.

Sending an Event

You can send an event using the included Action Tasks SendEvent or SendEvent<T> in case its a value event. When using the SendEvent<T> you will also be able to provide a value for the event to transfer from within the editor inspector. Following is a SendEvent<float>.



You can also send an event easily from code of course. You can do this by having a reference to the GraphOwner. For example.

```
public class Example : MonoBehaviour{

 public BehaviourTreeOwner owner;

 public void SendNormalEvent(){
  owner.SendEvent( "MyEventName" );
 }

 public void SendValueEvent(){
  owner.SendEvent<float>( "MyEventName", 1.2f );
 }
}
```

Checking an Event

To Check for an event in the Graph, you should use the included Condition Task, CheckEvent or CheckEvent<T> in case of a value event.

Using the CheckEvent<T> also allows you to store the received event's value to a Blackboard Variable for later usage. The CheckEvent condition will return true for only one frame, not until the event is consumed.

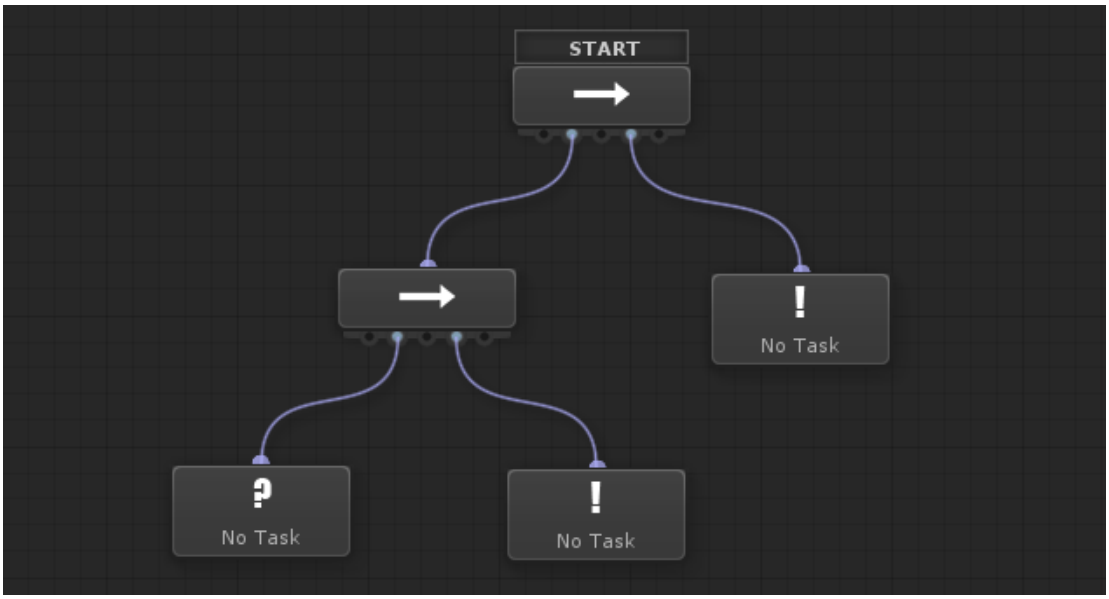
In regards to FSMs, using events is very straigtforward, but using events in BehaviourTrees along with the various Decorators included allows for a lot of flexibility in designing a behaviour tree. For example:

- You can use a CheckEvent along with an Interruptor Decorator to interrupt the decorated child if the event is received.
- You can use a CheckEvent along with a Conditional Decorator to trigger access to the decorated child when the event is received.

10. JSON Import/Export

All graphs in NodeCanvas can be exported to or imported from a JSON file. This proves usefull for sharing graphs, designs or patterns with others in a simple format. Importing and Exporting to/from a JSON file can be done either in the Graph Inspector, or from within the NodeCanvas editor under the "File" menu.

So for example the following behaviour tree which is the common If-Then-Else pattern…



…can be exported or imported to/from the following JSON file.

```
{
    "version": 2.0,
    "type": "NodeCanvas.BehaviourTrees.BehaviourTree",
    "name": "BehaviourTree",
    "comments": "",
    "translation": {
        "x": -5202.998046875,
        "y": -4921.998046875
    },
    "nodes": [
        {
            "dynamic": false,
            "random": false,
            "_collapsed": false,
```

```
        "_position": {
            "x": 5708.0,
            "y": 5031.0
        },
        "_name": null,
        "_tag": null,
        "_comment": null,
        "_isBreakpoint": false,
        "$type": "NodeCanvas.BehaviourTrees.Sequencer",
        "$id": "1"
    },
    {
        "dynamic": false,
        "random": false,
        "_collapsed": false,
        "_position": {
            "x": 5608.0,
            "y": 5147.0
        },
        "_name": null,
        "_tag": null,
        "_comment": null,
        "_isBreakpoint": false,
        "$type": "NodeCanvas.BehaviourTrees.Sequencer",
        "$id": "2"
    },
    {
        "_condition": null,
        "_collapsed": false,
        "_position": {
            "x": 5512.998046875,
            "y": 5259.998046875
        },
        "_name": null,
        "_tag": null,
        "_comment": null,
        "_isBreakpoint": false,
        "$type": "NodeCanvas.BehaviourTrees.ConditionNode",
        "$id": "3"
    },
    {
        "_action": null,
        "_collapsed": false,
        "_position": {
            "x": 5705.0,
            "y": 5262.0
        },
        "_name": null,
        "_tag": null,
        "_comment": null,
        "_isBreakpoint": false,
        "$type": "NodeCanvas.BehaviourTrees.ActionNode",
        "$id": "4"
    },
    {
```

```
            "_action": null,
            "_collapsed": false,
            "_position": {
                "x": 5822.0,
                "y": 5150.0
            },
            "_name": null,
            "_tag": null,
            "_comment": null,
            "_isBreakpoint": false,
            "$type": "NodeCanvas.BehaviourTrees.ActionNode",
            "$id": "5"
        }
    ],
    "connections": [
        {
            "_sourceNode": {
                "$ref": "1"
            },
            "_targetNode": {
                "$ref": "2"
            },
            "_isActive": true,
            "$type": "NodeCanvas.BehaviourTrees.BTConnection"
        },
        {
            "_sourceNode": {
                "$ref": "1"
            },
            "_targetNode": {
                "$ref": "5"
            },
            "_isActive": true,
            "$type": "NodeCanvas.BehaviourTrees.BTConnection"
        },
        {
            "_sourceNode": {
                "$ref": "2"
            },
            "_targetNode": {
                "$ref": "3"
            },
            "_isActive": true,
            "$type": "NodeCanvas.BehaviourTrees.BTConnection"
        },
        {
            "_sourceNode": {
                "$ref": "2"
            },
            "_targetNode": {
                "$ref": "4"
            },
            "_isActive": true,
            "$type": "NodeCanvas.BehaviourTrees.BTConnection"
        }
```

```
    ],
    "primeNode": {
        "$ref": "1"
    },
    "canvasGroups": null,
    "localBlackboard": {
        "_name": "Local Blackboard",
        "_variables": [],
        "$type": "NodeCanvas.Framework.Internal.BlackboardSource"
    }
}
```

11. Scripting

The following sub-sections are relevant to scripting

11.1. Interfacing with your code

NodeCanvas is build around the premise that it has to work in your way instead of against it. What this mean for you, is that you can use NC at a higher level to interact with your existing code and systems, without necessarily writing custom Actions & Conditions for the NC system.

Here is a collection of documentation links relevant to the subject:

- Using Script Control Actions(http://nodecanvas.paradoxnotion.com/documentation/?section=the-script-control-tasks)
  to call any function or get/set any property on any Component.
- Using Script Control Conditions(http://nodecanvas.paradoxnotion.com/documentation/?section=the-script-control-tasks)
  to check any property or non void function on any Component.
- Using Blackboard Variable Data Bindings(http://nodecanvas.paradoxnotion.com/documentation/?section=data-binding-variables)
  to bind a variable to a property of any Component.
- Having the ability to work with any Type(http://nodecanvas.paradoxnotion.com/documentation/?section=working-with-custom-types)
  .

11.2. GraphOwner API

The GraphOwner class has the following important methods and properties:

```
//The Graph currently assigned
Graph graph {get;set;}

//The Blackboard currently assigned
Blackboard blackboard {get;set;}

//Is the Graph running?
bool isRunning {get;}

//Is the Graph paused?
bool isPaused {get;}
```

```
//Starts the behaviour
void StartBehaviour()

//Starts the behaviour and get a callback when and if it's finished
void StartBehaviour(System.Action callback)

//Start a new behaviour graph
void StartBehaviour(Graph newGraph)

//Both of the above
void StartBehaviour(Graph newGraph, System.Action callback)

//Stops the behaviour
void StopBehaviour()

//Pause the behaviour
void PauseBehaviour()



//Sends an event
void SendEvent(string eventName)

//Sends an value event
void SendEvent<T>(string eventName, T value)

//Send a global event. Same as Graph.SendGlobalEvent
static void SendGlobalEvent(string eventName)

//Send a global value event. Same as Graph.SendGlobalEvent
static void SendGlobalEvent<T>(string eventName, T value)

//Calls a public method of name "methodName" on all tasks of the assigned graph
void SendTaskMessage(string methodName, object value = null)
```

BehaviourTreeOwner : GraphOwner<BehaviourTree>

The BehaviourTreeOwner has the following on top of the inherited GrapOwner ones:

```
//Should the BehaviourTree repeat?
bool repeat {get;set}

//The update interval in seconds the tree is ticked
float updateInterval {get;set;}

//The current root Status of the behaviour
Status rootStatus {get;}
```

```
//Tickes the current behaviour
Status Tick()
```

FSMOwner : GraphOwner<FSM>

The FSMOwner has the following on top of the inherited GrapOwner ones:

```
//The current State's name
string currentStateName {get;}

//The previous State's name
string previousStateName {get;}

//Triggers a state of the current FSM by name. Returns the FSMState node triggered.
FSMState TriggerState(string statename)

//Return the names of all states of the current FSM assigned to the FSMOwner
string[] GetStateNames()
```

11.3. Blackboard API

The Blackboard can easily be accessed manualy to read/write it's variables. All you need is a reference to it of course. Within tasks and nodes, you get a reference through the .blackboard inherited properties, but you can always of course use GetComponent<Blackboard>() since blackboard is a component.

```
//Adds a new variable to the Blackboard of type and name specified.
public Variable AddVariable(string, Type)

//Gets the Variable instance of name and type specified.
public Variable GetVariable(string, Type)

//The generic version of the above.
public Variable<T> GetVariable<T>(string)

//Get a variable value that is of specified name and of type T or assignable to type
 T.
//This is easier because it returns the value directly instead of the variable insta
nce.
public T GetValue<T>(string)

//Set the value of a variable with specified name to the new value passed.
public void SetValue(string, object)

//The blackboard can also be indexed for even easier access. For example...
blackboard["myVar"] = 5f;
Debug.Log( blackboard["myVar"] );
```

```
///
///Following is the API for saving/loading the blackboard variables for your conenie
nce
///

//Save the variables state to in PlayerPrefs with the key specified.
//Returns the serialized blackboard json string.
public string Save(string)

//Loads the variables back from PlayerPrefs from the key specified. Returns true if
success.
public bool Load(string)

//Returns the serialized blackboard to json.
public string Serialize()

//Deserialize a json serialized blackboard. Returns true if success.
public bool Load()
```

11.4. Runtime Instantiation Tips

There is often the need to proceduraly instantiate a number of different agents using the same Behaviour Tree or FSM at runtime. Due to the way NodeCanvas decouples the Behaviour Tree data from the agent, this is very easy to do with a number of different options.

Option 1

You have a game object prefab with a BehaviourTreeOwner. You also have a created BehaviourTree.

At runtime you can instantiate the BehaviourTreeOwner game object prefab and call StartBehaviour(Graph newGraph) on it, providing the BehaviourTree like this:

```
using UnityEngine;
using NodeCanvas.BehaviourTrees;

public class Example : MonoBehaviour {

 public int count;
 public BehaviourTreeOwner agent;
 public BehaviourTree bt;

 void Start(){

  for (int i = 0; i < count; i++){
   var newAgent = (BehaviourTreeOwner)Instantiate(agent);
   newAgent.StartBehaviour(bt);
  }
 }
}
```

Option 2

You only have a BehaviourTree created and no BehaviourTreeOwner at all. You can add the BehaviourTreeOwner component at runtime and similarily to above, Start a new graph for them:

```
using UnityEngine;
using NodeCanvas.BehaviourTrees;

public class Example : MonoBehaviour {

 public int count;
 public BehaviourTree bt;

 void Start(){

  for (int i = 0; i < count; i++){
   var agent = new GameObject("Agent").AddComponent<BehaviourTreeOwner>();
   agent.StartBehaviour(bt);
  }
 }
}
```

Of course in most cases you will also want to assign a blackboard for the BehaviourTreeOwner to use.

```
using UnityEngine;
using NodeCanvas;
using NodeCanvas.BehaviourTrees;

public class Example : MonoBehaviour {

 public int count;
 public BehaviourTree bt;
 public Blackboard blackboard;

 void Start(){

  for (int i = 0; i < count; i++){
   var agent = new GameObject("Agent").AddComponent<BehaviourTreeOwner>();
   agent.blackboard = blackboard;
   agent.StartBehaviour(bt);
  }
 }
}
```

Going Further

Using the StartBehaviour(Graph) or SwitchBehaviour(Graph) you can go further and even switch BehaviourTrees at runtime. Here is an example:

```
using UnityEngine;
using NodeCanvas.BehaviourTrees;
```

```
public class Example : MonoBehaviour {

 public BehaviourTree idleBehaviour;
 public BehaviourTree fleeBehaviour;
 public BehaviourTree attackBehaviour;

 private BehaviourTreeOwner agent;

 void Start(){

  agent = new GameObject("Agent").AddComponent<BehaviourTreeOwner>();
  agent.StartBehaviour(idleBehaviour);
 }

 void Update(){

  if (Input.GetKeyDown(KeyCode.Alpha1))
   agent.SwitchBehaviour(idleBehaviour);

  if (Input.GetKeyDown(KeyCode.Alpha2))
   agent.SwitchBehaviour(fleeBehaviour);

  if (Input.GetKeyDown(KeyCode.Alpha3))
   agent.SwithBehaviour(attackBehaviour);
 }
}
```

While this is quite possible, it is best design wise, to have only ony BehaviourTree and switch behaviours within using Nested Behaviour Trees.

All the above examples are also true for State Machines as well.

Manual Tick *(BehaviourTrees only)*

Sometimes you may want to Tick a BehaviourTree manualy instead of relying on StartBehaviour, PauseBehaviour and StopBehaviour. Here is how you could do this:

```
using UnityEngine;
using System.Collections.Generic;
using NodeCanvas;
using NodeCanvas.BehaviourTrees;

public class Example : MonoBehaviour {

 public int count;
 public BehaviourTree bt;
 public Blackboard blackboard;

 private List<BehaviourTreeOwner> agents = new List<BehaviourTreeOwner>();

 void Start(){
```

```
  for (int i = 0; i < count; i++){
   var agent = new GameObject("Agent").AddComponent<BehaviourTreeOwner>();
   agent.blackboard = blackboard;
   agent.behaviour = bt;
   agents.Add(agent);
  }
 }

 void Update(){

  foreach (BehaviourTreeOwner agent in agents){
   agent.Tick();
  }
 }
}
```

Remember that the Tick method also returns the root node status in case you care about it. Alternatively you can get the root node status with the rootStatus property on the BehaviourTreeOwner, which in essence returns the currently assigned Behaviour Tree's root node status.

12. Playmaker Integration

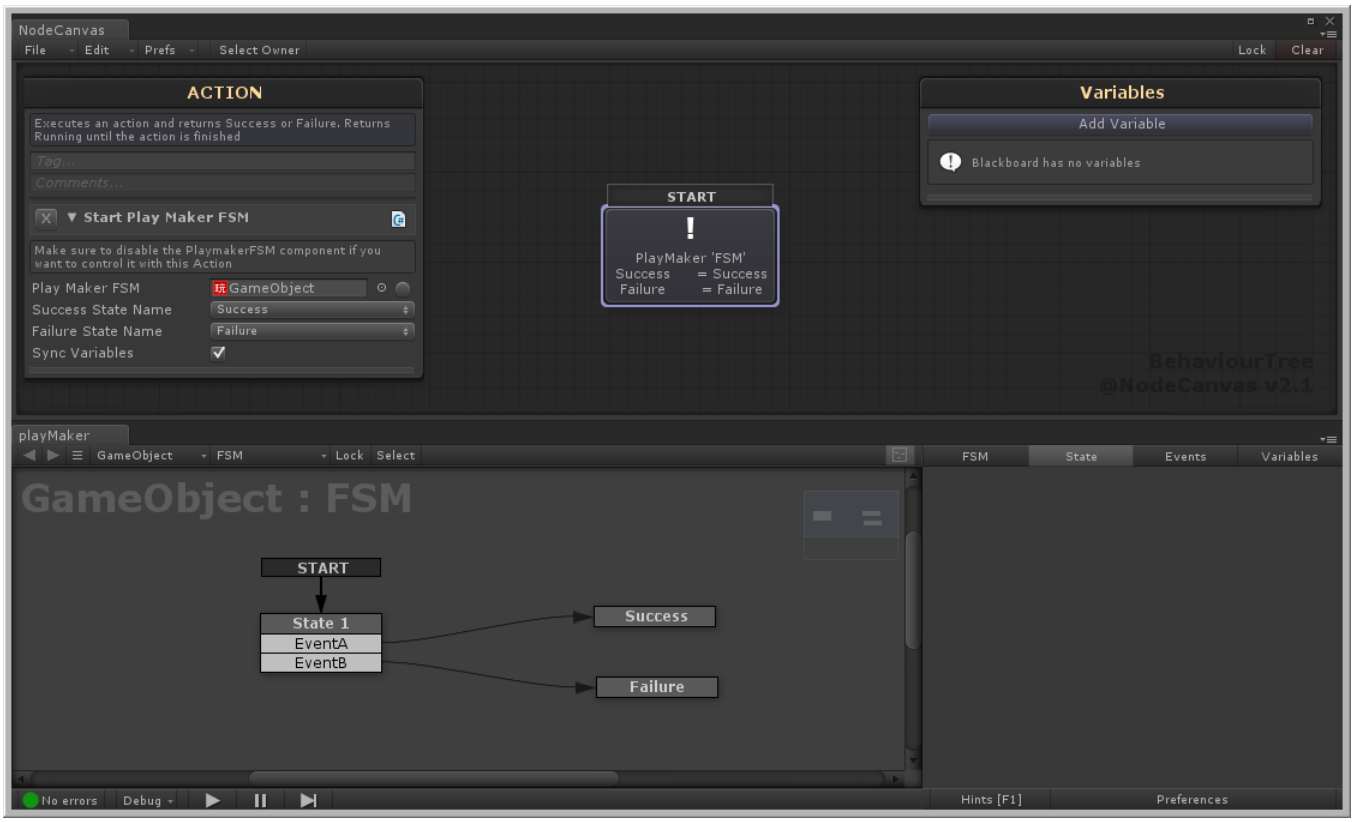
ntegration of NodeCanvas with playMaker is two-fold and in both cases it is possible to **Sync Variables** from one system to the other.

NodeCanvas > PlayMaker

A whole playMaker FSM can be part of a Behaviour Tree as an Action. Simply use the included NodeCanvas Action **Playmaker/Start Playmaker FSM.**

In that action you have to select the PlayMakerFSM to execute. The selected playMaker FSM will get disabled on awake. As soon as the action is executed it will get enabled and start. In the action you have to chose two playMaker states; one for Success and one for Failure. As soon as either state is entered, the playMaker FSM will stop and the action will return Success or Failure based on that state entered. These two states can just be empty states.

If the option to **Sync Variables** is checked, then the value of all common variables (name + type) between the Behaviour Tree blackboard and playMaker FSM, will be copied over to the playmaker FSM on execution, which then will be possible to change and modified by playMaker and then copied back to the Behaviour Tree blackboard!

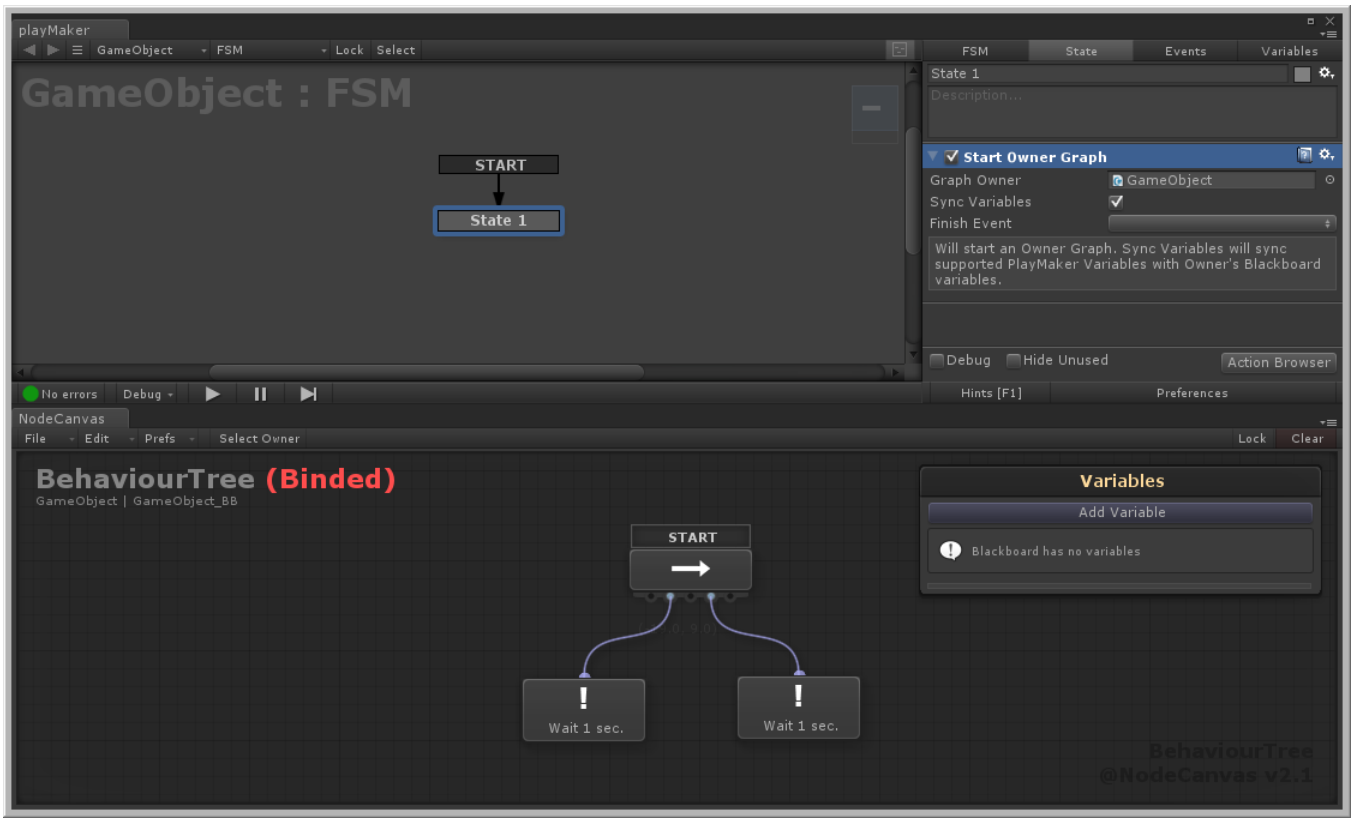


Playmaker > NodeCanvas

You can make a Behaviour Tree Owner as normal set it to ‘Do Nothing’ OnEnable. Then in a playMaker state add the action **Start Owner Graph,** found under the NodeCanvas category in the Action Browser. In that action, assign the Behaviour Tree Owner. When the Playmaker FSM state is Entered, it will start the Behaviour Tree of the owner. When the state Exits, that Behaviour Tree will stop. The best use scenario would be to set Run Forever in the BehaviourTreeOwner to false and get a FinishEvent in playMaker.

There is no special setup from the Behaviour Tree Owner side in this case.

The **Sync Variables** option works like before, but in this case the variables are copied from the playMaker FSM and to the Behaviour Tree blackboard and then back to the FSM instead!



13. Complete API(http://paradoxnotion.com/files/nodecanvas/Docs/html/index.html)