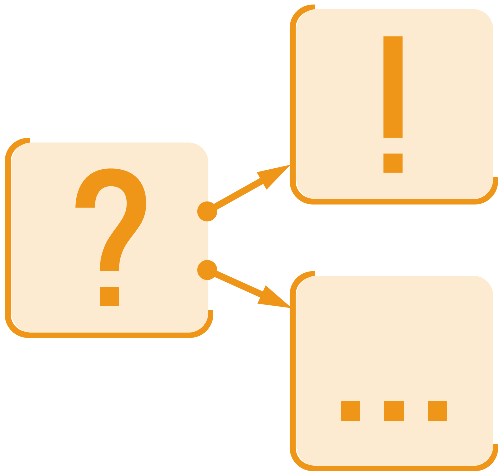
# DIALOGUE NODES

Documentation v1.0 2019/05



Plugin created by Thomas Lamson (aka Crafty Weazel)

# Table of contents

# Plugin presentation

This plugin is a Blueprint-integrated Dialogue system for Unreal Engine v4.22. It was made to make designer's work easier and faster. It can be used in any project which requires dialogues at some point and will be directly integrated into your own object's Blueprints.

**DIALOGUE NODES**

**INTUITIVE – INTEGRATED – FLEXIBLE – COMPLETE**

With this plugin, you will be able to design full branching dialogues directly into the Blueprint editor. You will directly use your Blueprint variables and functions into the dialogue branching to make your narrative design dynamic, conditional and adaptive.

Main features are:

- New Blueprint nodes, called **Dialogue Nodes**, which allow you to design and execute dialogues in the most graphic and intuitive way.
- Complete **integration in Blueprint** editor with read and write options at runtime.
- **Dynamic** and **rich text** writing: you will be able to use text parameters to make your text depend on in-game data, and you will be able to enrich it with text styles.
- **In-editor features** to ease and accelerate the design process.
- **"Dialogue" objects** which hold all the designed dialogues and manage their execution.
- New **Project Settings** to set custom parameters for all dialogue elements: sounds and animations to play, character's name or identification…
- Ground rules of this plugin: game agnosticism, user experience, easy integration.

What this plugin **isn't**:

- You will need to implement UI yourself, or use the one implemented in the example project.
- This isn't a new graph editor, as you can find in some other plugins, everything is done inside Blueprint's Event Graphs.
- This isn't an RPG framework, it is only dealing with dialogues and can be integrated to any other system of your making because of its external simplicity.

The full documentation is given here for your version of the plugin, have fun making dialogues with it!

# Terminology

Let’s go over a few terms that will be significantly used in this document.

- **dialogue**: a set of lines and possible answers that NPCs and Player will say at some point in the game. This includes branching over conditions, triggering events of all kind, using variables, etc.
- **Dialogue** (with a capital D): an object from the Dialogue class, responsible of holding all the dialogue branching, executing it, etc.
- **Blueprint / BP**: the Unreal Engine 4 graphical scripting system. Watch out! It doesn’t include all the graph editors of the engine, only the one called Blueprint editor (or Kismet 2).
- **Node**: term referring to a Blueprint node, a rectangle box you place in the Blueprint editor and from which you drag and connect wires.
- **Event**: a starting point in an object’s script that can be triggered by anything else that holds a reference to that object. It is widely used in this dialogue system.
- **Event Graph**: in the Blueprint editor, the graph where you define all the event actions. Typically, you find Event Tick, Event Begin Play over here, and this is where you’re going to work with dialogues.
- **Pin**: the place in the node where you connect wires.
- **Execution Pin**: the white arrowish pin in several nodes that corresponds to execution flow. When a node doesn’t have an execution pin, it is called Pure.
- **Data Pin**: pins that aren’t execution pins, which works in another way and which carry data.
- **Dialogue Flow Pin**: looking a lot like execution pins, those dialogue flow pins are colored in orange and are there to draw the sequential flow of the dialogue you’re designing. Disclaimer here: though the pins are orange, the wires are white because of some hard-coded settings in the editor that prevent us to make them another color.
- **Line**: a text line of the dialogue that will be pronounced by an NPC (typically, but you can do whatever you want after all).
- **Option**: a text line of the dialogue that will be suggested to the player, most of the time among other options, for him-her to choose (again, feel free to use it however you want, it’s just a convention).
- **Condition**: something that can be true or false depending on the context it is evaluated in. In this system, those are Boolean Pins that are read during Dialogue’s execution. It is highly practical when you want your dialogue to branch in different ways depending on some game logic.
- **NPC**: a non-playable character, referring to external characters in the game.

# Overall process

Before getting our hands dirty and diving into the details, let’s take a general look over the dialogue designing process. With this plugin installed, you will be able to design as much dialogues as you need and execute them whenever you need to.

The first step is to **design** the dialogue. Dialogues are designed only once, in one of your object’s Blueprint Event Graph. The designing process consists in drawing a dialogue graph representing the branching of your dialogue lines and options: who says what, among which options the player will chose, what happens when a certain point is reached?

The design process finally produces a single object, called **Dialogue Object**, which is self-sufficient. Think about this object as a capture of your graph. The graph you designed is read once and is registered definitely in this output object. You take that object, store it somewhere, and then you’re ready for the next step!

Whenever your player is starting a conversation with an NPC for which you designed a dialogue, you will have to **execute** it. Keeping details out for now, you just have to take that Dialogue Object you made, feed it into a processing node and here you go.

During the dialogue execution, you will have to display lines and options to the screen: this part is your job. You will receive text and options from the dialogue itself and you will have to display it into your own user interface. Then take the player’s choice thanks to your controller and feed it back into the dialogue to progress into the branching you designed.

You can make as much Dialogue Objects as you want, each one of them containing a different dialogue branching. For example, in an RPG project, you would design a dialogue for each character in the game directly into their Blueprint graph. Then, when interacted with by the player, these characters would send their dialogue to the HUD or the Controller to tell “hey, launch that dialogue, would you?”.

Let’s say you designed a long and complex dialogue and sometime later you want to edit it. Then you just have to go back where you designed it and modify the dialogue graph: change the branching, change the text, or any other options. Click Compile & Save, launch the game, that’s it! Your dialogue was updated.

In this document, we will go over the details of each step above. You will find all the features and different options offered by this plugin, as well as a more technical explanation at the end of the document for those who want to understand the backend behavior of the system.

Let’s begin with the most important part: designing dialogues.

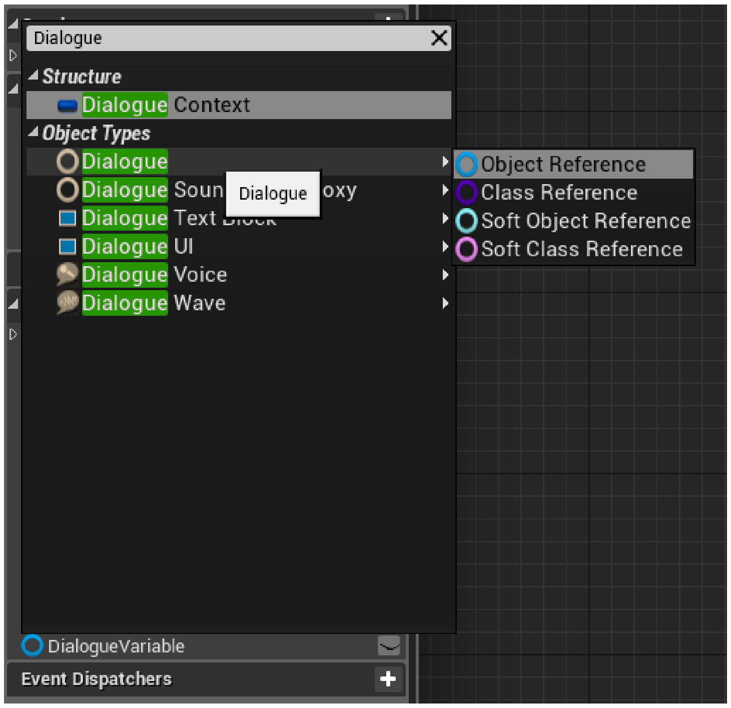
# Designing dialogues

Now that the overall process is hopefully clear enough, let's dive into the details of each step and of every possibility the system has to offer.

## Begin creation

To begin the creation of a Dialogue object, which is what contains a full dialogue branching, you'll need to open the Blueprint object of your project where you want to design the dialogue. It might be an object dedicated to this dialogue, an object where a set of dialogues are reunited (like a character or NPC object), or anything else that contains Blueprints.

Add a Dialogue (Object Reference) variable in that Blueprint and go into the Event Graph. As the Dialogue system strongly relies on hidden events (the nodes are creating events at compile time), you absolutely need to design and execute dialogues from here. However, it is recommended that you create a second Event Graph that you would call "Dialogue(s)", to keep things clean.



*Picture 1 - Dialogue variable creation*

## Working with Dialogue Nodes

The nodes added by this system are a bit special. They almost work as any other Blueprint node, but you should know important facts before working with them.

### Node properties

Some of the nodes have properties. Usually, BP nodes don't have such, but many other objects in the editor do. When you select a dialogue node that has properties, you will see it appear in the "Detail" panel, on the right of the BP editor screen by default. If you don't see it, you can always click on the "Windows" tab on the top of the screen and select "Details" to re-open the details panel.

When you've selected one node (select only one of them at a time), its properties appear in the detail panel where you can edit them. One of the most common properties in this dialogue system is the text field where you can type the dialogue lines. You can find some explanations about the behavior of this field in the section "Writing dynamic and rich text" of this document.

Otherwise, you will be able to find several fields. Some of them are check-boxes which make new pins appear on the node itself. For example, most of the dialogue nodes have a condition pin hidden by default. You can make it appear and link a Boolean value to it by checking the box "Has Condition" in the detail panel.

### Dynamic pins

Some of the fields in the detail panel are check-boxes which make new pins appear on the node itself. For example, most of the dialogue nodes have a hidden condition pin. You can make it appear and link a Boolean wire to it by checking the box "Has Condition" in the detail panel.

Some pins may appear by editing the text into a dialogue node. For example, if you type a text between [brackets], it will create what we call a "Text Parameter", which is a pin where you can connect a Text wire in order to change its value while the game is running.

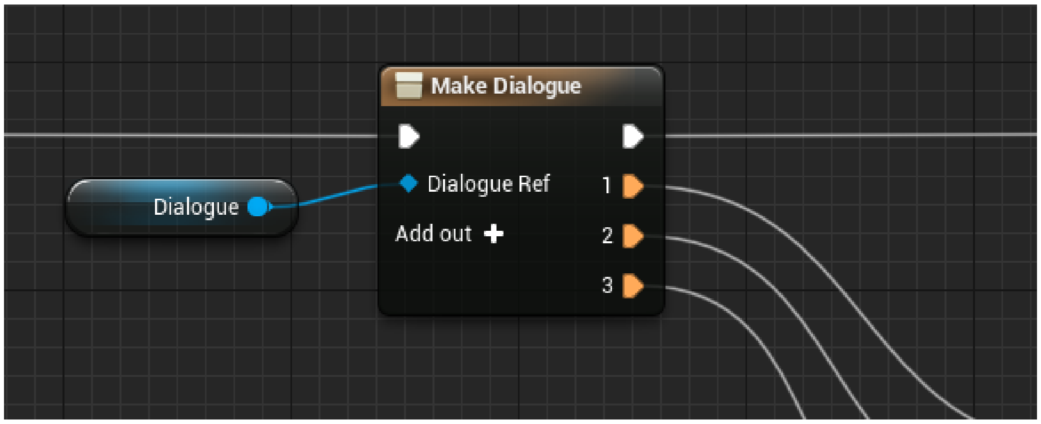
Also think about right-clicking on nodes and pins, as it may offer several options related to dialogue editing. By this mean, you can add or delete outcome pins, as we will see later.

### Dialogue Flow

The white wires connecting orange arrowish pins are dialogue flow wires. They cannot be connected to white Execution pins. They represent the flow of the dialogue and you can use the "To BP" node to transform the dialogue flow into an execution flow.

## Node: Make Dialogue

The first node you want to add in order to begin designing a Dialogue. This is the starting point of a new Dialogue object. To find all the nodes of the dialogue system, just right-click anywhere clear in the Event Graph and type “dialogue”. There are also other keywords that you can find in each section.



*Picture 2 – Make Dialogue Node*

This node isn’t pure, which means you must connect it to the normal execution flow of your object. Once it’s triggered, it will read all the following dialogue graph and will output the result. You should only trigger this node once, at game loading, as it will create a new dialogue containing all your branching. Then, take the output and store it into the Dialogue variable you created earlier. Later, this variable will be used to execute the dialogue.

***Execution pins, in and out***: these pins are the ones passed through by the execution flow. It works the same as other executable nodes.

***Dialogue output***: this output pin contains the final object, created after reading the following dialogue graph. You should store it into a variable.

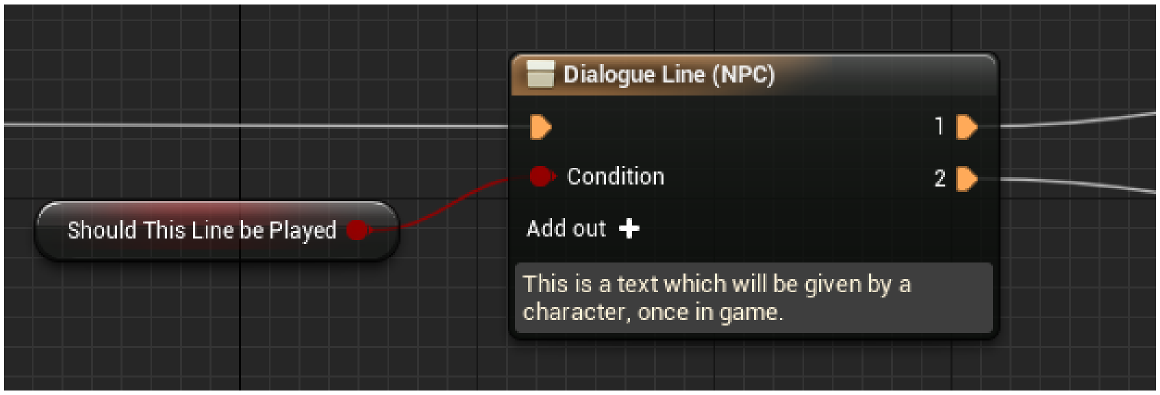
*+ **Add out***: this button adds a new dialogue outcome to the node, connected to nothing yet. You can also create new outcomes by right-clicking on the node and selecting the right option.

***Outcomes:*** the Dialogue Flow Pins on the right are outcomes of the node. They are the different options that will be evaluated when the Dialogue is being executed. You can have as many as you want and connect them to any other dialogue node. Right-click on a pin to see quick editing options.

[**Keywords:** Dialogue, Make, Construct, Create, Begin]

## Node: Dialogue Line (NPC)

From any dialogue flow pin, you can connect a Dialogue Line node, either by selecting it in the context sensible menu, or by right-clicking on the previous pin and selecting "Link new line". The Dialogue Line will register a text line that will be pronounced by any character of your choice.


Dialogue Line (NPC)
1
Condition
2
Should This Line be Played
Add out +
This is a text which will be given by a character, once in game.

*Picture 3 - Dialogue Line Node*

This node only has Dialogue Flow Pins, plus data pins. This means it will only be usable in a Dialogue graph, after a Make Dialogue node. This is a *solo* event, which means it will be the only one executed when encountered in the previous node outcomes list, admitting its condition is true if it was specified.

***Income:*** this pin should be connected to the previous node(s) in the dialogue graph.

***Outcomes:*** these pins lead to the different outcomes of the node. Right-click on a pin for quick editing options.

***+ Add out***: adds a new empty outcome. Right-click on the node for quick options.

***Condition:*** this Boolean pin is only available if you check the "Has Condition" box in the properties of the node. You should connect a Boolean value to be evaluated at runtime here, determining if the node should be considered or not in the dialogue branching.

***Text Parameters:*** text pins appear when you create text parameters in the Text property of this node. See "Writing dynamic and rich text" section for more information.

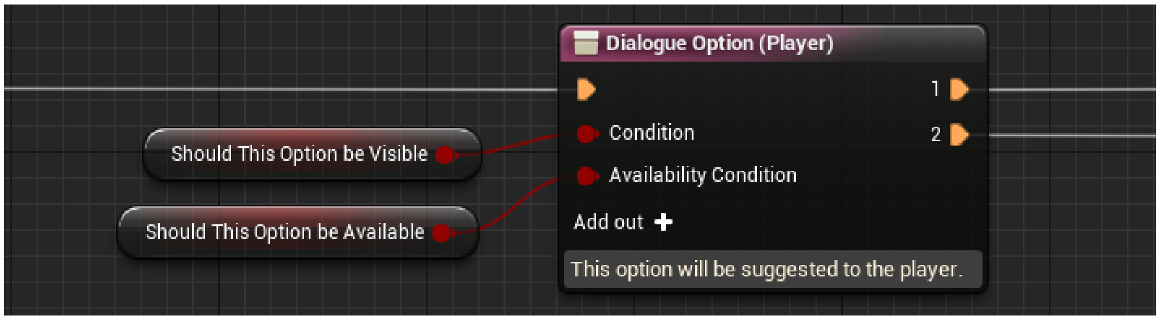
***Details Panel:***

***Text*** – the dynamic and rich text of this dialogue node. Multi-line, localized.
***Has Condition*** – does this node have a **Condition** pin? False by default.
***Metadata*** – fields depending on project settings, for any kind of information.

[**Keywords:** Dialogue, Line, Text, NPC]

## Node: Dialogue Option (Player)

From any dialogue flow pin, you can connect a Dialogue Option node, either by selecting it in the context sensible menu, or by right-clicking on the previous pin and selecting “Link new option”. The Dialogue Option will register a new answer option that will be suggested to the player, often among other options. This node is a lot like Dialogue Line node, but has one more condition pin.


Dialogue Option (Player)
1
2
Condition
Availability Condition
Add out
This option will be suggested to the player.
Should This Option be Visible
Should This Option be Available

*Picture 4 - Dialogue Option Node*

***Income:*** this pin should be connected to the previous node(s) in the dialogue graph.

***Outcomes:*** these pins lead to the different outcomes of the node. Right-click on a pin for quick editing options.

***+ Add out***: adds a new empty outcome. Right-click on the node for quick options.

***Condition:*** this Boolean pin is only available if you check the “Has Visibility Condition” box in the properties of the node. It determines if this option is considered in branching, hence visible or not to the player.

***Availability Condition:*** this Boolean pin is only available if you check the “Has Availability Condition” box in the properties of the node. Assuming this option is visible, this will determine if it should be grayed out from the player (*false*) or available (*true*).

***Text Parameters:*** text pins appear when you create text parameters in the Text property of this node. See “Writing dynamic and rich text” section for more information.

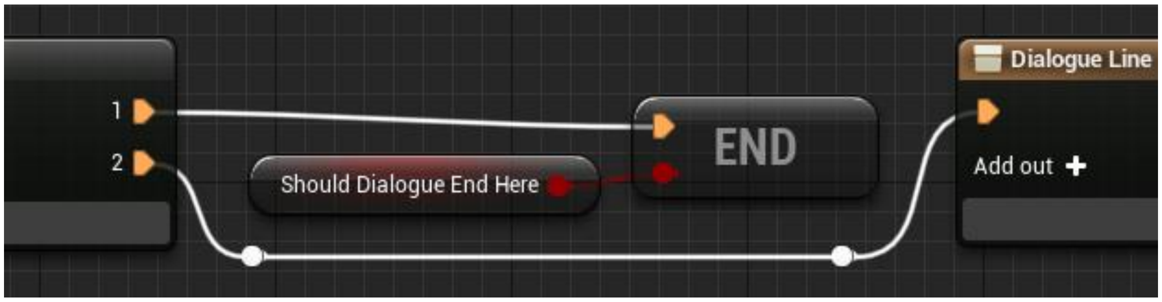
***Details Panel:***

***Text*** – the dynamic and rich text of this dialogue node. Multi-line, localized.
***Has Visibility Condition*** – does this node have a **Condition** pin? False by default.
***Has Availability Condition*** – does this node have an **Availability Condition** pin?
***Metadata*** – fields depending on project settings, for any kind of information.

[**Keywords:** Dialogue, Option, Text, Player]

## Node: Dialogue End

From any dialogue flow pin, you can connect a Dialogue End node by selecting it in the context sensible menu. This Dialogue End will mark the end of the Dialogue, ending all outcome processing when reached. However it can be made optional.



*Picture 5 - Dialogue End Node*

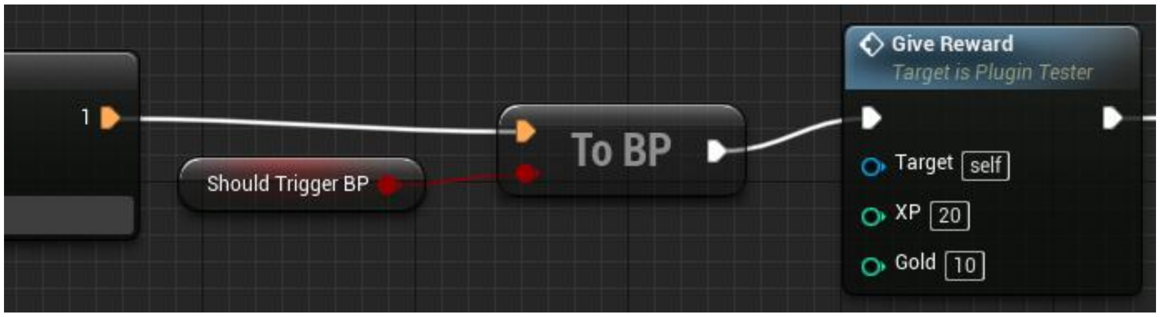
***Income:*** this pin should be connected to the previous node(s) in the dialogue graph.

***Condition:*** available by checking the “Has Condition” box in the node’s ***Details Panel***. It determines if the end of the dialogue should be reached at this point.

[**Keywords:** Dialogue, End, Stop]

## Node: To Blueprint

From any dialogue flow pin, you can connect a To Blueprint node by selecting it in the context sensible menu. This node converts the dialogue flow into an execution flow, which allows you to trigger any Blueprint behavior from any point in the dialogue. See “Dialogue Branching Process” for further details on whether this node is executed or not.



*Picture 6 - To Blueprint Node*

***Income:*** this pin should be connected to the previous node(s) in the dialogue graph.

***Outcome:*** the starting point of the execution flow that will be triggered, like an event.

***Condition:*** available by checking the “Has Condition” box in the node’s ***Details Panel***. It determines whether the Blueprint execution flow should be triggered.

[**Keywords:** Dialogue, To, Blueprint, BP, Event]

## Dialogue branching process

As it was explained in the previous sections, you can link dialogue nodes to one another pretty much as you want, through Dialogue Flow Pins. The true power of the systems is that you can connect multiple nodes to the same one, merging dialogue branches. But despite its simple look, the branching evaluation system may become quite complicated when you begin to mix different nodes together.

Here, we will go over the whole evaluation process, which is what happens when the dialogue is executed and your nodes are browsed.

1. First thing to know, outcomes are evaluated **from top to bottom**. You should use this fact to make prioritized choices.

2. If a node have a **condition**, it is evaluated and the node will only be considered if the condition is true at this moment. If it doesn't have a condition, it's *true* by default.

3. **Dialogue Options** are executed altogether and constitute the set of visible options to the player. Note: only the "*Condition*" pin is evaluated for branching, as the "*Availability Condition*" only tells if this option should be grayed out or not.

4. **Dialogue Lines** and **Dialogue Ends** are *solo* events, which means they are the only ones that are executed if their condition is evaluated as *true*. When the evaluation process meets a *true* Line/End, it stops and only executes this Line/End, forgetting about Dialogue Options that might have been encountered before.

5. **Dialogue To Blueprints** nodes are *extra* events, which means they will still be executed if a Line/End event is met afterward in the outcomes list. However, as a Line/End encounter stops the evaluation process, To Blueprints found afterward won't be considered, only the ones found before will be executed.

You can use these rules at your own advantage by mixing node types together. For example, you might mix a set of options with an optional line, which will display options only if the line condition isn't met. You can also place To Blueprint node between two lines: the BP event will only be triggered if the first line's condition isn't met, and will be followed by the second text line instead.

*Notes:* you can also make loops in the Dialogue Flow. Just connect a node's outcome to an earlier node's income in your graph, and a loop is created. Don't stuck the player though! You can also connect multiple nodes to a common income. Loops and common incomes won't change a thing in the dialogue branching process, feel free to use them.

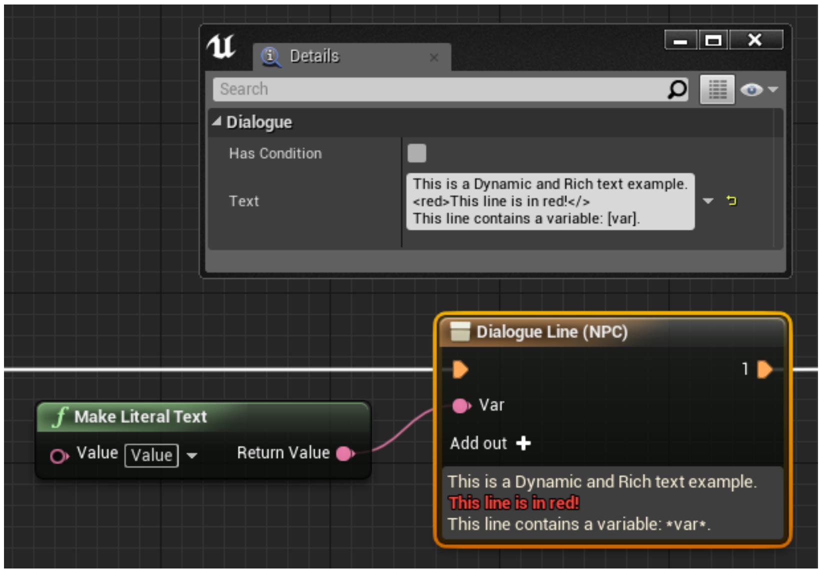
## Writing dynamic and rich text

The text property that you can see in Dialogue Line and Dialogue Option nodes contains a wide range of possibilities. By simply editing the text content itself, you can create text parameters and use different text styles.

Text parameters are custom text variables that are represented by a pin on the node itself. They are very useful as they let you dynamically change parts of the text depending on in-game data. Basically, you just need to write a text parameter's name in brackets: it will be detected and registered as soon as you press Enter or click outside the text field. A pin will appear with the name you wrote, and you will be able to connect it to your Blueprint logic with no harm. Further details in next section.

Text styles are those of Unreal Engine. What you write in the text field is what is called a Rich Text. Once it's finished editing, Unreal Engine displays it in the corresponding node by applying different text styles you will have defined elsewhere. Basically, you just put chevron tags throughout your text to specify which style you want to use. <bold>Writing this will display this sentence in bold.</> Further details in next sections.

Those two features are compatible together. You can use text parameters inside a text style tag for example, which will allow you to change the style of a portion of the text depending on any external game logic.

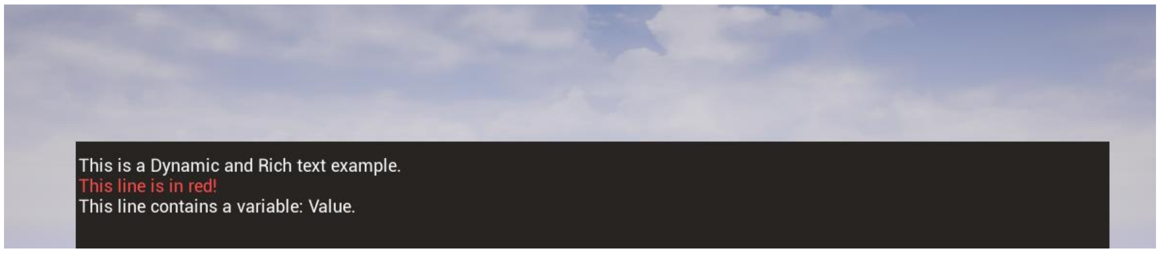


This is a Dynamic and Rich text example.
<red>This line is in red!</>
This line contains a variable: [var].

This is a Dynamic and Rich text example.
This line is in red!
This line contains a variable: *var*.

*Picture 8 - Node display and Details Panel (left)*

*Picture 7 - Dialogue display example in game (below)*



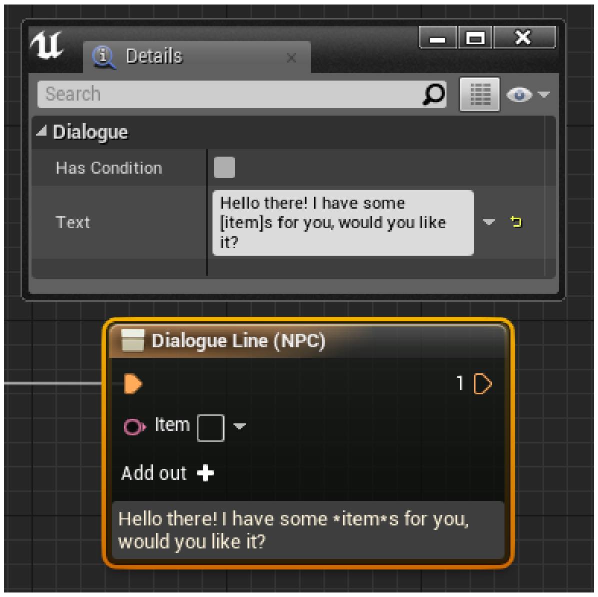
This is a Dynamic and Rich text example.
This line is in red!
This line contains a variable: Value.

## Text Parameters

Text parameters are variables inside the text of a dialogue line/option. They come as in-brackets names and are automatically detected when you finish editing your text. For example, let's say you wrote this into the Text field of a line/option node:

```
Hello there! I have some [item]s for you, would you like it?
```

As soon as you finish editing this text, a new pin called "Item" appears on the corresponding node:

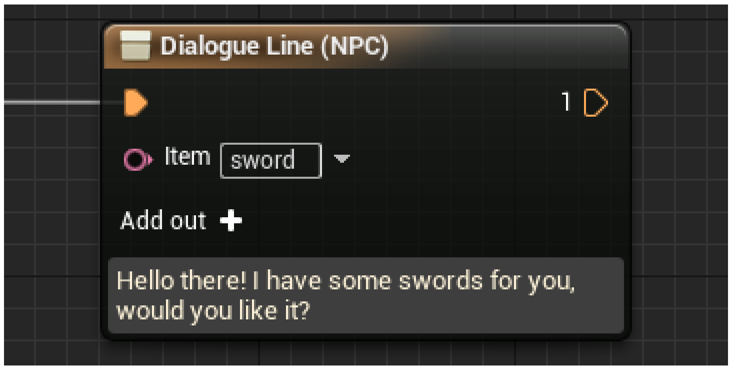


Text: Hello there! I have some [item]s for you, would you like it?

Hello there! I have some *item*s for you, would you like it?

Picture 9 - Line Node with a text parameter

You might also notice that the text in the node was replaced by `some *item*s for you`. It means that your text parameter was detected but no replacement value was added yet. Let's try to write a value inside the "Item" pin's field.



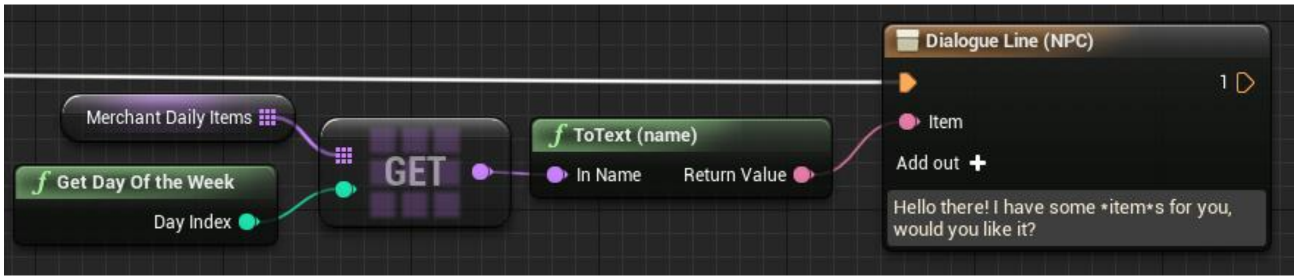
Item: sword

Hello there! I have some swords for you, would you like it?

Picture 10 - Line Node with a filled text parameter

Now the text in the node is saying `some swords for you`, as `*item*` was replaced by `sword`. You might also notice that we first wrote `[item]s` with an “s” after the final bracket. In fact, there is no need for spaces before and after brackets so you can use that fact to mark the plural or do anything else that matters.

Finally, it wouldn’t be interesting to let things as they are because until now our value is still static. It won’t change when the game is running because we only typed a literal value in the pin. The true power of text parameters appears when you connect that pin to anything else. You can connect it to any other system of your project. Here is an example of what can be achieved in our case:



*Picture 11 - Text parameters usage example*

In this final example, depending on the in-game day of the week during which the player is talking to the NPC – assumed to be a nice merchant – it will offer a different item, taken from the “Merchant Daily Items” array. From now, you are free to do whatever you want, but remember to only use **pure** nodes to retrieve data (without execution pins).

### *Pins management*

You can write the same parameter several times and it will be driven by the same pin. You can also delete a pin by right-clicking on it and selecting “Delete parameter”, in case you don’t want a certain parameter anymore. If it happens that you deleted a parameter and want it back, you can regenerate all parameter pins by right-clicking in the middle of the node and selecting “Create text pins” or by modifying the text field.

### *Convenient parameter names*

Supposedly, the plugin shouldn’t consider inconvenient names for text parameters, like an empty parameter, a punctuated name, a brackets mistake…

```
[], [n,a!m?e], [[]name]... -> Don’t do that.
```

But it might happen with special symbols that weren’t supported. Thus, you should always use convenient names: you should respect the same conventions as Blueprint variable names.

## Rich Text Styles

Since a recent version of Unreal Engine, Rich Text were reimplemented. They are UI components which display text with a wide range of styling effects, and can be extended with custom decorators – classes that add inline UI components or do some effects to the text, depending on the original content. They are welcomed when developing rich RPG games as they allow designers to mix various text styles together (color, alpha, boldness, font, outline, shadow…), add icons, hyperlinks, or tooltips to certain parts of their text. A dialogue plugin on UE4 wouldn't be complete if it didn't allow designers to use Rich Text components.

To that end, this plugin was made so that designers can have a live feedback of their text during the writing process. As you write raw text in the Details Panel of a dialogue node, it is displayed as a Rich Text in the corresponding node. As explained in the previous part, text parameters are also processed in that live feed. In this plugin, you will be able to set different text styling for both editor nodes and runtime.

### *Rich Text Style database*

Firstly, you need to define Rich Text style tables. The first one you should design is the editor styles table. To do so, create a new asset in your project browser and select *Miscellaneous / Data Table*. In the new dialog, select *Rich Text Style Row* in the list and click ok. A new text styles table is created. You can open it with a double-click.

In the table editing window, by default, the upper part is a list of the different text styles. At the beginning it is empty. Start by creating a new row with the "+" button in the lower half of the screen. Call this new style "Default". It will be the default text style when no tags are used. Then, you can create as many styles as you want, and you will specify the text effects in the lower half of the window.

Do it for the editor styles table, then do it again for the runtime styles table. This second table will contains styles that you will use once in game, in your UI components definition.

### *Setup text styles for editor and runtime*

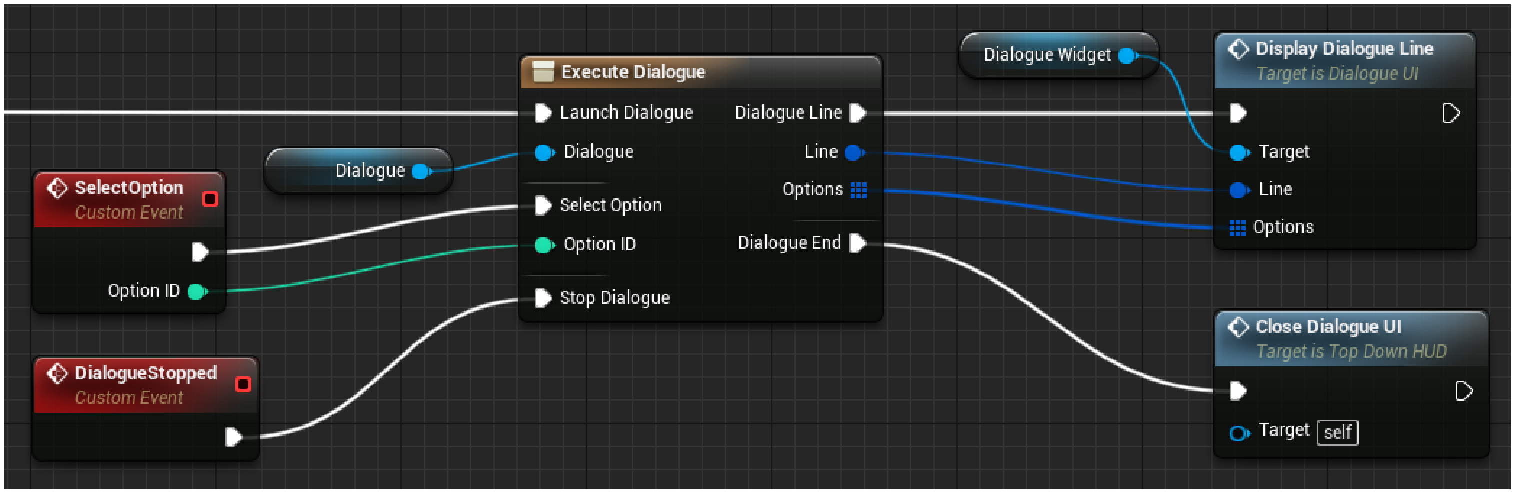
To select the first table you created as the editor text styles table, go into Project Settings / Editor / Dialogue and select your table in the *Text Style Set* field. You might need to re-compile some blueprints to see the new styles.

To select the second table you created as the runtime text styles table, you will need to use it manually when you design your UI components. Use a RichTextBox UMG component for your dialogue display, select your table, and feed processed text into it.

To begin to use a text style at some point in a text, just write its name between chevrons. Example: `<bold_red>`. Don't forget to close the tag with `</>` at some point.

# Executing dialogues

Dialogues are executed through the Execute Dialogue node, which takes a Dialogue object as an input and launches it. From this sole node, you will be able to manage all the UI behavior and logic behavior linked to your own in-game dialogue system. Remember that this plugin is only taking in charge the abstract part of the dialogue, also called game logic: you will have to implement your own displaying functions. This guide will however give you some advice on how to do so, and you will find an implementation example in the example project shipped with the plugin.



*Picture 12 – Execute Dialogue Node, with typical events and functions*

## Node: Execute Dialogue

This node acts as a hub: it contains several sections that could have been separated in distinct nodes, but things are clearer and less prompt to misusage this way. Let's go over the details on this particular node.

As you can see in the picture above, this node contains three input and two output execution pins. Each of them corresponds to a function: inputs are ways to communicate with the Dialogue object, outputs are the ways the Dialogue object uses to communicate with you.

### **Input:** Launch Dialogue

This first input pin is used to begin the Dialogue execution. You should connect the Dialogue variable you created earlier and in which you previously stored the result of a Make Dialogue node.

Once triggered, the dialogue is launched with its first events, whatever they are. As long as the dialogue isn't waiting for input or hasn't reached an end, it will trigger the Dialogue Line pin.

Just before launching the dialogue, you should do all the UI setup in order for it to be ready when the first lines are fired.

### **Output:** Dialogue Line

This output pin is triggered every time a dialogue line is encountered and executed. Output data is composed of a Line Struct containing the line text for the current localization and metadata, and of an Option Struct Array which contains all the data related to the options that come with the dialogue line.

You should display the line's text and the different options - if any - when this pin is triggered. The UI design is all yours at this point.

Note that the text of the line and options is already processed at this point, which means that all Text Parameters will have been replaced by their dynamic value and you only have to display the result.

**Option Struct Array:**

Option Structs are Blueprint structures containing all the relevant data for a dialogue option: processed text, is the option available or grayed out, metadata, etc. The Option Struct Array is containing an ordered list of these Option Structs, which is the order you designed in the dialogue graph. If there are no visible option, the array is empty.

### **Input:** Select Option

When a line has been displayed, the dialogue begins to wait for an input before proceeding to next events. If there are options to chose among, it's waiting for a choice of one of them. If it's not the case, it will still wait for an input before displaying the next line or ending the dialogue.

In either case, this goes through the Select Option input pin, where you simply indicate the chosen option index in the Option Struct Array from earlier. If no options are available, you can put any index, it won't be considered and dialogue will continue.

Once you've triggered Select Option pin, the next dialogue event is executed.

### **Input:** Stop Dialogue

Whenever you want, you can trigger this pin to put an end to the dialogue execution. This is to be done when player is closing dialogue UI on its own, or for any other reason that might end the dialogue sooner than expected.

This will reset the dialogue state, marking it as unlaunched. Note that when a dialogue is under execution, it cannot be executed a second time elsewhere. You would have to copy the Dialogue object and store it in another variable to launch it from another spot. Therefore, don’t forget to use the Stop Dialogue pin if the dialogue isn’t allowed to reach an end on its own, as you wouldn’t be able to launch it again if the player talked to the same NPC again.

Then, if you trigger the Stop Dialogue pin, or if the dialogue is reaching an end on its own (Dialogue End node, or no more events to execute in the current branch), it will trigger the Dialogue End output.

### **Output:** Dialogue End

Whenever the dialogue is ending, manually or by reaching a proper branch end, this pin is triggered. You should close the UI there and do whatever you want to be done as soon as the dialogue is ended.

When this pin is triggered, it means that the dialogue can be launched again from the beginning. You won’t be able to launch the dialogue a second time if that pin wasn’t triggered yet, except if you copy the Dialogue object before launching it and store the copy in another variable (then you’re simply manipulating two independent Dialogue objects).

### Side notes

*Note:* this hub node is replaced by several sub-nodes at compile time. Because some of them are event nodes (like Event Tick or Event Begin Play), you will only be able to use the Execute Dialogue node, as well as any other dialogue node, in a Blueprint Event Graph.

*Note:* during dialogue execution, some events are triggered back in the dialogue designing graph. It happens when you are reading dynamic values in the dialogue, or when you used To Blueprint nodes. This is what is referred to in this documentation when you see “at runtime” mentions.

You can find a complete dialogue executing implementation in the example project shipped with this plugin. On a side, you will find the dialogue design, on the other the dialogue executing, along with an UI example. Feel free to reuse the work here, and adapt the UI to your project!

## Workflow enhancement

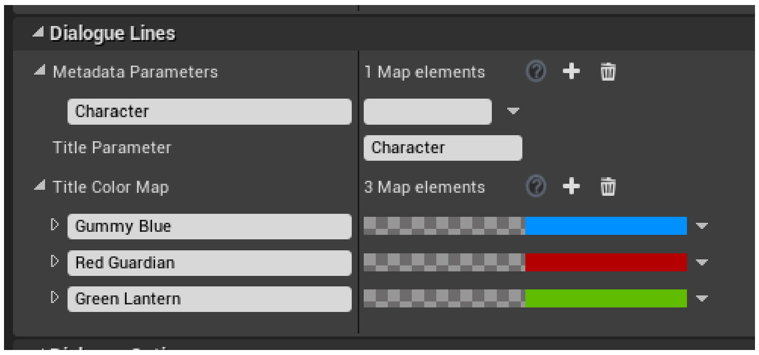
Some features were added to enhance the dialogue design process. Most of them rely on metadata, and you also have the opportunity to associate a sound and/or a dialogue wave to each dialogue nodes, in order to use voice acting in your dialogues. In this section, we will go over these features, which are optional but very useful in the end.

### Metadata parameters

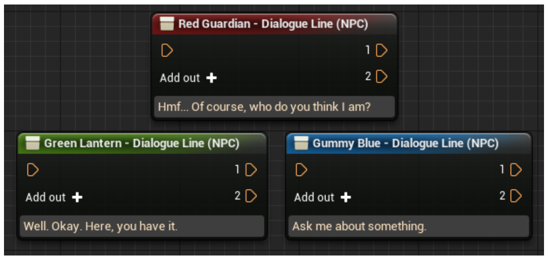
First of all, on each Dialogue Line/Option node, you will find a Metadata in the Details panel. This is a Name to String map that can store a lot of data of your choice for each line or option you have in your dialogue. Examples: speaker's name, animation to play, speaker's mood, required skill level or score for a special option, etc. It's a map, so you can add anything you want in it.

Usually, you would define default entries that will show up in every node, but then you can add specific entries to certain nodes, depending on your needs. Default entries are defined in the Project Settings > Editor > Dialogue. You will find two separate sections here, one for the lines and one for the player's options. In each of them, you can setup three things:

1. **Metadata parameters:** this is the default map to use as metadata parameters. You can have as many entries as you want, and you can set default values or let them empty.
2. **Title parameter:** here, you can type the name of one of the entries. In the editor, whenever this entry is found into a node, it will put the corresponding value and display it directly into the node's title! This is very useful for displaying the speaker's name more clearly, for example.
3. **Title color map:** once you've defined a Title parameter, you can specify custom node colors! When a specific value is found for the Title parameter, the corresponding color will be applied to the node, which makes it even more practical to highlight the main characters' lines.


Dialogue Lines

| | |
|---|---|
| Metadata Parameters | 1 Map elements |
| Character | |
| Title Parameter | Character |
| Title Color Map | 3 Map elements |
| Gummy Blue | 🟦 |
| Red Guardian | 🟥 |
| Green Lantern | 🟩 |


Red Guardian - Dialogue Line (NPC)
Hmf... Of course, who do you think I am?

Green Lantern - Dialogue Line (NPC)
Well. Okay. Here, you have it.

Gummy Blue - Dialogue Line (NPC)
Ask me about something.

*Picture 13 – Example of Dialogue Lines settings.*

## Sounds and Dialogue Waves

On each Dialogue Line/Option, you can link Sound assets and Dialogue Waves assets. They can be retrieved during dialogue execution in the Line Struct and in the Option Structs to be played accordingly to the player's choices.

## Right-click magic

Many elements of the Dialogue Nodes offer shortcuts when you right-click on them, making your dialogue design process faster and lighter.

When you right click on an outcome Dialogue pin, you will be offered the options of removing that pin, or linking a new Line or a new Option node to it. You have the same options when you right-click on the node itself or on the income pin, which then also creates a new outcome pin.

Newly created Dialogue Lines this way (and only them), will copy the metadata parameters from the previous Dialogue Line node. If the previous node is an Option, it will look one level further to see if there's a Line there. Then it stops and just adds a default Line. This will considerably accelerate your dialogue writing as the speaker's name will be copied from a line to another without requesting you to type it everywhere or to copy-paste the nodes.

*More workflow enhancements will come in the next versions of the plugin.*
*For them to be accurate, they will exclusively rely on user feedbacks.*

*If you have any good idea, please share it!*

# Technical details

The source code of this plugin can be found in the Source folder. However, you might want some explanations to understand the general behavior of the system, as it goes against some Unreal Engine basic rules. Here, we will go over the way this system works, without entering into too much details.

## Blueprint compilation process

If you are familiar with the Blueprint compilation process from Unreal Engine 4 teams, you might have understood that this dialogue system is going against the rules at many levels. First of all, although Dialogue Flow Pins are looking a lot like execution pins, they don't carry an execution flow but plain old data as any data pin, and bring that "back" to the Make Dialogue node which builds a self-contained Dialogue object from this data. Therefore, we have a pin following connection convention of execution pins, but carrying data "backward" like reversed data pins.

Secondly, once you notice Dialogue nodes are somewhat pure nodes, and are only gone through once at Dialogue registration, you might ask how conditions, text parameters and other data can ever be dynamically evaluated. Indeed, they should take data once at registration time and never read it again, right?

In order to counteract in a proper way these Blueprint conventions, this system might have been done in an external graph editor (you can find several dialogue plugins on the marketplace with new graph editors), but the direct integration into Blueprints is really valuable for a fast and easy dialogue design process.

### Node expansion

The solution that was chosen to create this system was to take advantage of a Blueprint compilation step called Node Expansion. When you hit the "Compile" button in the BP editor, it launches a complex process which goes through all your BP graphs and transforms them into executable code. See Unreal Engine documentation for more information about this process. Here, we will focus on an early step: just after the graphs are gone through a node-developing pass and purging pass (nodes that aren't relied on by executed nodes, like unconnected nodes or groups of nodes, are purged), each remaining node is given a chance to **expand**.

Expand means that a node is replacing or augmenting itself with other nodes, created at compile time and invisible to the designer. In the case of this dialogue system, every single node you see in the editor is a frontend node that is replaced at compile time by several other nodes. Sometime, one frontend node can be replaced with up to 5 backend nodes! There are mainly two things happening during node expansion of dialogue nodes:

1. Nodes are replaced by their pure equivalent, which are unavailable to designer but basically are just nodes from a blueprint function library in C++. This is used to convert from frontend nodes to backend nodes which create the main component of dialogues: dialogue events.

2. Each node which is relying on some dynamic data (any node taking data from a classical data pin) generates more nodes: an event node to trigger at runtime, setters, new properties, etc. Further details in next section.

What you should remember there is that dialogue nodes don't "do" anything when they are browsed at runtime. Their role is only to build Dialogue Events which are objects containing references to other Dialogue Events, plus a set of properties.

The Make Dialogue node is the first one to be triggered, which triggers a data-dependency chain reaction towards the end leaves of the graph. Then, the data connections are traced back, forming a consistent chain of Dialogue Events, which is finally brought back to the Make Dialogue node. The starting point of this chain is stored in a new Dialogue object which is finally returned as the output of the Make Dialogue node.

### Events and properties generation

As mentioned above, when a node is relying on dynamic data, or when its expected behavior is to trigger a Blueprint execution flow (like To Blueprint node), it will generate a Custom Event node at compile time and give it a unique generic name.

Then, for each data value that it should retrieve, a new property is created and is given a unique generic name. These properties are there to store the value temporarily for the Dialogue object to retrieve it afterward. These properties are set by setter nodes: one of them is created for each property and they are connected in chain to the event node.

At runtime, when the Dialogue event is executed and if we ever come to the point corresponding to that dialogue node, the custom event is triggered, which retrieves all the dynamic data values at once, and then these values are read by the Dialogue object thanks to the generated properties whose names have been stored before.

### Loops and common outcomes

If you're familiar with graph theories, you might see a problem there: if there is a loop in the dialogue graph, or more generally common outcomes (several node's outcomes going to the same node's income), we might create a dependency loop. This is forbidden by Unreal Engine Blueprint Editor, and you can test it by making your own loop with pure nodes. Execution wires aren't ruled by this restriction as they don't represent data flows, which is why all dialogues nodes work with execution lookalike pins.

But indeed, this is a hack. Although Dialogue Flow Pins are a lot like Execution Pins, we saw earlier that they only are disguised data pins. To counteract this rule, all loops are broken at compile time, during node expansion. In fact, classical node expansion happens quite randomly among nodes, which isn't usually an issue as long as each node is given a chance to expand. However, for dialogue nodes, we overrode the classical expand method and made a recursive one: beginning with the Make Dialogue node, each dialogue node is responsible of triggering the outcoming nodes expansion phase.

Then, if we detect that one of the outcomes was already expanded, it means that a loop or a common outcome was detected! Then, we break the direct data connection between the two nodes and replace it by special "GoTo" nodes responsible of establishing a reference bridge into the dialogue. These GoTo nodes will be forgotten in the end, once Dialogue object is reaching its final check pass, inside the Make Dialogue node, replacing them by direct pointers on already created objects.

In conclusion, this whole process might look quite complicated, especially when dialogue graphs are getting more complex, but actually this is the only solution I found to make the dialogue design process as most user friendly and game agnostic as I could. Full Blueprint integration required this system to go against the conventions, only to get back on tracks again in the end with intuitive frontend nodes.

# Troubleshooting

Contact me on Unreal Engine's forums if you encounter any issue with the plugin, or have any improvement to suggest. I try to be as available and helpful as possible, but remember that I'm developing this plugin alone on my free time!

| | |
|---|---|
| **Nickname:** | CraftyWeazel |
| **Plugin thread:** | https://forums.unrealengine.com/community/work-in-progress/1562806-blueprint-ready-dialogue-system-c |