Questryn — User & Customization Guide

> This guide covers two things together, on purpose: how to **use** the Questryn editor to author dialogues, and how the **example Blueprint project** included with the plugin implements everything around those dialogues (conditions, text tags, rich text, sounds, widgets). They're not really separable — features like path conditions, `{Tag}` text substitution and rich text markup only work end-to-end once you've adapted the Blueprint side to your own project (your items, your data tables, your character class). Read this guide top to bottom once, then come back to individual sections as reference.

Table of Contents

1. What Questryn Is

Questryn — Node-Based Dialogue System is a dialogue plugin for Unreal Engine 5. A dialogue is authored as a small **graph**: a Start Node feeds into a Dialog Node, and each Dialog Node can branch into further Dialog Nodes through labeled **paths**. Each Dialog Node holds one or more lines ("SubDialogues") — dialog title (usually the speaker), text(s), image(s), sound(s), and duration.

The graph reads top-to-bottom, like a script, instead of the left-to-right layout most Unreal graph editors use — that's a deliberate choice, so a branching conversation stays readable as a conversation.

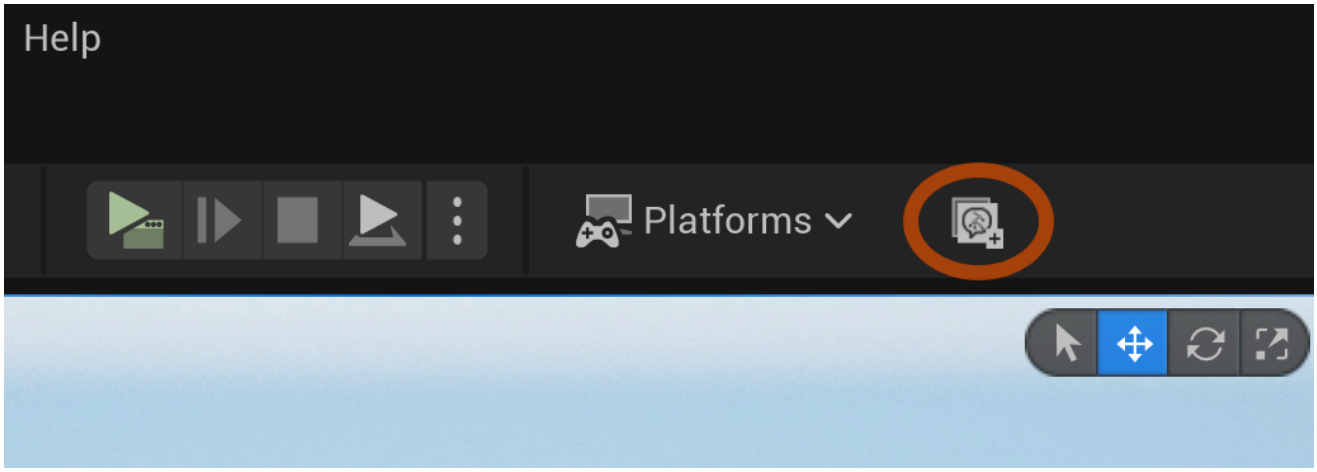
Everything you build in the graph is **data**. The plugin's C++ side owns the dialogue's actual flow — advancing through nodes, the priority/Lower Priority Behavior system, blocking actions — but hands off all *presentation* to Blueprint: which widget shows up, how text is revealed, what happens when a condition is checked, is driven by an example **Blueprint project** shipped alongside the plugin, which you're meant to adapt (Section 5).

2. Getting Started: Setting Up Your Own Copy

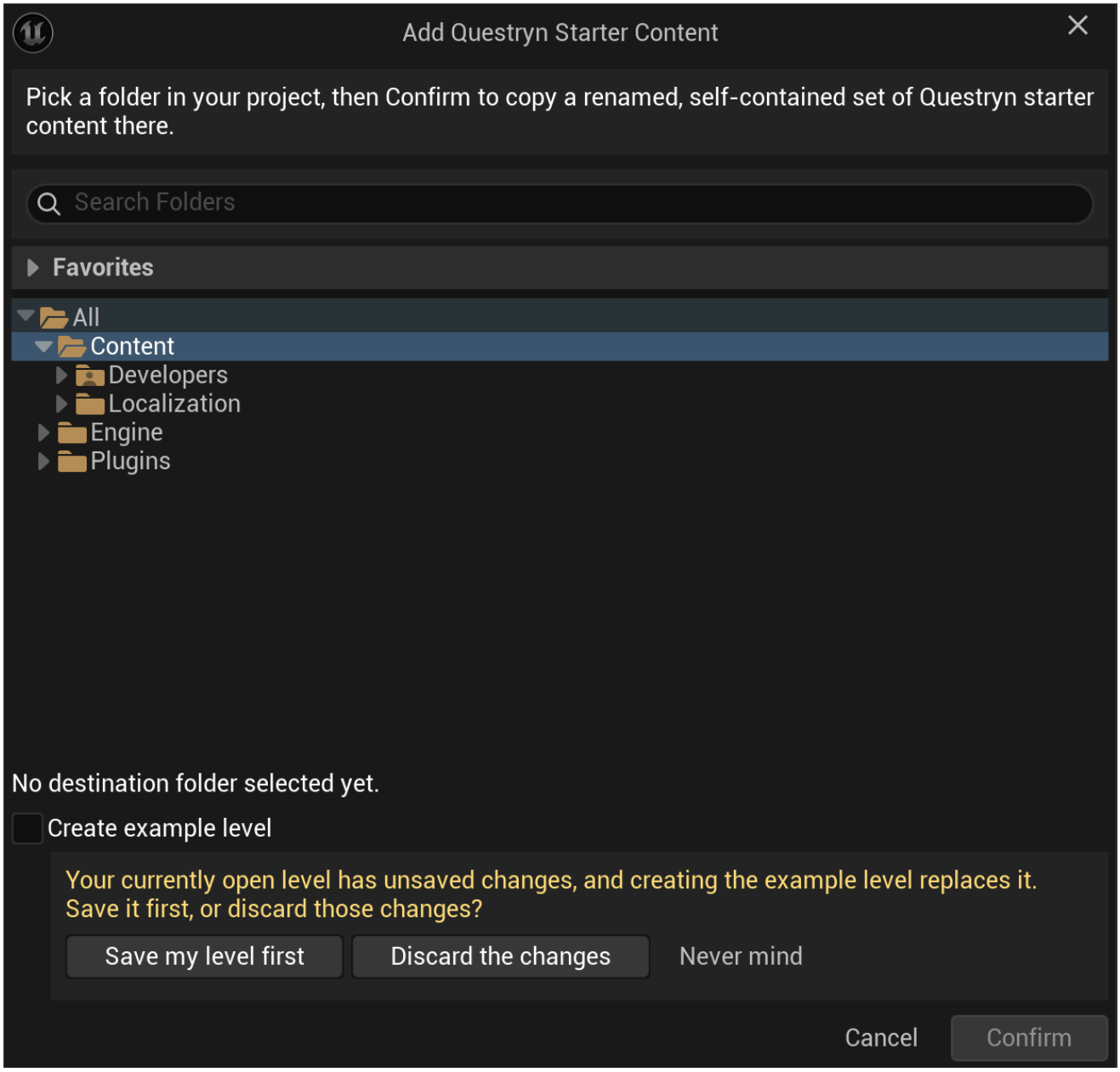
The plugin ships with a full set of example content — Blueprints, DataTables, Widgets, an example map — living under the `/Questryn/` content mount point, used to demonstrate every feature end-to-end (`PC_Questryn_Example` , `Character_Questryn_Example` , `BP_Questryn_Trigger_Base` , `WB_Questryn_Dialogue_Example` , and so on). This example content is meant to be **read, copied, and eventually replaced** — not built on top of directly.

Recommended way to start your own project:

1. **Decide what you actually need.** You rarely need all of it on day one. A common starting set is: your Player Controller's blocking-actions handling, your condition-check functions, one Trigger Blueprint (a **copy** of `BP_Questryn_Trigger_Base` — copying is the supported path here, not subclassing it), the main dialogue widget and its companion widgets, and the 4 Blueprint Interfaces.
2. **Copy the assets into your project.** Two ways to do this:
   - **"Add Starter Content" button (recommended)** — a one-click shortcut for the common case. In the Level Editor toolbar, click **Add Starter Content**, pick a destination folder in your project in the picker that opens, then **Confirm**. This copies the full starter set — every Blueprint, Widget, Interface, the example Trigger Box, the example dialogues, the procedural speech-bubble material, and the supporting DataTables/Structs/Enums they depend on — into `<your folder>/QuestrynStarterContent/` (split into `Blueprints/` , `UI/` , `Materials/` , `Data/` , `Dialogues/` and `Badge/` subfolders), with each asset already renamed with a `Starter_` prefix (e.g. `Starter_BP_Questryn_Trigger_Base`) and cross-references between the copies already pointing at each other — no manual "Migrate", no risk of missing a dependency. The one exception is textures and sounds: those stay referenced to the plugin's own `/Questryn/` example media, since they're not meant to be edited and duplicating read-only art assets wouldn't buy you anything.



It also builds you a ready-to-play `Maps/Starter_Map_Questryn` — a fresh level (not a copy of the plugin's own example map; bulk-copying a Level isn't reliable in UE5's asset tools) populated with the copied Trigger Box placed three times, each wired to one of the copied example dialogues, a PlayerStart, a floor, and basic lighting. Open it and press Play to test the duplicated assets immediately, fully independent of the plugin's own `/Questryn/` content. This is controlled by the **"Create example level"** checkbox in the panel, which is ticked for you unless the level you currently have open has unsaved changes — creating the example level replaces that open level, so in that case you have to tick it yourself, and you'll be asked right then whether to **save** the open level first, **discard** its changes, or leave the box unticked. If `Starter_Map_Questryn` already exists at the destination from a previous run, it's left alone (delete it first if you want a fresh one) and the panel says so under the checkbox. If any of the copied assets already exist at the destination, you'll get a confirmation prompt before anything is overwritten. Don't need this button around? It can be hidden per-project via **Project Settings → Plugins → Questryn → "Show 'Add Starter Content' Button"**.



- **Manual copy from inside the Editor** — use this instead if you want to hand-pick a subset rather than take the whole starter set. The example content lives at `/All/EngineData/Plugins/Questryn` in the Content Browser — if you'd rather click through than type a path, enable *Show Engine Content* and *Show Plugin Content* in the Content Browser's view options, then navigate **All** → **Engine** → **Plugins** → **Questryn - Node-Based Dialogue System Content**. From there, use copy/paste or "Migrate..." into your own Content subfolder. Don't copy `.uasset` files outside the Editor (through the OS file explorer); Blueprint/DataTable assets are binary and editing or moving them outside the Editor risks corrupting them.

3. **Rename your copies** to match your project's naming convention (drop the `Starter_` / `_Example` naming, use your own prefixes). This keeps them clearly distinct from the plugin's reference copies, and means your game no longer depends on the plugin's `/Questryn/` example content once you've made the switch.
4. **Re-point references** in your copies: DataTable Row Struct types, the Data Table defaults used by your Dialog Nodes, which classes implement the 4 interfaces, the Widget Class / Dialog Trigger Class fields on your dialogues, etc.
5. **Only then** start working through the `CHANGE RECOMMENDATION` / `POSSIBLE CHANGE` comments (Section 3) with your own logic.

You don't have to do this all at once. Plenty of projects keep using `BP_Questryn_Trigger_Base` itself for a while and only swap out the pieces they need (commonly just `CheckCondition` and `RunData`), while everything else still runs on the example implementation. One piece isn't optional, though: if you keep the example widget's rich-text rendering (the `<bigtext>`-style markup), you need to keep or adapt the Rich Text Style Table it reads from — Unreal's Rich Text Block can't resolve any markup without a valid style table assigned.

3. Reading the Comments: POSSIBLE CHANGE vs. CHANGE RECOMMENDATION

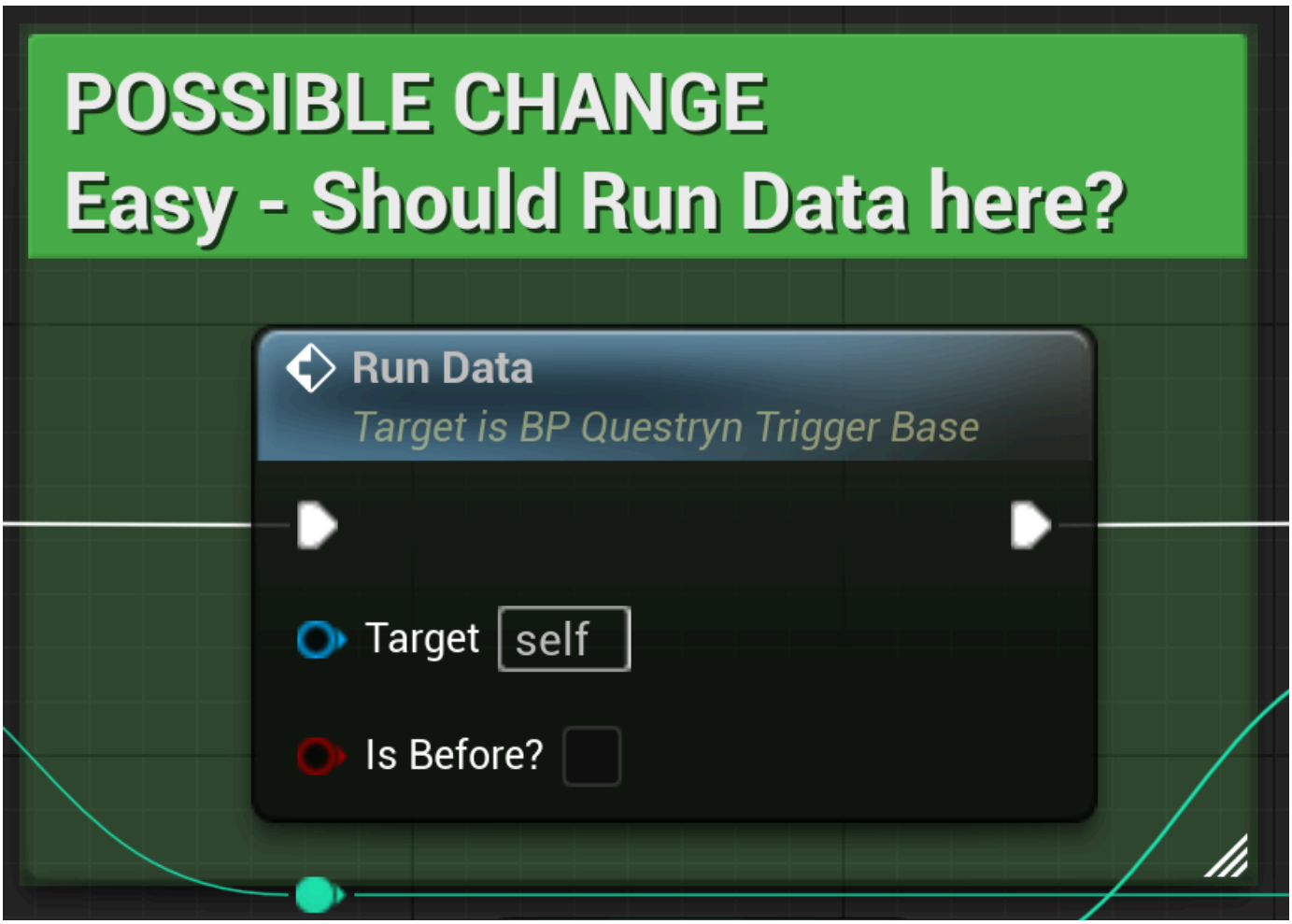
Throughout the example Blueprints you'll find color-coded comment boxes marking points meant for you to look at. **None of them are required for the plugin to work** — the example project runs as-is. They exist to tell you where and how to adapt it to your own game.

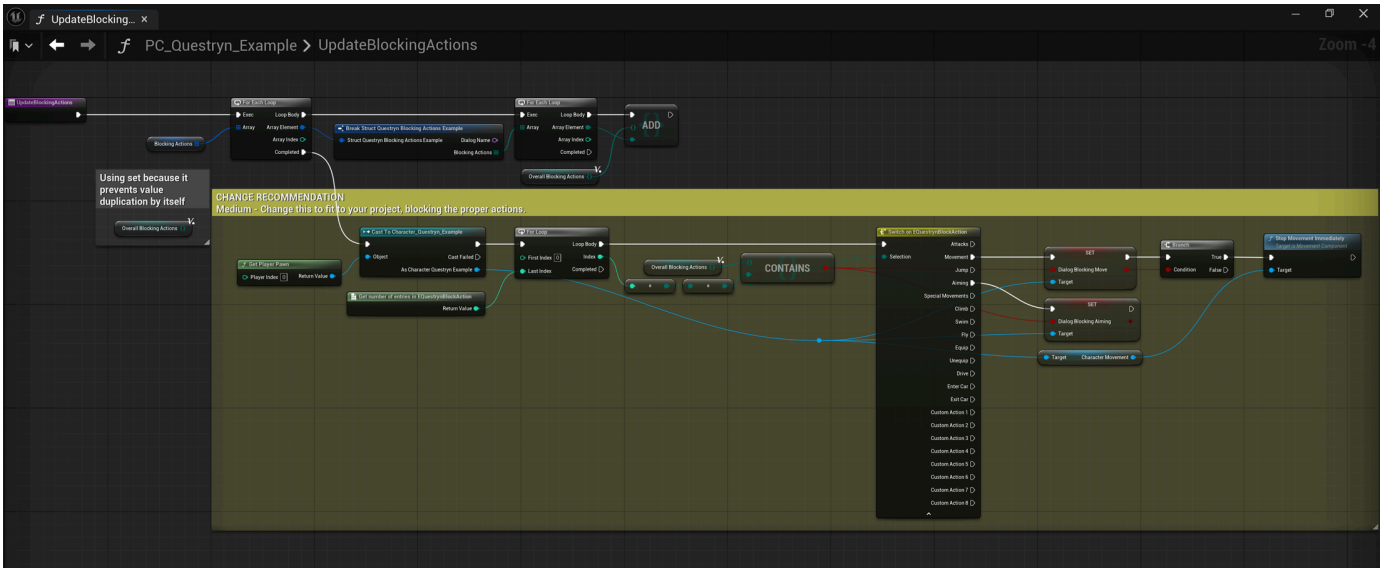
	Meaning
POSSIBLE CHANGE	Something you *can* change if it doesn't fit your project, or can remove entirely if you don't need the feature. Not a recommendation either way.
CHANGE RECOMMENDATION	Something the author suggests you adapt, specifically so you get the plugin's full potential in your own project (e.g. wiring conditions to your actual item/quest system instead of the example one).

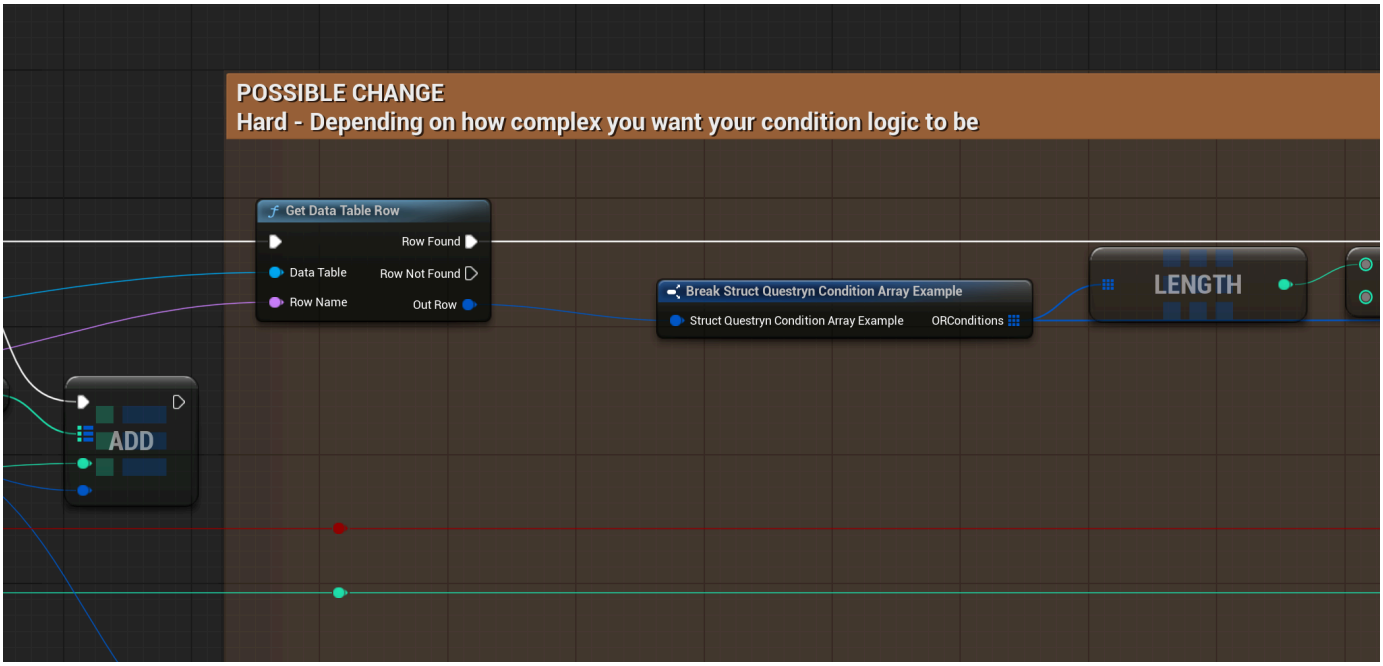
Color marks **difficulty**, independent of the category above:

Color	Difficulty
Green	Easy
Yellow	Medium
Orange	Hard — usually "hard" because it depends on how complex *you* want your own logic to be, not because the change itself is exotic.

Examples, straight from the Blueprints:





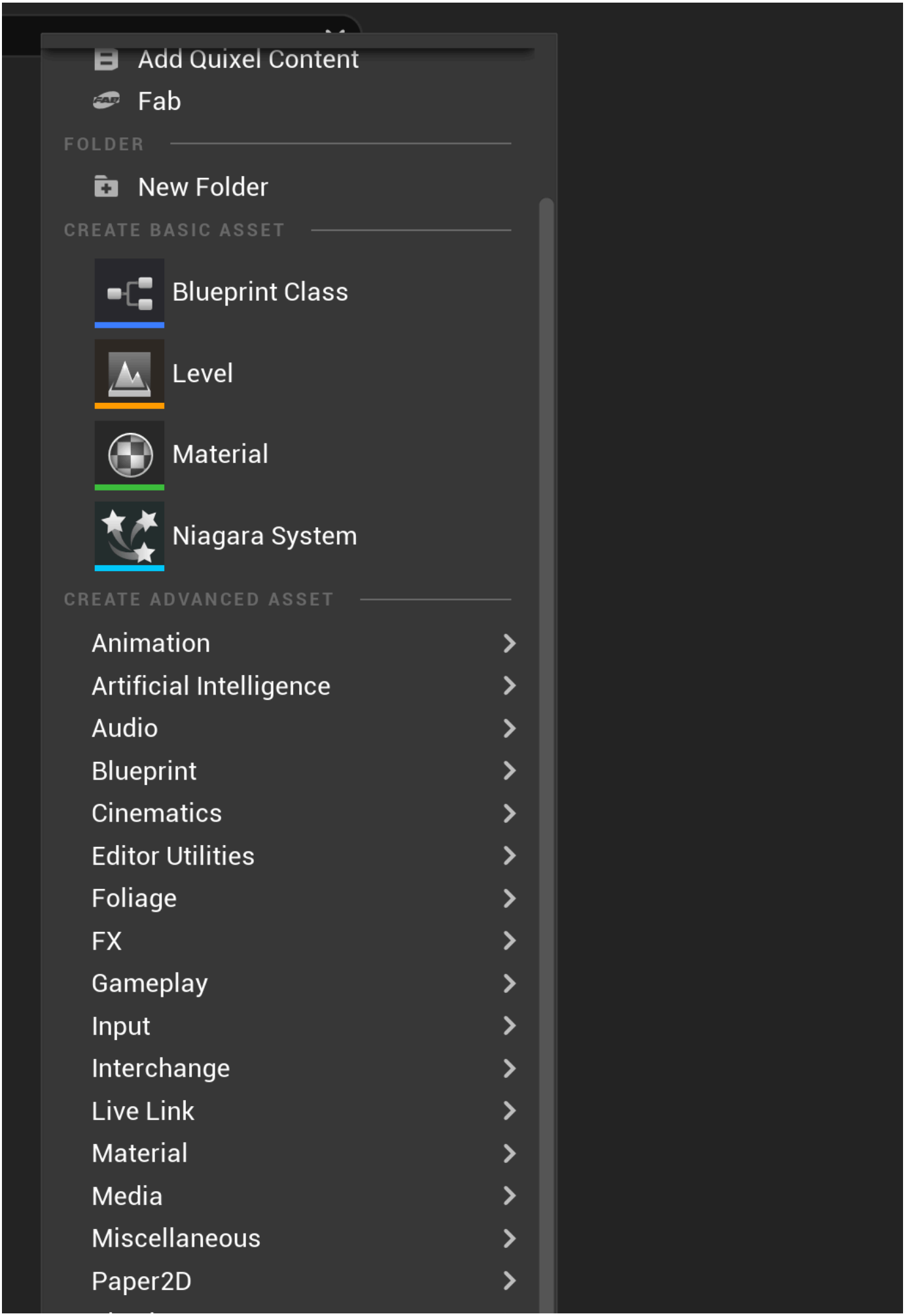


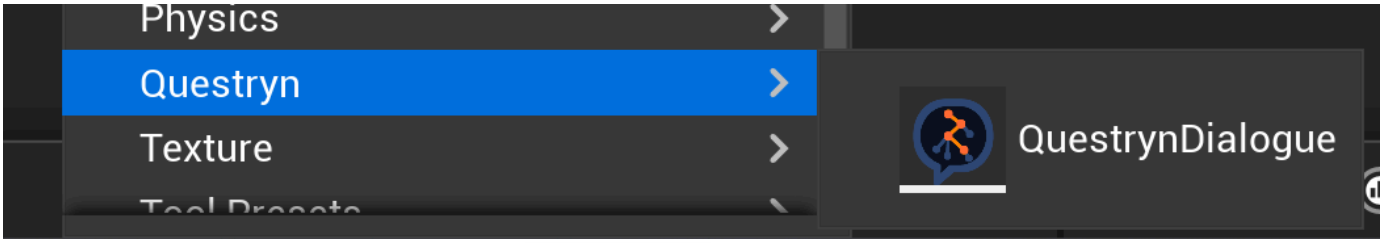
Keep this legend in mind for the rest of Section 5 — instead of repeating every single comment box, this guide summarizes what each function does and calls out the noteworthy ones.

4. The Editor Workflow

4.1 Creating a QuestrynDialogue Asset

A dialogue lives in its own asset type, `QuestrynDialogue`. Create one through the Content Browser's right-click menu — **Create Advanced Asset → Questryn → QuestrynDialogue** (or just type "Questryn" into the create-menu search):

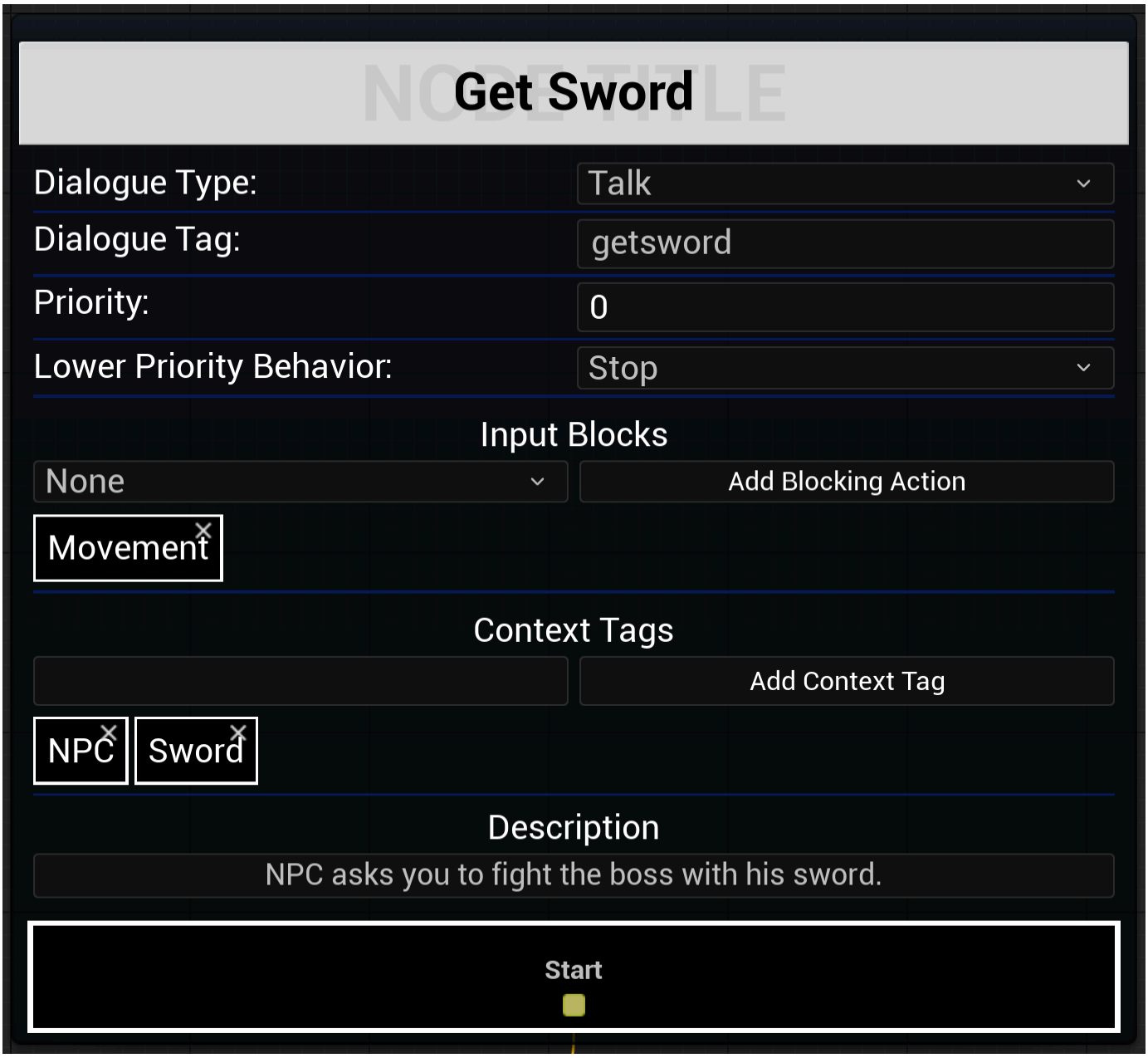

Add Quixel Content
Fab
FOLDER
New Folder
CREATE BASIC ASSET
Blueprint Class
Level
Material
Niagara System
CREATE ADVANCED ASSET
Animation
Artificial Intelligence
Audio
Blueprint
Cinematics
Editor Utilities
Foliage
FX
Gameplay
Input
Interchange
Live Link
Material
Media
Miscellaneous
Paper2D



This is the only supported way to create a dialogue. `QuestrynDialogue` does **not** appear as a selectable parent class under **Add** → **Blueprint Class** — that route would create an unrelated, empty Blueprint asset (opened in the generic Blueprint Editor, not the Questryn graph editor), not a usable dialogue.

4.2 Start Node

Every `QuestrynDialogue` asset has one fixed Start Node. Instead of hiding the dialogue's configuration in the Details panel, it's exposed directly on the node:

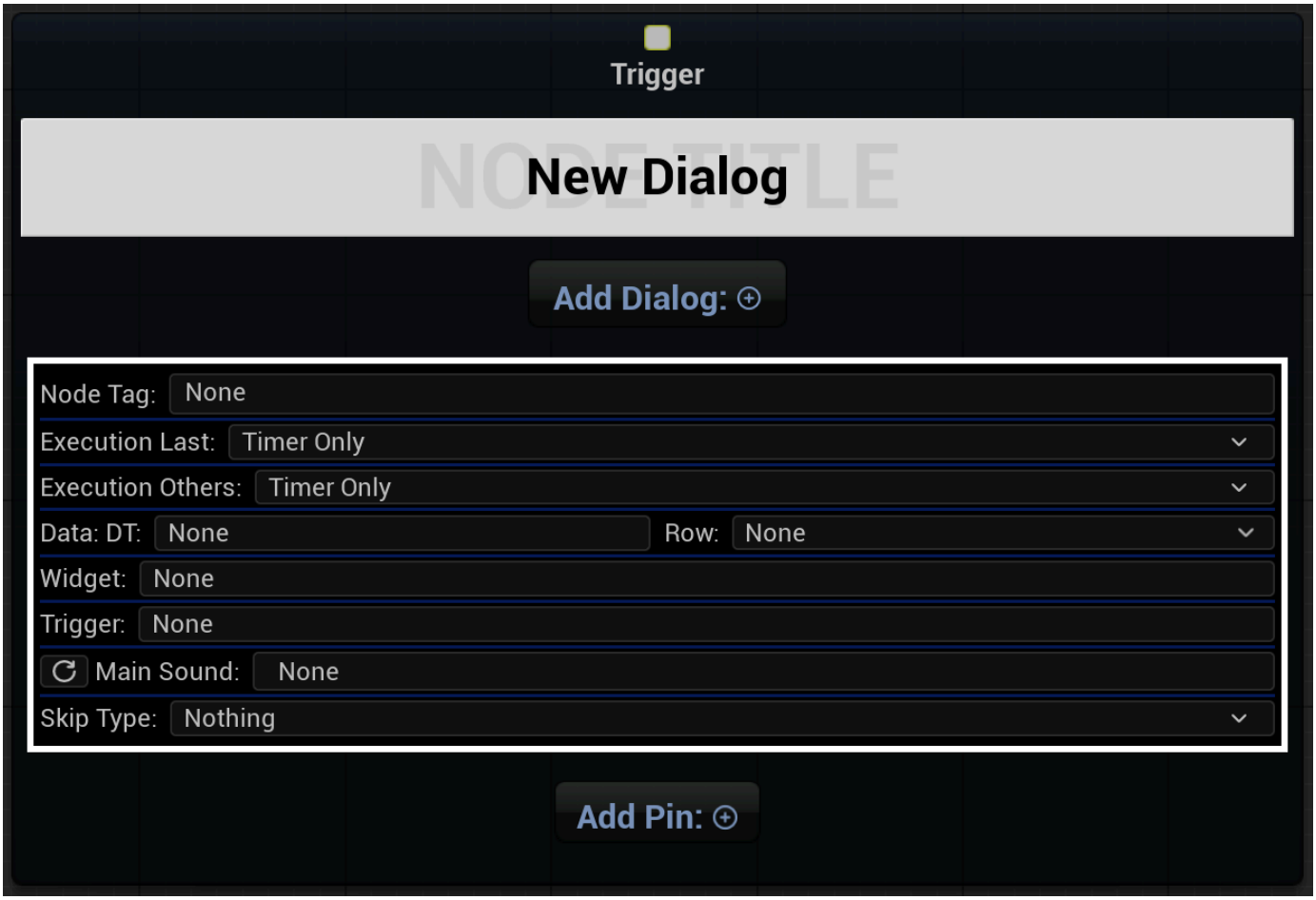


- **Dialogue Type** — a category (Talk, Quest, Notification, HUD, Status Bar, Battle, ...). It reads like a label, but its real job is the priority scope described two bullets down: **dialogues only compete with others of the same type**. That is what lets a HUD, a Status Bar, an Enemy Status Bar and a Talk all sit on screen at once, and it is also the trap to watch — four companion health bars sharing one type will push each other aside unless their Lower Priority Behavior is *Continue*. Four **Custom** slots are there for categories the built-in list does not name.

- **Dialogue Tag** — the identifier gameplay code uses to target this exact dialogue (`Start Dialogue` , `Force Pause/Resume/Destroy Dialogue` , `Execute Next Dialogue` — see 4.9 — all take a Dialogue Tag).
- **Priority** + **Lower Priority Behavior** — what happens to already-running dialogues when this one starts, if they're lower priority (e.g. Stop). This comparison is scoped to **Dialogue Type** and to the **player**: a higher-priority dialogue only forces the Lower Priority Behavior on that same player's running dialogues of the *same* Dialogue Type — a dialogue of a different type, or one belonging to another local player, keeps running untouched, regardless of priority. The same scope decides who comes back afterwards: when a dialogue ends, the highest-priority dialogue still paused *in that scope* is the one resumed.
- **Input Blocks** — player actions blocked while this dialogue runs (Movement, Jump, Attacks, Interact, …) — see how the example Player Controller/Character actually enforce these in 5.1–5.2.
- **Context Tags** / **Description** — free text used only by the Dialogue Browser (4.7); no runtime effect.

4.3 Dialog Node

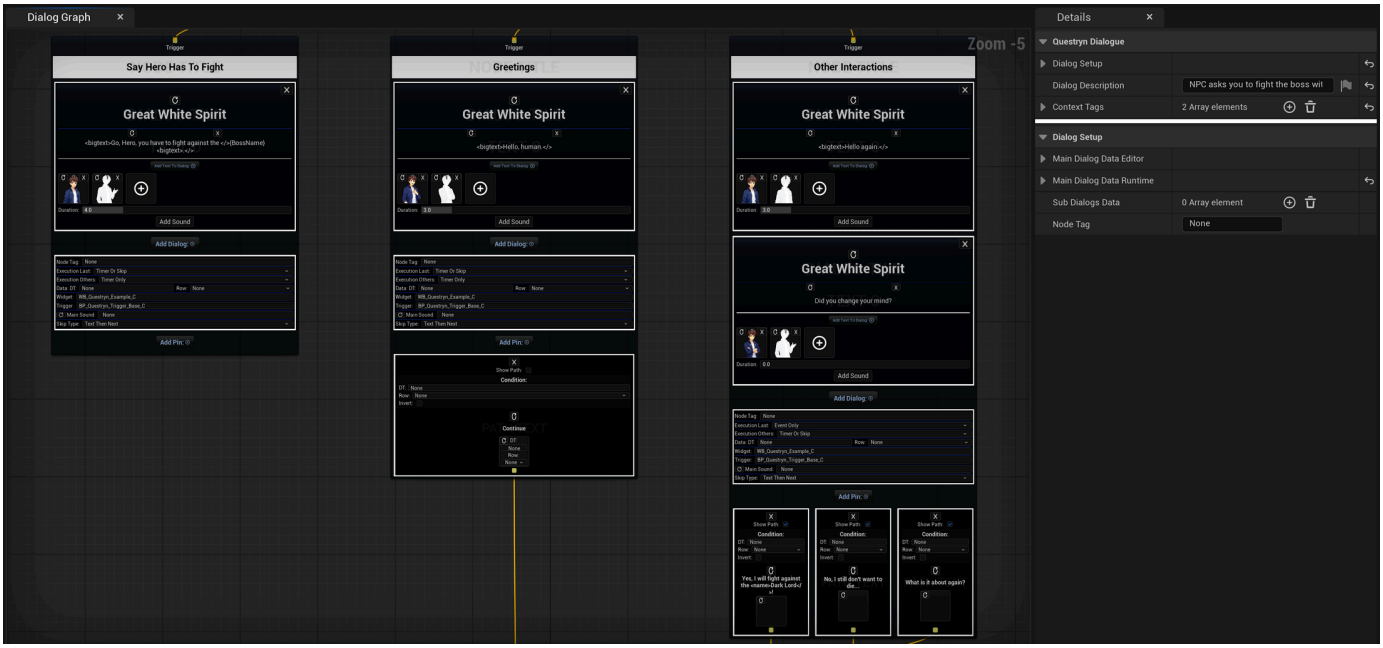
A Dialog Node is one "beat" of the conversation. Empty, it looks like this:



Each **SubNode** (one block per line spoken) has: dialog title (usually the speaker), text (raw rich-text markup, e.g. `<bigtext>Hello.</>`), one or more texture buttons (portraits/images, toggle-able between a direct value or a DataTable row), **Duration**, and **Add Sound**. "Add Text To Dialog" adds another line of text to the same SubNode; "Add Dialog" adds a whole new SubNode.

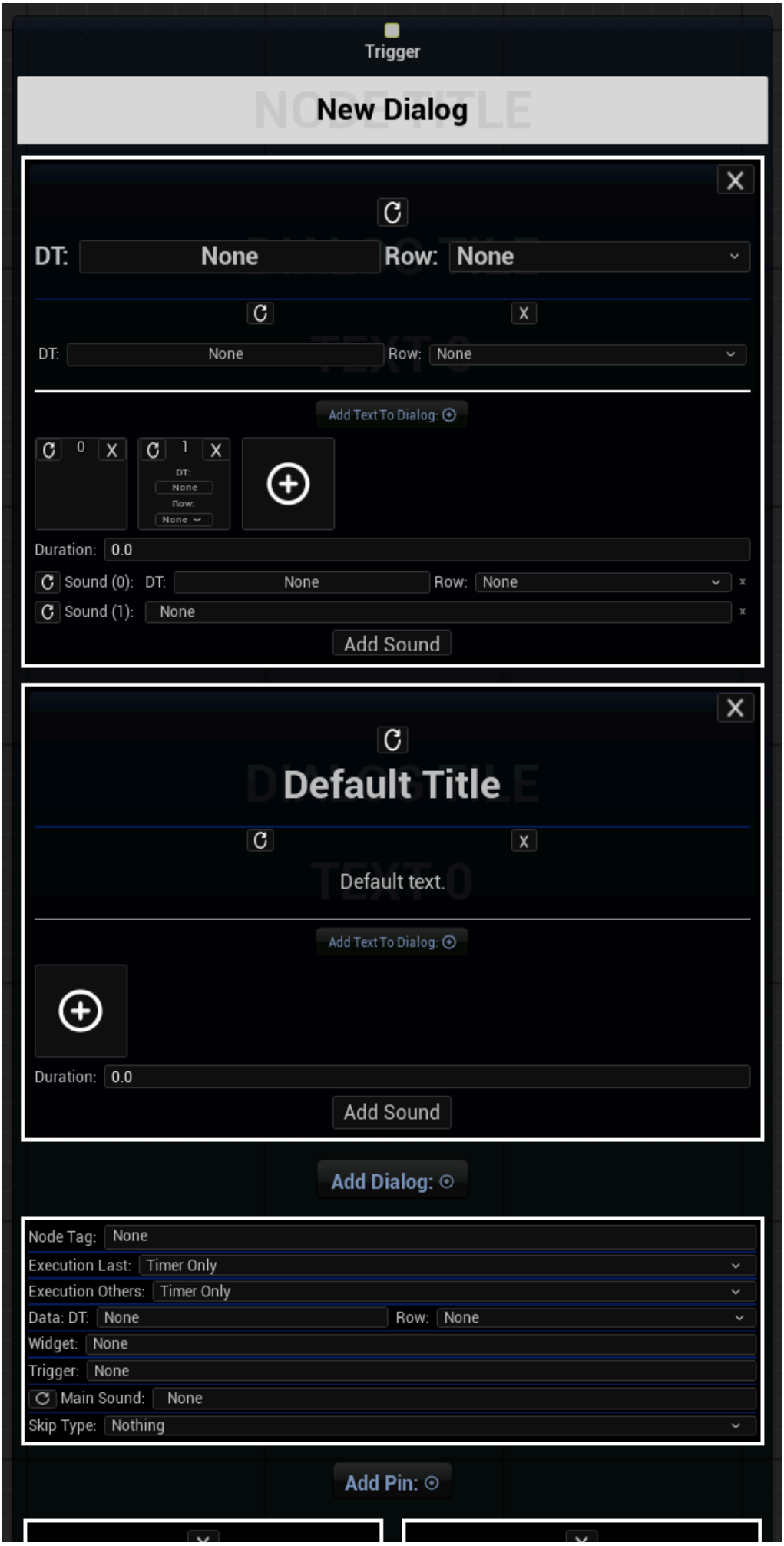
Below the SubNodes, the execution fields control pacing and behavior: **Node Tag** (an optional freeform name for this exact node — on its own it does nothing, but `Start Dialogue` (4.9) can target it to begin the dialogue right here instead of at the beginning, e.g. to resume a tutorial where the player left off), **Execution Last** / **Execution Others** (how the last vs. earlier lines advance — Timer, Timer or Skip, Skip, Event Only, Event with Time Limit), **Data DT/Row** (see `RunData` , 5.3), **Widget** (which widget class to spawn — defaults to the node's own if unset), **Trigger** (which Trigger class handles this node — see 5.3), **Main Sound**, and **Skip Type** (what a "skip" input does here: nothing, skip text, skip-then-advance, advance directly, jump to the last line, stop, or finish the dialogue).

A filled example, alongside the Details panel showing the same data:



4.4 Branching, Paths & Conditions

Every output pin of a Dialog Node is a **path**, and each path is its own customized pin widget — not the engine's default pin:

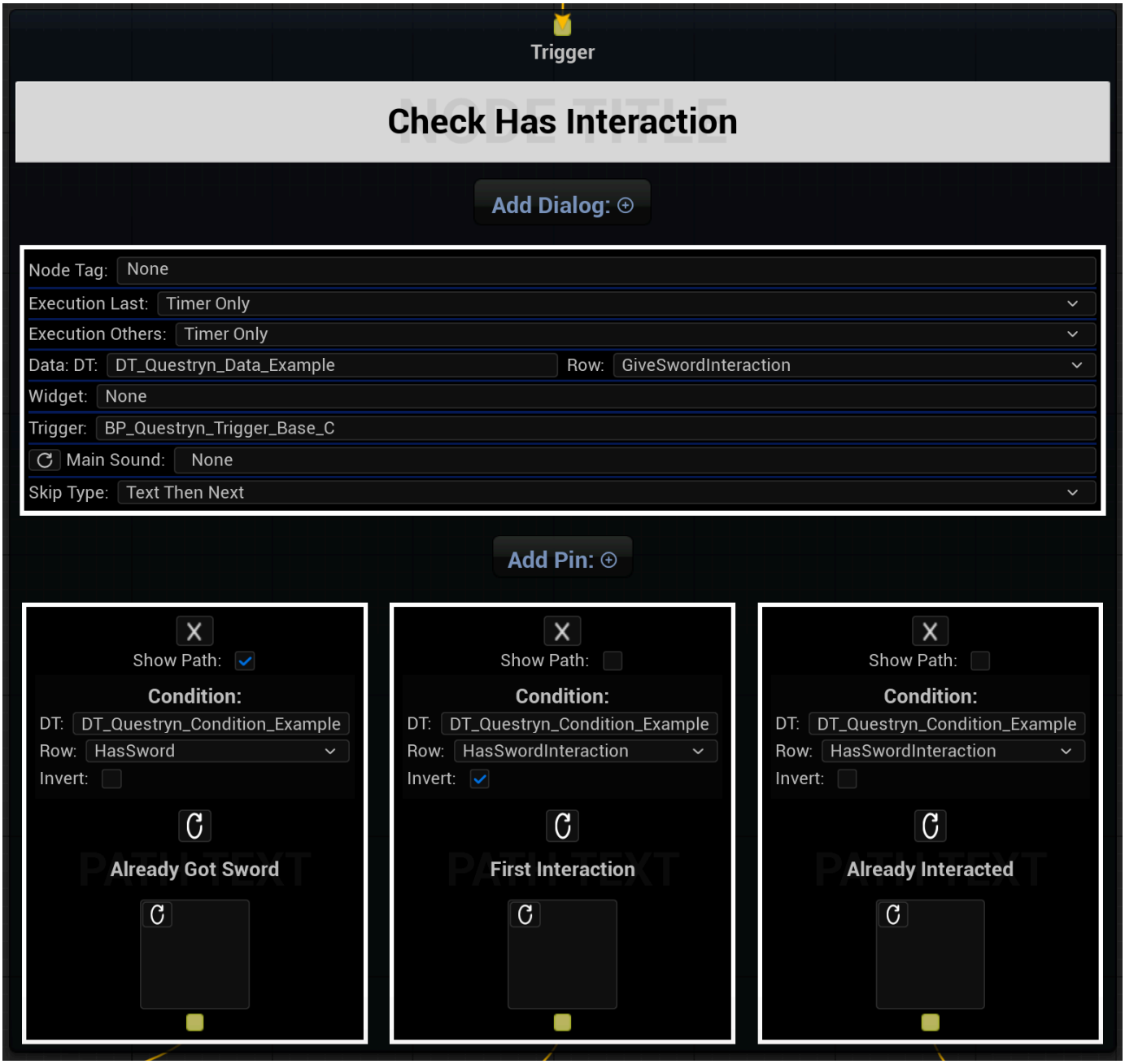

Trigger
New Dialog
DT:
None
Row:
None
DT:
None
Row:
None
Add Text To Dialog:
0
1
DT:
None
Row:
None
Duration:
0.0
Sound (0):
DT:
None
Row:
None
Sound (1):
None
Add Sound
Default Title
Default text.
Add Text To Dialog:
Duration:
0.0
Add Sound
Add Dialog:
Node Tag:
None
Execution Last:
Timer Only
Execution Others:
Timer Only
Data: DT:
None
Row:
None
Widget:
None
Trigger:
None
Main Sound:
None
Skip Type:
Nothing
Add Pin:

Graph connections follow one rule throughout: an output (a path) can only ever connect to a single input, but an input (a Dialog Node) can receive incoming connections from any number of outputs — so separate branches are free to converge back into a shared continuation node.

- **Show Path** — whether this path is presented to the player as a visible/selectable choice, or is an automatic/hidden path.
- **Condition** (DataTable + Row + **Invert**) — gates the path on a condition read from a DataTable row. See `CheckCondition` (5.3) for how the condition is actually evaluated.
- The path's own name (editable), and — when Show Path is checked — an icon representing the choice.

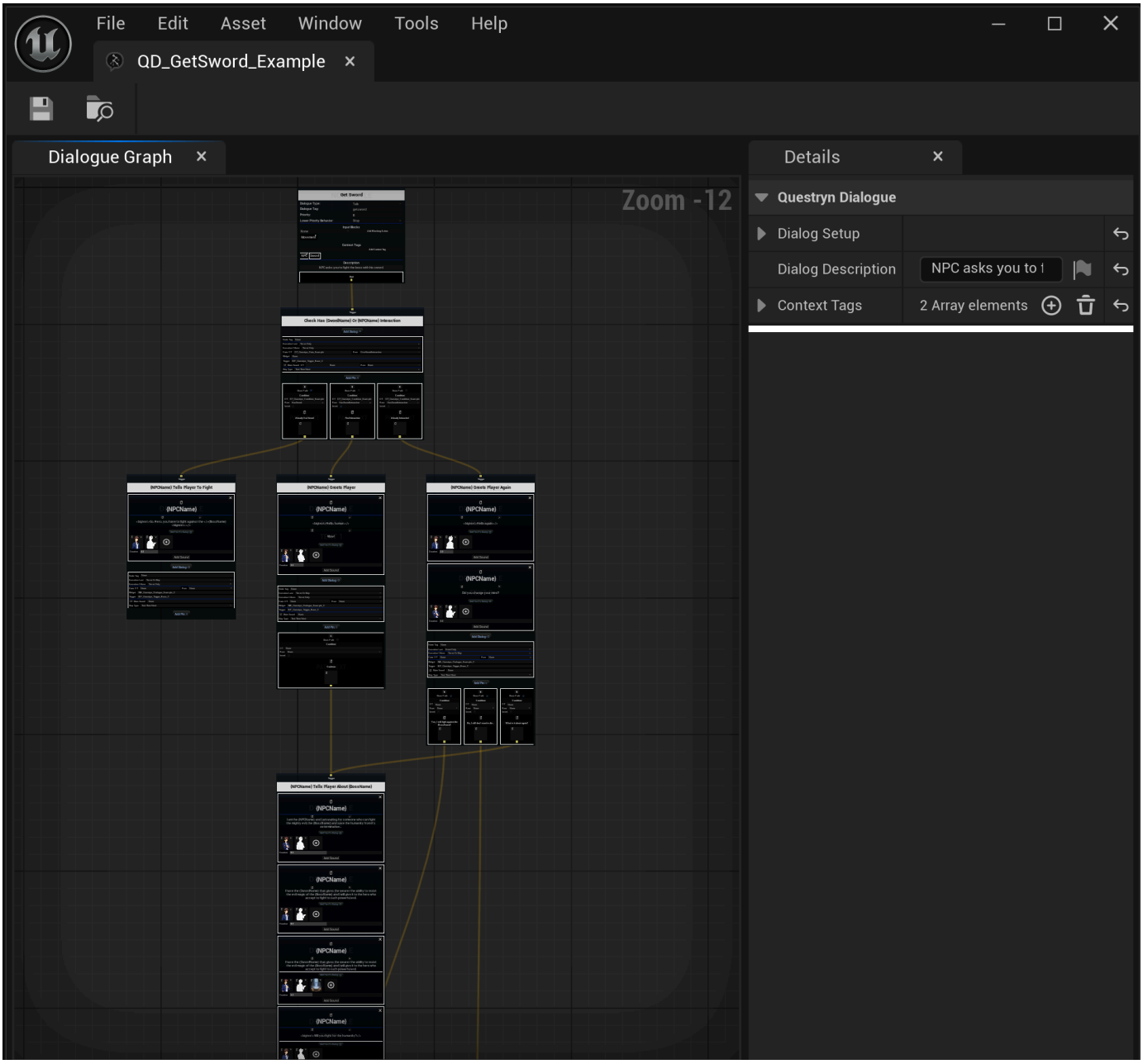
Almost any piece of content on a node (text, sound, texture, …) can alternate between a **direct** value typed/selected right on the node, or a reference to a **DataTable row** — the small toggle icons switch between the two. This is what lets the same dialogue read from live game data instead of hardcoded text.

A Dialog Node doesn't need any SubNodes at all — it can be used purely as branching logic, e.g. to check several conditions and route to different follow-up nodes without showing anything to the player:



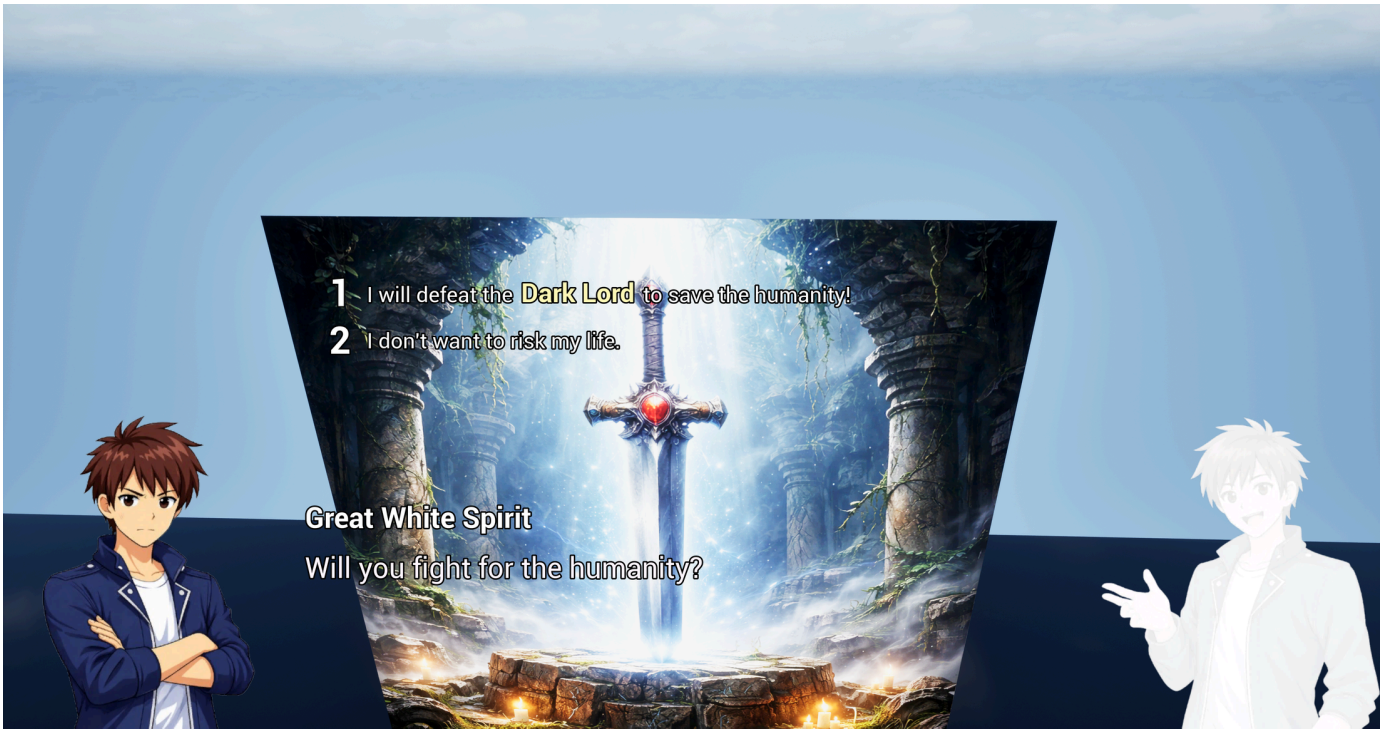
4.5 A Larger Dialogue Graph

Zoomed further out, here's part of a bigger dialogue in the same style — Start Node at the top, branching into a preparatory check (4.4) and from there into several further Dialog Nodes, with the graph continuing below what's shown here. Details panel on the right showing the asset-level setup:



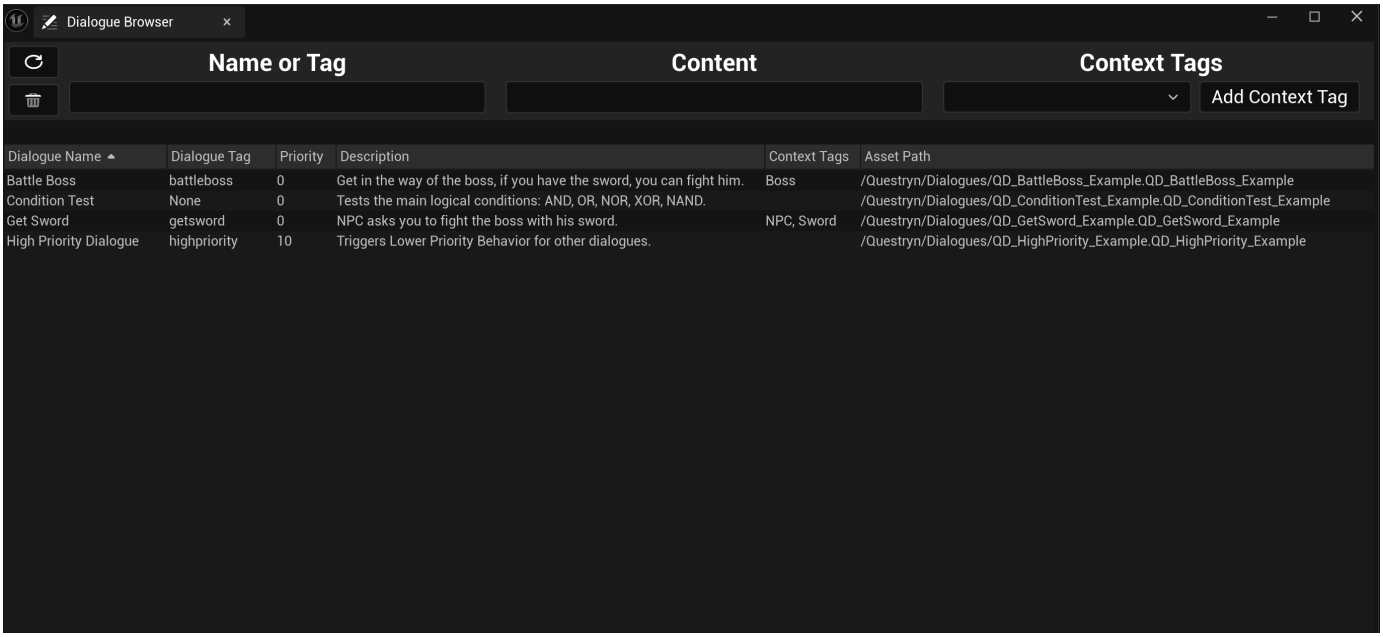
4.6 What the Player Sees

The same dialogue at runtime, using the example widget (portraits, title, numbered choices):



4.7 Dialogue Browser

As the number of `QuestrynDialogue` assets in a project grows, finding them by folder/filename in the Content Browser stops being practical. The **Dialogue Browser** (`Window` menu) lists every dialogue in the project and lets you filter it three ways, which stack: **Name or Tag** matches a dialogue's Dialogue Name or its Dialogue Tag — the same tag you pass to **Start Dialogue** and read off every Player Interface event (4.10), so a tag you found in code is enough to find the asset; **Content** searches inside the dialogues themselves and, when you open a result, carries the search straight into Find in Graph (4.8); and **Context Tags** narrows to dialogues carrying every tag you add to the list.

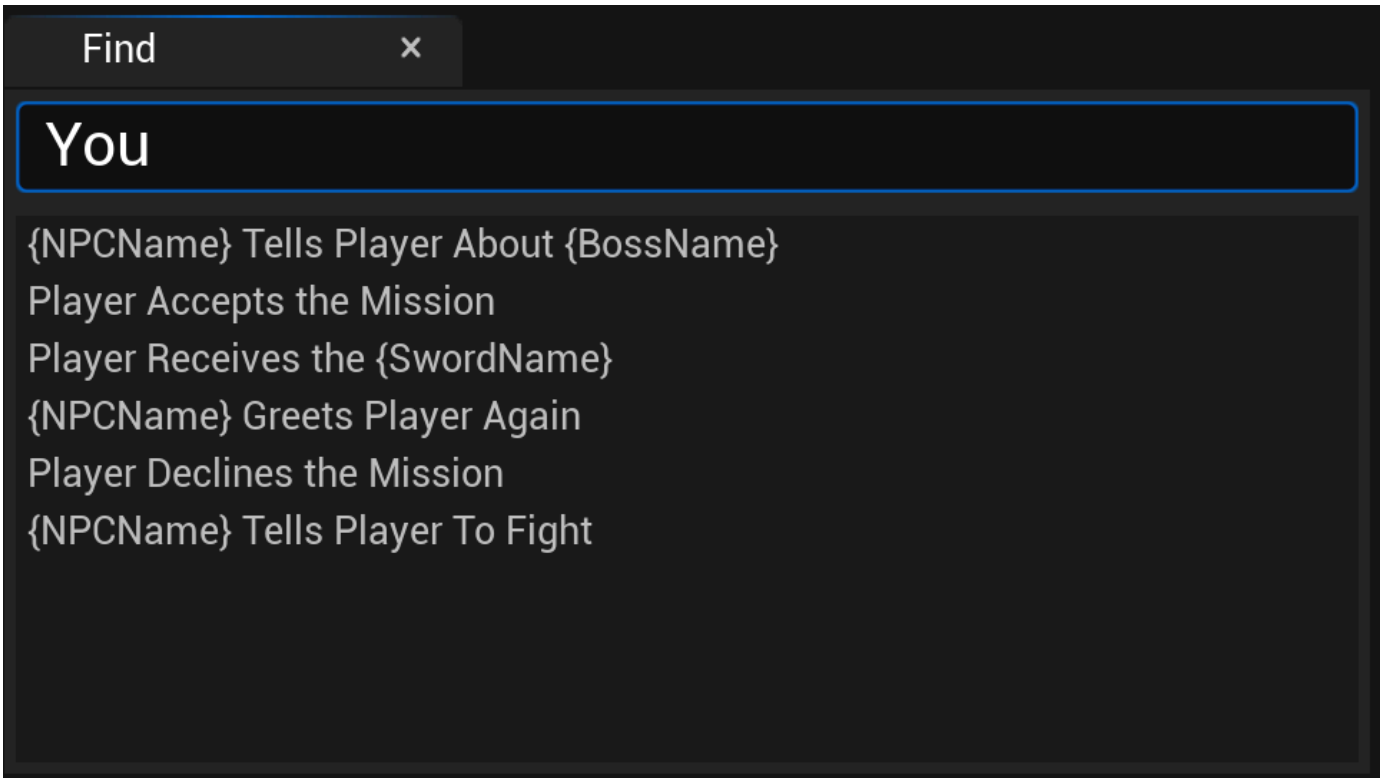


Columns, left to right: **Dialogue Name**, **Dialogue Tag**, **Priority**, Description, Context Tags, Asset Path — the three that identify a dialogue and decide which one wins come first. Every column sorts; click its header.

4.8 Find in Graph

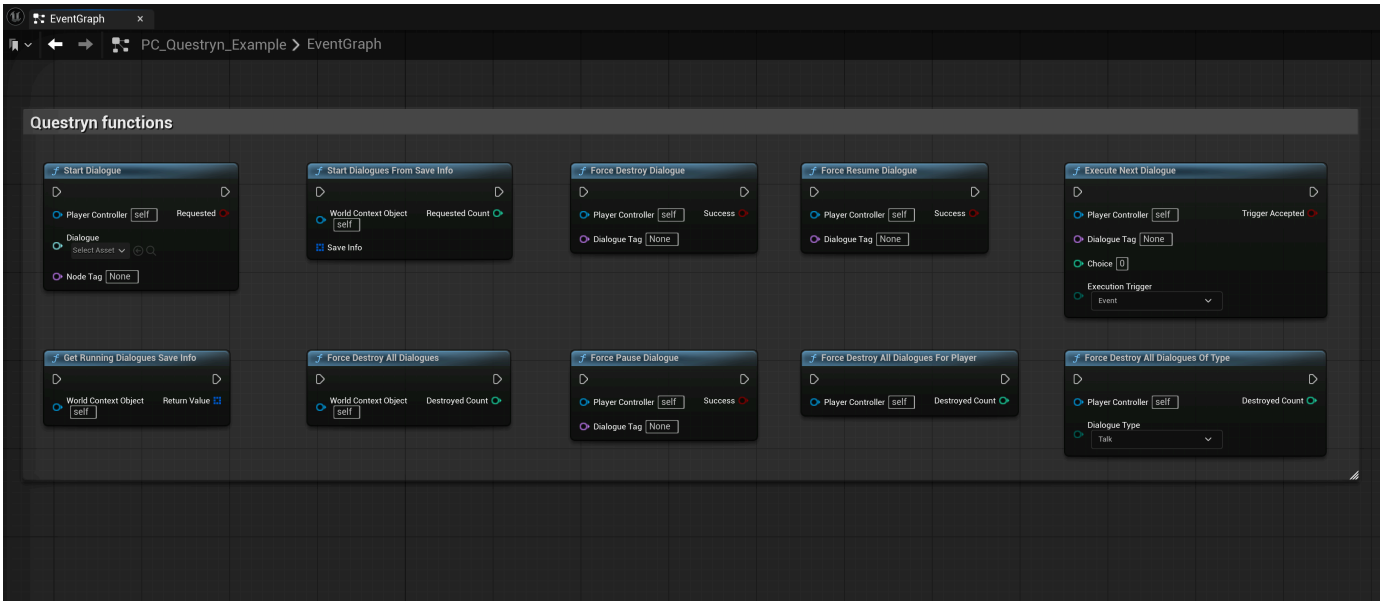
The Dialogue Browser (4.7) finds a dialogue *asset* across the whole project; **Find** does the equivalent search *inside* the one dialogue graph you currently have open — useful once a graph grows past the point where scanning it visually for one specific node is practical.

It's a dockable tab (open by default alongside Details; reopen it anytime via **Ctrl+F** or the graph editor's Window menu). Type into the search box and results update as you type, matching against every Dialog Node's title, Node Tag, DATA table reference, Widget/Trigger class, and main sound, plus — per path option — its text/condition/texture, and — per line (SubDialogue) — its title, text(s), texture(s), and sound(s). Double-click a result, or press Enter with at least one result showing, to jump the graph view straight to that node.



4.9 Calling Dialogues From Anywhere

`UQuestrynFunctionLibrary` exposes Blueprint-callable functions to drive dialogues from any gameplay code — a Player Controller, a Trigger, a cutscene, etc.:



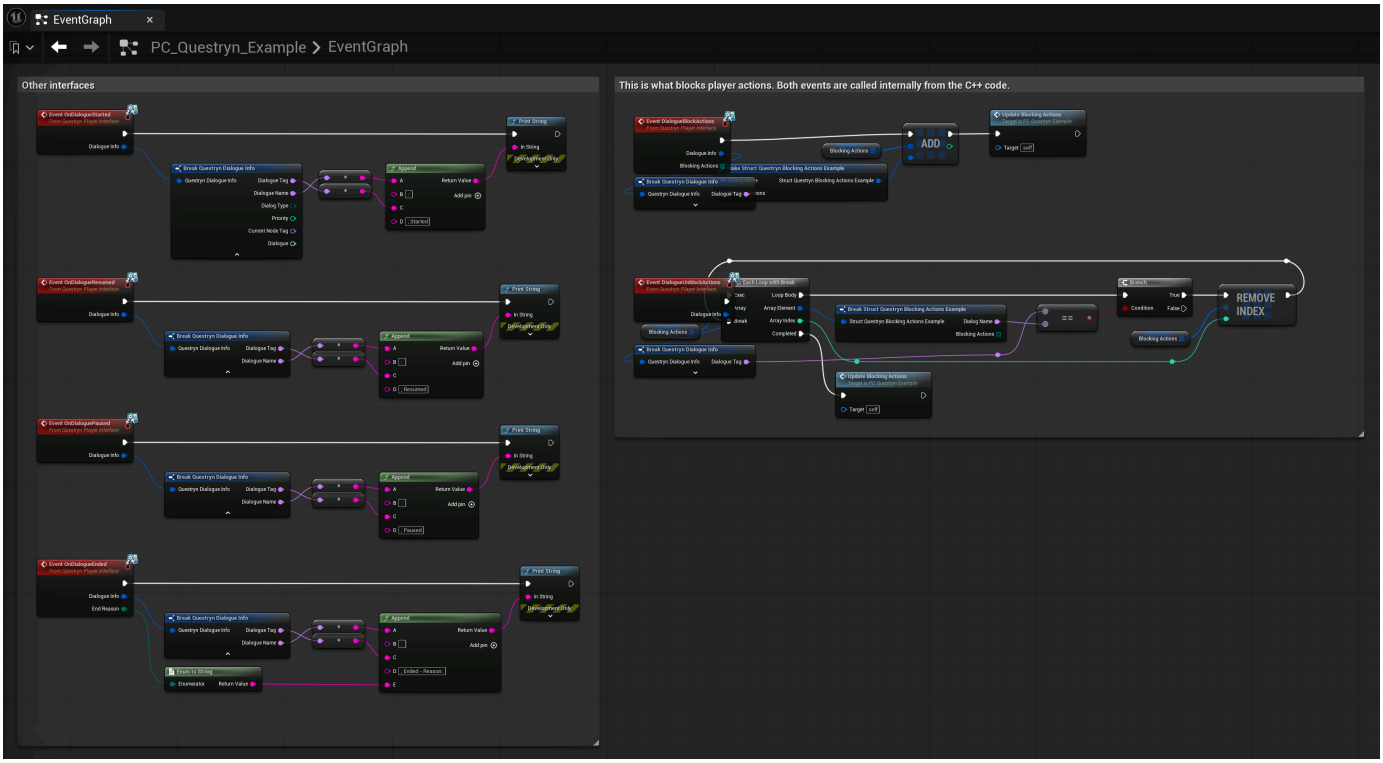
- **Start Dialogue** (Player Controller, Dialogue asset, Node Tag) → *Requested* — Node Tag is optional; leave it empty to start at the dialogue's normal beginning. If set, it searches the graph for a Dialog Node whose own **Node Tag** (4.3) matches and starts there instead; if no node has that tag, it falls back to the normal beginning and logs a warning.
- **Execute Next Dialogue** (Player Controller, Dialogue Tag, Choice, Execution Trigger) → *Trigger Accepted*
- **Force Pause / Force Resume Dialogue** (Player Controller, Dialogue Tag) → *Success*
- **Force Destroy Dialogue** (Player Controller, Dialogue Tag) → *Success*
- **Force Destroy All Dialogues** (World Context) / **Force Destroy All Dialogues For Player** (Player Controller) → *Destroyed Count*

- **Force Destroy All Dialogues Of Type** (Player Controller, Dialogue Type) → *Destroyed Count* — clears one surface for that player: every Notification, every Status Bar, whatever the type. It ignores priority and **Lower Priority Behavior** entirely, which is the difference between this and starting a high-priority dialogue to push the others out — that only reaches the ones set to *Stop*.
- **Get Running Dialogues Save Info** (World Context → array of `FQuestrynSaveData`, one entry per currently running dialogue across every local player: Dialogue asset, Dialogue Type, current Node Tag, and Local Player Index) — call this whenever your own save system writes a save, and persist the returned array in it.
- **Start Dialogues From Save Info** (World Context, array of `FQuestrynSaveData`) — the inverse of the function above: call it after your save system loads, passing back the array it got from **Get Running Dialogues Save Info**. Resolves each entry's Player Controller from its Local Player Index and calls **Start Dialogue** at the saved Node Tag, and returns how many entries it was able to request; an entry whose Local Player Index no longer maps to a valid Player Controller (e.g. a split-screen player who isn't there anymore) is skipped with a warning in the log.

> **Calling without a tag**: any of these functions that takes a Dialogue Tag, when called with an empty/none tag, acts on the highest-priority dialogue currently running **for the Player Controller you passed in** — and if more than one shares that top priority, the most recently created one.
>
> **Return pins**: the synchronous functions return a **Success** pin you can branch on. **Start Dialogue** and **Execute Next Dialogue** cannot — both may still be streaming assets in when they return — so their pins are named *Requested* and *Trigger Accepted* instead: true means Questryn took the request, not that the dialogue is already on screen.
>
> **When something silently does nothing**: every failure and every step of a dialogue's life is logged under the `LogQuestryn` category. Mistakes you'd want to catch (a Dialogue Tag that matches nothing, a Node Tag that doesn't exist, an unset Dialogue reference, an asset with no playable graph) are logged as **Warnings** and show up with no setup. The routine play-by-play (dialogue created, paused, resumed, ended, a Lower Priority Behavior firing, an input refused because the node's Execution Option doesn't accept that trigger) is **Verbose**, so it stays out of your log until you ask for it with the console command `Log LogQuestryn Verbose`. If a Blueprint call seems to do nothing, that command is the first thing to try.
>
> **Save/load is your responsibility**: Questryn deliberately has no save system of its own — the two functions above only get data in and out of memory. Where and how you persist the `FQuestrynSaveData` array (a savegame slot, a cloud save, whatever your project already uses) is entirely up to your own save system.

4.10 Reacting to a Dialogue's Life Cycle

Section 4.9 is your game talking to a dialogue. This is the other direction: the **Questryn Player Interface**, which Questryn calls on your Player Controller as dialogues come, go and change hands. Add it in **Class Settings** → **Interfaces** → **Add** → **Questryn Player Interface**, then implement only the events you care about — every one of them is optional.



Blocking player actions

- **Dialogue Block Actions** (Dialogue Info, Blocking Actions) — a dialogue is on screen and asks for these actions to be blocked. Fires every time a dialogue starts *and* every time one resumes. Questryn never enforces the list itself: what "blocked" means is entirely your Player Controller's decision.
- **Dialogue Unblock Actions** (Dialogue Info) — that dialogue no longer needs its blocks, because it was paused or it ended.

Both carry the same **Dialogue Info** struct as the life-cycle events below, so what you key on is the **Dialogue Tag** inside it: with more than one dialogue running, keep a per-dialogue list and apply the union of it, exactly like `PC_Questryn_Example` does in `UpdateBlockingActions` (5.1). Getting the whole struct rather than just the tag also means you can decide *how* to block from the dialogue's own **Dialog Type** or **Priority**, not only from the list it sent — freeze the camera for a `CutScene` but not for a `Notification`, ignore blocking entirely below a certain priority — without Questryn having to model any of that itself.

Life cycle

- **On Dialogue Started** (Dialogue Info) — the dialogue reached the screen for the first time.
- **On Dialogue Resumed** (Dialogue Info) — it came back after having been paused. It never fires on the first run; that is always *On Dialogue Started*.
- **On Dialogue Paused** (Dialogue Info) — it was pushed aside but is still alive and can come back.
- **On Dialogue Ended** (Dialogue Info, End Reason) — it is over and will not come back.

Three guarantees worth relying on: every dialogue that fires *Started* fires exactly one *Ended* later; a dialogue that never got to run — one stopped by a higher-priority dialogue the moment it was created — fires neither; and a dialogue that is ending does **not** fire *Paused* on the way out, so *Ended* is always the last thing you hear about it.

End Reason tells you why it is over:

End Reason	What happened
Completed	The dialogue reached a node with no path left to take and ended on its own.
Forced Finish	Something called **Finish Dialogue** with *Trigger Events* on, so the rest of the graph ran hidden and the dialogue ended at its last node.
Forced Destroy	The dialogue was destroyed outright — **Force Destroy Dialogue**, or **Finish Dialogue** with *Trigger Events* off.
Stopped By Priority	A higher-priority dialogue started and this one's **Lower Priority Behavior** is *Stop*.
Finished By Priority	A higher-priority dialogue started and this one's **Lower Priority Behavior** is *Trigger Finish*.
Invalid Data	The dialogue could not keep running because its graph data is broken. `LogQuestryn` says what was wrong.

Dialogue Info carries who the event is about:

Field	Use it for
Dialogue Tag	The ID. This is what you branch on.
Dialogue Name	The display label, for logs and debug UI. Not unique.
Dialog Type	The dialogue's category, and the scope its priority competes in.
Priority	Its priority within that type.
Current Node Tag	Which node it is on right now. On *Ended* this is **where it stopped**, which is how you tell "accepted the quest" from "walked away" — give those nodes a **Node Tag** (4.3) and read it here.
Dialogue	The Dialogue asset itself.

> **Why blocking and life cycle are separate**: *Resumed* always blocks and *Paused* always unblocks, so the four life-cycle events could have carried the blocking list themselves. They deliberately don't. Blocking actions are an instruction ("apply this list now"), the life-cycle events are a notification ("this happened") — keeping them apart means a Player Controller that only wants to swap the camera on pause doesn't have to think about blocking at all, and one that only blocks input doesn't have to implement four events to do it.

5. Blueprint Architecture of the Example Project

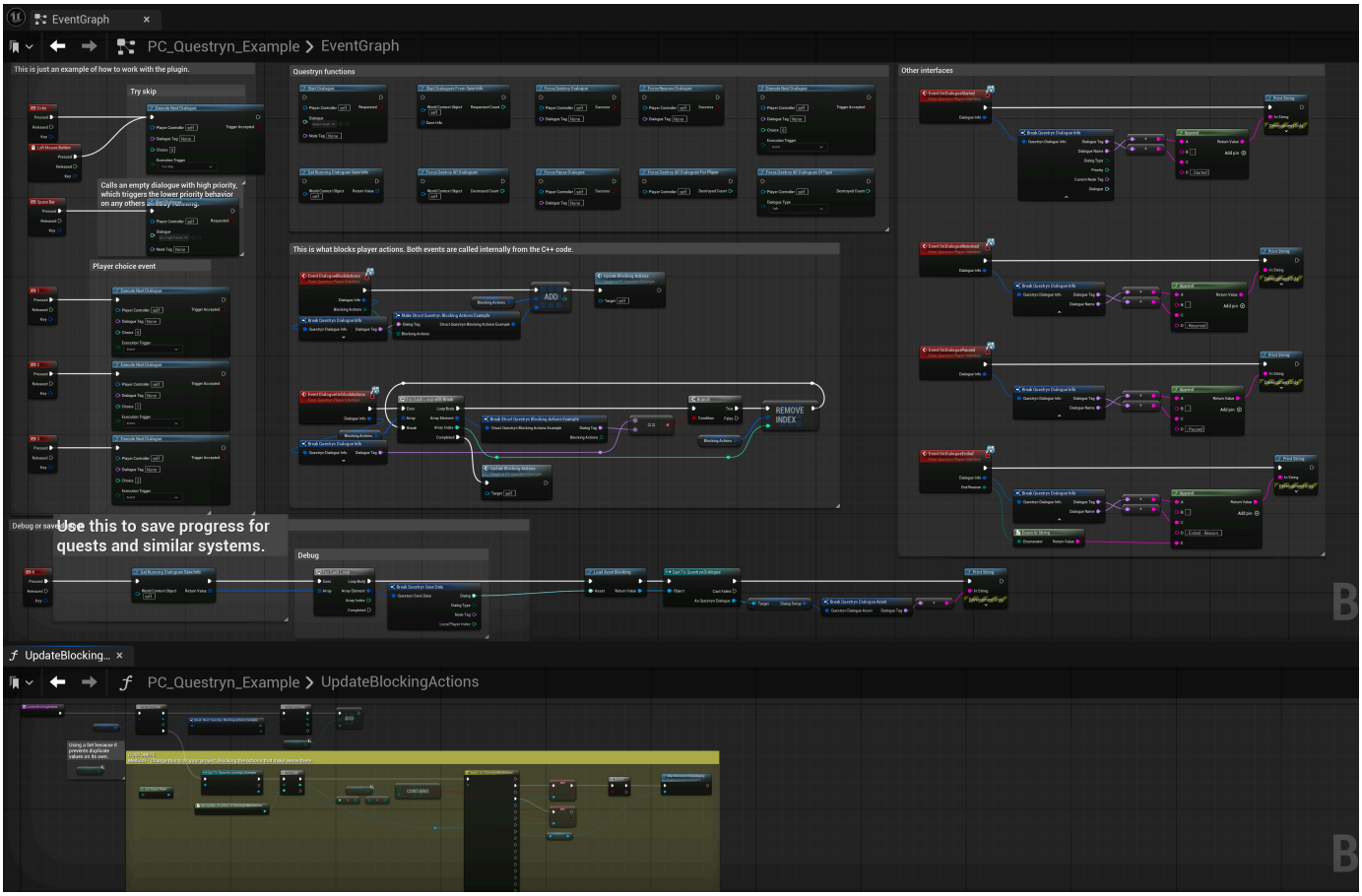
Everything in this section lives in **Content**, not in the plugin's C++ — it's Blueprint, yours to inspect, change, rename or delete (see Section 2). It's what turns a `QuestrynDialogue` asset into an actual on-screen, interactive conversation. Comment-box call-outs below use the legend from Section 3.

5.1 `PC_Questryn_Example` (Player Controller)

`UpdateBlockingActions` — takes the union of `Blocking Actions` currently requested by every running dialogue (`EQuestrynBlockAction`: Attacks, Movement, Jump, Aiming, Special Movements, Climb, Swim, Fly, Equip, Unequip, Drive, Enter Car, Exit Car, Interact, Abilities, Use Item, Reload, Crouch, Camera Look, Open Menu, Custom Action 1–8) into a `Set` (chosen specifically to avoid duplicate entries), then hands the result to the Character (5.2) to actually enforce.

> 🟡 **CHANGE RECOMMENDATION** — *Medium*: adapt this so it blocks the actions that actually matter for your project.

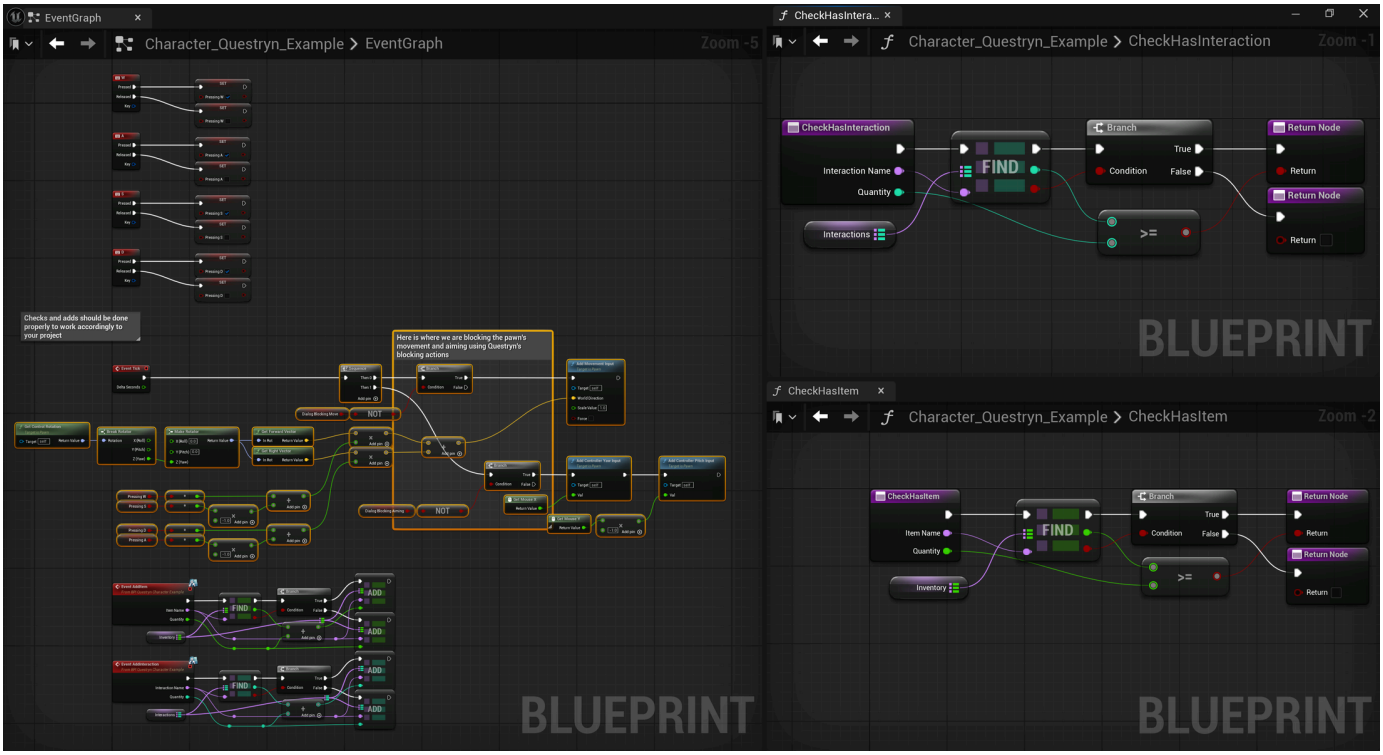
Event Graph — mostly example/demo wiring, meant to be copied and adapted rather than used verbatim. Shown below together with `UpdateBlockingActions` itself:



- Left-click → **Try Skip** on the current dialogue's *Execute Next Dialogue*.
- Space → starts an empty, high-priority dialogue (demonstrates the priority/"Lower Priority Behavior" system pushing other dialogues aside).
- Keys 1/2/3 → **Execute Next Dialogue** with Choice 0/1/2 (answering a path-choice prompt).
- `Event Dialogue Block Actions` / `Event Dialogue Unblock Actions` (from the Questryn Player Interface, 4.10) — maintain a per-dialogue array of requested blocking actions and call `UpdateBlockingActions` whenever it changes. The same interface also offers **On Dialogue Started / Resumed / Paused / Ended** for reacting to a dialogue's life cycle rather than to its blocking list; the example Player Controller implements them as stubs that log, so you can see the order they fire in.
- **K** → debug: dumps the data of every currently running dialogue via `Get Running Dialogues Save Info` to the log.
- A "Questryn functions" region shows every `UQuestrynFunctionLibrary` call available (same list as 4.9), ready to wire up.

5.2 `Character_Questryn_Example` (Pawn)

Event Graph — tracks WASD as booleans and, on Tick, applies movement/looking input built from those booleans — standard third/first-person movement, included only so the blocking demo has something to block. The part relevant to Questryn: movement input and mouse look are each gated behind `NOT DialogBlockingMove` / `NOT DialogBlockingAiming`, two booleans driven by `PC_Questryn_Example`'s blocking-actions handling.



> 🟠 "*Checks and adds should be done properly to work accordingly to your project*" — the movement/aiming block here is a minimal example; a real project usually needs more granular checks.

Functions — small inventory/interaction helpers, exposed through the `BPI_Questryn_Character_Example` interface (5.5) so `FilterPathOptions` / `CheckCondition` (5.3) never need to know the concrete class implementing them, visible on the right of the same screenshot above:

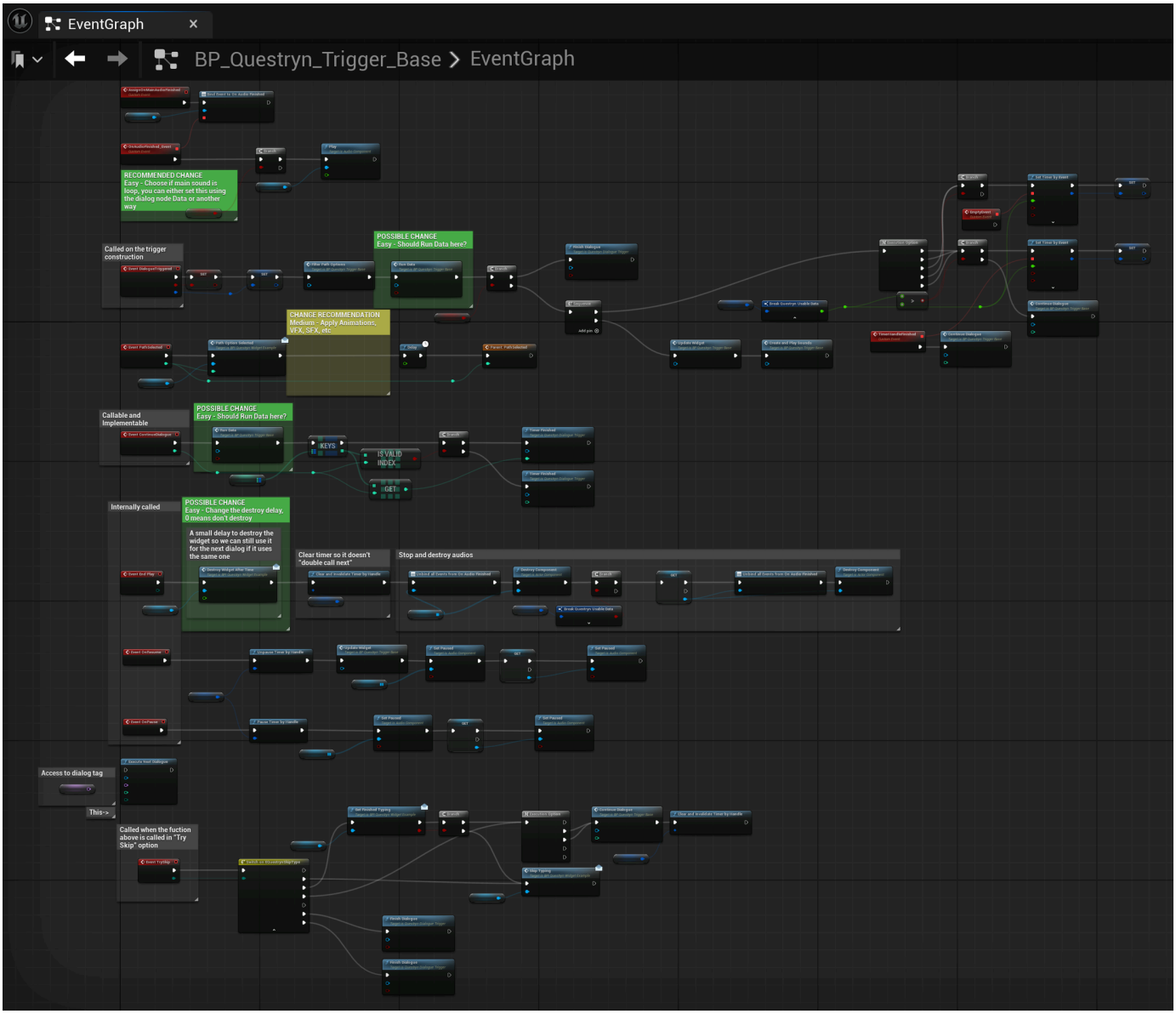
- `Inventory` (Map: item name → quantity) and `Interactions` (Map: interaction name → done/not done) are the two backing variables.
- `CheckHasItem` / `CheckHasInteraction` (implemented as Functions) and `AddItem` / `AddInteraction` (implemented as Events) — straightforward find/add on those maps, wired up as the interface's four functions.

This is the class most likely to be swapped for your own character early on — since `BP_Questryn_Trigger_Base` only ever reaches it through the interface, any actor implementing `BPI_Questryn_Character_Example` can stand in for it without touching the Trigger Blueprint at all.

5.3 `BP_Questryn_Trigger_Base` — the Main Extension Point

This is **the** class to look at first. Every Dialog Node references a Trigger class (4.3); this is the example implementation of `QuestrynDialogueTrigger`, and the one place where dialogue data, conditions, text/texture/sound resolution, widget updates and gameplay side-effects all meet. When you build your own, copy this Blueprint rather than subclassing it — copies are the supported customization path here.

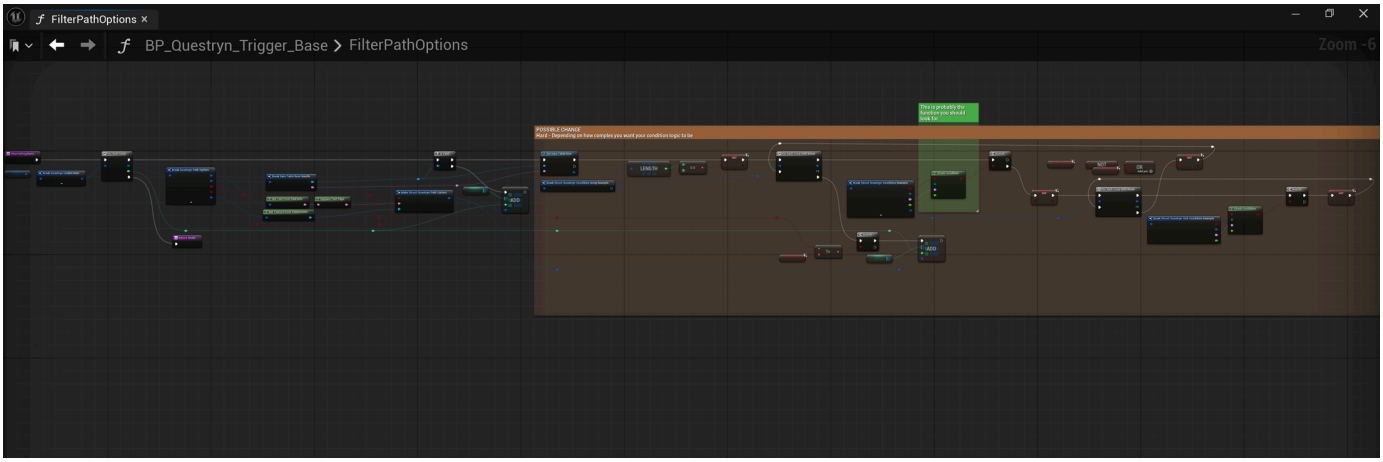
Event Graph — the core lifecycle, plus pause/resume, skip, and tag-based access, all in the same graph:



- `AssignOnMainAudioFinished` binds the main audio component's finish event to `OnAudioFinished_Event`, which — 🟢 *Easy*, **CHANGE RECOMMENDATION** — replays the main sound if `Loop Main Sound` is set (you can drive this from the Dialog Node's data instead).
- `Event DialogTriggered` (called on trigger construction) — calls `FilterPathOptions` (below), then `RunData` (🟢 *Easy*, POSSIBLE CHANGE: "should Run Data run here?"). If the node is Trigger-Only, it finishes the dialogue right away (with Trigger Events). Otherwise it calls `UpdateWidget` + `CreateAndPlaySounds`, and — depending on the node's Execution Option — sets a timer to advance automatically.
- `Event PathSelected` (a protected native event — implementable in Blueprint, but only callable from within the class hierarchy, not from arbitrary external Blueprints) — calls `PathOptionSelected` on the dialogue widget, waits (🟡 *Medium*, CHANGE RECOMMENDATION: "Apply Animations, VFX, SFX, etc." during this delay), then calls the parent's `PathSelected` to continue.
- `Event ContinueDialog` (same protected-native-event pattern) — `RunData` again (🟢 *Easy*, same "should Run Data run here?" note), then finishes the current line by checking the `Choice` is valid and calling `TimerFinished`.
- `Event End Play` — destroys the widget after a short delay (🟢 *Easy*, POSSIBLE CHANGE: "change the destroy delay" — kept so the same widget instance can still be reused by the *next* dialogue if it uses one), clears the timer, and unbinds/destroys the audio components (only destroying the shared main component if this is the last sub-dialogue).
- `Event OnPause` / `Event OnResume` — pause/unpause the internal timer and every audio component (main + secondary); `OnResume` also refreshes the widget.
- `Event TrySkip` — switches on the node's `Skip Type` (`EQuestrynSkipType`: Nothing, Text Only, Text Then Next, Next Directly, Skip to Last Sub Dialog, Stop Dialog, Finish Dialog) and reacts accordingly — typically checking whether typing has finished (`Get Finished Typing`) before deciding whether to skip the typing animation or advance.
- Shows `Execute Next Dialogue` being called by Dialogue Tag as a plain access-pattern example.

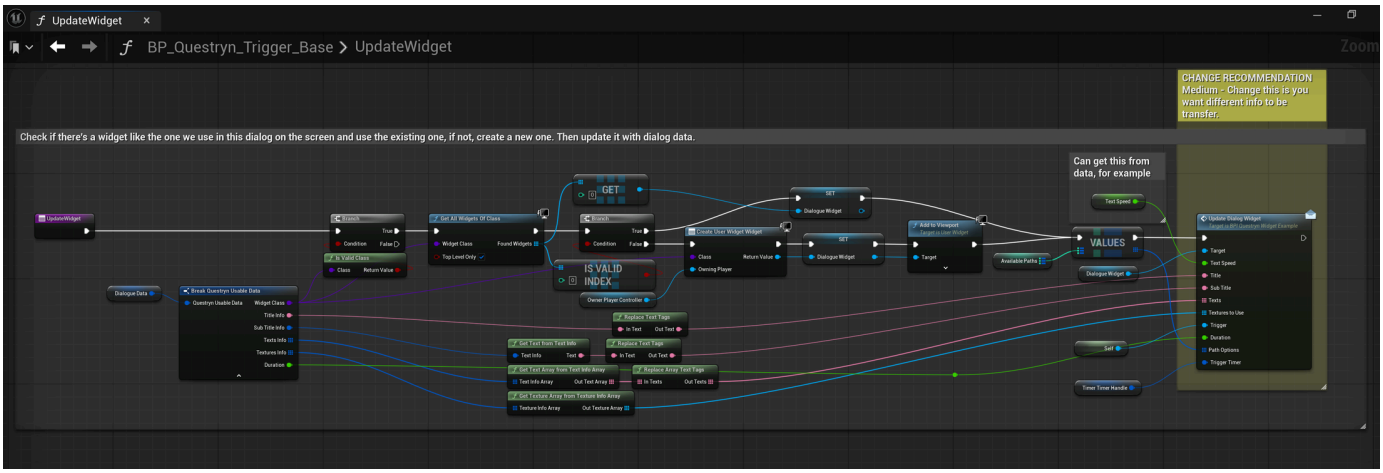
Functions

`FilterPathOptions` — evaluates every path's condition(s) and returns only the paths currently valid, reaching the controlled pawn's condition checks through `BPI_Questryn_Character_Example` (5.5) rather than any hard-coded character class:



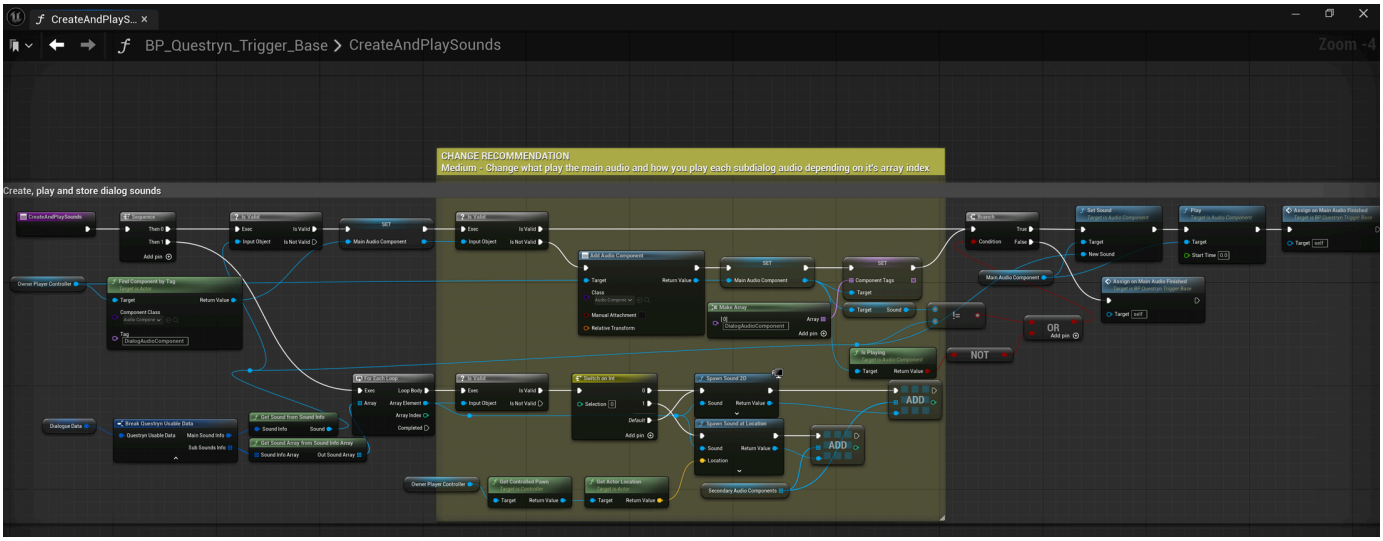
> 🔴 **POSSIBLE CHANGE** (Hard, "depending on how complex you want your condition logic to be") — the actual filter logic: conditions are grouped as **an array of OR groups, each containing an array of AND conditions** — a path is valid if *any* OR group has *all* of its AND conditions true. This is the function to look at if you need more elaborate condition logic than AND-of-ORs.

`UpdateWidget` — finds an existing dialogue widget on screen and reuses it, or creates a new one; gathers the node's title/subtitle/text(s)/texture(s) (resolving `{Tag}` substitutions and DataTable references along the way) and hands it all to the widget through the `BPI_Questryn_Widget_Example` interface:



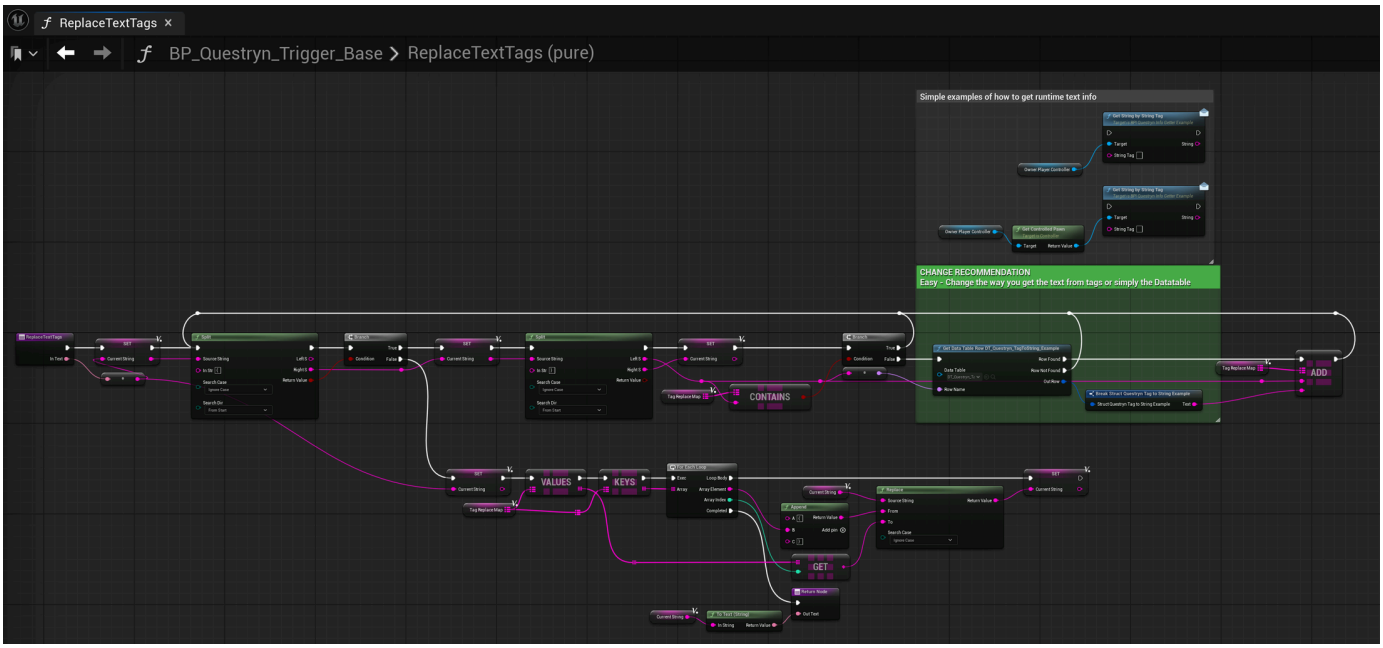
> 🟠 **CHANGE RECOMMENDATION** (Medium) — adapt `UpdateDialogWidget` (the interface call) if you want different information transferred to your own widget.

`CreateAndPlaySounds` — creates and plays the main sound plus each sub-dialogue's sound as dynamically added Audio Components, choosing 2D vs. 3D-at-location by array index in the example:



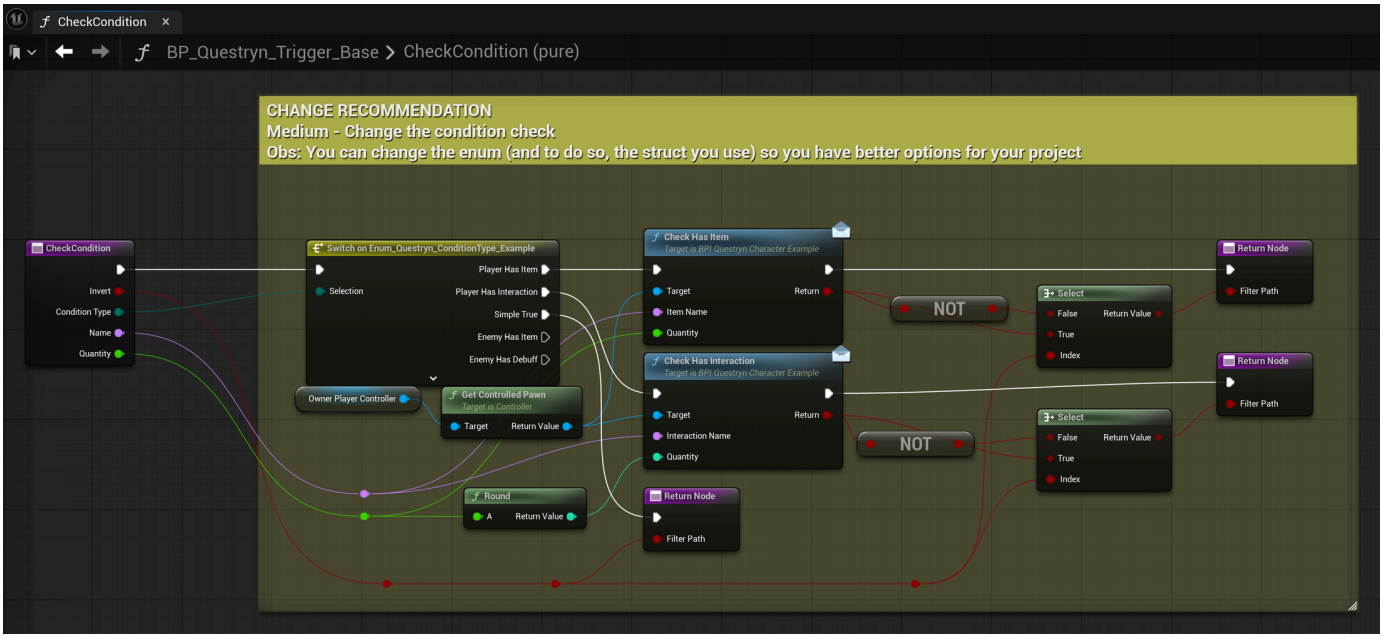
> 🟡 **CHANGE RECOMMENDATION** (Medium) — change what plays the main audio, and how each sub-dialogue's sound is played depending on its array index.

`ReplaceTextTags` (pure) — finds `{Tag}`-style placeholders in a string (distinct from `<RichText>` markup — see the plugin's design notes) and replaces each with a value looked up from `DT_Questryn_TagToString_Example`:



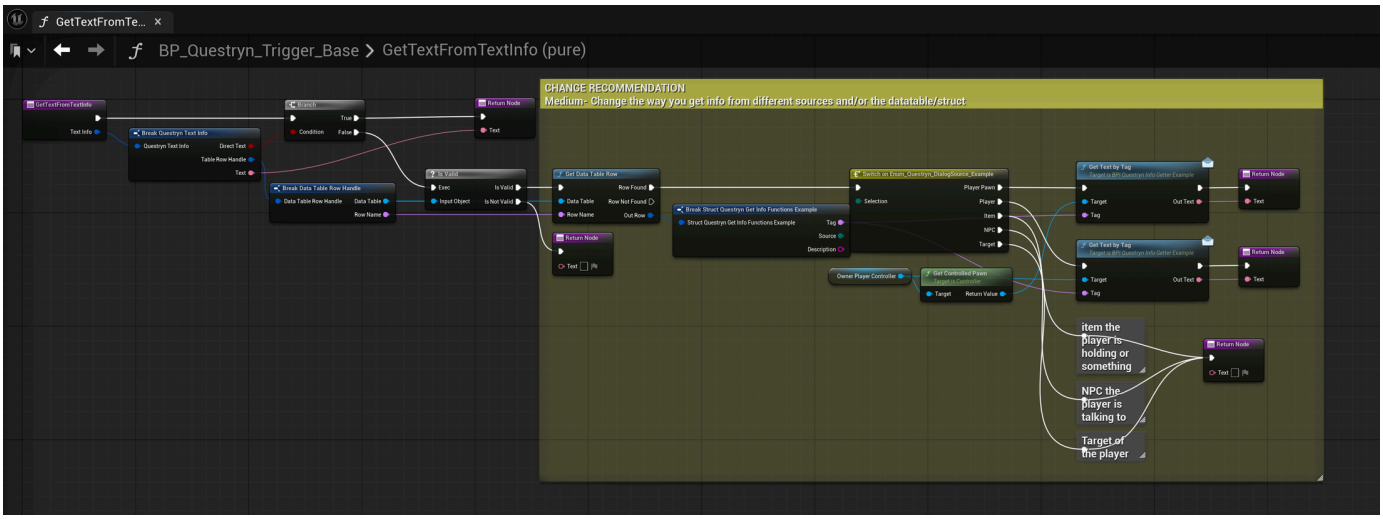
> 🟢 **CHANGE RECOMMENDATION** (Easy) — change where the replacement text comes from (a different table, a function, whatever fits your project) instead of this one DataTable.

`CheckCondition` (pure) — switches on `Enum_Questryn_ConditionType_Example` (**Has Item**, **Has Interaction**, **Simple True**), calls `CheckHasItem` / `CheckHasInteraction` on the controlled pawn through `BPI_Questryn_Character_Example` (5.2, 5.5) rather than casting to a specific class, and applies **Invert** if set:



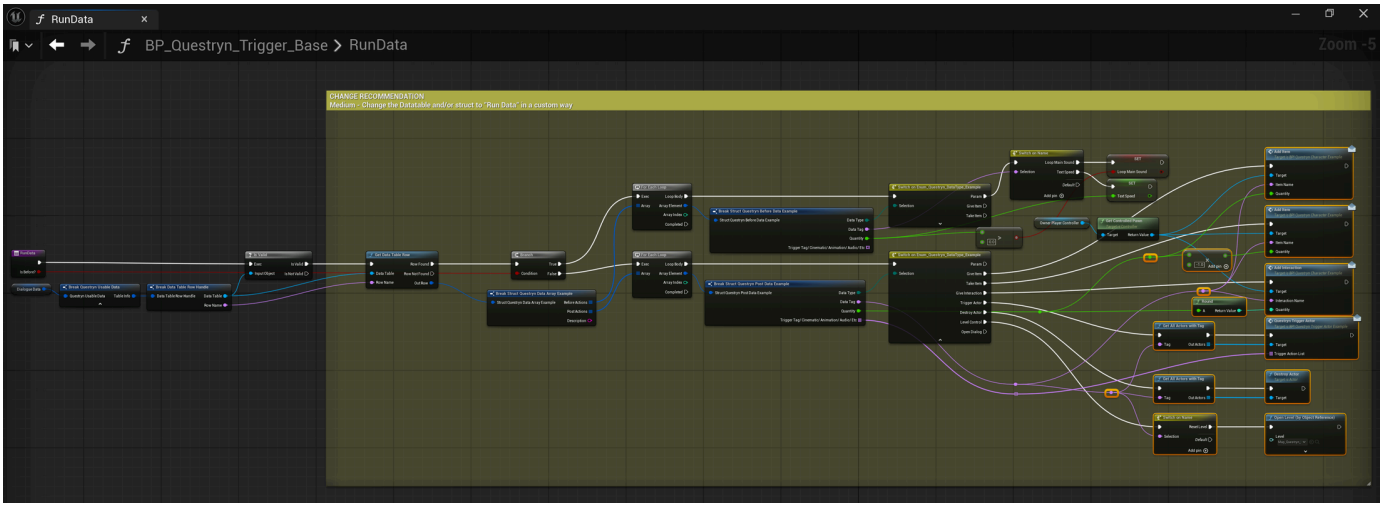
> 🟡 **CHANGE RECOMMENDATION** (Medium) — change the condition check itself; you can also extend the enum (and the struct backing it) for more condition types.

`GetTextFromTextInfo` / `GetTextureFromTextureInfo` / `GetSoundFromSoundInfo` (pure) — the direct-vs-DataTable resolvers behind every "soft" info field (4.4). If the field is set to a direct value, it's returned as-is; otherwise a DataTable row supplies a **Source** + **Tag**, and an `Enum_Questryn_DialogSource_Example` switch routes the lookup to the right object (the player pawn, the player controller, an item reference, the NPC, or a generic target) via `BPI_Questryn_InfoGetter_Example`:



> 🟡 **CHANGE RECOMMENDATION** (Medium, all three functions) — change how you fetch info from different sources and/or the DataTable/struct shape.

`RunData` — the general-purpose gameplay side-effect dispatcher, driven by a DataTable row referenced from the node's **Data: DT/Row** field (its **Give Item** / **Give Interaction** actions call `AddItem` / `AddInteraction` through `BPI_Questryn_Character_Example`, same as `CheckCondition`). Each row holds two ordered lists — **Before Actions** and **Post Actions** — and `RunData`'s `Is Before?` input picks which list runs (once when the node triggers, once when the player finishes/continues past it). Each action has a **Data Type** (`Enum_Questryn_DataType_Example`: **Give Item**, **Take Item**, **Give Interaction**, **Trigger Actor**, **Destroy Actor**, **Level Control**, **Open Dialog**), a **Data Tag** identifying what it applies to, a **Quantity**, and an extra field for a trigger tag/cinematic/animation/audio reference depending on the type:



> 🟠 **CHANGE RECOMMENDATION** (Medium) — adapt the DataTable/struct and the actions themselves to your project; there's a couple of special-cased data tags in the example too (e.g. one that toggles main-sound looping, one that resets the level) — check the graph directly if you want to see exactly how those are wired.

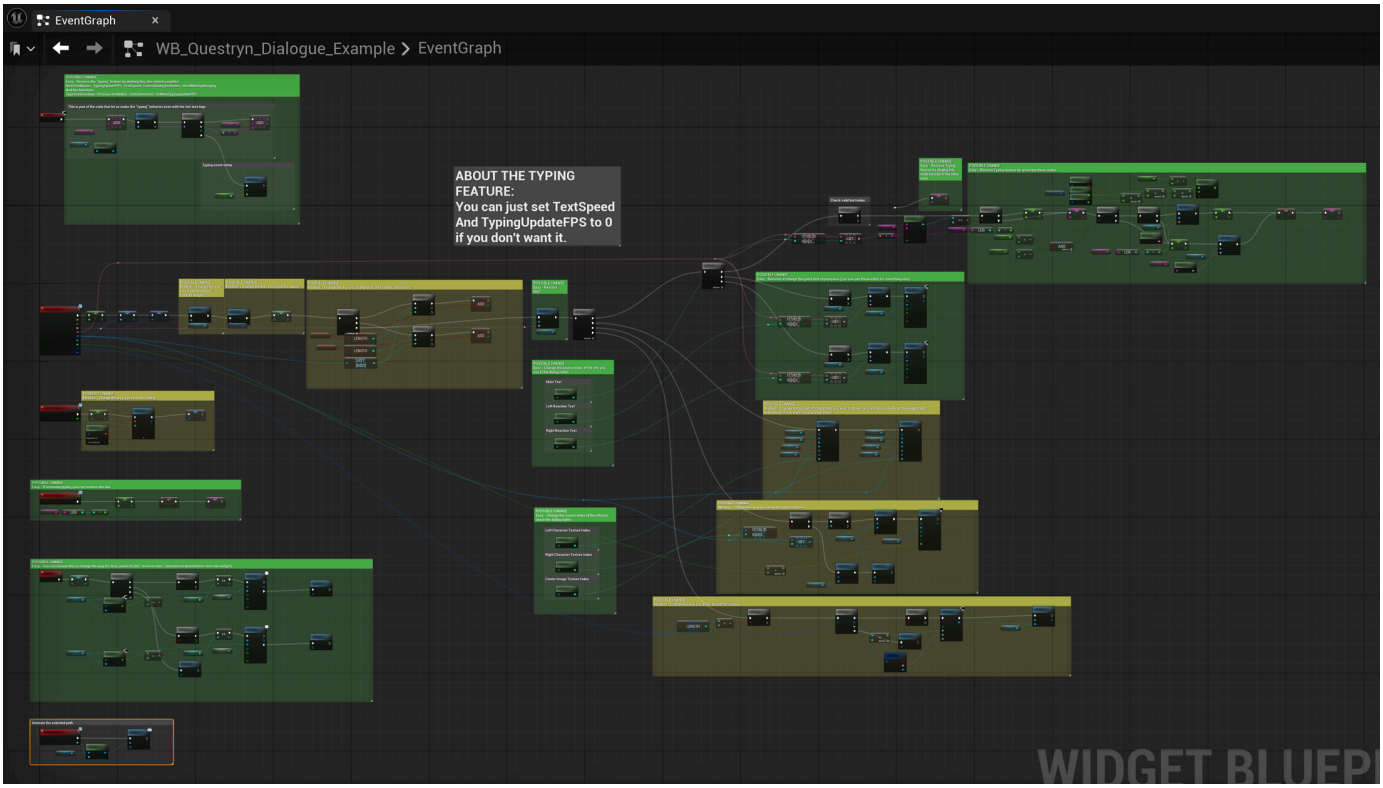
This function alone can drive almost anything — setting a parameter, giving/removing an item, spawning/destroying/otherwise manipulating an actor, running level commands, opening another dialogue, calling a specific function some other way, etc. — the example only covers a handful of simple cases.

> **Driving live values (HUD, Status Bar, Action Bar and friends)**: Questryn has no data binding — it never watches a variable and re-renders when it changes. A dialogue whose Dialogue Type is **Status Bar** works, but it works because *your game drives it*: you call **Execute Next Dialogue** (4.9) whenever the value moves, and the node's path conditions route to the right state (healthy → wounded → critical). To get the number itself on screen you extend `RunData` the same way you would for any other gameplay action — add a Data Type for it, add a function to `BPI_Questryn_InfoGetter_Example` (or your own interface) that returns the value, and use the action's **Data Tag** to say *which* value to fetch (`PlayerHealth` , `BossHealth` , `Stamina` , …). That is the same Data Tag mechanism the example already uses to say which item to give; nothing new to learn, but it is work you have to do — the plugin does not ship a status readout.

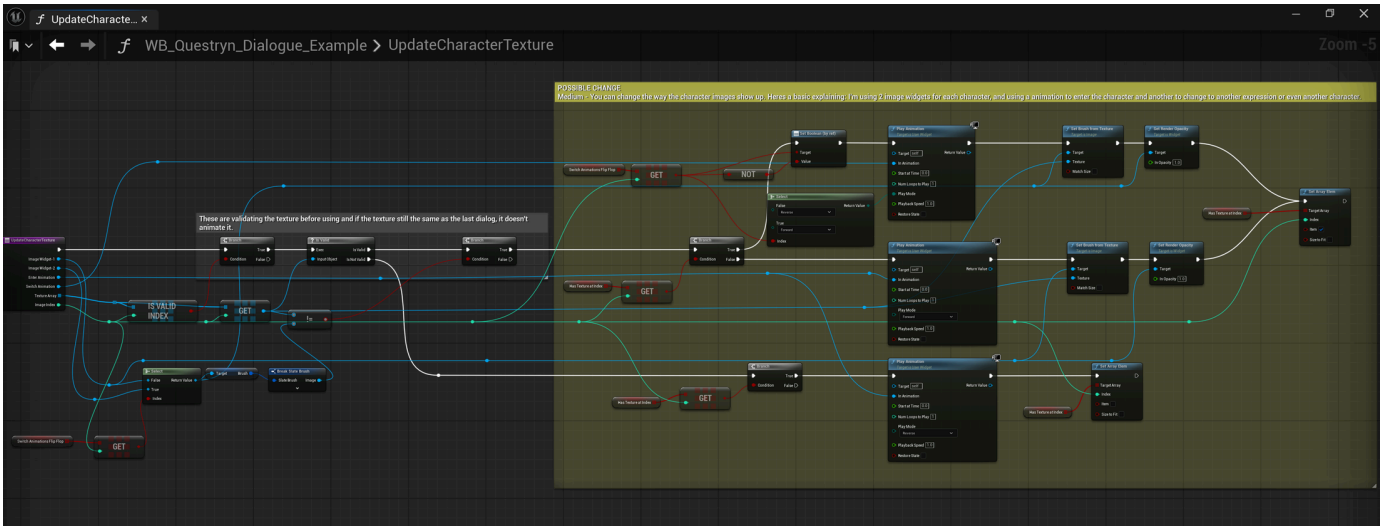
5.4 `WB_Questryn_Dialogue_Example` — the Dialogue Widget

This is the widget doing the actual work of turning a Dialog Node's data into a fully functional, animated dialogue screen: text/array/texture processing, the typing-machine effect, positioning, choice widgets. As of this writing it has **16 `POSSIBLE CHANGE` points in the Event Graph and 1 in `UpdateCharacterTexture`** — all related to presentation/behavior choices, none required.

The typing effect is implemented as its own self-contained block — variables `RichTextNames` , `TypingUpdateFPS` , `TextSpeed` , `CurrentDialogTextIndex` , `HasAMarkupHanging` , and functions `TypeTextOvertime` , `PreviousTextIndex` , `GetCurrentText` , `SetNewTypingUpdateFPS` — and can be removed entirely if you don't want it (🟢 *Easy*, POSSIBLE CHANGE, called out wherever the typing-driven text node is used instead of a plain one). Other notable customization points in the same Event Graph: how the dialogue closes (waits for a "reverse enter" animation before destroying the widget — 🟢 *Easy* to change or remove), how paths/choices are shown, how character portrait images and their enter/expression animations are swapped, the "quick text expression" shortcuts, and — twice — a set of `Make Literal Int` constants (`Main Text` / `Left Reaction Text` / `Right Reaction Text` , and `Left` / `Right` / `Center Character Texture Index`) that simply say *which array index* the widget reads for each purpose, in case you rearrange the source arrays:



`UpdateCharacterTexture` — validates whether a portrait texture actually changed since the last line before playing any animation for it (so it doesn't needlessly replay), then updates the image and plays an "enter" or "switch" animation. The example uses two Image widgets per character (one to animate the incoming image over the outgoing one):



> 🟡 **POSSIBLE CHANGE** (Medium) — you can change how character images show up entirely; this is just one way to do it (2 image widgets, animate-in on enter, animate-change on expression change).

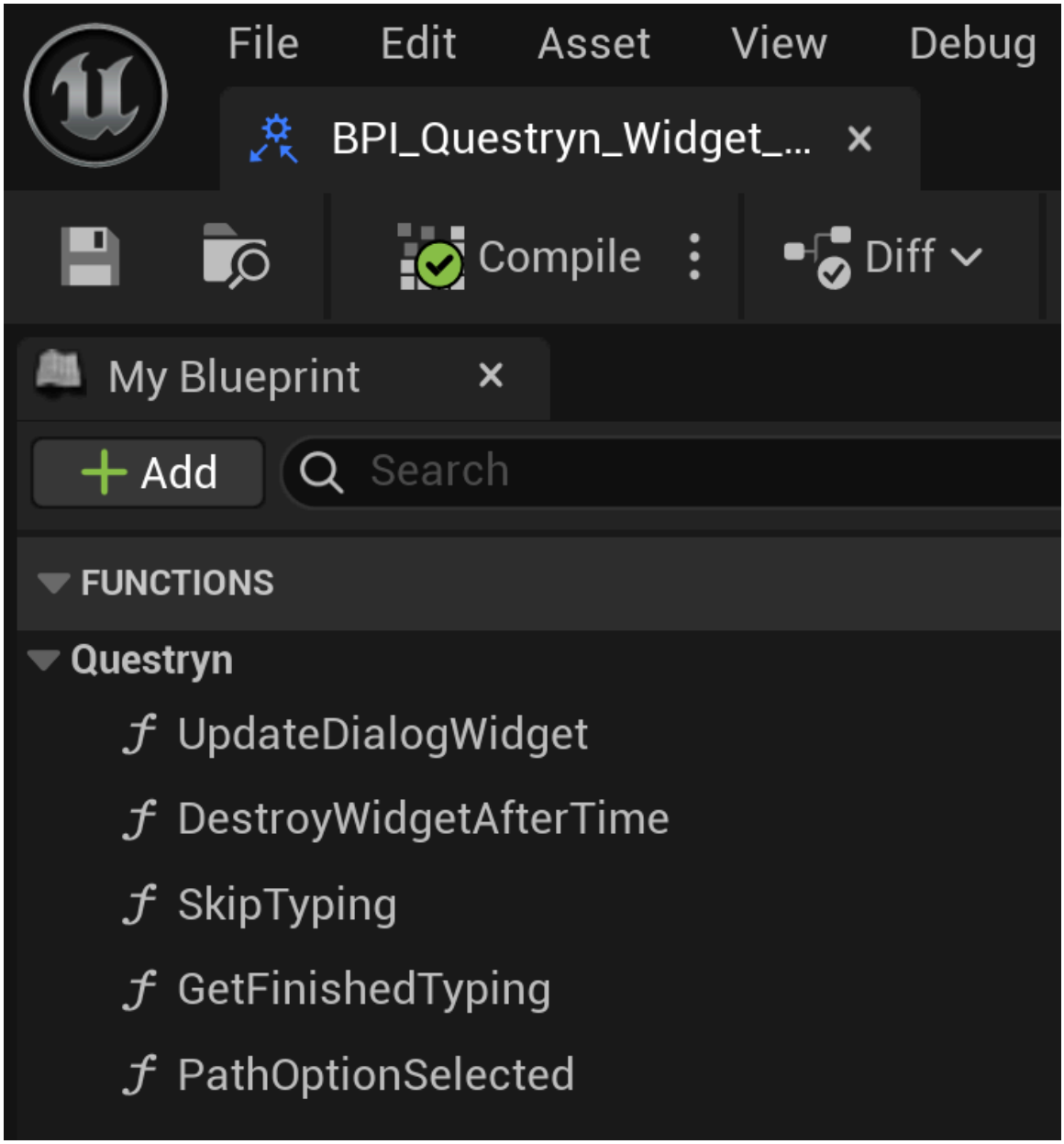
Companion widgets, each with a narrow job:

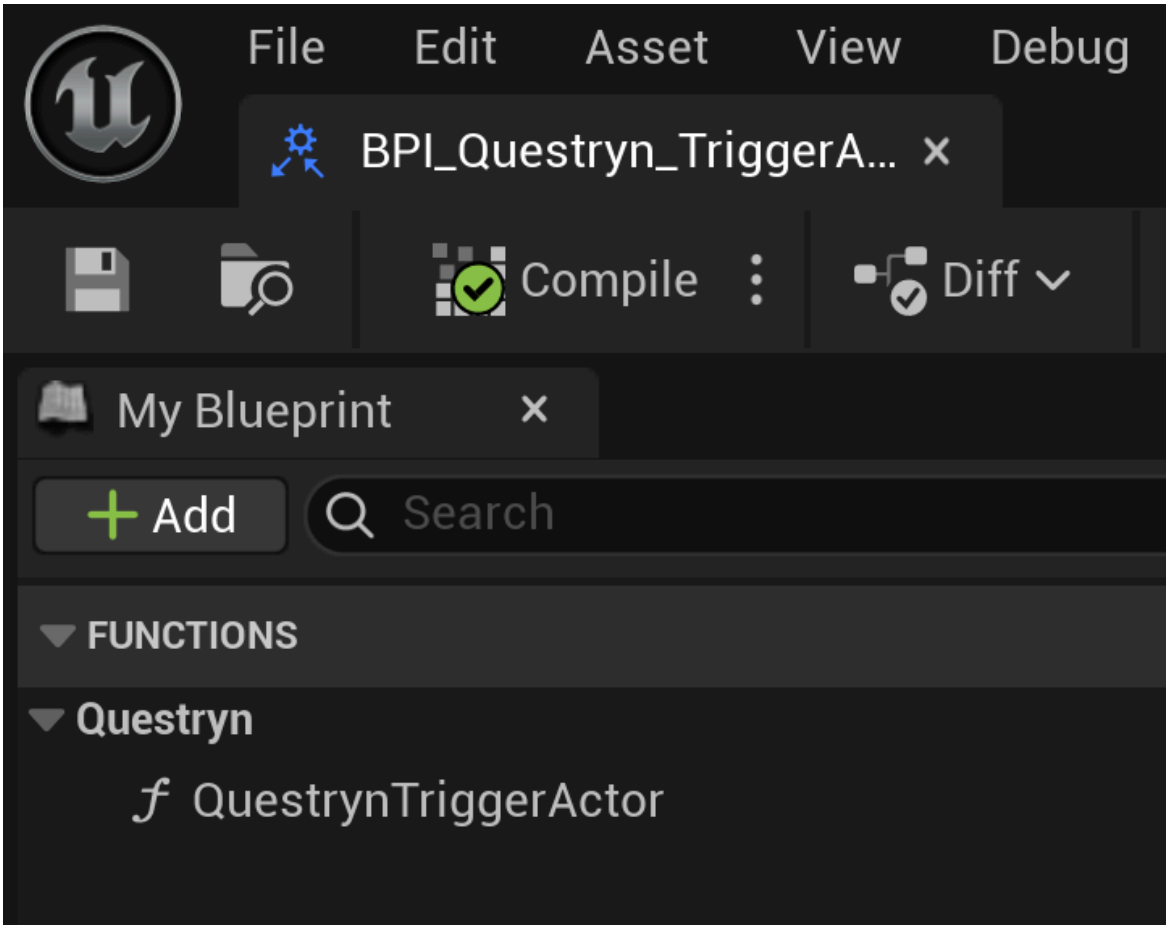
- `WB_Questryn_Notification_Example` — shows a single notification (image + text).
- `WB_Questryn_NotificationWindow_Example` — arranges multiple Notifications on screen.
- `WB_Questryn_PathOption_Example` — a single path/choice entry; stays on screen (parented inside `WB_Questryn_Dialogue_Example`) until the player picks one, then plays a short selection animation (triggered from `WB_Questryn_Dialogue_Example`, see `Event PathSelected` in 5.3).

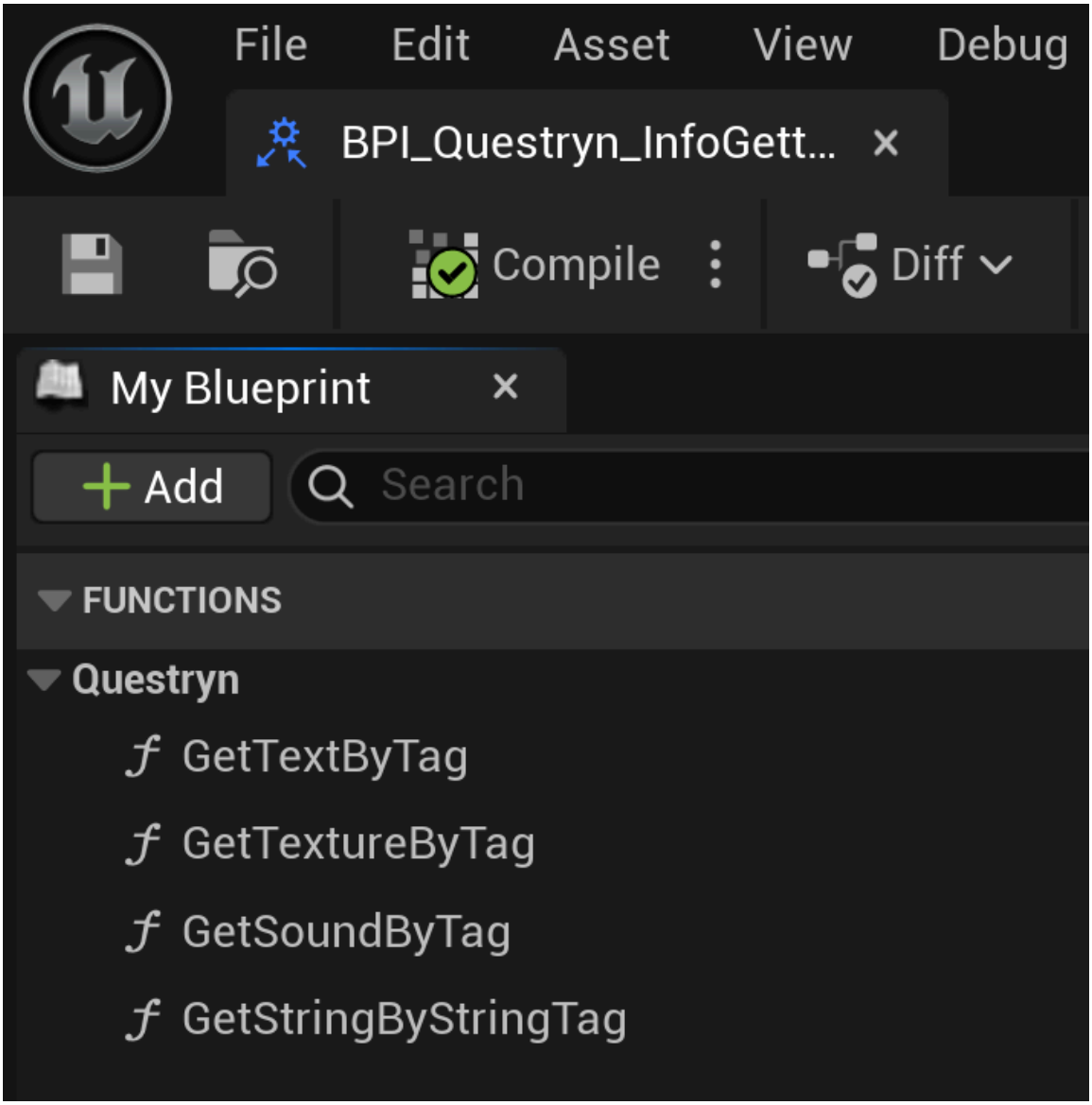
5.5 The Four Interfaces

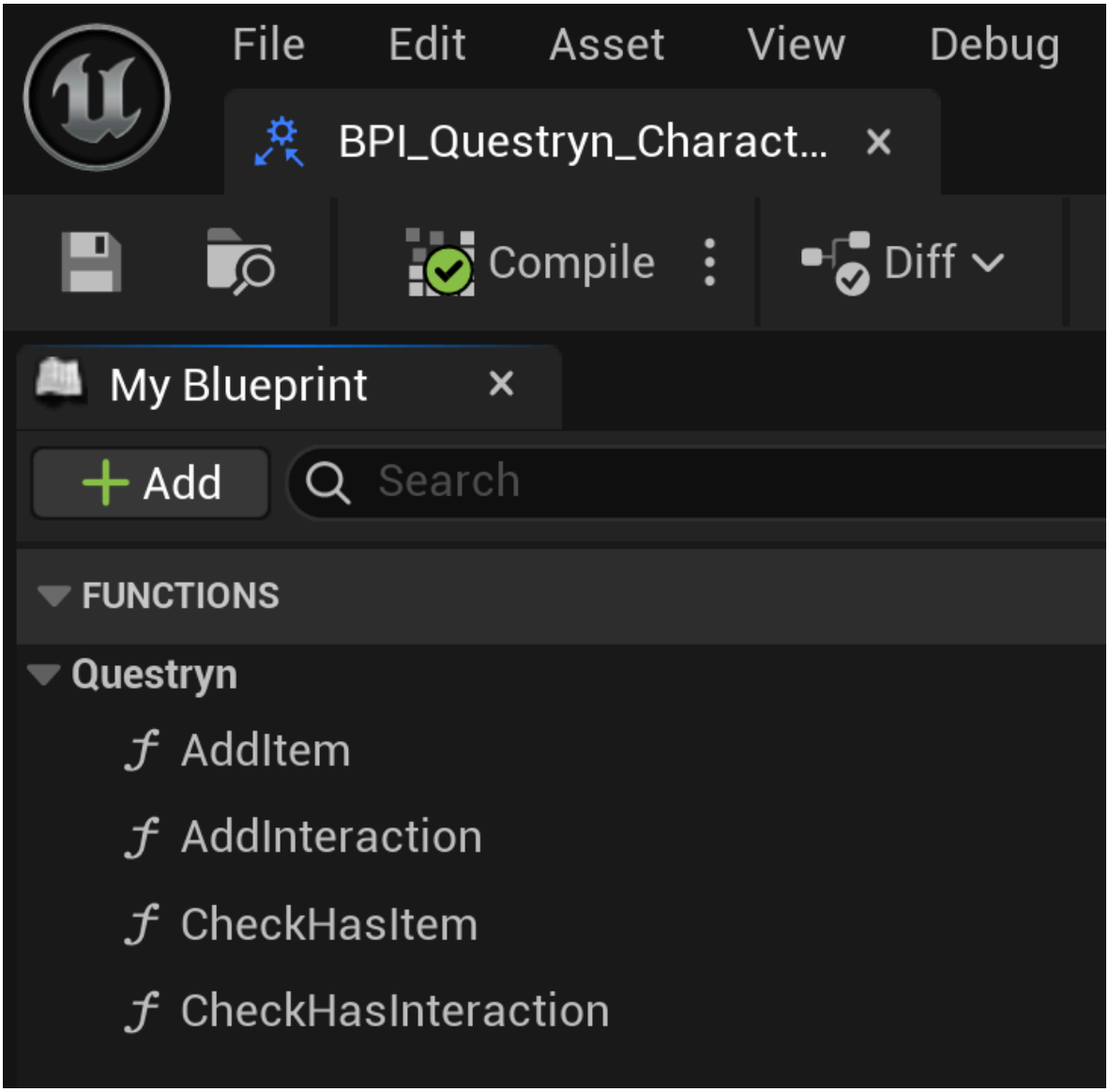
These exist specifically so `BP_Questryn_Trigger_Base` doesn't need to know the concrete class of the widget, the trigger actor, the character, or wherever info comes from — swap in your own classes that implement the same interface and the Trigger keeps working unchanged.

Interface	Functions	Implemented by (example)	Called from
`BPI_Questryn_Widget_Example`	`UpdateDialogWidget`, `DestroyWidgetAfterTime`, `SkipTyping`, `GetFinishedTyping`, `PathOptionSelected`	`WB_Questryn_Dialogue_Example`	`BP_Questryn_Trigger_Base` (`UpdateWidget`, `TrySkip`, `Event End Play`, `Event PathSelected`)
`BPI_Questryn_TriggerActor_Example`	`QuestrynTriggerActor`	Any actor you want `RunData`'s **Trigger Actor** / **Destroy Actor** action to reach	`BP_Questryn_Trigger_Base` (`RunData`)
`BPI_Questryn_InfoGetter_Example`	`GetTextByTagAndIndex`, `GetTextureByTagAndIndex`, `GetSoundByTagAndIndex`	Player Pawn / Player Controller / item / NPC / target, as needed	`BP_Questryn_Trigger_Base` (`GetTextFromTextInfo` and its Texture/Sound siblings)
`BPI_Questryn_Character_Example`	`CheckHasItem`, `CheckHasInteraction`, `AddItem`, `AddInteraction`	`Character_Questryn_Example` (5.2)	`BP_Questryn_Trigger_Base` (`FilterPathOptions`, `CheckCondition`, `RunData`)


File
Edit
Asset
View
Debug
BPI_Questryn_Widget_...
Compile
Diff
My Blueprint
Add
Search
FUNCTIONS
Questryn
UpdateDialogWidget
DestroyWidgetAfterTime
SkipTyping
GetFinishedTyping
PathOptionSelected

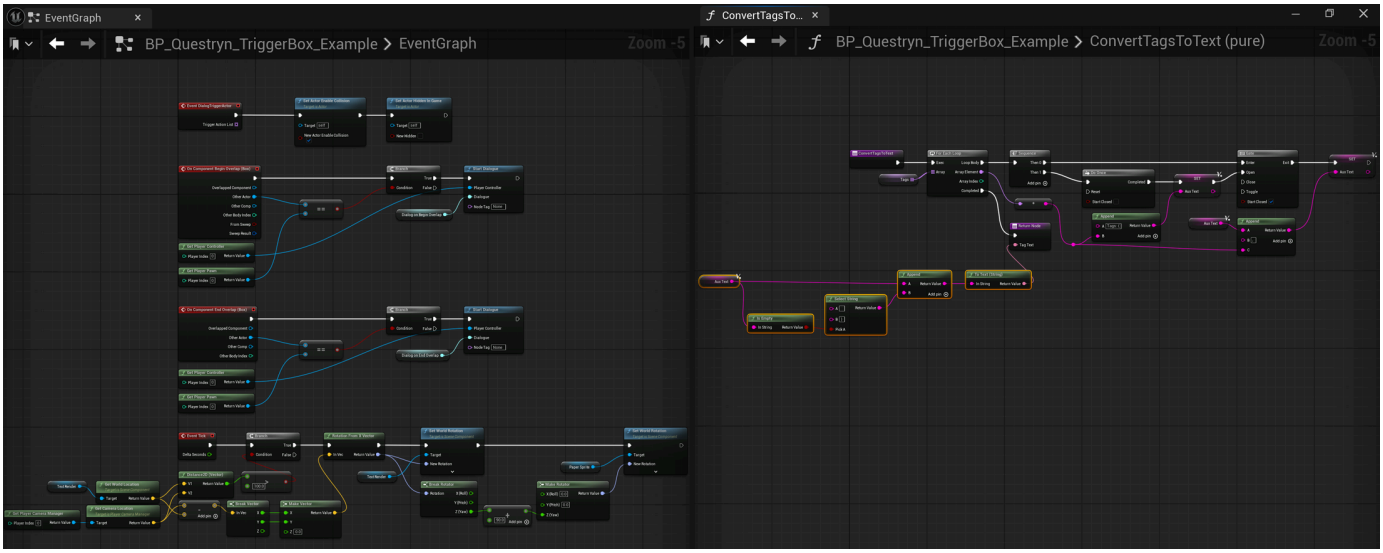

File
Edit
Asset
View
Debug
BPI_Questryn_TriggerA...
Compile
Diff
My Blueprint
Add
Search
FUNCTIONS
Questryn
QuestrynTriggerActor


File
Edit
Asset
View
Debug
BPI_Questryn_InfoGett...
Compile
Diff
My Blueprint
Add
Search
FUNCTIONS
Questryn
GetTextByTag
GetTextureByTag
GetSoundByTag
GetStringByStringTag



5.6 `BP_Questryn_TriggerBox_Example` — Putting It All Together

A concrete, minimal example of triggering a dialogue from the world: an overlap volume that starts the dialogue on `Begin Overlap` and ends it on `End Overlap` (each guarded by a check that the overlapping actor is actually the player pawn), plus an unrelated bit of polish — a world-space prompt text that continuously faces the player camera (billboard/look-at, recalculated on Tick) — and its own small pure helper, `ConvertTagsToText`, which turns an array of gameplay tags into a single display string:



This is a good template to copy when building your own dialogue-starting actors (NPCs, interactables, cutscene triggers, ...).

6. Where to Start

If you only read one section besides this one, make it **5.3 (`BP_Questryn_Trigger_Base`)** — it's where dialogue data, conditions, and every other system in this guide actually connect. A practical order:

1. Follow Section 2 and get your own copies of the classes you need in place.
2. Implement `BPI_Questryn_Character_Example` (5.5) on your own character/condition system, so `FilterPathOptions` and `CheckCondition` reach it.
3. Point `GetTextFromTextInfo` / `Texture` / `Sound` and the `BPI_Questryn_InfoGetter_Example` implementations at your own data sources.
4. Adapt `RunData` to the gameplay actions your project actually needs.
5. Only then, if you want a different look and feel, work through `WB_Questryn_Dialogue_Example`'s presentation-level `POSSIBLE CHANGE` points.

Everything else in Sections 4–5 keeps working as shipped in the meantime.

7. LLM-Assisted Dialogue Authoring

Questryn ships a second file alongside this one — `Questryn_LLM_Authoring_Prompt.md` — that you're not meant to read yourself. Instead, copy the whole thing into a chat with an AI assistant (ChatGPT, Claude, etc.). You can paste it on its own: the assistant's first reply explains what kind of dialogue you can build with Questryn, what it needs from you, and what's optional — then asks one question at a time, in plain language, in whatever language you're writing in. Paste your story or script along with it if you already have one; you don't have to, and you can also ask the assistant to help you write the story itself. If you've already set up your reusable tables, it will show you a summary of everything you have available — your conditions, node presets, and ready-made texts, sounds and images — before it asks anything about the story, so you're choosing from a list instead of trying to remember what's in your project. You never need to open the file to know what's going on. Behind the scenes, it teaches the AI Questryn's data model and how to turn your answers into a dialogue graph: a human-readable plan for you to build by hand, or a ready-to-run Python generation script if the AI has Editor Python execution access to your project.

7.1 Getting the prompt file

Both documents — plus the community Discord — are reachable from the Editor without hunting through folders — **Help menu** → **Questryn**:

- **Questryn User Guide** — opens this document (PDF).
- **Copy Questryn LLM Prompt to Clipboard** — copies the whole file's text, ready to paste straight into an AI chat. There's no separate "open" option — you're not meant to read the file yourself, just hand it to an AI, so copying straight to clipboard is the only path.
- **Join the JN Softworks Discord** — opens the support server (see Section 8).

7.2 The Setup Table and Profile Table (optional, but recommended)

Two kinds of DataTable make the AI's job — and yours — easier, by letting you describe your reusable content once, ahead of time, instead of answering questions about it table-by-table in chat:

- **Setup Table** — one row per reusable DataTable you want the AI to draw on: conditions that gate a branch, sound/text/image lookups, generic gameplay data, or a Profile Table (below). Each row just says which category the table belongs to, a reference to the table itself, an optional description of what it's for, and — if you only want specific rows available — their names. Create one via **Content Browser** → **Add** → **Miscellaneous** → **Data Table**, Row Type `QuestrynLLMSetupEntry`. A filled-in example ships at `/Questryn/Data/DT_Questryn_LLMSourceTable_Example`.
- **Profile Table** — one row per reusable "node behavior" preset: which pacing/skip settings, Trigger class, and Widget class a node should use (the same fields described in 4.3), bundled under a name like `Default` or `Notification` so you can just say "use the Notification profile" instead of setting all four fields by hand on every node. Row struct `FQuestrynNodeProfile`, example table `/Questryn/Data/DT_Questryn_NodeProfile_Example`. This is purely an authoring-time convenience for the AI's generation script — it's not something the graph editor itself has a "Profile" picker for, so you can also ignore it entirely and set each node's fields by hand as usual.

Hand the AI your Setup Table directly if it has Editor Python access to your project. Otherwise, run a small Python script bundled inside `Questryn_LLM_Authoring_Prompt.md` once — Unreal Editor → **Tools** → **Execute Python Script...** — which writes a JSON file into your project's `Saved` folder; open that file and paste its contents into the chat instead. Either way, you fill in the setup once, in the DataTable editor, rather than one table at a time in chat.

Note that the **Trigger** field on both Dialog Nodes and Profile rows only ever offers actual Trigger classes (e.g. your copy of `BP_Questryn_Trigger_Base`) — the plugin's own `QuestrynDialogueTrigger` base class is abstract and deliberately never shows up as a selectable option.

8. Sharing Your Game (Optional)

Neither of these affects your license in any way (see `LICENSE.txt`, Section 4) — they're just two ways to let us know Questryn is part of your game, if you'd like to:

- **Drop the "Made with Questryn" badge into your credits.** A ready-to-use, transparent PNG ships at `Docs/Badge/QuestrynBadge.png` (also copied into `Badge/` by the **Add Starter Content** button, Section 2). It's designed for dark backgrounds.

- **Post in #showcase on the JN Softworks Discord** (Help menu → Questryn → Join the JN Softworks Discord, or the link in `LICENSE.txt`) when your game ships. We regularly feature community projects there and on JN Softworks' own channels.