Quest System

Quest System

Version 1.1.2 - updated 31/07/2025

Overview

Solaris Quest System is my attempt to create an intuitive and easy way to implement quests for Solaris. It was somewhat inspired by how my beloved game Neverwinter Nights 2 deals with quests and journal systems.

All necessary assets can be found in Content/QuestSystem and Source/Solaris/QuestSystem. It is currently located on the quest-system branch on our github project. The system is designed in a way that lets us have multiple quests at once.

Some documents that might be useful to look at:

- **Solaris Dialogue System documentation (link broken [*])**
- **Main quest designs**

Breaking down the system

QuestData

QuestData is a DataAsset that stores the information about the quest - QuestID (used to add/update the quest), QuestName and QuestEntries. A **QuestEntry** is basically information about the phases of a quest. Each entry consists of:

- **Objective -** info on what the player needs to do, with the aim to display somewhere in the UI - i.e. *Talk to Panam.*
- **Description** - intended to use as a journal entry text, so definitely longer and more in-detail in terms of story (i.e. *The Luskan forces attacked the Hollows in reprisal for your attack on the Sea Ghost. They made mistakes in the attack that allowed Kana to find their base of operations in the Dock District. Potentially the Hollows has been infiltrated, so only you can marshall a counter-attack against them).* Not sure if we will actually want to use it, but I added it just in case.
- **QuestEntryID** - used only for clarity and has no real impact on how the system works. I added it because I figured it could make some people's lives easier - if it doesn't for you, you can just ignore this variable;)

JournalEntry

Due to the fact that QuestData is a DataAsset, it couldn't really be used to represent the quests in-game in a way I wanted. To fix it, I came up with a solution which I called a **JournalEntry.** Unlike QuestData, a JournalEntry is a class that inherits from UObject, so you can create instances of it and do all sort of magic you want. It has some parameters similar

to QuestData, but is used as a representation of a current state of the quest instead of its whole data. It consists of:

- QuestName (fstring) - derived from QuestData on constructing.
- QuestID (int32) - derived from QuestData on constructing.
- CurrentEntryID (int32) - set to 0 on constructing; it is changed on every quest update and represents its current state.
- CurrentObjective (fstring) - set to the objective of index 0 of the “parent” QuestData on constructing; it is changed on every quest update. Used to display the player’s objective on screen.
- CurrentDescription (fstring) - set to the description of index 0 of the “parent” QuestData on constructing; it is changed on every quest update.
- Finished (bool) - set to 0 by default; determines whether a quest is considered completed.

A JournalEntry has a few functions that may be worth looking at. They are declared in C++, but are all marked BlueprintCallable. To get current parameters of a quest, you should use GetCurrentDescription(), GetCurrentObjective(), GetQuestName(), GetQuestID() and GetCurrentEntryID(). Other functions are used for changing the parameters - they are SetQuestNameAndID(FString **NewQuestName**, int32 **NewQuestID**), which is used on adding the quest to the journal; **SetQuestParameters**(int32 **NewCurrentEntryID**, FString **NewCurrentObjective**, FString **NewCurrentDescription**) is used on adding and updating the quest; finally, **MarkFinished**() changes the Finished variable to *true*.

QuestDatabase

A **QuestDatabase** is probably the most important asset when it comes to adding and updating quests. It is basically an array of QuestData in order (the order is very important for the system to work - see more in **Creating a quest step by step**. There is intended to be one DataAsset of type QuestDatabase - you can find it under Content/QuestSystem/DA_QuestDatabase. Remember to add every quest you create to this asset and give it a matching index.

A QuestDatabase has 3 functions (traditionally, all BlueprintCallable). **PassDescription**(int32 **QuestID**, int32 **EntryID**) and **PassObjective**(int32 **QuestID**, int32 **EntryID**) return the description/objective of a quest of given QuestID’s entry of index EntryID. There is also a function **GetQuestOfID**(int32 **ID**) which returns a pointer to a quest of given ID. *(This paragraph feels a little spaghettified to me, but I hope you get what I mean;))*

BP_JournalComponent

The brain of this system is **BP_JournalComponent.** It is a blueprint-based Actor Component attached to BP_MainCharacter. It stores the data of currently active quests and deals with adding, updating, finishing and printing (development only screen messages) the quests. It also has a reference to the QuestDatabase from which it takes the quest data to use in JournalEntries. Because it is entirely created in blueprints, naturally all functions on this component are blueprint callable. The functions that you can currently use are:

- **AddQuest** (int **QuestID**) - used for adding a quest of given ID to JournalEntries array. The quest always starts with a Quest Entry of ID 0 - keep that in mind when designing the quests.
- **UpdateQuest** (int **QuestID**, int **EntryID**) - searches for an instance of a quest of given ID in JournalEntries array (so a JournalEntry with a matching QuestID) and updates its description, objective and entry index so that it matches the given EntryID.
- **PrintAllQuests** - development only function that goes through the JournalEntry array and prints the quest name and current objective of each active quest to screen. I'll probably get rid of it once we create a UI for displaying quests.
- **FinishQuest** (int **QuestID**, int **EntryID**) - it works in a similar way to UpdateQuest. The only difference is that it changes the Finished variable of the JournalEntry to true. The given EntryID should reference an entry with a description marking the end of a quest - the reason I did the EntryID thing here is because we might want to have multiple endings to some quests one day. While creating an ending entry, you can leave the objective empty, as it won't be displayed anywhere.

UpdateQuest and FinishQuest both return a boolean variable that determines whether the action was successful - used primarily in **adding, updating and finishing quests in-game**.

There will come a day when I (or someone more competent than me) will create a UI to display the quests and make them actually accessible for the player, but for now we'll need to settle for debug messages.

Creating a quest step by step

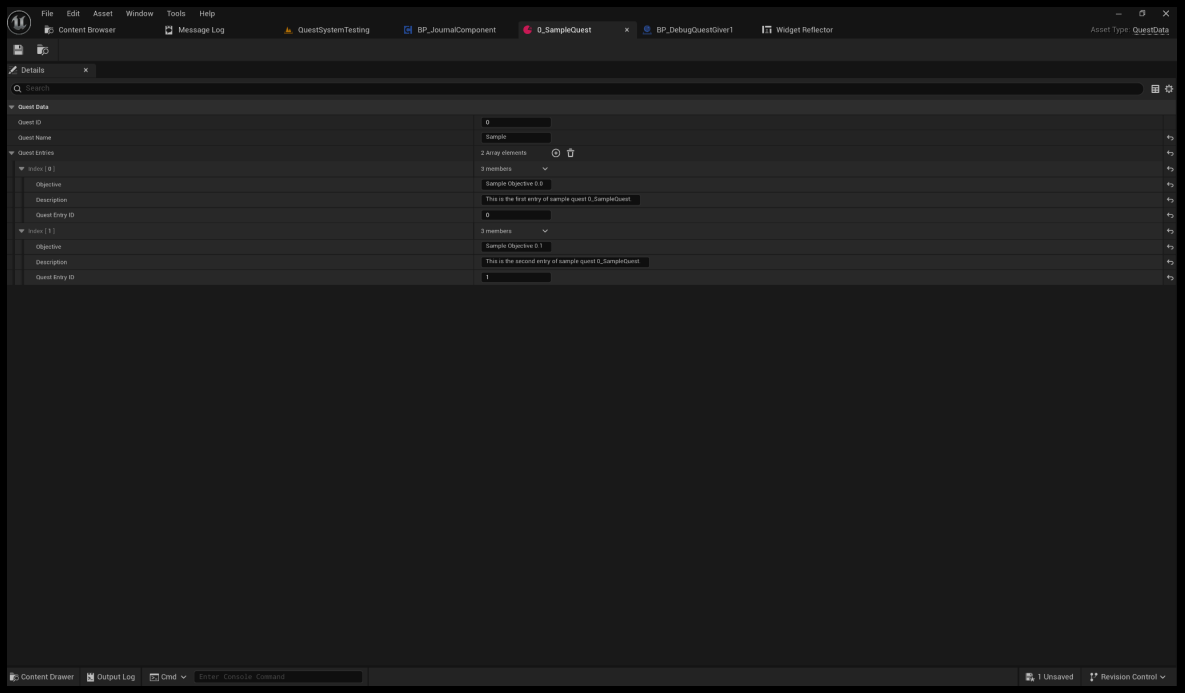
To create a quest, you first need to create a DataAsset of type **QuestData**. For the sake of tidiness, put them in Content/QuestSystem/QuestData and keep the naming convention of ID_QuestName (i.e.: *0_SampleQuest*, *12_NeverFadeAway*). Don't use spaces. The ID should be the next available number and we start from 0.

Once you've created the asset, open it and you'll see a bunch of variables that you will eventually need to fill. The most important one is QuestID - it must be unique for each quest, or else unexpected errors may occur (I mean, expected, but still annoying). Make the QuestID match the ID in the asset name Unlike the QuestEntryID, this is necessary for the system to work.

Next, you have the QuestName variable - keep it similar to the name you have given to the asset, but use spaces this time.

The final thing is the **QuestEntries** array. Fill the objective and description to your liking. If filling the QuestEntryID in a way that it matches the QuestEntry's ID in the array will make your life easier, do it - but don't worry if you don't do that, the system will still work just fine. Create the objectives you want to have in your quest - again, we're starting from 0[1].

[1] Please note that if you suddenly have an idea to add a new entry between other entries, you don't have to completely reorganise everything (doing that could be really painful if you already started implementing the quest and adding/updating entries in-game). Just add another entry with another index and use that numeration to update the quest accordingly. I'll make an excel spreadsheet for the quests when we actually get there to keep it clean and organised, especially for situations like that.

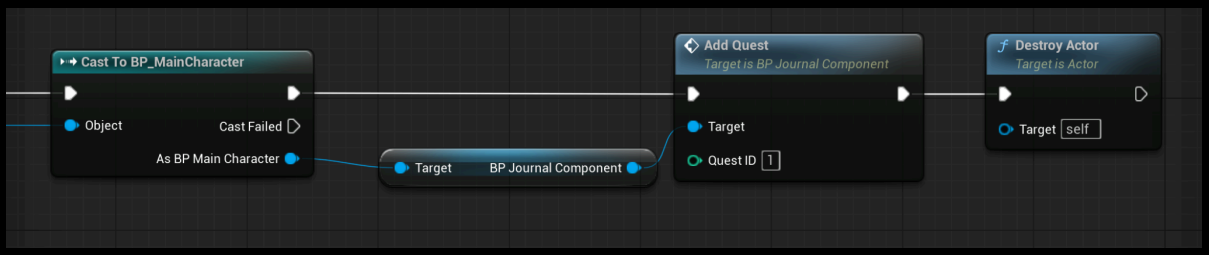


Save the QuestData asset and proceed to Content/QuestSystem. Open DA_QuestDatabase. Put your quest in the array that you find there - make sure that the QuestID matches the array index, because again, (un)expected errors may occur.

Congratulations! You've now created your very first quest!

Adding, updating and finishing quests

All functions needed to make this kind of quest magic are a part of **BP_JournalComponent**. To use them, you need to get a reference to the player object, Cast To BP_MainCharacter, get BP_JournalComponent and then call the function - like so:



If you want to use it from the BP_MainCharacter, just get a reference to BP_JournalComponent and go on with calling the function.

WARNING: Don't try to add a quest of an index that is not specified in the **QuestDatabase** - doing so crashes the editor.

Design note: I think ideally we'll need 3 ways to deal with quests:

- by dialogue - (created 23/07/2025 by Natalia/aranara)
- by walking into a certain place - so by a quest trigger (created 17/07/2025)
- by interacting with an item

All of these will be described below as they get developed.

Walk-in quest triggers

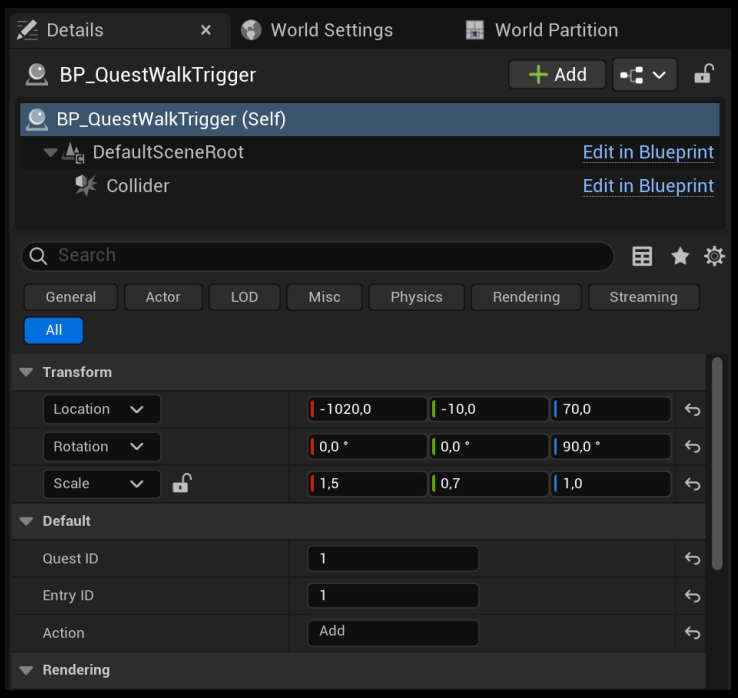
Walk-in quest triggers (or just quest triggers) are a way to deal with quests without the player needing to perform any special kind of action. The quest gets updated when the player enters the quest trigger placed on a map. You can find a blueprint for it under Content/QuestSystem/BP_QuestWalkTrigger.

BP_QuestWalkTrigger is essentially an Actor that doesn't have a mesh, but has a Box Collider which size can be adjusted to your liking. If an actor of class BP_MainCharacter enters this collider, it tries to add, update or finish a quest of specified QuestID (and possibly EntryID).

To do it, place an instance of BP_QuestWalkTrigger on your map and scale it to your liking. In the Details panel, search for Default and fill the variable values - QuestID, EntryID and Action. Action is a string variable that is meant to take in 3 possible values: add, update or finish (not case sensitive, so you can as well enter them as Add, Update or Finish). They all cast to **BP_JournalComponent** and call a function there. If it succeeds, the trigger is destroyed so it can't be used anymore - if not, it will exist in the world and wait until the player enters it again.

Note: when you hover over "Action" in the Details panel, an editor prompt will appear to remind you of the form of the string value.

Note 2: If choose to "add" a quest, EntryID is not used at all - you can either leave it empty or set it to any number you want.



Updating quests via dialogue

A quest can be updated through an event called on a dialogue node. To learn more about that, see Natalia/aranara's **Solaris Dialogue System**.

Letter System

Letter System

Version 1.0 - updated 04/04/2025

Overview

Solaris Letter System is a fairly simple system for managing collectible letters and notes for *Solaris.* Its main goal is to be based on designer-friendly data assets, which are translated in-engine to physical actors and UI elements representing the letters.

The source code is located under *Source/Solaris/Letters,* while all the necessary data can be found in *Content/Letters.*

Breaking down the system

Letter_Data Class

Letter_Data is a C++ based class inheriting from DataAsset. Its only property is a LetterDataStruct, which stores the core data concerning the letter:

- Name - name of the letter, intended to represent where the letter was found or who wrote it (ie. *Cyprian's Letter)*
- Content - content of the letter, stored as a text variable in order to be displayed properly in-game.
- BackgroudImage - material used by the letter's physical representation.
- BackgroundTexture - texture used in the UI after the letter is collected.

The class also provides functions that allow us to read the values of these variables - called, respectively, GetName(), GetContent(), GetMaterial() and GetTexture().

Letter Class

Letter is also a C++ based class. However, unlike Letter_Data, it inherits from UObject, which lets us change its value in-game without corrupting the data assets. It stores a pointer to a LetterData and a boolean describing whether the letter was found by the player. Its main goal is to provide information to the UI and it does so with the following functions:

- FillLetterData(ULetter_Data* Data) - used to initialize an instance of a Letter class in-engine. As bool Found is set to false by default, the only thing that needs to be done is to set the LetterData pointer to a corresponding DataAsset.
- GetLetterData() - returns the LetterData pointer.
- IsFound() - returns if the letter was collected by the player, aka whether it can be displayed in the UI.
- MarkFound() - called when the player collects the letter, changes Found value to true.

LetterDatabase

LetterDatabase is a DataAsset C++ class, which purpose is to keep all Letter_Data assets in place. It provides an array of LetterData pointers to keep an order in the collection. It has two functions: GetDataByID(int id), which returns the Letter_Data of a given index (NULL if the index is invalid) and GetDataArray(), which returns the whole array.

BP_LetterComponent

BP_LetterComponent is a component attached to the player character. It is responsible for managing the letters and whether they have already been found. Its only event is SetupLetterComp, which fills the LetterCollection array with Letter class instances.

BP_BaseLetter

BP_BaseLetter is an Actor blueprint class that handles how the player interacts with the letter in the game world. On BeginPlay, assuming it got provided with a valid LetterID, it takes the information about a letter from LetterDatabase and changes its appearance accordingly. On ActorBeginOverlap and ActorEndOverlap, it shows and hides the UI prompt for interacting with the letter. When Interact input is received, it marks the letter found in BP_LetterComponent's LetterCollection and then destroys itself.

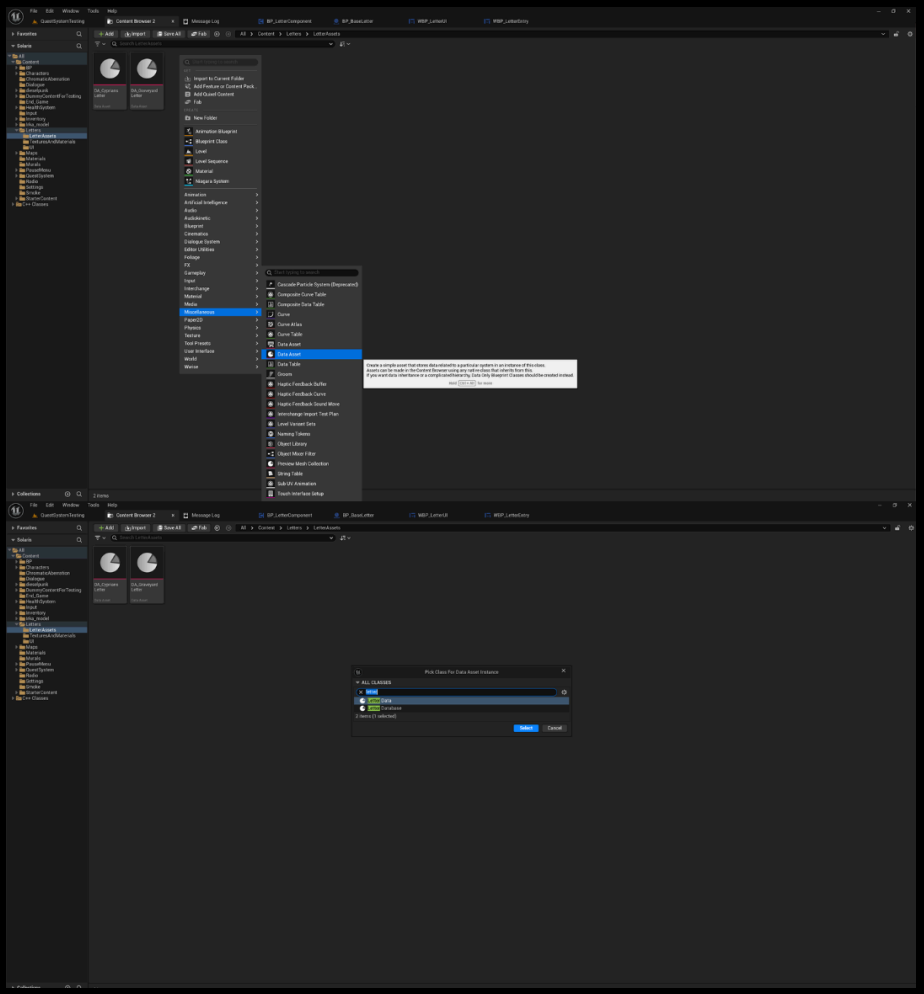
WBP_LetterUI

WBP_LetterUI handles the display of the letter collection. It consists of a GridPanel used to display all letters in game as buttons (no matter if they were collected or not) and a letter preview, which is the background and content of a certain letter. If the letter has not been collected, its image in the button is greyed out and the letter name that should appear on hovering over the button is replaced with "????".

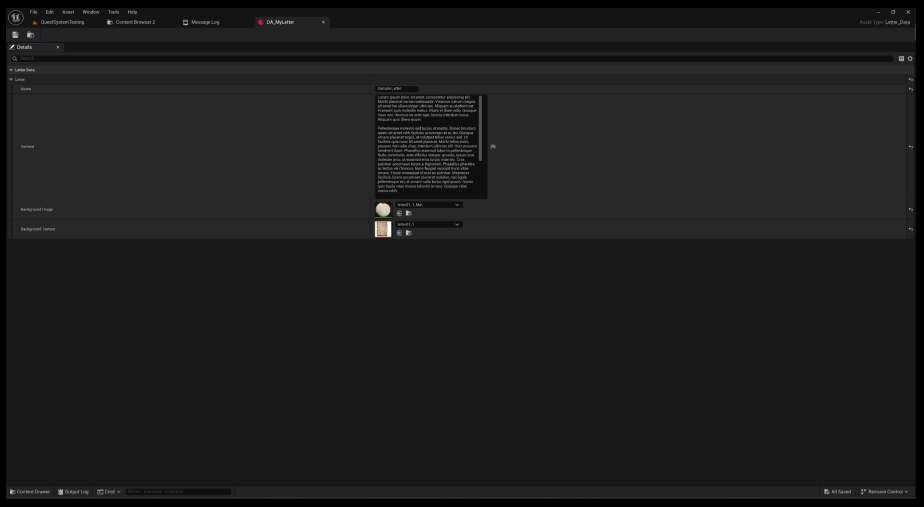
Creating a letter

Now that you have the idea on how the system works, you can create your own letter for Irka to enjoy:)

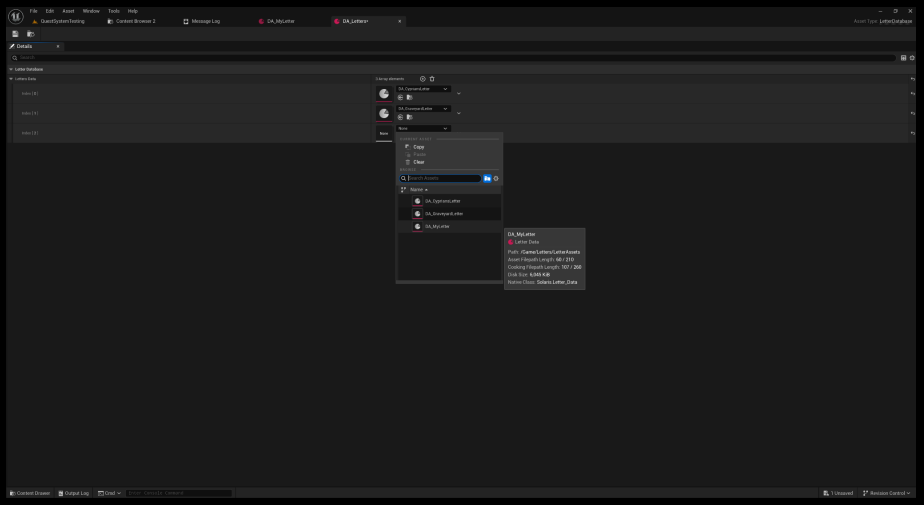
1. Go to *Content/Letters/LetterAssets* and add a new Data Asset of type Letter_Data. Name it in a way that will make it clear what letter it is.



2. Open your new-made asset and fill in all the data. For materials and textures, all of the approved ones start with *letter.*



3. Go to *Content/Letters* and open DA_Letters. Add a new element to the LettersData array and put in your asset.



4. Place a BP_BaseLetter (found in *Content/Letters*) instance in the world and change the default LetterID in the details panel to the one from DA_Letters (here it's 2). Play the game and find your letter already working!