

Contents

DialogueQuest Developer Manual	2
DialogueQuest for non-coders	2
Installation and Setup	2
The Data Directory	3
Exporting	3
Examples	3
Writing Dialogue	3
Creating Characters	4
Exporting Characters	4
See Also	4
Creating Dialogue	4
Playing Dialogue	5
Scene setup	5
Starting the dialogue	5
Stopping the dialogue	5
Settings	6
Settings Options	6
Also see	6
Extending DialogueQuest	6
Theming	6
See Also	6
Custom Statements	7
Custom Logic	8
The Flags system	9
Also see	9
DialogueQuest signals	9
The error signal	10
DQSignals	10
See also	10

DialogueQuest Developer Manual

DialogueQuest for non-coders

DialogueQuest features a standalone program called [DialogueQuestTester](#) that allows running dialogues without a Godot environment.

Installation and Setup

The recommended way to install DialogueQuest is via Godot's builtin Asset Library.

However if you want to get the latest features, you should install via the repository as such:

On Linux / Mac:

```
1 cd my_godot_project
git clone https://github.com/hohfchns/DialogueQuest
3 mkdir -p addons/
mv ./DialogueQuest/addons/DialogueQuest ./addons
5 rm -rf DialogueQuest
```

Or in one line: `git clone https://github.com/hohfchns/DialogueQuest && mkdir -p addons/ && mv ./DialogueQuest/addons/DialogueQuest ./addons && rm -rf DialogueQuest`

On Windows:

In your Godot project:

- Clone the repository
 - Make directory called addons
 - Move the folder `DialogueQuest\addons\DialogueQuest` inside addons
 - Delete the cloned repository
-

Open your Godot project, go to Project -> Project Settings -> Plugins and enable DialogueQuest

Reload the project.

Notice - During the setup process, Godot will show errors due to the plugin not yet being enabled. Until the plugin is enabled and the project is reloaded, you can safely ignore any errors.

The exception to the above are errors that prevent enabling the addon.

The Data Directory

Go to Project -> Project Settings -> General and search for Dialogue Quest, then set Data Directory to a folder where you will store DialogueQuest files (characters, dialogues, etc.)

This folder is by default set to `res://dialogue_quest/`

This folder is where you will be storing your [characters](#) and [dialogues](#)

Exporting

When exporting your game, it is vital to include `.dqd` files in your build, as well as the plugin configuration file.

- Go to Project -> Export
- Select the platform you are exporting to (Linux/Windows/Web/etc.)
- Select the Resources tab
- Under Filters to export non-resource files/folders, add `*.dqd,*.conf`

Examples

To see a basic example of DialogueQuest, see the [examples](#) folder of the repository.

A more advanced example would be the [DialogueQuestTester](#) application.

Writing Dialogue

Writing dialogue is done in the DQD format and is explained in detail in the [User Manual](#).

This manual is for usage and extension of DialogueQuest within a Godot project.

For implementing the dialogue, see [Creating Dialogue](#)

Creating Characters

Before we start creating dialogues, we need to know how to create characters.

Creating characters is quite simple, simply create a new `DQCharacter` Resource in your [Data Directory](#), and DialogueQuest will automatically be able to find and use it.

When creating a character, you must provide a `character_id`. This is how the character will be referred to in [DQD](#).

Exporting Characters

DialogueQuest allows for exporting your characters to files that can then be imported in DialogueQuestTester or another Godot project using DialogueQuest.

Exported DialogueQuest characters are saved in the `.dqc` format.

*Note - Exported characters are **not** Godot resources. Their only functionality is exporting and importing.*

To export, go to the DialogueQuest tab in the Godot editor, select your characters, and click `Export`. You will then be prompted for the directory to save the characters in.

To import, go to the DialogueQuest tab in the Godot editor, and press `Import`, then select all relevant `.dqc` character files.

See Also

[DialogueQuestTester](#)

Creating Dialogue

To create a dialogue, simply create a new `.dqd` file.

If you put the file in your [data directory](#), you will not have to specify a full path for it. See [Starting the Dialogue](#)

Playing Dialogue

Scene setup

To play dialogue, you should set up your scene as follows:

```
...
2 CanvasLayer
    DQDialoguePlayer
4     DQDialogueBox
    DQChoiceMenu
6 ...
```

Click on the `DQDialoguePlayer` and provide it with the `DQDialogueBox`, as well as `DQChoiceMenu`.

Also create a `DQDialoguePlayerSettings` for it. It is recommended to save this resource as a file in your project.

You can also do this setup through code, however make sure the `DQDialoguePlayer` node is set up before it is added to the scene.

When setting up the scene, make sure you instantiate the scene for each component, rather than instantiating the script object.

The scenes can be found at the following paths:

```
prefabs/systems/dqd/dialogue_player.tscn
2 prefabs/ui/dialogue/components/dialogue_box/dialogue_box.tscn
prefabs/ui/dialogue/components/choice_menu/choice_menu.tscn
```

Starting the dialogue

In order to start the dialogue, use the `DQDialoguePlayer.play()` method.

```
dialogue_player.play("my_dialogue_name")
2 # These are also valid ways to provide the dialogue
# dialogue_player.play("my_dialogue_name.dqd")
4 #
    dialogue_player.play("res://dialogue_quest/my_dialogue_name.dqd")
```

Take note - If your `.dqd` file is not in your [data directory](#), you will have to provide the full filepath.

Stopping the dialogue

If you want to stop the dialogue early, you can call the `DQDialoguePlayer.stop()` method which will end the dialogue early.

Settings

Settings are saved in the `.dialogue_quest_settings.conf` file.

When running from the Godot Editor, the settings will be decided by the Project Settings.

Settings Options

- `data_directory`
 - The data directory is explained in the [data directory](#) section.
- `say_by_name`
 - If enabled, the `say` statement will be optional, and will automatically be used if the statement is a valid character ID. This may save time for writers, and is enabled by default.

Also see

[data directory](#)

The user manual entry on the `say` statement

Extending DialogueQuest

Theming

DialogueQuest uses mostly Godot's native Theme system for designing how the interface looks.

You can create a new Theme and import the settings from the default DialogueQuest theme, or create one completely from scratch as the default theme is quite small.

The main way of customizing dialogue components in DialogueQuest is simply creating an inherited scene, and changing it however you like.

Some nodes such as `DQDialogueBox` have `settings` objects, for example `DQDialogueBoxSettings`, which provides some common customizations.

See Also

[Theme](#)

[Using the theme editor](#)

Custom Statements

DialogueQuest allows you to add custom statements to DQD and extend the featureset of DqdParser

To do so, do the following:

```
## my_node_or_autoload.gd
2 class SectionMySection extends DQDqdParser.DqdSection:
4     var statement: String
6
8     func solve_flags() -> void:
9         pass
10
11    func _ready() -> void:
12        DQDqdParser.statements.append(
13            DQDqdParser.Statement.new("my_statement",
14                _my_statement_func)
15        )
16
17    ## Returns SectionPipeline on success
18    ## Returns DQDqdParser.DqdError on failure
19    static func _my_statement_func(pipeline:
20        PackedStringArray):
21        if pipeline.size() <= 2:
22            var error := DQDqdParser.DqdError.new("Error!
23                Cannot parse statement my_statement, please
24                provide at least 2 arguments.")
25            return error
26
27        var sec := SectionMySection.new()
28        sec.statement = pipeline[1] + pipeline[2]
29        return sec
```

First we create a new section class which extends `DQDqdParser.DqdSection`, this can be either a locally defined class like the example, or a new script with `class_name` definition.

We can give it the `solve_flags()` method which will define how the `${flag}` syntax works in [DQD](#).

Now we need to create our parser function, in this case `_my_statement_func`. Note that the `static` is optional, however the rest of the signature is critical.

The function must take in an argument of type `PackedStringArray`, and must return either an object of class inheriting `DqdSection` indi-

cating it is successful, or `DqdError` indicating it has failed.

The `pipeline` argument is an array of every pipe-separated argument in the line the statement was found in.

Note that:

- It contains the statement itself (always, at index 0)
- It contains whitespace, you can use the helper functions `DQScriptingHelper.remove_whitespace`, `DQScriptingHelper.trim_whitespace`, `DQScriptingHelper.trim_whitespace_prefix`, `DQScriptingHelper.trim_whitespace_suffi`

Lastly we must add a new `DQDqdParser.Statement` object to `DQDqdParser.statements`.

The `DQDqdParser.Statement` constructor takes 2 arguments:

- The statement itself, the word that will be referred to in DQD.
- The callback function that will be used to parse the the statement.

Right now, your statement is parsed, however it cannot actually do anything until you implement it's logic. See [Custom Logic](#)

Custom Logic

The logic of DialogueQuest is handled in the `DQDialoguePlayer` class.

In order to add custom logic, you must create a new class extending `DQDialoguePlayer`.

Once you do, you can handle your custom statement like so:

```
## my_dialogue_player.gd
2 extends DQDialoguePlayer

4 func _ready() -> void:
    self.section_handlers.append(
        SectionHandler.new(SectionMySection,
            _handle_my_section),
    )
8
10
12 func _handle_my_section(section: SectionMySection) ->
void:
    # Here you have access to all parts of the
    DQDialoguePlayer
14     print("This section doesn't do anything yet... It's
statement is %s" section.statement)
```

To add a handler, we must add a `SectionHandler` object to the `section_handlers` array.

The constructor of `SectionHandler` takes two parameters, a class (object of type `GDSscript`), and a `Callable`.

When the `parser` returns the class you provided, the function you provided will be called with the parser's returned object.

Now we need to create our handler function, in this case `_handle_my_section`.

The function must take in an argument of your section class, in this case `SectionMySection`. It does not return anything.

The Flags system

Flags are global variables that can be accessed from both code and dialogue.

They are accessible via the global `DQFlags` instance `DialogueQuest.Flags`

An example:

```
2 DialogueQuest.Flags.raise("flag1")
4 DialogueQuest.Flags.set_flag("flag2", 2)
6 DialogueQuest.Flags.set_flag("flag3", "a third flag")
8 # Outputs 2
  print(DialogueQuest.Flags.get("flag2"))
10 # Outputs 'A third flag'
12 print(DialogueQuest.Flags.get("flag3"))
```

Also see

The user manual entry on the `flag` statement

DialogueQuest signals

There are a few important signals in `DialogueQuest`:

The error signal

DialogueQuest uses assert statements for it's critical errors, which will pause the game when running in the editor, however will not do so in a release build.

For the purpose of handling errors in release builds as well as GUI, DialogueQuest emits the `DialogueQuest.error(message: String)` signal when an error occurs.

DQSignals

Other main signals are available via the `DQSignals` instance `DialogueQuest.Signals`

The signals are:

- `dialogue_started(dialogue_id: String)`
- `dialogue_ended(dialogue_id: String)`
- `dialogue_signal(params: Array)`
 - Emitted via the `signal` statement in dialogue.
- `choice_made(choice: String)`
 - Emitted when a player makes a choice during dialogue.

See also

The `signal` statement in the user manual.

The `choice` statement in the user manual.