

Tech Document

Money Doors Tech Guide

Big Money's guide to keeping a well-maintained and organized project

1. Introduction.....	1
2. Files.....	1
2.1. Folders and Files.....	1
2.2. File Names.....	2
2.3. Red Folders.....	2
3. Code.....	3
3.1. Private/Public Variables.....	3
3.2. Lines/Indentation.....	4
3.3. Naming Convention.....	5
3.4. Order.....	6
3.5. Commenting.....	7
3.6. Debugging.....	7
3.7. Globals/Autoloads.....	9
4. Git/GitHub.....	9
4.1. Naming Commits.....	9
4.2. Branching Strategy.....	9

1. Introduction

As the project grows, it gets more difficult to organize it. This is why this document exists, so that everyone can follow the same style of coding and organizing their files. This, ideally, should prevent others from being confused about where to find files and read each others' code.

This is a **living document**. Changes might be added in the future as new information and problems are found.

This document uses the official [Godot guide](#) and some unofficial user guides. In case of any confusion, look under these links:

- [Project organization](#)
- [GDScript style guide](#)
- [How to Write a Git Commit Message](#)
- [Developer Tip: Keep Your Commits "Atomic"](#)
- [Git feature branch workflow](#)

2. Files

Source: [Project organization](#)

2.1. Folders and Files

All files of the same type should be grouped in the same parent folder. They can still be broken down into smaller folders if necessary.

Example:

```
scenes/  
  characters/  
    player.tscn  
    host.tscn  
scripts/  
  characters/  
    player.gd  
    host.gd  
  puzzles/  
    puzzle1.gd  
    puzzle2.gd  
textures/  
  characters/  
    player.png  
    host.png  
  puzzles/  
    puzzle1.png  
    puzzle2.png
```

.gd - script, .tscn - scene, .png - image

2.2. File Names

Godot will automatically generate file and node names for you. When renaming them, stick to the original conventions:

- Folders and files should be named with **snake_case**.
- Nodes should be named with **PascalCase**.
- Root nodes of each puzzle need to be named with **CONSTANT_CASE** due to the integration with [translation document](#).

2.3. Red Folders

Any folders marked with **red color** have a no-touch policy:

- Do not move the folders or files in them
- Do not rename the folders or files in them
- Do not edit the content of the files without prior mention unless you are the creator
- Do not remove any files from the folders unless you are the creator and made sure it would not cause any issues
- You can add files to these folders freely as long as it fits the place

These folders contain files with specific paths mentioned in code. Changing the pre-defined paths will cause errors.

3. Code

Source: [GDScript style guide](#)

3.1. Private/Public Variables

By default, Godot doesn't have built-in variable security, so all variables are technically public. Adding `_` helps knowing which variables and functions are not meant to be used outside of the script. It will prevent Godot from suggesting the variables in other scripts, but not from referencing them.

If a variable is not used or intended to be used by other scripts, make it private.

Example:

```
# Puzzle statistics might be read from other scripts, for example to read how many puzzles were completed, therefore it's public.
var puzzle_statistics: PuzzleStatistics

# Other scripts don't need to know about the save path, therefore it's private.
var _save_file_path = "res://puzzle_data.tres"

# _ready() should never be called by other scripts so it's private.
func _ready() -> void:
>     if not OS.has_feature("editor"):
>         _save_file_path = "user://puzzle_data.tres"
>
>     # Load data
>     puzzle_statistics = load_data()

# Other scripts will need to call save data so it's public.
func save_data(data: PuzzleDataResource):
>     puzzle_statistics.stats.push_back(data)
>     ResourceSaver.save(puzzle_statistics, _save_file_path)
```

3.2. Lines/Indentation

Godot will most of the time automatically arrange indentation for you. Stick to these suggestions, as they are currently the easiest to follow.

- Do not use spaces for indentation. Press **TAB** to add, and **SHIFT+TAB** to remove.
- Each indent level should be one greater than the block containing it.
- Keep line-length under **100 characters**. If the line is too long, separate it into multiple lines.

😊**Good:**

```
seconds = float(  
    current_time.second +  
    current_time.minute * 60 +  
    current_time.hour * 3600  
)
```

😞**Bad:**

```
seconds = float(current_time.second + current_time.minute * 60 +  
current_time.hour * 3600)
```

- When you separate one line into multiple, use two indentations (press TAB twice), except for enums and arrays.
- Separate parts of the code (variables, functions, etc.) with two lines.
- Separate functions by two lines.
- Separate types of variables (public, exported, private etc.) with one line.
- Separate long loops and if statements with one line.
- Separate logical sections inside the function with one line.

3.3. Naming Convention

Type	Convention	Example
File names	snake_case	yaml_parser.gd
Class names	PascalCase	class_name YAMLParser
Node names	PascalCase	Camera3D, Player
Functions	snake_case	func load_level():
Variables	snake_case	var particle_effect
Signals	snake_case	signal door_opened
Constants	CONSTANT_CASE	const MAX_SPEED = 200
Enum names	PascalCase	enum Element
Enum members	CONSTANT_CASE	{EARTH, WATER, AIR, FIRE}

- **Private variables** - Add _ (floor symbol) before their names.
- **Signals** - Use past tense.
For example: door_opened, score_changed.
- **Booleans** - Use past tense and write them as questions.
For example: is_grounded, is_dead.
- When naming variables, do not use shortcuts, unless the word is a shortcut (like hp).
Good: speed, gravity, time_scale.
Bad: spd, g, ts.
- Names should explain what the variable/function does.

3.4. Order

1. Script declarations:

- 1.1. class_name
- 1.2. extends
- 1.3. comment

2. Variables

- 2.1. signals
- 2.2. enums
- 2.3. constants
- 2.4. static
- 2.5. @export
- 2.6. regular
- 2.7. @onready

3. Methods

- 3.1. static
 - 3.2. built-in, (like *ready()* and *process()*) in the order of their activation
For example, ready() runs before process(), so put it before.
 - 3.3. remaining methods
- Public methods come before private.

3.5. Commenting

Comment your code where it's needed. Do not assume that just because you know what it does, everyone else will.

- All public variables and methods should be commented!
 - This is to allow usage of the variables outside of the class
 - If it's not intended to be used outside of the class, make it private
 - You can still add comments to private variables
- Use # sign for writing comments about your code.
- Add space after # sign for comments, and don't add it for disabled code.
- Avoid in-line comments unless very short.
- Use ## to explain what variable / method does. Especially when using @export. The comment will be attached to the next variable below.
- Write comments about what the code does in imperative form.
- Write explanations to methods in third person.
- End your comments with a dot, except for commented out code.

Example:

```
## How fast the character moves in units per ms.
@export speed: float = 1.0
## The character won't be able to rotate past this point.
## Counted in degrees.
@export rotation_limit: float = 45.0

## Calculates current speed based on initial speed and outside factors.
func calculate_speed():
>   var _current_speed = speed
>   _current_speed = speed * slow_factor * speed_boost
>   #print("Speed is: " + current_speed)
>
>   # Stop moving completely if slowed too much.
>   if (_current_speed >= 0.1):
>       _stop_moving()
```

3.6. Debugging

- You can debug your code with `print()` or `print_rich()` functions.
- When debugging your code, make sure that the debug points towards what it's debugging.
- Debug should be descriptive, even if it's not intended to stay.
- If the debug is not meant to be relevant in the build, it should be either deleted or commented out.
- For multiple lines of debug, add indentation for readability. You can do that by pressing **[TAB]**:

```
print(">Rounds left: " + str(Global.rounds_left))
```

Good:

```
func _ready() -> void:
    test_resource = load_data()
    print("Variable is ", test_resource.variable, ". Adding +1.")
    test_resource.variable += 1
    print("Test save file variable: ", test_resource.variable)

func load_data():
    if ResourceLoader.exists(_save_file_path):
        print("UID: ", ResourceLoader.get_resource_uid(_save_file_path))
        var _resource = load(_save_file_path)
        print("Save file found and loaded.")
        return _resource

    var _new_resource = TestResource.new()
    print("Save file not found. New save file was created")
    return _new_resource
```

Bad:

```
func _ready() -> void:
    test_resource = load_data()
    print(test_resource.variable)
    test_resource.variable += 1
    print(test_resource.variable)
    save_data()

func load_data():
    if ResourceLoader.exists(_save_file_path):
        print(ResourceLoader.get_resource_uid(_save_file_path))
        var _resource = load(_save_file_path)
        print("works")
        return _resource

    var _new_resource = TestResource.new()
    print("nope")
    return _new_resource
```

In debug:

Good:

```
UID: -1
Save file found and loaded.
Variable is 6. Adding +1.
Test save file variable: 7
```

Bad:

```
-1
works
7
8
```

3.7. Globals/Autoloads

When making an Autoload, do not add it through Project Settings.

Do this:

1. Add your .gd file to `res://scripts/globals/`
2. Create a Node scene in `res://scenes/globals/` and attach the script to it.
3. Go to `res://addons/autoloader/` and open `autoloader.gd`.
4. Add your Autoload to AUTOLOADS dictionary where key is the class name, and value is the file name of the scene. Each Autoload should be in a separate line, and have a comma at the end.

Example:

```
const AUTOLOADS: Dictionary[String, String] = {  
    "Global" : "global.tscn",  
    "MoneyManager" : "money_manager.tscn",  
    "PlayerData" : "player_data.tscn",  
}
```

4. Git/GitHub

4.1. Naming Commits

Source: [How to Write a Git Commit Message](#)

Summary:

- Always explain what was done in the summary.
- Keep summary in one sentence.
- Limit it to 50 characters.

GitHub Desktop will warn you if your summary is too long. If you struggle to keep a short summary, you are probably not committing often enough.

- Capitalize the first letter.
- Do not end with a period.
- Use the imperative mood.

Description:

- Add description unless the commit is very simple (like fixing a typo).
- Since a description can have multiple sentences, end it with a period.
- Use the first paragraph to explain “what”.
- Add another paragraph, with one blank line between, to explain “why” if needed.

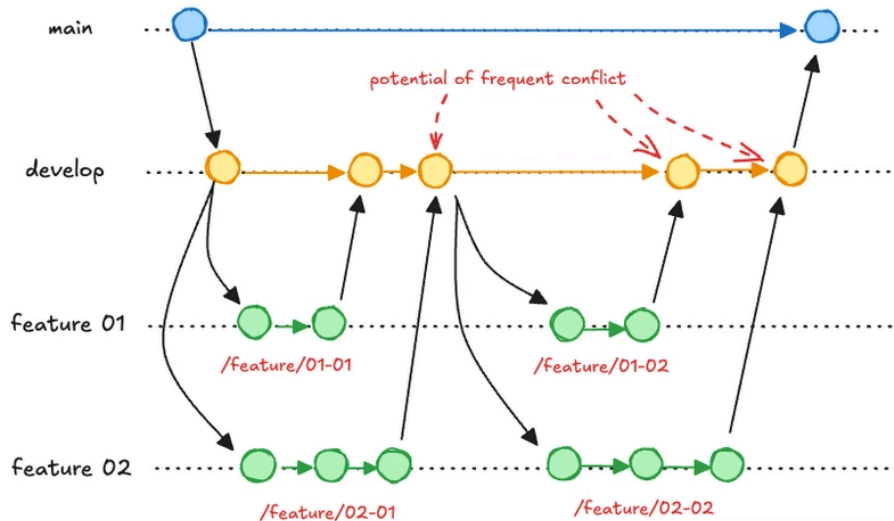
4.2. Branching Strategy

Source: [Git feature branch workflow](#)

For the current project, **Feature Branching** is used as a main strategy. This means that:

- main branch is never worked on
- Each feature is a separate branch
- Features are merged into develop branch

Feature Branching



When starting your work on the project, follow these steps:

1. Create a new branch with the name of the feature
 - You do not need to publish the branch unless you leave work or someone asks to work on it
2. Commit changes regularly and when taking a break
3. Finish work on the feature
4. Make a pull request
 - At this point the branch needs to be published
5. Wait till your request is approved
6. Squash-merge the branch into develop
 - This part can be done by someone else