Making your Tools #1
Data Based
Inventory System
with an Editor

Written by Threeclaws

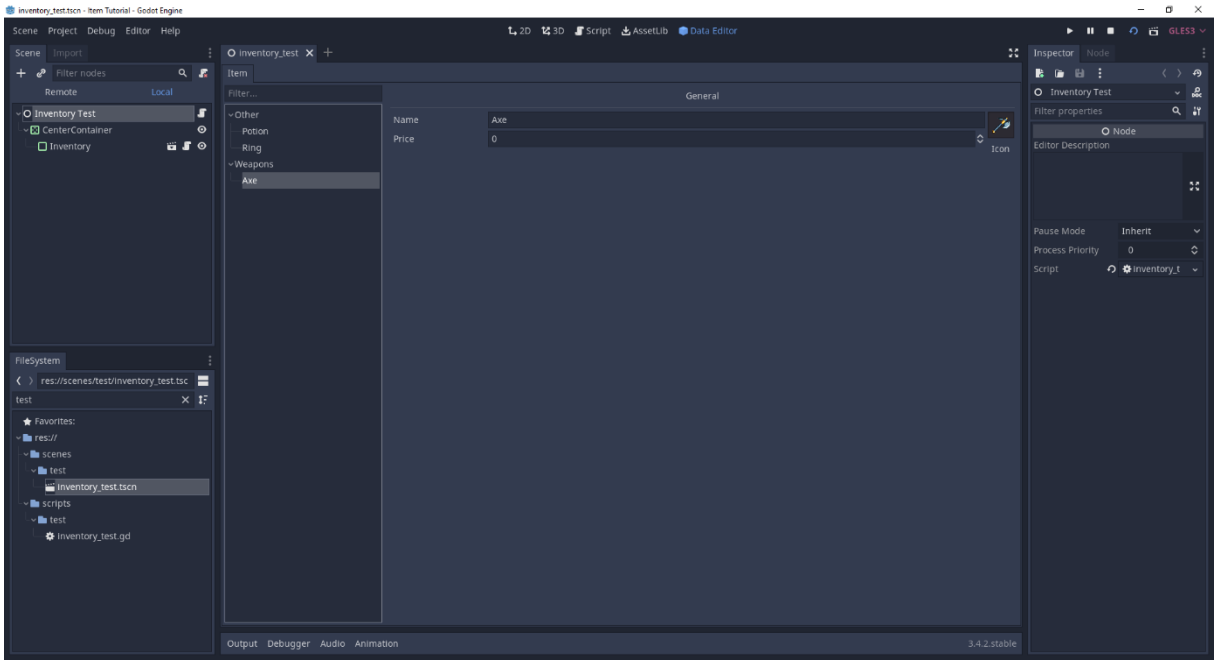
Table of Contents

Introduction

If you're reading this, you probably already have an idea what an inventory system is. You might even have made one already. Even so, this tutorial may still be of use. For one, this tutorial links to several other tutorials which could prove helpful. For another, this tutorial goes beyond the level of simply making a UI element to store pictures of items in. It even goes beyond the level of making resource files for the item data. Rather this tutorial intends to demonstrate on how to make an inventory system that can not only be used by its creator, but also other co-developers.

This tutorial intends make the process of creating and editing the item data easier and less error prone. No longer will you have to swap between applications to edit the item data. Neither will you have to search through the file browser to find the item resources.

To achieve this, you will build your own plugin data editor that can be used inside the Godot Editor in addition to a simple, but easily expandable inventory system, the item data, and the beginnings of a save system for the runtime data.



Requirements

To be able to understand this tutorial the following foreknowledge is required:

- Understands Godot's basic node structure.
- Able to read and understand GDScript(variables, functions, autoload/singletons, constants, signals/observer pattern, export hints, static typing, inheritance).
- Knows how to load PackedScenes and instantiate them.
- Understands tool scripts.
- Understands how main screen plugins are made.

Download Links

This tutorial should be read with the source code on the side since the code of several script files is too long to be decently pasted into a text file. You can find the project by using the following download link:

https://github.com/Threeclaws90/godot_item_tutorial.git

The icons used in this tutorial have been made by LimeZu and can be downloaded using the following page:

https://limezu.itch.io/rpg-arsenal

Item Data

First off, we begin by setting up the data structures and the data manager. The data manager will be the singleton which loads the item data and icons when the game or editor are started, and which saves the item data to a file. The data structure used to store the items will be a hierarchy of dictionaries. An array may also be used; however, it'll be more work to implement a hierarchical structure.

Savable, IDValue, and Item

The item class inherits from the IDValue class, which inherits from the Savable class.

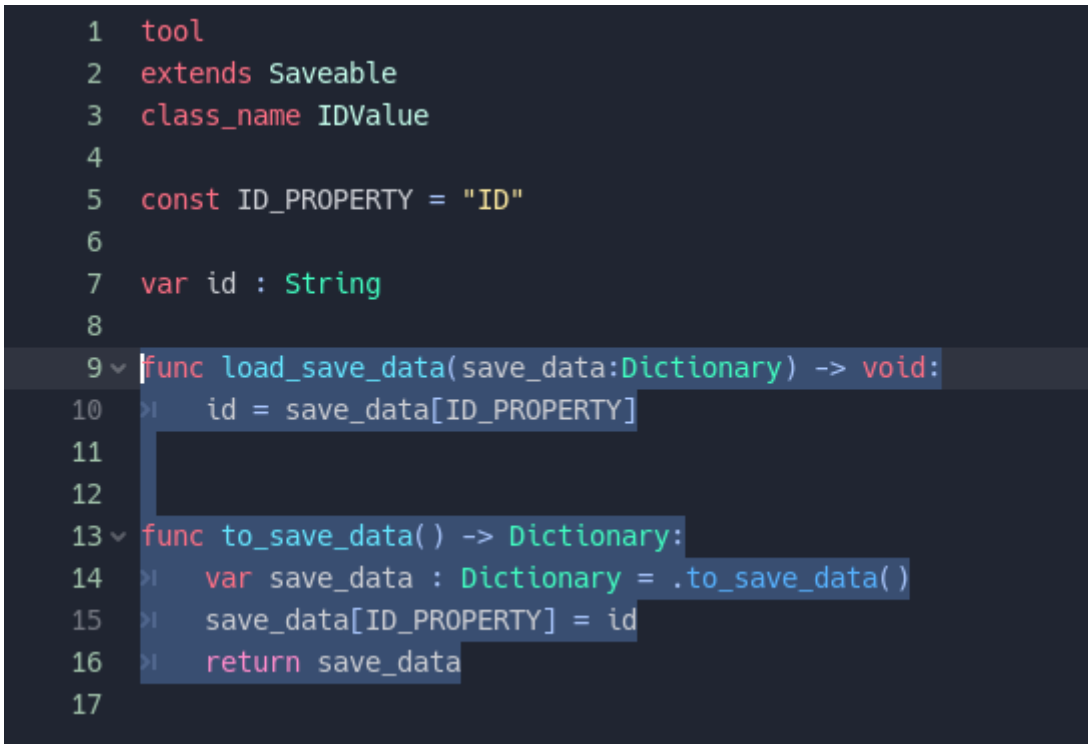
The Savable class is simply an object that converts the object data to a dictionary which can be easily saved via the *JSON.print* method. This data can also easily be loaded by the Savable object. The most notable methods of the savable object are *variant_to_save_data* and *load_variant*. The method *variant_to_save_data* converts primitives, arrays, dictionaries, resources objects, savable objects, vector2, vector3, and colors to values that can be easily printed via the *JSON.print* method.

```
25 static func variant_to_save_data(variant):
26     match typeof(variant):
27         TYPE_BOOL, TYPE_INT, TYPE_REAL, TYPE_STRING:
28             return variant
29         TYPE_ARRAY:
30             return array_to_save_data(variant)
31         TYPE_DICTIONARY:
32             return dictionary_to_save_data(variant)
33         TYPE_OBJECT:
34             return object_to_save_data(variant)
35         TYPE_VECTOR2:
36             return vec2_to_save_data(variant)
37         TYPE_VECTOR3:
38             return vec3_to_save_data(variant)
39         TYPE_COLOR:
40             return color_to_save_data(variant)
41     return null
42
```

Contrarily the method *load_variant* loads the JSON data and converts it back to the original values. Be careful however when loading numbers. Numbers in JSON are generally always floats, which the savable does not take care of.

```
124 static func load_variant(save_data, object_creator:FuncRef = null):
125     match typeof(save_data):
126         TYPE_BOOL, TYPE_INT, TYPE_REAL, TYPE_STRING, TYPE_VECTOR2, TYPE_VECTOR3, \
127         TYPE_COLOR, TYPE_OBJECT, TYPE_ARRAY:
128             return save_data
129         TYPE_DICTIONARY:
130             match save_data.get(SAVEABLE_TYPE_PROPERTY):
131                 ARRAY_TYPE:
132                     return load_array(save_data[SAVE_DATA_PROPERTY], object_creator)
133                 DICTIONARY_TYPE:
134                     return load_dictionary(save_data[SAVE_DATA_PROPERTY], object_creator)
135                 SAVEABLE_TYPE:
136                     return load_saveable(save_data, object_creator)
137                 RESOURCE_TYPE:
138                     return load_external_resource(save_data[RESOURCE_PATH_PROPERTY])
139                 VEC_2_TYPE:
140                     return load_vec2(save_data)
141                 VEC_3_TYPE:
142                     return load_vec3(save_data)
143                 COLOR_TYPE:
144                     return load_color(save_data)
145                 _:
146                     return save_data
147     return null
148
```

The ID Value class on the other hand is simply a savable object that has an id. It simply retrieves the id from the save data when loading the value and stores it in the save data when saving.



```
1  tool
2  extends Saveable
3  class_name IDValue
4
5  const ID_PROPERTY = "ID"
6
7  var id : String
8
9  func load_save_data(save_data:Dictionary) -> void:
10     id = save_data[ID_PROPERTY]
11
12
13 func to_save_data() -> Dictionary:
14     var save_data : Dictionary = .to_save_data()
15     save_data[ID_PROPERTY] = id
16     return save_data
17
```

Finally, the items. To keep this example simple, items will also only have an id, a name, a price, and an icon. Here the use of the load_variant and variant_to_save_data methods can also be observed as they're used to convert the texture to its resource_path and store said resource_path instead of the entire image. When loading the resource_path is then used to reassign the texture. However, if the texture file was moved outside the editor or deleted. In this case the icon will be set to null and hence not be visible in the inventory.

```
1  tool
2  extends IDValue
3  class_name ItemData
4
5  const NAME_PROPERTY = "Name"
6  const PRICE_PROPERTY = "Price"
7  const ICON_PROPERTY = "Icon"
8
9  var name : String
10 var price : int
11 var icon : Texture
12
13
14 func load_save_data(save_data:Dictionary) -> void:
15     .load_save_data(save_data)
16     name = save_data[NAME_PROPERTY]
17     price = save_data[PRICE_PROPERTY]
18     icon = load_variant(save_data[ICON_PROPERTY])
19
20
21 func to_save_data() -> Dictionary:
22     var save_data : Dictionary = .to_save_data()
23     save_data[NAME_PROPERTY] = name
24     save_data[PRICE_PROPERTY] = price
25     save_data[ICON_PROPERTY] = variant_to_save_data(icon)
26     return save_data
27
```

Data Manager

The data manager contains a list of all files that data will be loaded from and saved to, as well as a list of all directories that contain resources which will be used by the editor, which will be expounded upon later.

First, the data manager contains 1 method for loading the data, *_load_data*, which is called in the *_ready* method and 1 *save_data* method which will be called whenever the developer saves the scene data.

The data manager also contains methods to access the items, *get_item* to fetch the data of a specific item, *get_items* to receive the entire data structure, *list_items* to receive all items in an array, and *list_item_paths* to receive all absolute paths to item values.

```
4   const ITEM_FILE = "res://data/items.json"
5
6   const ICON_DIRECTORY = "res://graphics/icons/"
7
8   var _items : Dictionary
9
10  func _ready() -> void:
11      _load_data()
12
13
14  func _load_data() -> void:
15      _items = Saveable.load_saveable_from_file(ITEM_FILE, null, {})
16
17
18  func save_data() -> void:
19      if not Saveable.save_to_file(ITEM_FILE, _items):
20          printerr("Could not save items.")
21
22
23  func get_item(item_path:String) -> ItemData:
24      return _find_value(item_path, _items)
25
26
27  func get_items() -> Dictionary:
28      return _items
29
30
31  func list_items() -> Array:
32      return _accumulate_values(_items)
33
34
35  func list_item_paths() -> Array:
36      return _accumulate_paths(_items)
37
38
39  func list_icon_paths() -> Array:
40      return _get_directory_paths(ICON_DIRECTORY, ["jpg", "png"])
41
```

The utility method *_find_value* used in *get_item*, returns a value found in the hierarchy which follows the given path (if it helps think of it like a path in the file system on your pc).

```
43 func _find_value(path:String, dictionary:Dictionary):
44     var segments : Array = Array(path.split("/"))
45     var value : Reference = null
46     while segments.size() > 0:
47         var segment : String = segments.pop_front()
48         if segments.empty():
49             value = dictionary.get(segment)
50         else:
51             dictionary = dictionary[segment]
52     return value
```

The *_accumulate_values* method and the *_accumulate* paths gather all the values and absolute paths inside the hierarchy.

```
55 func _accumulate_values(dictionary:Dictionary) -> Array:
56     var values : Array = []
57     for value in dictionary.values():
58         if typeof(value) == TYPE_DICTIONARY:
59             var sub_values : Array = _accumulate_values(value)
60             values.append_array(sub_values)
61         else:
62             values.append(value)
63     return values
64
65
66 func _accumulate_paths(dictionary:Dictionary, path:String = "") -> Array:
67     var paths : Array = []
68     for key in dictionary:
69         var value = dictionary[key]
70         var full_path : String = path.plus_file(key)
71         if typeof(value) == TYPE_DICTIONARY:
72             var sub_paths : Array = _accumulate_paths(value, full_path)
73             paths.append_array(sub_paths)
74         else:
75             paths.append(full_path)
76     return paths
```

Finally, the *_get_directory_paths* method collects all files in a directory that end with the given file extensions. This method is not only useful for fetching paths to image files, but also other file types, such as audio files (ogg, wav), video files (ogv, webm), scene files (tscn), or other godot resource files (tres) if you need them.

```
79 func _get_directory_paths(directory_path:String, allowed_file_types:Array) -> Array:
80     var directory : Directory = Directory.new()
81     var paths : Array = []
82     if directory.open(directory_path) == OK:
83         if directory.list_dir_begin(true, true) == OK:
84             while true:
85                 var file_name : String = directory.get_next()
86                 if file_name.empty():
87                     break
88                 var full_path = directory_path.plus_file(file_name)
89                 if directory.current_is_dir():
90                     var sub_paths = _get_directory_paths(full_path, allowed_file_types)
91                     paths.append_array(sub_paths)
92                 elif allowed_file_types.has(file_name.get_extension()):
93                     paths.append(full_path)
94         directory.list_dir_end()
95     else:
96         push_warning("Could not open directory \"%s\"" % [directory_path])
97     return paths
```

Making Changes to the Data Manager and Data Scripts

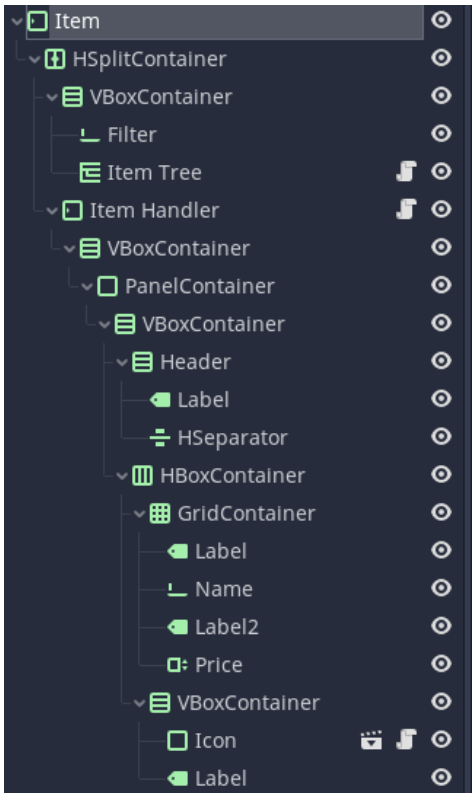
THIS PART IS IMPORTANT! Whenever you make changes to the data manager, or any of the scripts used for the data loaded inside the data manager YOU SHOULD NOT SAVE NORMALLY AND ONLY SAVE THE SCRIPTS VIA THE SCRIPTS EDITOR OR ALL DATA MAY BE LOST!

To avoid irreparable damage to your data, it is recommended to use tools for source control, such as Github or Bitbucket.

Editor

Item Editor

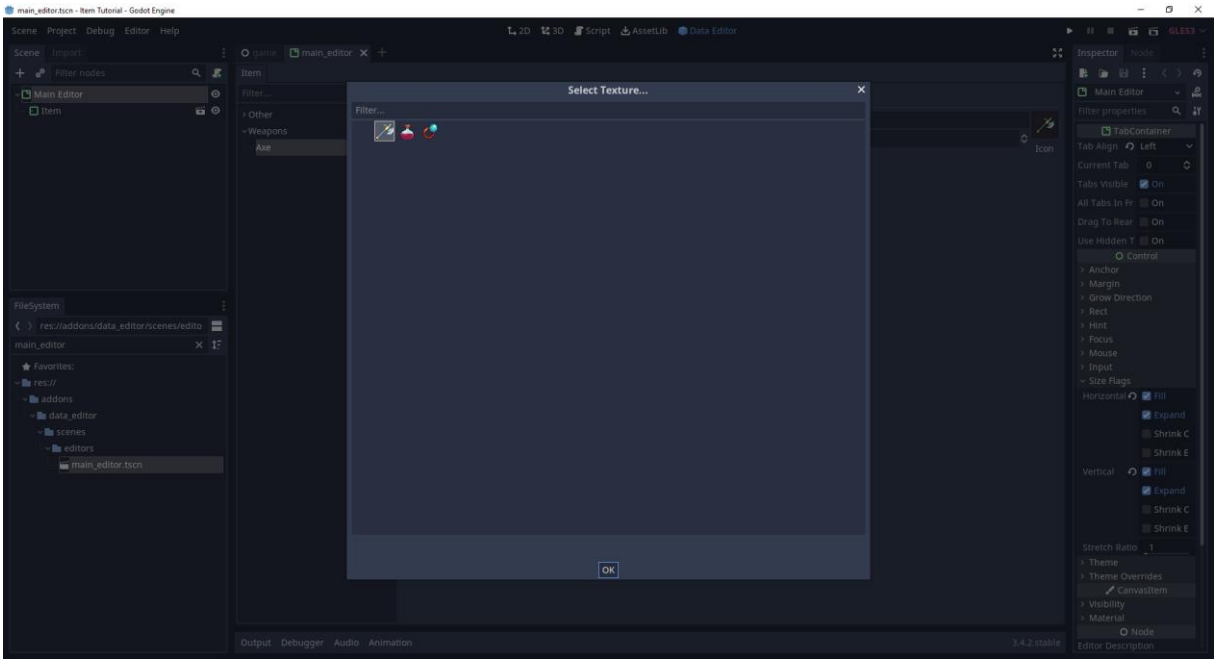
The item editor is built using the following node structure. Mind the icon, as it's built by inserting the *texture_selection.tscn* scene, which is a custom made component for selecting a texture from a given array of textures. The Filter Line Edit was given minimum width of 250 and the vertical size flag of the Item Tree is set to expanded.



The editor looks as follows.

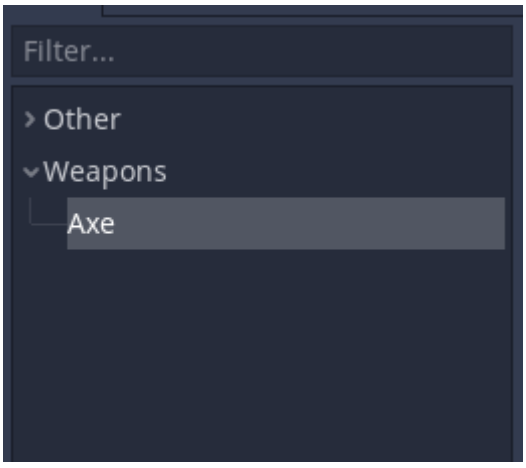
Texture Selection

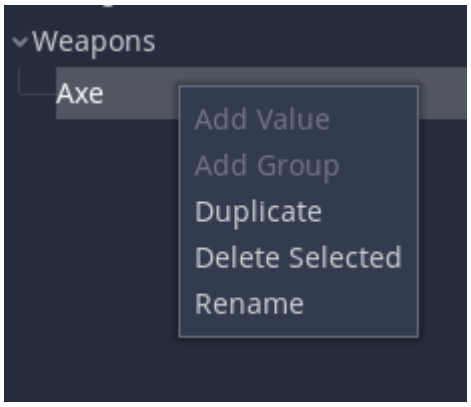
The texture selection added to the editor is a custom component. It consists of three parts. The button that is clicked to open the popup, the popup that contains a filter Line Edit, and a Grid Container which contains all the texture selection buttons. The texture selection buttons do exactly as their name implies. If the user clicks onto the texture selection button, the given texture is selected. If the user double clicks on the texture selection button, or if the user presses the ok button, the selection is confirmed, and the popup is closed.



Item Tree

The item tree is a script that handles the filling and the interaction with the tree node and extends the utility script of the dictionary tree. To make the work of the developers easier, a filter Line Edit is added. To add new values, to add a group, to duplicate a value, to delete a value, or to rename a value a context menu will be displayed whenever the user right clicks onto the tree.





The item tree mostly overrides methods from the dictionary tree, inserting the missing behavior such as generating a new value, or selecting a given tree item and setting the value of the handlers.

```
4  export(NodePath) var item_handler_path : NodePath
5
6  onready var _item_handler = get_node(item_handler_path) as Control
7
8  func _ready() -> void:
9      connect("selection_changed", self, "_on_item_selected")
10     # Wait for the _ready function to be executed
11     yield(get_tree(), "idle_frame")
12     _dictionary = DataManager.get_items()
13     refresh()
14
15 func _new_value(id:String) -> IDValue:
16     var new_item : ItemData = ItemData.new()
17     new_item.id = id
18     new_item.name = id
19     return new_item
20
21
22 func _duplicate_value(value:IDValue) -> IDValue:
23     var copy = value.copy()
24     return copy
25
26
27 func _on_item_selected(selected:TreeItem) -> void:
28     if selected == null:
29         _item_handler.value = null
30     else:
31         var meta = selected.get_metadata(0)
32         if typeof(meta) == TYPE_DICTIONARY:
33             _item_handler.value = null
34         else:
35             _item_handler.value = meta
36
37
38 func _on_visibility_changed() -> void:
39     ._on_visibility_changed()
40     if get_dictionary() != DataManager.get_items():
41         set_dictionary(DataManager.get_items())
```

Dictionary Tree

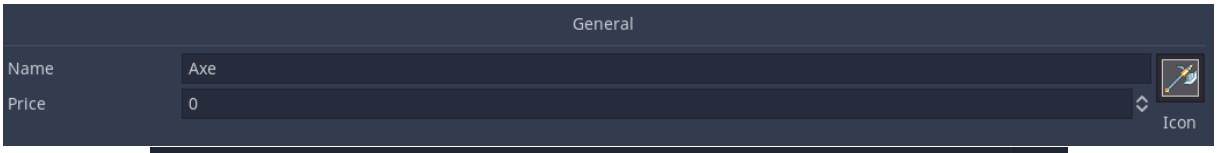
The dictionary tree handles most stuff that'll be similar for most data trees, such as input handling (context menu, drag and drop, selection & deselection) or filling the tree with tree items. It also circumvents the somewhat buggy tree item selection and tries to store and restore the collapsed items when being refreshed. If you aren't already experienced with the use of the Tree node in Godot and want to learn more about how to use it, this class may serve as a useful introduction.

Below follows an example of how the tree items are being added to the Tree node in the Dictionary Tree.

```
219 func _create_dictionary_item(parent:TreeItem, dictionary:Dictionary,
220         name:String) -> TreeItem:
221     var item : TreeItem = _create_value_item(parent, dictionary, name)
222     var path = _get_item_path(item)
223     var keys : Array = dictionary.keys()
224     keys.sort()
225     for key in keys:
226         var value = dictionary[key]
227         var sub_path = path.plus_file(key)
228         if typeof(value) == TYPE_DICTIONARY and _is_dictionary_filtered(sub_path, value):
229             _create_dictionary_item(item, value, key)
230     for key in keys:
231         var value = dictionary[key]
232         var sub_path = path.plus_file(key)
233         if typeof(value) != TYPE_DICTIONARY and _is_path_filtered(sub_path):
234             _create_value_item(item, value, key)
235     item.collapsed = ((item != get_root()) and (_collapsed_items.has(path)))
236     return item
237
238
239 func _create_value_item(parent:TreeItem, value, name:String) -> TreeItem:
240     var item : TreeItem = create_item(parent)
241     item.set_text(0, name)
242     item.set_metadata(0, value)
243     return item
```

Item Handler and Value Handler

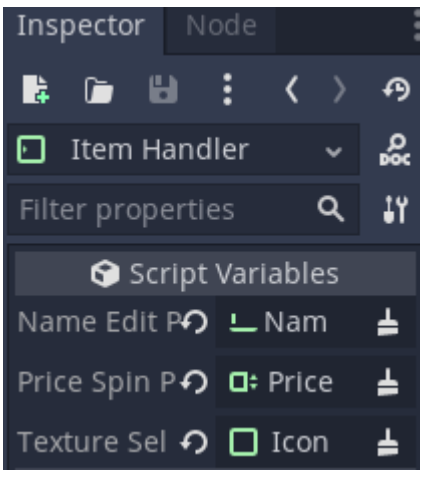
The item handler is a script that extends from the value handler scripts and is used for updating and connecting the fields used for editing the item data.




```gdscript
tool
extends ValueHandler

export(NodePath) var name_edit_path : NodePath
export(NodePath) var price_spin_path : NodePath
export(NodePath) var texture_select_path : NodePath

onready var _name_edit = get_node(name_edit_path) as LineEdit
onready var _price_spin = get_node(price_spin_path) as SpinBox
onready var _texture_select = get_node(texture_select_path) as TextureSelect

func _ready_connections() -> void:
    _name_edit.connect("text_changed", self, "_on_value_property_changed",
            ["name"])
    _price_spin.connect("value_changed", self, "_on_value_property_changed",
            ["price"])
    _texture_select.connect("texture_selected", self, "_on_value_property_changed",
            ["icon"])


func _refresh_controls_enabled() -> void:
    _refresh_control_enabled(_name_edit, "editable")
    _refresh_control_enabled(_price_spin, "editable")
    _refresh_control_enabled(_texture_select, "disabled", true)


func _refresh_control_values() -> void:
    _refresh_control_value(_name_edit, "text", "name", "")
    _refresh_control_value(_price_spin, "value", "price", 0)
    _set_node_value(_texture_select, "all", DataManager.list_icon_paths())
    _refresh_control_value(_texture_select, "texture", "icon")

```

Nodepaths

First the item handler contains export hints for the name Line Edit, the price Spin Box and the icon Texture Select. This is done using export hints, rather than hardcoded node paths, to make changes to the editor structure easier (if node paths are used the editor automatically updates the path to the given node when the scene changes). Also, node paths are chosen in the inspector, hence allowing even non-programmers on the team to easily set them.



Value Handler

The main function of the value handler is to refresh itself once the selected value changes. It provides some utility functions for observing changes to the node values as well as for updating the nodes.

```
25 func set_value(value) -> void:
26     var prev = _value
27     _value = value
28     if prev != value:
29         refresh()
```

```
36 func refresh() -> void:
37     _refresh_control_connections()
38     _refresh_controls_visible()
39     _refresh_controls_enabled()
40     _refresh_control_values()
```

```
75 func _refresh_control_enabled(control:Control, enabled_property:String,
76         inverted:bool = false) -> void:
77     if control != null:
78         var value = (is_enabled() != inverted)
79         control.set(enabled_property, value)
```

```
82 func _refresh_control_value(control:Control, control_property:String,
83         value_property:String, default_value = null) -> void:
84     if control != null:
85         var value = _get_value_property(value_property, default_value)
86         control.set(control_property, value)
```

```
113 func _on_value_property_changed(value, property:String) -> void:
114     _set_value_property(property, value)
115
```

```
98 func _set_value_property(property:String, value) -> void:
99     if _value != null:
100         if _value is Dictionary:
101             _value[property] = value
102             emit_signal("property_changed", property)
103         elif _value is Object:
104             _value.set(property, value)
105             emit_signal("property_changed", property)
```

Main Editor

Theoretically, we could add the item editor to the main screen plugin now. However, if more editors were to be added, such as an armor or weapon editor, this would easily start to clutter the screen and hence, we're going to create a main editor, which contains the item editor and future data editors.

I've found the node that works best for this purpose is the Tab Container, though it has the drawback of getting difficult to handle if there are a lot of editors and the user must scroll through the tab bar.

Plugin Script

Before continuing with this tutorial, make sure you understand how main screen plugins are made. You can find the tutorial onto how to make them here.

The plugin script of the data editor plugin mainly adds the main editor to the editor viewport and tells the data manager to save the data when the user saves scenes or the editor autosaves before the game is run or closed.

```
tool
extends EditorPlugin

const MAIN_EDITOR_PREFAB : PackedScene = preload("res://addons/data_editor/scenes/editors/main_editor.tscn")
var _main_editor : Control

func _enter_tree() -> void:
    _main_editor = MAIN_EDITOR_PREFAB.instance()
    get_editor_interface().get_editor_viewport().add_child(_main_editor)
    make_visible(false)


func _exit_tree():
    if _main_editor != null and is_instance_valid(_main_editor):
        get_editor_interface().get_editor_viewport().remove_child(_main_editor)
        _main_editor.queue_free()


func save_external_data():
    DataManager.save_data()


func has_main_screen() -> bool:
    return true;


func make_visible(visible:bool) -> void:
    if _main_editor:
        _main_editor.visible = visible


func get_plugin_name() -> String:
    return "Data Editor"


func get_plugin_icon() -> Texture:
    return get_editor_interface().get_base_control().get_icon(
        "MaterialPreviewCube", "EditorIcons")
```

Runtime Data

The runtime data is the data that is all the data of the player's current playthrough, such as the current player progress, the player location, npc locations and their current state, and so on. This data too will inherit from the savable script, though the risk of losing all data when editing the scripts in the editor is gone. The reason why these scripts should inherit from the savable script is to make saving the game easy, i.e., we don't have to write a lot more code for converting the game data to save data, which can be stored in save file.

Game Inventory Slot

We start with writing the game inventory slot script. This script tracks the state of an inventory slot, which item is stored in the slot, and how many items of the given type are stored in the slot. If this changes a signal is sent to the observers, which in this tutorial, will be the item slot node.

```
 9  var item_id setget set_item_id, get_item_id
10  var count setget set_count, get_count
11
12  var _item_id : String
13  var _count : int
14
15  func get_item_id() -> String:
16      return _item_id
17
18
19  func set_item_id(value:String) -> void:
20      _item_id = value
21      emit_signal("content_changed")
22
23
24  func get_count() -> int:
25      return _count
26
27
28  func set_count(value:int) -> void:
29      _count = value
30      emit_signal("content_changed")
31
```

The game inventory slot also implements the methods *to_save_data* and *load_save_data* so that the state of the inventory can be saved and loaded when the player saves the game.

```
51  func load_save_data(save_data:Dictionary) -> void:
52      _item_id = save_data[ITEM_ID_PROPERTY]
53      _count = save_data[COUNT_PROPERTY]
54
55
56  func to_save_data() -> Dictionary:
57      var save_data : Dictionary = .to_save_data()
58
59      save_data[ITEM_ID_PROPERTY] = _item_id
60      save_data[COUNT_PROPERTY] = _count
61
62      return save_data
63
```

Finally, the game inventory also allows for swapping its contents with another inventory slot, making sorting an inventory, or transferring items from one inventory to another (for instance the inventory of a chest) easier.

```
33 func swap_with_slot(other_slot:Reference) -> void:
34     var other_slot_item : String = other_slot._item_id
35     var other_slot_count : int = other_slot._count
36     other_slot.item_id = _item_id
37     other_slot.count = _count
38     _item_id = other_slot_item
39     _count = other_slot_count
40     emit_signal("content_changed")
41
```

Game Inventory

The game inventory is an extremely simple script. It contains an array of all the slots and saves and loads them from the save data. For testing purposes, the start size of the inventory is set to 36 and the slots are added to the inventory when it is created.

```
1  extends Saveable
2  class_name GameInventory
3
4  const DEFAULT_SIZE = 36
5  const SLOTS_PROPERTY = "Slots"
6
7  var slots : Array
8
9  func _init():
10     slots = []
11     # warning-ignore:unused_variable
12     for i in range(DEFAULT_SIZE):
13         slots.append(GameInventorySlot.new())
14
15
16 func load_save_data(save_data:Dictionary) -> void:
17     slots = save_data[SLOTS_PROPERTY]
18
19
20 func to_save_data() -> Dictionary:
21     var save_data : Dictionary = .to_save_data()
22     save_data[SLOTS_PROPERTY] = variant_to_save_data(slots)
23     return save_data
24
```

Game Data

The game data is the savable that contains all the stuff that will be saved and will be the main access point for other scripts. For now, it only contains the player inventory, but other data such as a list of flags, variables, or the state of the player character would be added to this.

```
1  extends Saveable
2  class_name GameData
3
4  const INVENTORY_PROPERTY : String = "Inventory"
5
6  var inventory : GameInventory
7
8  func _init() -> void:
9      inventory = GameInventory.new()
10
11
12 func load_save_data(save_data:Dictionary) -> void:
13     inventory = save_data[INVENTORY_PROPERTY]
14
15
16 func to_save_data() -> Dictionary:
17     var save_data : Dictionary = .to_save_data()
18
19     save_data[INVENTORY_PROPERTY] = inventory.to_save_data()
20
21     return save_data
```

Game Manager

The game manager is the singleton that creates and contains the game data and makes it accessible to all other scripts.

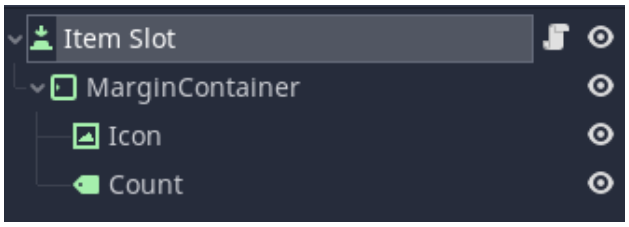
```
1  extends Node
2
3  var data : GameData
4
5  func _ready() -> void:
6      data = GameData.new()
7
```

Inventory Scene

The inventory scene contains the UI for the player inventory. It displays all the items in the player inventory and allows the player to drag and drop the items inside the inventory. When the player hovers over an item, a tooltip will also be shown displaying the name of the item.

Item Slot Scene

The item slot scene will be used for filling the inventory scene with item slots. Each item slot consists of a button that can be pressed, a margin container to keep the icon within the borders of the slot, a Texture Rect which displays the icon, and a Label, which displays the item count.




```gdscript
func _ready() -> void:
    _is_ready = true
    # warning-ignore:return_value_discarded
    get_slot_data().connect("content_changed", self, "refresh")
    refresh()


func get_slot_index() -> int:
    return _slot_index


func set_slot_index(value:int) -> void:
    _slot_index = value
    refresh()

func get_slot_data() -> GameInventorySlot:
    return GameManager.data.inventory.slots[_slot_index]


func get_item_data() -> ItemData:
    return get_slot_data().get_item_data()


func refresh() -> void:
    if _is_ready:
        var item_data : ItemData = get_item_data()
        if item_data == null:
            _button.disabled = true
            _icon_rect.texture = null
            _count_label.text = ""
            hint_tooltip = ""
        else:
            _button.disabled = false
            _icon_rect.texture = item_data.icon
            _count_label.text = String(get_slot_data().count)
            hint_tooltip = item_data.name
```

Drag and Drop

To make drag and drop possible for the inventory slots, the *get_drag_data*, *can_drop_data*, and *drop_data* methods need to be implemented. For user friendliness the drag preview is also set, so that the player knows what item they're currently dragging.

The *get_drag_data* method should return a value that is easy to identify in the *can_drop_data* method, to not confuse the dragged data with another value. However, first, if there is no item in the slot, null is returned, hence telling the engine that there is nothing to drag. As for the return value in this example, a dictionary containing a type property, which can be easily checked for, is used.

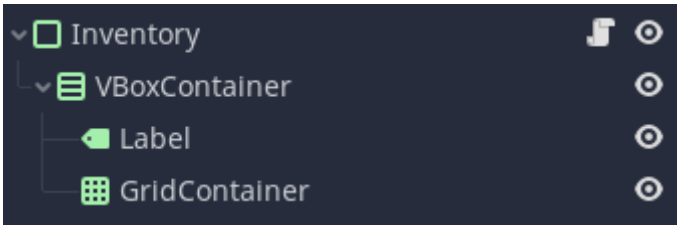
The *can_drop_data* method checks if a dragged value can be dropped into the given inventory slot. In this example we only allow the value to be dropped if it is a dictionary and if it contains the previously specified type.

Finally, the *drop_data* method happens when the player drops an item onto the slot and the drop action is accepted. It gets the slot data that was being dragged and the slot data of the dropped onto slot, then swaps their values and finally the dropped onto slot grabs the focus (the *grab_focus* can be ignored, but it feels smoother in my opinion).

```
59  # warning-ignore:unused_argument
60  func get_drag_data(position):
61      if get_item_data() == null:
62          return null
63      set_drag_preview(_create_drag_preview())
64      return {
65          DRAG_DATA_TYPE_PROPERTY: INVENTORY_SLOT_TYPE,
66          SLOT_PROPERTY: get_slot_data(),
67      }
68
69
70  func _create_drag_preview() -> Control:
71      var prefab : PackedScene = load(filename)
72      var drag_preview = prefab.instance()
73      drag_preview.slot_index = _slot_index
74      return drag_preview
75
76
77  # warning-ignore:unused_argument
78  func can_drop_data(position:Vector2, data) -> bool:
79      if typeof(data) != TYPE_DICTIONARY:
80          return data
81      var dictionary : Dictionary = data
82      if dictionary.get(DRAG_DATA_TYPE_PROPERTY) != INVENTORY_SLOT_TYPE:
83          return false
84      return true
85
86
87  # warning-ignore:unused_argument
88  func drop_data(position:Vector2, data) -> void:
89      var dragged_slot : GameInventorySlot = data.get(SLOT_PROPERTY)
90      var dropped_slot : GameInventorySlot = get_slot_data()
91      dropped_slot.swap_with_slot(dragged_slot)
92      grab_focus()
93
```

Inventory Scene

The inventory prefab will be used for displaying the player inventory. The root node will user a Panel Container with a Vertical Box Container, which contains Label to tell the player that this is an inventory (this could also be used for telling the player the name of the inventory), and a Grid Container with a column count of 6 to store the item slots.

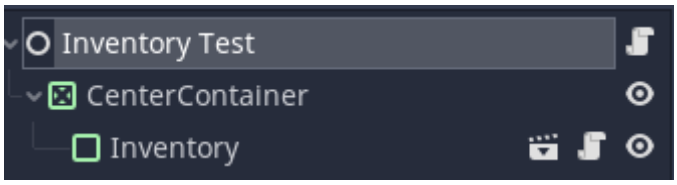


Right now, the main roll of the inventory scene is to fill the grid container with the item slots. This could be further expanded to make the inventory scene handle various inventories.

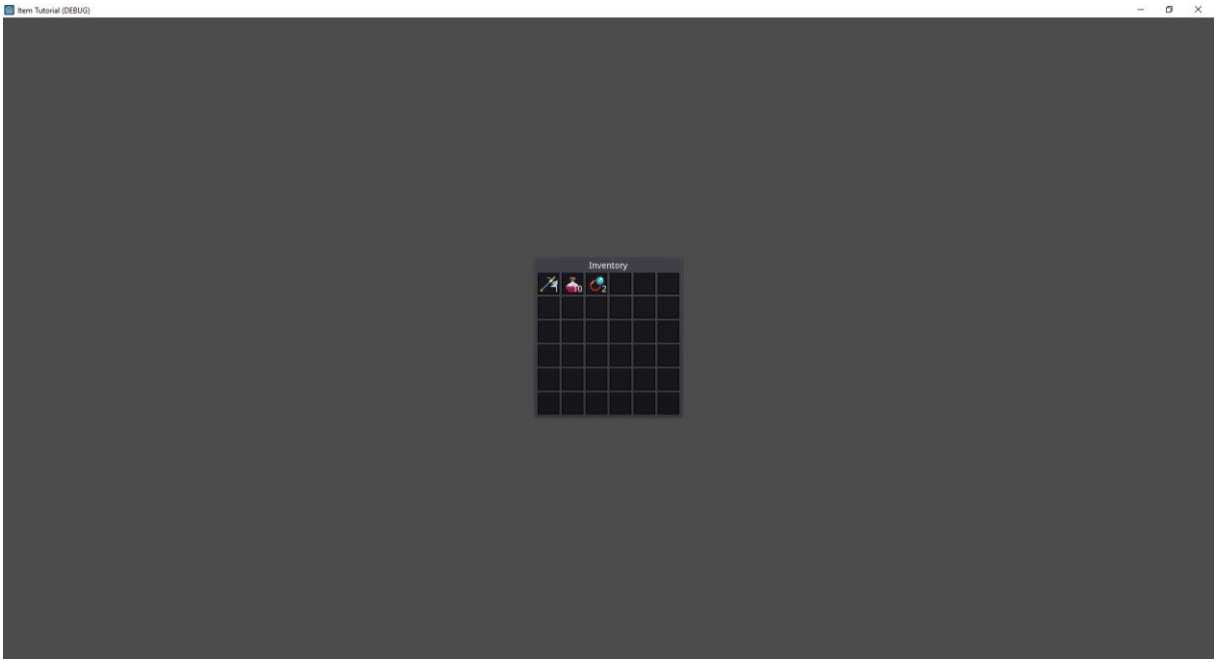
```
1  extends Control
2
3  const ITEM_SLOT_PREFAB : PackedScene = preload("res://scenes/menu/item_slot.tscn")
4
5  export(NodePath) var slot_parent_path : NodePath
6
7  onready var _slot_parent = get_node(slot_parent_path) as Control
8
9  func _ready() -> void:
10     refresh()
11
12
13 func get_slot_count() -> int:
14     return GameManager.data.inventory.slots.size()
15
16
17 func refresh() -> void:
18     _refill()
19
20
21 func _refill() -> void:
22     NodeUtil.queue_free_children(_slot_parent)
23     for i in range(get_slot_count()):
24         var item_slot = ITEM_SLOT_PREFAB.instance()
25         item_slot.slot_index = i
26         _slot_parent.add_child(item_slot)
27
28
```

Test

To test the inventory system, items were added in the item editor and a test scene was built, which fills the player inventory at startup and then places the inventory scene displaying the player inventory within the center of the screen.




```gdscript
extends Node

func _ready() -> void:
    fill_inventory_slot(0, "Weapons/Axe", 1)
    fill_inventory_slot(1, "Other/Potion", 10)
    fill_inventory_slot(2, "Other/Ring", 2)


func fill_inventory_slot(index:int, item_id:String, count:int) -> void:
    var slot : GameInventorySlot = GameManager.data.inventory.slots[index]
    slot.item_id = item_id
    slot.count = count
```



Further Improvements

This tutorial only gives a simple example of how an inventory system built upon data, which can be edited in a custom editor can be made. Still, it contains many scenes and a lot of code, each of which can be easily improved upon. Some of the code may even contain bugs, which simply haven't been found yet. Hence this tutorial can be seen more of a template or a starting point from which to continue work from. There are however a few specific improvements I could recommend adding to the project.

Item Types

If the game contains different item types with somewhat different behavior, it may be prudent to split it up. For instance, if the game contains items, weapons, and armor, a good idea may be to have 3 different editors for each of them with 3 different dictionaries in the data manager.

Item Count

Adding a maximum stack count for items will likely be needed to manager the game economy. Further it may prove useful to only display the count for items where there can be more than 1 item of the given type in the slot.

Improved Tooltip

A tooltip that displays more than the name of the item, such as an item's price, description, and its functions would also be helpful to players. To add a customized tooltip, the *_make_custom_tooltip* function will need to be implemented in the item slot script.

More Inventories

Another point previously mentioned would be to add more inventories. For instance, an additional storage where players can put their items. Or perhaps players can equip different bags, each being its own inventory. Another addition would be to make it possible for the player to create their own storages and allow for a seemingly infinite amount of them. Though do be mindful that each storage will need to be stored in the game data and the save files as well.

Load on Demand

Rather than having textures, such as the icon texture, permanently loaded in the RAM, it may be better to only have them saved as the paths to their files and load them when needed by the game. This may not be necessary for images such as icons, but it may prove useful for big image files like backgrounds.