gs-ecs

Release 3.2

Oct 23, 2020

General

Tip: This is documentation for the development (master) branch. Looking for documentation of the current stable branch? Have a look here.

Welcome to the official documentation of the gs-ecs library. A lightweight Entity Component System for the Godot Engine.

The table of contents below and in the sidebar should let you easily access the documentation for your topic of interest. You can also use the search function in the top left corner.

Note: gs-ecs is an open source project.

Submit an issue or pull request on the GitLab repository.

CHAPTER 1

About

This is a **Framework** for adding a simple **Entity Component System** using **Godot**.

The **Framework** puts less emphasis on performance, and instead tries to focus on improving **Workflow** and **Code Reuse**.

CHAPTER 2

Installation

This library requires that two additional libraries be added to your Godot project. The first is the *gs-ecs* library itself. The second, is a logging system called *gs-logging*.

Note: In the instructions below, pick the BRANCH that is most appropriate for your particular project. For example, if you are working on a 3.1.1 project, pick the 3.1 branch. For a 3.0 project, use the 3.0 branch and so on.

2.1 Manual Installation

- download the package from https://gitlab.com/godot-stuff/gs-ecs
- unzip the folder gs_ecs into your project
- download the package from https://gitlab.com/godot-stuff/gs-logger
- unzip the folder gs_logger into your project

2.2 Installation With gs-project-manager

- add the following entry in the *assets:* section of your **project.yml**

```
gs-ecs:
  location: https://gitlab.com/godot-stuff/gs-ecs.git
  branch: 3.2
  includes:
    - dir: gs_ecs

gs-logger:
  location: https://gitlab.com/godot-stuff/gs-logger.git
  branch: 3.2
```

(continues on next page)

(continued from previous page)

```
includes:
  - dir: gs_ecs
```

2.3 Installation Using Asset Library

- open up the Asset Library from Godot and search for godot-stuff
- Download and Install the ECS and Logger libraries

2.4 Activate Plugin

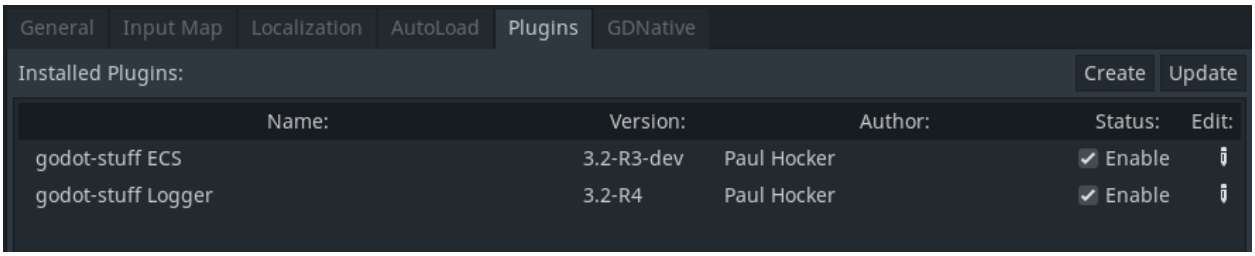
- after it is installed open your Project Settings and Activate the Plugin

CHAPTER 3

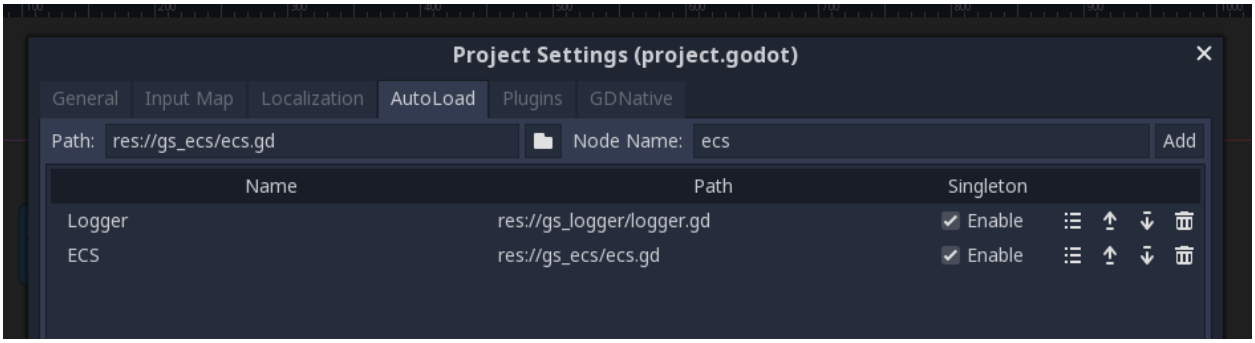
Activation

The main ECS library, along with it's supporting library Logger, must be activated to make sure that the proper Autoload objects are included in your Project.

- Open your Project Settings and open the Plugins tab
- Activate the Logger and ECS Plugins



After that is complete, you should see something similar to this in your Autoload



Note: The Logger autoload needs to come before the ECS autoload in order to work properly.

CHAPTER 4

Simple Example

4.1 Overview

This Tutorial will show you some of the basics of setting up and using the ECS framework. In this simple example, we will be creating a Sprite that will move across the screen.

4.2 Project Setup

You can download `gs-ecs-simple.zip` a startup project to get you started. It contains the ECS framework, and the Logging system it needs. The startup project was created with Godot 3.2, but anything after that should work as well, with the exception of 4.0

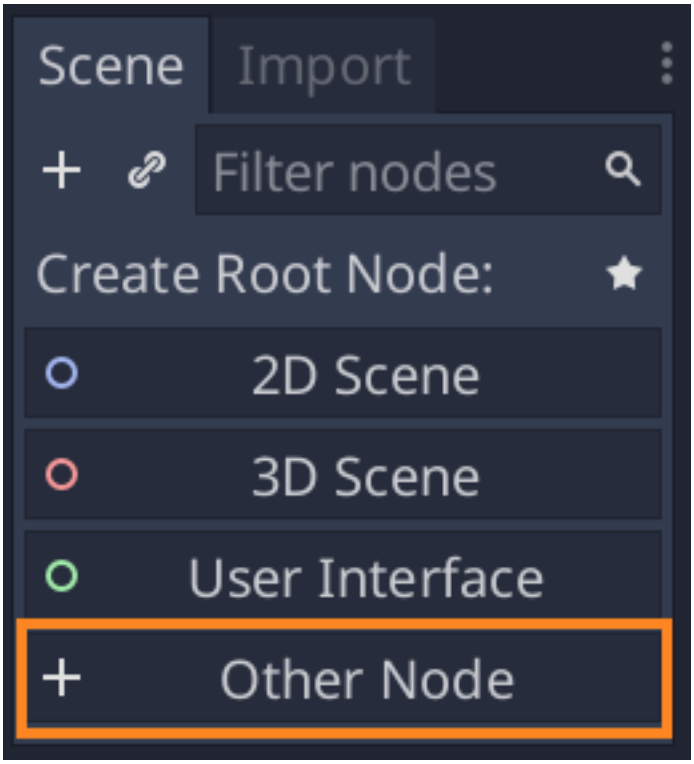
You may also clone the project and open it that way.

```
$ git clone https://gitlab.com/godot-stuff/gs-ecs-simple.git
```

Note: Godot 4.0 was still in early development when this Tutorial was written. It will be updated when it is released and the ECS has been modified to work with the new version of Godot.

4.3 Entity

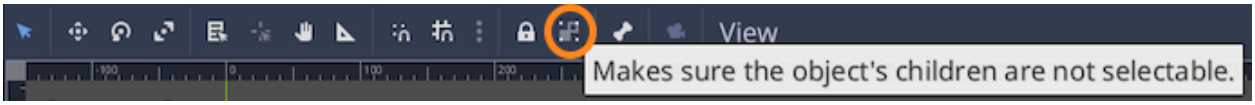
The first thing we are going to do is add an Entity to the project. We need to choose a root node for the player object. As a general rule, a scene's root node should reflect the object's desired functionality - what the object *is*. Click the "Other Node" button and add an Area2D node to the scene.



Godot will display a warning icon next to the node in the scene tree. You can ignore it for now. We will address it later.

With `Area2D` we can detect objects that overlap or run into the player. Change the node's name to `Entity` by double-clicking on it. Now that we've set the scene's root node, we can add additional nodes to give it more functionality.

Before we add any children to the `Entity` node, we want to make sure we don't accidentally move or resize them by clicking on them. Select the node and click the icon to the right of the lock; its tooltip says "Makes sure the object's children are not selectable."



Let's add the `entity.gd` script to the Node we just created. This script contains some boilerplate code to add this scene as an entity. Look here for more information on *Entity*.

Save the scene and call it `entity.tscn`. Click Scene -> Save, or press `Ctrl + S` on Windows/Linux or `Cmd + S` on macOS.

With that, let's run the scene and see what shows up in the log. Press `F6` to run the scene, and your log should look similar to this.

```
** Console Appender Initialized **

TRACE       1         [entity] _init
TRACE       2         [entity] on_init
TRACE       3         [entity] _ready
TRACE       4         [ECS] add_entity
TRACE       5         [ECS] has_entity
TRACE       6         [entity] on_before_add
```

(continues on next page)

(continued from previous page)

```
DEBUG       7         - entity [Area2D:1187]:Entity has been added
TRACE       8         [entity] on_after_add
WARN        9         No Components found
TRACE       10        [entity] on_ready
TRACE       11        [ECS] _exit_tree
```

The key message to look for is that the Area2D has been added to the framework.

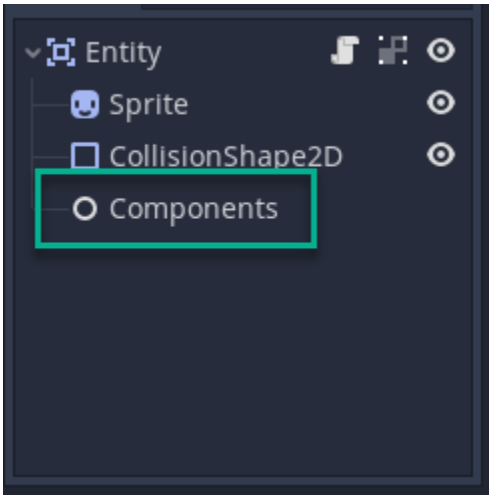
Let's add a component to our Entity now.

4.4 Component

The data for your Entities are stored in Components. In the ECS Framework, we usually create Components visually by creating an empty node, and adding a script that subclasses the `component.gd` script.

To prepare our Entity to find the components we add, we need to create an empty Node called `Components` first, and then you can add your components to that node. When your project starts, any Component attached to that parent Node will be added as Component.

This is how your Entity should look.



Now let's create a new Scene, then add a new Node and rename it to `Velocity`.

Save this scene with the name "velocity.tscn", and then attach a new Script to the Node and call it "velocity.gd".

Velocity needs both a direction, and speed so lets modify our Script to include those, and expose them as parameters for the Component itself.

```
class_name Velocity
extends Component

export var speed : float = 100.0
export var direction : Vector2 = Vector2.RIGHT
```

So now we have a component that will travel at 100 pixels/second in the Right direction. Notice that there is no game logic at all in this Code. That will come later when we talk about Systems.

Save the Velocity Component, and then open up the Entity Scene again.

Now you can drag the Velocity Component you just created onto the "Components" Node in your Entity.

Now when we press F6 to run the Entity Scene again, the output should now show us that a new Component called "velocity" was registered and added to the entity.

```
** Console Appender Initialized **

TRACE      1        [entity] _init
TRACE      2        [entity] on_init
TRACE      3        [entity] _ready
TRACE      4        [ECS] add_entity
TRACE      5        [ECS] has_entity
TRACE      6        [entity] on_before_add
DEBUG      7        - entity [Area2D:1192]:Entity has been added
TRACE      8        [entity] on_after_add
TRACE      9        [ECS] entity_add_component
TRACE      10       [ECS] has_entity
TRACE      11       [ECS] has_component
TRACE      12       [ECS] add_component
DEBUG      13       - new component velocity was registered
DEBUG      14       - added velocity component for entity [Area2D:1192]:Entity
TRACE      15       [entity] on_ready
TRACE      16       [ECS] _exit_tree
```

4.5 System

Systems are the parts of your game that contain the logic to make things go. For our example, we want to create a system that will Move anything that has a Velocity Component attached to it.

To do this, let's create a new Scene and add a Node, and call it "VelocitySystem". Save the Scene as "velocity_system.tscn", and then attach a new Script called "velocity_system.gd" to it. Replace the Code for the Script with this.

```
class_name VelocitySystem
extends System

func on_process_entity(entity : Entity, delta: float):
    var _component = entity.get_component("velocity") as Velocity
    entity.position += _component.direction * _component.speed * delta
```

Because we are subclassing the System class, we get some inherited methods. The one shown here is called "on_process_entity". This method is called for every Entity registered that has a Component matching the list of Components for a System.

The other two important areas in this example are the reference to the Velocity component using the get_component method on the Entity Scene. When you use this method, it pulls the reference of that Component from memory in the ECS framework, not from your Scene itself. It then uses those values to calculate the new position of the Entity. You can update values in a Component, just like you would any other Script.

Now when we press F6 to run this Scene, we should get output similar to the following.

```
** Console Appender Initialized **

TRACE      1        [ECS] add_system
TRACE      2        [ECS] has_system
TRACE      3        [system] on_before_add
```

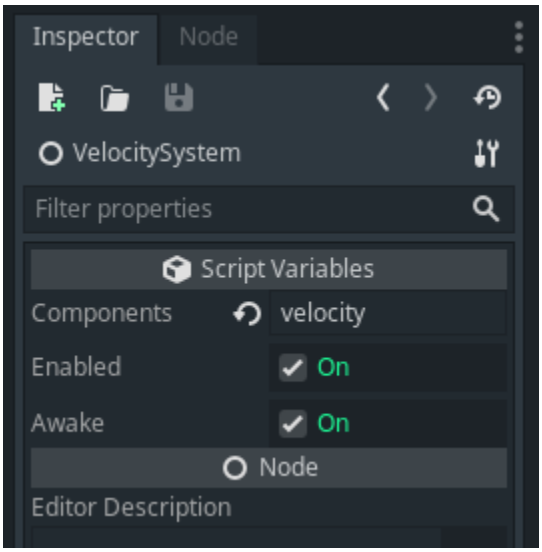
(continues on next page)

(continued from previous page)

```
DEBUG       4         - system velocitysystem has been added
TRACE       5         [system] on_after_added
TRACE       6         [ECS] _exit_tree
```

Before we go to the next step, let's make sure that we mark this System to handle all Entities that have a Velocity Component.

To do this, Select the Root node of your System, and you will notice that there are a few properties. The first, called Components, lets you enter in a list of Component names, separated by commas, that will be processed by this System.

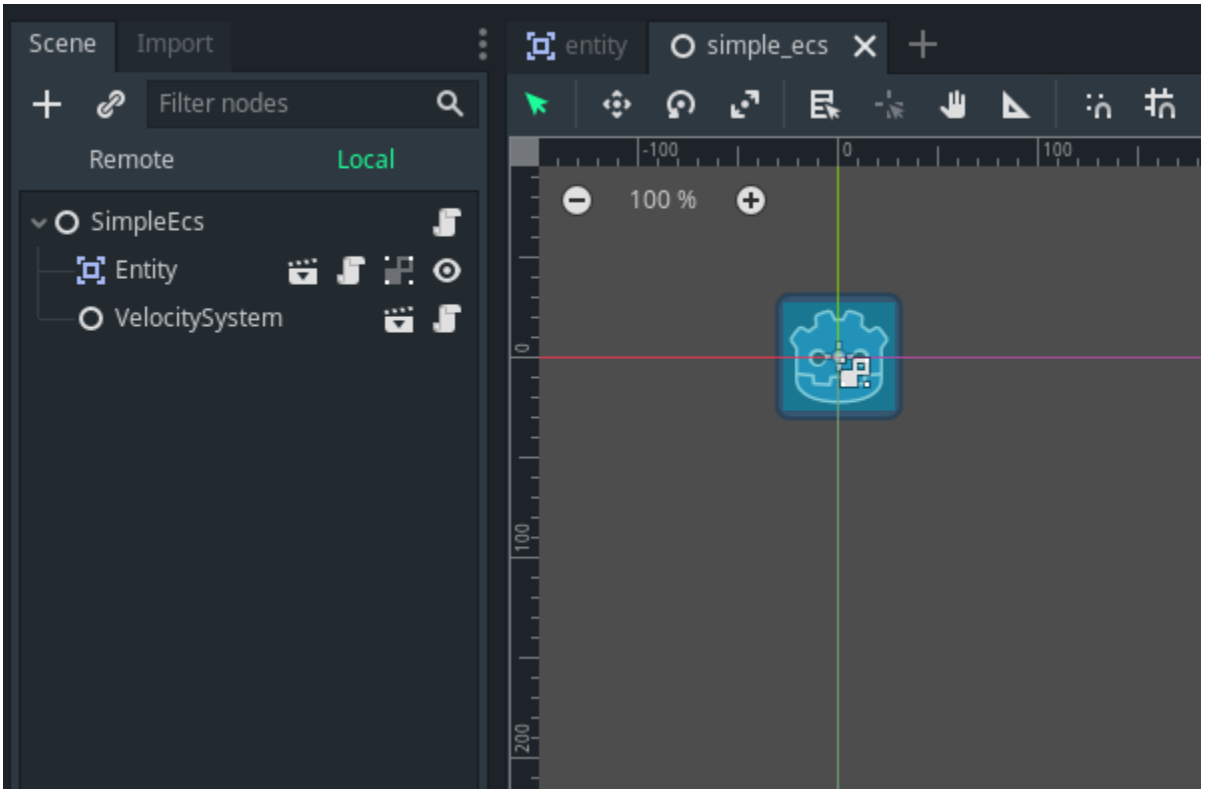


The next two properties, should be left to the default of "enabled". These two properties are used to Activate the System, and force the System to start running immediately when the Scene starts.

4.6 Putting It All Together

Now it is time to put everything together.

Create a new Scene with a new Node called "SimpleEcs". Now, drag the Entity and VelocitySystem Scenes onto your new Scene. It should look like this when you are done.



Save your Scene as "simple_ecs.tscn" and then add a new Script called "simple_ecs.gd" to it. Replace the code with this instead.

```
extends Node

func _process(delta):
    ECS.update()
```

Save your Scene, and press F6 to run it. You should see the Godot sprite move across the Scene. That's it, really it is that simple. Calling the update function will take care of everything.

4.7 Congratulations

Awesome! You now know some of the key concepts on how to use this simple ECS Framework for the Godot game engine. We are working on more Tutorials and Guides to help you get started, and to show you more advanced techniques when working with it.

Download some of the Demos available to see what else you can do with this simple ECS Framework.

CHAPTER 5

Singletons

5.1 ECS

Inherits: Node

A singleton for managing the simple Entity Component System for Godot.

5.1.1 Description

This framework does not try to add a lot of functionality, instead it tries to provide a low level API for managing objects for an ECS like environment.

Inherits: Node

5.1.2 Properties

Dictionary	*entities*
Dictionary	*component_entities*
Dictionary	*systems*
Dictionary	*system_components*
Dictionary	*system_entities*
Dictionary	*groups*
Dictionary	*group_systems*
Dictionary	*entity_remove_queue*
bool	*is_dirty*

5.1.3 Methods

void	*add_component* (*Component* component)
void	*add_entity* (*Entity* entity)
void	*add_group* (*Group* group)
void	*add_system* (*System* system)
void	*clean* ( )
void	*entity_add_component* (*Entity* entity, *Component* component)
Component	*entity_get_component* (*Entity* entity, String name)
bool	*entity_has_component* (*Entity* entity, String name)
void	*entity_remove_component* (*Entity* entity, String name)
Dictionary	*get_all_components* ( )
Component	*get_component* (String name)
bool	*has_component* (String name)
bool	*has_entity* (String name)
bool	*has_group* (String name)
bool	*has_system* (String name)
void	*rebuild* ( )
void	*remove_component* (String name)
void	*remove_entity* (String name)
void	*remove_group* (String name)
void	*remove_system* (String name)
void	*update* (String group = `null`, float delta = `null`)

5.1.4 Property Descriptions

- Dictionary **entities**

A list containing all the entities that have been registered.

- Dictionary **component_entities**

A list containing all the entities with a component that have been registered.

- Dictionary **systems**

A list containing all the systems that have been registered.

- Dictionary **system_components**

A list containing all the components in a system that have been registered.

- Dictionary **system_entities**

A list containing all the entities in a system.

- Dictionary **groups**

A list containing all the groups in the framework.

- Dictionary **group_systems**

A list of systems in a group.

- Dictionary **entity_remove_queue**

A list of entities to remove after all systems have completed running.

- bool **is_dirty**

Returns `true` when the framework needs to be cleaned up. This usually occurs when any objects have been added or removed in the framework. You can set this property to true at any time to force a rebuild of the systems.

5.1.5 Method Descriptions

- void **add_component** (*Component* component)

Adds a *Component* to the framework. If the component already exists, it will not be added again.

- void **add_entity** (*Entity* entity)

Adds an *Entity* to the framework. If the entity has already been added, it will not be added again.

- void **add_group** (*Group* system)

Adds a *Group* to the framework. If the group has already been added, it will not be added again.

- void **add_system** (*System* system)

Adds a *System* to the framework. If the system has already been added, it will not be added again.

- void **clean** ( )

Removes all odbjects from the framework.

- void **entity_add_component** (*Entity* entity, *Component* component)

For a given *Entity*, add a *Component* to it.

- *Component* **entity_get_component** (*Entity* entity, String name)

For a given *Entity*, return a *Component* from it based on the name given.

- bool **entity_has_component** (*Entity* entity, String name)

For a given *Entity*, returns `true` if the named component is available.

- bool **entity_remove_component** (*Entity* entity, String name)

For a given *Entity*, remove the named component.

- Dictionary **get_all_components** ( )

Returns all components in the framework.

- *Component* **get_component** (String name)

Returns a *Component* from the framework in `name`

- bool **has_component** (String name)

Returns `true` if the *Component* in `name` exists.

- bool **has_entity** (String name)

Returns `true` if the *Entity* in `name` exists.

- bool **has_group** (String name)

Returns `true` if the *Group* in `name` exists.

- bool **has_system** (String name)

Returns `true` if the *System* in `name` exists.

- bool **rebuild** ( )

Rebuild the *system_entities*.

- void **remove_component** (String name)

Rmoves the *Component* from the framework in `name`.

- void **remove_entity** (String name)

Rmoves the *Entity* from the framework in `name`.

- void **remove_group** (String name)

Rmoves the *Group* from the framework in `name`.

- void **remove_system** (String name)

Rmoves the *System* from the framework in `name`.

- bool **update** (String group = `null`, float delta = `null`)

Update the *systems* in the framework. A *Group* may be specified in addition to a `delta`.

CHAPTER 6

Classes

This library comes with a number of Helper Classes that can be used with Nodes in Godot to make designing your ECS Scenes more visually. The Tutorials will help you get more familiar with the Nodes.

6.1 Entity

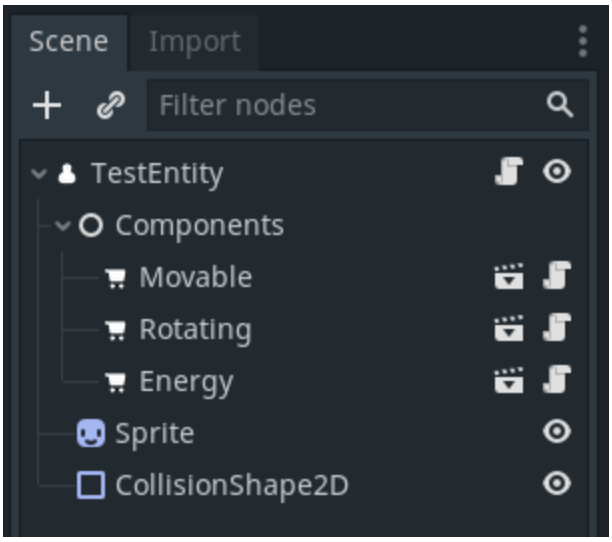
Inherits: Node

Entities represent the **things** in your game. It does not have any behavior or data; it does contain components and nodes which are needed for the entity. Systems provide the behavior, while Components contain the data.

6.1.1 Description

An entity is essentially a Node in Godot. In fact it is a base class of a Godot Node.

In the example below the Entity called **TestEntity** has three **Components** attached to it called **Movable**, **Rotating** and **Energy** respectively. The node called "Components" is used to keep the structure of the Entity node clean and gives a good visual representation for the developer.



6.1.2 Properties

int	*id*
bool	*enabled*

6.1.3 Methods

void	*on_init* ()
void	*on_ready* ()
void	*on_before_add* ()
void	*on_after_add* ()
void	*on_before_remove* ()
void	*on_after_remove* ()
void	*on_enter_tree* ()
void	*on_exit_tree* ()
void	*add_component* (*Component* component)
Component	*get_component* (String name)
bool	*has_component* (String name)
void	*remove_component* (String name)

6.1.4 Property Descriptions

- int **id**

The internal identifier for the Entity. This is essentially the same as the Godot object id.

- bool **enabled**

Lets you enable the entity by setting this to True. When an Entity is not enabled when its setting is False, it will not be included during System updates.

6.1.5 Method Descriptions

- void **on_init** ()

The framework will make a virtual call when the Entity is initialized. Use this in place of _init().

- void **on_ready** ()

The framework will make this virtual call when the Entity is ready. Use this in place of _ready().

- void **on_before_add** ()

The framework will make this virtual call when the Entity immediately before it is added to the framework.

- void **on_after_add** ()

The framework will make this virtual call when the Entity immediately after it has been added to the framework.

- void **on_after_remove** ()

The framework will make this virtual call just after the Entity is removed from memory.

- void **on_before_remove** ()

The framework will make this virtual call just before the Entity is removed from memory.

- void **on_enter_tree** ()

The framework will make this virtual call when the Entity enters the Tree. Use this in place of _enter_tree().

- void **on_exit_tree** ()

The framework will make this virtual call when the Entity exits the Tree. Use this in place of _exit_tree().

- void **add_component** (*Component* component)

Adds another *Component* to this Entity. If the *Component* has already been added it will not be added again.

- void **get_component** (String name)

Returns the *Component* based on the Name that it is given. If the *Component* is not found it will return null and a Warning will be logged.

- bool **has_component** (String name)

Returns True if the *Component* *name* does exist for the Entity, otherwise it will return False.

- bool **remove_component** (String name)

Removes the *Component* in *name*. If the *Component* is not found it will log a Warning.

6.2 Component

Inherits: Node

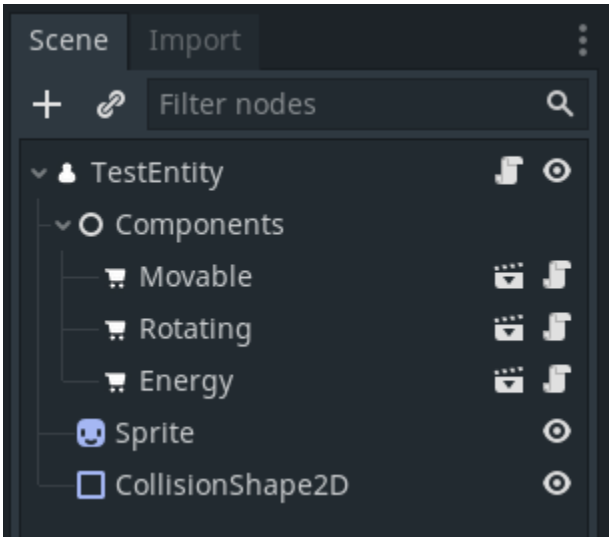
A Component contains the Data for an Entity in the ECS Framework.

6.2.1 Description

An entity is essentially a Node in Godot. In fact it is a base class of a Godot Node. In this **Framework** a Component is nothing more than a Script that is attached to a Node. The Script defines the data for the Component.

Components can be added to an Entity programatically or by using the Node Structure of an Entity by attaching any Component Node in the Tree.

In the example below the Entity called **TestEntity** has three **Components** attached to it called **Movable**, **Rotating** and **Energy** respectively. The node called "Components" is used to keep the structure of the Entity node clean and gives a good visual representation for the developer.



6.2.2 Methods

6.2.3 Method Descriptions

- void **on_init** ()

The framework will make a virtual call when the Entity is initialized. Use this in place of _init().

- void **on_ready** ()

The framework will make this virtual call when the Entity is ready. Use this in place of _ready().

6.3 System

Inherits: Node

A *System* is used to make the **things** in your Game go.

6.3.1 Description

A *System* will process entities that contain, or do not contain, a certain type of *Component*.

For example, say your game needs to be able to move an *Entity* across the scene. You would need some kind of *Component* that stores the direction and speed of an Entity, let's call it "Velocity". To move the *Entity*, we would create a *System* that would only run over Entities that have the "velocity" *Component*, and then calculate the velocity.

You might then create another *System* that uses Entities that have a "velocity" *Component* to move them to the next position in the Scene based on the Velocity that was previously calculated, and so on.

You can make your *System* as simple or complicated as needed, but we have found that breaking up your game into smaller, logical domains or behaviors makes development of your game easier, without any type of large performance hit.

Note: In some cases it does not make much sense to use a *System*. If you are, for example, creating a bullet-hell type of game, it might be more efficient to use the standard Godot processing loop to move those. However, the Collision Checking and Spawner for the Bullets might be created with a *System*. The Framework is flexible enough to let you decide the best place to use it.

Note: We suggest that you check out the simple example project to get a better understanding of how everything works together. It should help you understand where a *System* fits into the Framework.

6.3.2 Properties

bool	*enabled*
String	*components*

6.3.3 Methods

void	*on_init* ()
void	*on_ready* ()
void	*on_before_add* ()
void	*on_after_add* ()
void	*on_before_remove* ()
void	*on_after_remove* ()

6.3.4 Property Descriptions

- bool **enabled**

Lets you enable the entity by setting this to True. When an Entity is not enabled when its setting is False, it will not be included during System updates.

- String **components**

A very simple comma separated list of *Component* names to be processed when the updated.

The syntax is

{component}	any *Component* matching this name
!{component}	any *Component* not matching this name

In the example below, the *System* will process all Entities that have a Velocity and Move *Component*, but not an Enemy component.

```
velocity, move, !enemy
```

You can treat each comma as an AND statement for each *Component*. So another way to read the above is

```
**velocity** and *movement** and not **enemy**
```

6.3.5 Method Descriptions

- void **on_init** ()

The framework provides this in place of the _init() call.

- void **on_ready** ()

The framework provides this in place of the _ready() virtual call.

- void **on_before_add** ()

The framework will make this virtual call just before the *System* is registered.

- void **on_after_add** ()

The framework will make this virtual call just after the *System* has been registered.

- void **on_after_remove** ()

The framework will make this virtual call just after the *System* is removed.

- void **on_before_remove** ()

The framework will make this virtual call just before the *System* is removed.

6.4 Group

Inherits: Node

A Group is a way to organize *Systems*.

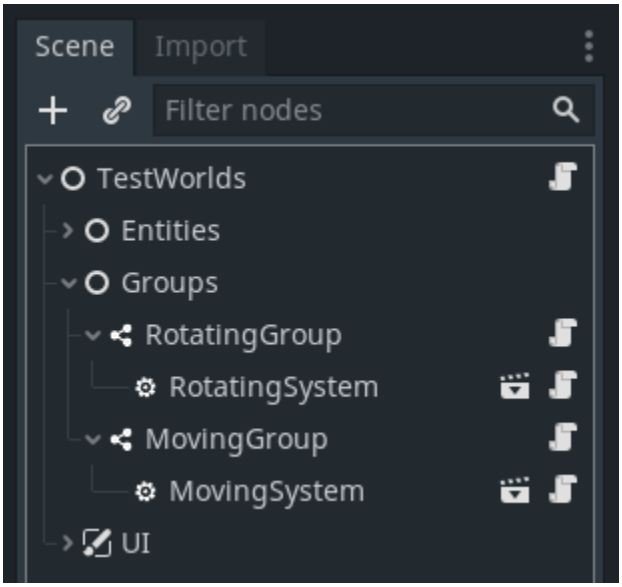
6.4.1 Description

Sometimes it is very useful to put similar Systems together in a Group so that they can be processed or not processed together. This is sometimes more simple than removing a *Component* from an Entity you want to control processing on.

In other ECS frameworks, this is sometimes refered to as a World.

6.4.2 How It Works

To use it, you simply add a Group node to your Scene, and attach Systems to that Node. The example below shows you how that might look.



In the Scene we have two Groups named **RotatingGroup** and **MovingGroup**, and each has a System attached to it respectively. When your game now runs, you can choose to *update* all Systems, or you can control which Systems to process based upon the Group they are in.

Then, when we want to *update* only one specific *Group* of *Systems* we can do this

```
func _process(delta):
    ECS.update('rotatingsystem')
```