encompass Documentation

Release 1.0

Evan Hemsley

Jun 19, 2019

Contents

encompass is a powerful engine-agnostic Hyper ECS framework ideal for game development.

CHAPTER 1

API Reference

1.1 WorldBuilder

```
import { WorldBuilder } from "encompass-ecs";
const world_builder = new WorldBuilder();
```

A WorldBuilder is used to build a World out of Engines and Renderers.

The WorldBuilder enforces certain rules about Engine structure. It is forbidden to have messages create cycles between Engines, and no Component may be mutated by more than one Engine.

The WorldBuilder uses Engines and their Message read/emit information to determine a valid ordering of the Engines, which is given to the World.

The WorldBuilder is also responsible for creating the World with an initial state of Entities, Components, and Messages.

It is a mistake to instantiate elements of Encompass directly except for WorldBuilder. These elements should be created either through appropriate WorldBuilder functions, or by Engines.

1.1.1 Functions

`create_entity`()

> **Returns** An instance of Entity.

`emit_message`(*MessageType*)

This function creates a new Message instance and makes it available to be read by other engines.

> **Arguments**
>
> - **`MessageType`** (`Type<Message>`) – A constructor reference to a subtype of Message.
>
> **Returns Message** An instance of Message of the given type.

`emit_component_message`(*ComponentMessageType*, *component*)
This function creates a new ComponentMessage instance and makes it available to be read by other engines.

Arguments

- **`ComponentMessageType`** (*`Type<ComponentMessage>`*) – A constructor reference for a subtype of ComponentMessage.
- **`component`** (*`Component`*) – An instance of a component.

Returns ComponentMessage An instance of ComponentMessage of the given type.

`emit_entity_message`(*EntityMessageType*, *entity*)
This function creates a new EntityMessage instance and makes it available to be read by other engines.

Arguments

- **`EntityMessageType`** (*`Type<EntityMessage>`*) – A constructor reference for a subtype of EntityMessage.
- **`entity`** (*`Entity`*) – An instance of Entity.

Returns EntityMessage An instance of EntityMessage of the given type.

`add_engine`(*EngineType*)

Arguments

- **`EngineType`** (*`Type<Engine>`*) – A constructor reference to a subtype of Engine.

Returns Engine An instantiated Engine of the given type.

`add_renderer`(*RendererType*)

Arguments

- **`RendererType`** (*`Type<Renderer>`*) – A constructor reference to a subtype of Renderer.

Returns Renderer An instantiated Renderer of the given type.

`build`()
Call this function when you are done adding engines.

Throws

- **`EngineCycleError`** – When messages between Engines create a cycle.
- **`EngineMutationConflictError`** – When two different Engines mutate the same Component.

Returns World An instantiated World.

1.2 World

```
import { WorldBuilder } from "encompass-ecs";
const world_builder = new WorldBuilder();
const world = world_builder.build();
```

A World is the glue that holds all the framework elements together.

The World's `update` function drives the simulation and should be controlled from your game engine's update loop.

1.2.1 Functions

`update` (*dt*)

Updates the simulation based on given delta time, advancing the simulation by one frame. This involves destroying marked Entities, updating Engines, and cleaning up the Messages at the end of the frame.

Arguments

- **`dt`** (*`number`*) – Delta time.

`draw` ()

Calls the render methods of all Renderers in the World, ordered by the layers on tracked DrawComponents and GeneralRenderers.

1.3 Entity

```
import { World } from "encompass-ecs";
const world = new World();
const entity = world.create_entity();
```

An Entity is composed of a unique internal ID and a collection of *Components*.

Entities do not have any implicit properties or behaviors but are granted these by their collection of Components.

There is no limit to the amount of Components an Entity may have, and Entities can have any number of Components of a particular type.

Entities are *active* by default and can be deactivated. While deactivated, Entities are still tracked by Engines, but are temporarily ignored. Note that the activation status of an Entity is independent of the activation status of its Components.

Entities typically are either instantiated at load time, or at runtime by a *Spawner*.

Defining logic on an Entity is an anti-pattern.

1.3.1 Functions

`add_component` (*ComponentType*, *properties?*)

Instantiates a Component of the given subtype and adds it to the Entity.

NOTE: Using the `properties` argument creates garbage. It may be better to set the component properties after instantiating the component.

Arguments

- **`ComponentType`** (*`Type<Component>`*) – A reference to a constructor for a subtype of Component.
- **`properties?`** (*`ComponentUserDataOnly<Component>`*) – An optional reference to an object containing properties of the Component subtype.

Returns Component An instantiated Component of the given type.

`get_components` (*ComponentType*)

Gets all components of the given Component type that belong to the Entity. Note that this does not include subtypes of the given type. Only includes active components by default.

Arguments

- **`ComponentType`** (*`Type<Component>`*) – A reference to a constructor for a subtype of Component.

Returns GCOptimizedList<Component> A list of active components of the given type.

`get_component`(*ComponentType*)

A convenience method for the common case of having a single component of a particular Type on an Entity. Note that if more than one component exists on the Entity, only the first one is returned. Be careful. Only checks active components.

Arguments

- **`ComponentType`** (*`Type<Component>`*) – A reference to a constructor for a subtype of Component.

Returns Component A Component of the given type, or null if none exist.

`get_components_including_inactive`(*ComponentType*)

Arguments

- **`ComponentType`** (*`Type<Component>`*) – A reference to a constructor for a subtype of Component.

Returns GCOptimizedList<ComponentType> A list of components of the given type.

`has_component`(*ComponentType*)

Checks if the Entity has a particular Component type. Only checks active components by default.

Arguments

- **`ComponentType`** (*`Type<Component>`*) – A reference to a constructor for a subtype of Component.

Returns boolean True if the Entity has a Component of the given type, otherwise false.

`has_component_including_inactive`(*ComponentType*)

Arguments

- **`ComponentType`** (*`Type<Component>`*) – A reference to a constructor for a subtype of Component.

Returns boolean True if the Entity has a Component of the given type including inactive components, otherwise false.

`remove_component`(*component*)

Removes a specific Component instance from the entity.

Arguments

- **`component`** (*`Component`*) – A Component.

`activate_component`(*component*)

Activates the component, making it tracked by engines. Does nothing if the component is already active.

NOTE: Components are active by default when added to an entity.

Arguments

- **`component`** (*`Component`*) – The component to deactivate.

`deactivate_component`(*component*)

Deactivates the component, making it untracked by engines. Does nothing if the component is already inactive.

Arguments

- **component** (*Component*) – The component to activate.

destroy()

Marks the Entity to be destroyed.

NOTE: The Entity will be destroyed at the end of the World `update`.

1.4 Components

1.4.1 Component

```
import { Component } from "encompass-ecs";
```

A Component can be thought of as a collection of data which is attached to an *Entity*.

Components are *active* by default and can be deactivated by the Entity they belong to. While inactive, Entities containing the Component will not be tracked by *Detectors* that track the Component type. This is useful for situations where you may want to temporarily disable a component without destroying information.

Note that the activation status of a Component is unrelated to the activation status of its Entity.

Components should not contain any game logic, but certain side-effect callbacks are available. For example, you could use callbacks to create or destroy bodies in the engine's physics system.

Defining non-side-effect logic on a Component is an anti-pattern.

Callbacks

`Component.`**on_initialize**()

Runs when the Component is added to an Entity. Deprecated.

`Component.`**on_activate**()

Runs when the Component is activated.

`Component.`**on_deactivate**()

Runs when the Component is deactivated.

`Component.`**on_destroy**()

Runs when the Component is removed from an Entity, or when its Entity is destroyed.

1.4.2 DrawComponent

```
import { DrawComponent } from "encompass-ecs";
```

A DrawComponent is a special subtype of *Component*.

The only difference is that it implicitly contains a `layer` property so it can be ordered properly by the World `draw` function.

NOTE: It is expensive to modify the `layer` property at runtime.

Properties

layer

The layer at which the DrawComponent will be drawn. Lower numbers mean it is drawn earlier.

Example

```
import { DrawComponent } from "encompass-ecs";

export class CanvasComponent extends DrawComponent {
    public canvas: Canvas;
    public w: number;
    public h: number;
}
```

```
import { CanvasComponent } from "src/components/draw/canvas";

const canvas_component = entity.add_component(CanvasComponent);
canvas_component.layer = 3;
canvas_component.w = 1280;
canvas_component.h = 720;
```

1.5 Messages

```
import { Message } from "encompass-ecs";
```

Similar to *Components*, Messages are collections of data.

Messages are used to transmit data between Engines so they can manipulate the game state accordingly.

Unlike Components, Messages are temporary and are destroyed at the end of each frame.

For performance reasons, it is discouraged to create new objects to pass to messages, as this will cause garbage collection pressure.

You should not define default property values on Messages because Messages are reused internally for memory usage reasons.

Defining any logic on a Message is an anti-pattern.

1.5.1 Example

```
import { Message } from "encompass-ecs";

class MotionMessage extends Message {
    public component: PositionComponent;
    public x_delta: number;
    public y_delta: number;
}
```

1.5.2 Special Types

ComponentMessage

```
import { ComponentMessage } from "encompass-ecs";
```

A ComponentMessage is a kind of message which has a `component` property.

Properties

`Component component`

A reference to an instance of Component.

Example

```
import { ComponentMessage, Message } from "encompass-ecs";

class MotionMessage extends Message implements ComponentMessage {
    public component: PositionComponent;
    public x_delta: number;
    public y_delta: number;
}
```

EntityMessage

```
import { EntityMessage } from "encompass-ecs";
```

An EntityMessage is a kind of message which has an `entity` property.

Properties

`Entity entity`

A reference to an instance of Entity.

Example

```
import { EntityMessage, Message } from "encompass-ecs";

class RemoveComponentMessage extends Message implements EntityMessage {
    public entity: Entity;
    public component_to_remove: Component;
}
```

1.6 Engine

```
import { Engine } from "encompass-ecs";
```

An Engine is the **encompass** notion of an ECS System. Engines are responsible for reading the game state and emitting messages, as well as mutating Entities and Components.

A particular type of Component may be modified by only one Engine. If you have two Engines which mutate the same Component type, an error will be thrown.

An Engine which Reads a particular Message is guaranteed to run *after* all Engines which Emit that particular Message.

1.6.1 Decorators

@Emits(*...message_type_args*)

Writes MessageTypes to the *emit_message_types* property.

Arguments

- **message_type_args** (*Type<Message>[]*) – MessageTypes which are emitted by the Engine.

@Reads(*...message_type_args*)

Writes MessageTypes to the *read_message_types* property.

Arguments

- **message_type_args** (*Type<Message>[]*) – MessageTypes which are read by the Engine.

@Mutates(*...component_type_args*)

Writes ComponentTypes to the *mutate_component_types* property.

Arguments

- **component_type_args** (*Type<Component>[]*) – ComponentTypes which are mutated by the Engine.

@Detects(*...component_type_args*)

Specifies that the given component types should be tracked by the Detector.

Arguments

- **component_type_args** (*Type<Component>[]*) – ComponentTypes which should cause the Detector to track an Entity.

1.6.2 Abstracts

update(*dt*)

Called by the World each frame. Place all of your logic in here.

Arguments

- **dt** (*number*) – Delta time.

1.6.3 Functions

create_entity()

Returns An instance of Entity.

emit_message(*MessageType*)

This function creates a new Message instance and makes it available to be read by other engines.

Arguments

- **MessageType** (*Type<Message>*) – A constructor reference to a subtype of Message.

Throws EmitUndeclaredMessageError – When this function is called on a MessageType which has not been declared in `emit_message_types`.

Returns Message An instance of Message of the given type.

emit_component_message (*ComponentMessageType*, *component*)

This function creates a new ComponentMessage instance and makes it available to be read by other engines.

Arguments

- **ComponentMessageType** (*Type<ComponentMessage>*) – A constructor reference for a subtype of ComponentMessage.
- **component** (*Component*) – An instance of a component.

Throws EmitUndeclaredMessageError – When this function is called on a MessageType which has not been declared in `emit_message_types`.

Returns ComponentMessage An instance of ComponentMessage of the given type.

emit_entity_message (*EntityMessageType*, *entity*)

This function creates a new EntityMessage instance and makes it available to be read by other engines.

Arguments

- **EntityMessageType** (*Type<EntityMessage>*) – A constructor reference for a subtype of EntityMessage.
- **entity** (*Entity*) – An instance of Entity.

Throws EmitUndeclaredMessageError – When this function is called on a MessageType which has not been declared in `emit_message_types`.

Returns EntityMessage An instance of EntityMessage of the given type.

get_entity (*entity_id*)

Arguments

- **entity_id** (*number*) – The ID of the Entity.

Returns Entity | undefined Either the Entity instance with given ID, or undefined if none exists.

read_components (*ComponentType*)

Arguments

- **ComponentType** (*Type<Component>*) – A constructor reference for a subtype of Component.

Returns GCOptimizedList<Readonly<ComponentType>> A GCOptimizedSet containing all active Components of the given ComponentType. The Components are readonly.

read_component (*ComponentType*)

Arguments

- **ComponentType** (*Type<Component>*) – A constructor reference for a subtype of Component.

Returns Readonly<TComponent> | null Returns a singleton Component of the given type or null if none exist.

read_components_mutable (*ComponentType*)

Arguments

- **ComponentType** (*Type<Component>*) – A constructor reference for a subtype of Component.

Throws IllegalComponentMutationError – When this function is called on a ComponentType which has not been declared in `mutate_component_types`.

Returns GCOptimizedList<ComponentType> A GCOptimizedSet containing mutable Component instances of the given ComponentType.

`read_component_mutable`(*ComponentType*)

Arguments

- `ComponentType` (*Type<Component>*) – A constructor reference for a subtype of Component.

Returns Readonly<TComponent> | null Returns a singleton Component of the given type or null if none exist.

`read_inactive_components`(*ComponentType*)

Arguments

- `ComponentType` (*Type<Component>*) – A constructor reference for a subtype of Component.

Returns GCOptimizedList<Readonly<ComponentType>> A GCOptimizedSet containing all inactive Components of the given ComponentType. The Components are readonly.

`make_mutable`(*component*)

Arguments

- `component` (*Readonly<Component>*) – A readonly Comopnent.

Returns Component A mutable version of the Component.

`read_messages`(*MessageType*)

Arguments

- `MessageType` (*Type<Message>*) – A constructor reference for a subtype of Message.

Throws `ReadUndeclaredMessageError` – When this function is called with a MessageType that has not been declared in `@Reads`.

Returns GCOptimizedList<Message> A GCOptimizedSet containing all Message instances of MessageType.

`some`(*MessageType*)

Arguments

- `MessageType` (*Type<Message>*) – A constructor reference for a subtype of Message.

Throws `ReadUndeclaredMessageError` – When this function is called with a MessageType that has not been declared in `@Reads`

1.6.4 Special Types

Detector

```
import { Detector } from "encompass-ecs";
```

A Detector is a subclass of Engine that provides a structure for a common pattern - performing a task for each Entity which has a particular combination of Components.

Detectors are defined by the *Component* types they track and the `detect` function they implement.

Abstracts

Type<Component>[] component_types

The Component types that will cause the Entity to be tracked by the Detector. Define using `@Detects` decorator.

detect (*entity*, *dt*)

This callback is triggered every frame when an Entity has all the required component types specified by the `component_types` property.

Arguments

- **entity** (*Entity*) – An entity that is being tracked by the Detector.
- **dt** (*number*) – Delta time.

Example

```
import { Detector, Entity } from "encompass-ecs";

import { PositionComponent } from "src/components/position";
import { VelocityComponent } from "src/components/velocity";
import { MotionMessage } from "src/messages/motion";

@Detects(PositionComponent, VelocityComponent)
@Emits(MotionMessage)
class MotionDetector extends Detector {
    protected detect(entity: Entity) {
        const position_component = entity.get_component(PositionComponent);
        const velocity_component = entity.get_component(VelocityComponent);

        const motion_message = this.create_component_message(MotionMessage, position_
↪component);
        motion_message.x_delta = velocity_component.x;
        motion_message.y_delta = velocity_component.y;
    }
}
```

ComponentModifier

```
import { ComponentModifier } from "encompass-ecs";
```

A ComponentModifier is a subclass of Engine that provides a structure for a common pattern - collecting all ComponentMessages of a particular type that reference the same Component instance.

The first message type given in the `@Reads` decorator is assumed to be the target ComponentMessage type.

Abstracts

modify (*component*, *messages*, *dt*)

This callback runs during the modify pass of World `update` when one or more ComponentMessages of `message_type` are produced during the detection pass.

Arguments

- **component** (*Component*) – The component attached to the ComponentMessage tracked by the ComponentModifier.
- **messages** (*GCOptimizedList<ComponentMessage>*) – A GCOptimizedSet of all the messages of *message_type* created during the detection pass.
- **dt** (*number*) – The delta time value given to the World *update* function.

Example

```
import { Component, ComponentModifier, Message, Mutates, Reads, Type } from
↪"encompass-ecs";
import { MusicComponent } from "hyperspace/components/music";
import { SoundComponent } from "hyperspace/components/sound";
import { AudioMessage } from "hyperspace/messages/component/audio";
import { TimeDilationBroadcast } from "hyperspace/messages/state/time_dilation";
import { AudioComponent } from "hyperspace/types/audio_component";
import { GCOptimizedSet } from "encompass-gc-optimized-collections";

@Reads(AudioMessage, TimeDilationBroadcast)
@Mutates(MusicComponent, SoundComponent)
export class AudioModifier extends ComponentModifier {
    protected modify(music_component: AudioComponent, messages: GCOptimizedSet
↪<AudioMessage>, dt: number) {
        let factor = 1;
        for (const broadcast of this.read_messages(TimeDilationBroadcast).iterable())
↪{
            factor *= broadcast.factor;
        }

        dt *= factor;

        const source = music_component.source;
        source.setPitch(factor);
        music_component.time += dt;
    }
}
```

EntityModifier

```
import { EntityModifier } from "encompass-ecs";
```

An EntityModifier is a subclass of Engine that provides a structure for a common pattern - collecting all EntityMessages of a particular type that reference the same Entity.

The first message type given in the `@Reads` decorator is assumed to be the target EntityMessage type.

Abstracts

modify (*entity*, *messages*, *dt*)

This callback runs during the modify pass of World *update* when one or more EntityMessages of `message_type` are produced during the detection pass.

Arguments

- **entity** (*Entity*) – The Entity attached to the EntityMessage tracked by the EntityModifier.
- **messages** (*GCOptimizedList<EntityMessage>*) – A GCOptimizedSet of all the messages of *message_type* created during the detection pass.
- **dt** (*number*) – The delta time value given to the World *update* function.

Example

```
import { Component, Entity, EntityModifier, Message, Reads, Type } from "encompass-ecs
↪";
import { AddComponentMessage } from "hyperspace/messages/entity/add_component";
import { GCOptimizedSet } from "encompass-gc-optimized-collections";

type MessageSet = GCOptimizedSet<AddComponentMessage<Component>>;

@Reads(AddComponentMessage)
export class AddComponentModifier extends EntityModifier {
    protected modify(entity: Entity, messages: MessageSet) {
        for (const message of messages.iterable()) {
            entity.add_component(message.component_to_add, message.args);
        }
    }
}
```

Spawner

```
import { Spawner } from "encompass-ecs";
```

A Spawner is a subclass of Engine that provides a structure for a common pattern - reading a Message and creating a new *Entity* in response.

Spawners are defined by the Message type they track, and the `spawn` function they implement.

The first message type given in the `@Reads` decorator is assumed to be the spawn message type.

Abstracts

Spawner:spawn (*message*)

Arguments

- **message** (*Message*) – A message that has been read by the Spawner.

This callback is triggered when a Message of the specified prototype is produced.

Example

```
import { Engine, Message, Reads, Type } from "encompass-ecs";
import { CellComponent } from "game/components/cell";
import { SpawnCellMessage } from "game/messages/spawn_cell";
```

(continues on next page)

(continued from previous page)

```
@Reads(SpawnCellMessage)
export class SpawnCellEngine extends Spawner {
    public spawn(message: SpawnCellMessage) {
        const cell_entity = this.create_entity();

        const cell_component = cell_entity.add_component(CellComponent);
        cell_component.i = message.i;
        cell_component.j = message.j;
        cell_component.alive = true;
    }
}
```

1.7 Renderer

A Renderer is responsible for reading the game state and drawing to the screen.

It is an error to modify Entities or Components from a Renderer.

1.7.1 Functions

`get_entity`(*entity_id*)

Arguments

- **`entity_id`** (*number*) –

Returns Entity An instance of Entity with the given `entity_id`.

`read_components`(*ComponentType*)

Arguments

- **`ComponentType`** (*Type<Component>*) – A constructor reference to a subtype of Component.

Returns GCOptimizedList<Readonly<ComponentType>> A set of readonly instances of the given ComponentType.

`read_component`(*ComponentType*)

Arguments

- **`ComponentType`** (*Type<Component>*) – A constructor reference to a subtype of Component.

Throws `SingletonReadError` – If more than one Component of the given type exists.

Returns Readonly<TComponent> | null A readonly reference to a singleton Component of the given type, or null if none exist.

1.7.2 Types

EntityRenderer

```
import { EntityRenderer } from "encompass-ecs";
```

An EntityRenderer provides a structure for the common pattern of drawing an Entity which has a particular collection of Components and a specific type of DrawComponent. They also have the ability to draw DrawComponents at their specific layer.

EntityRenderers are defined by the *Component* types and *DrawComponent* type they track, and the `render` function they implement.

Decorators

`@Renders`(*draw_component_type*, *...component_type_args*)

Writes MessageTypes to the *emit_message_types* property.

Arguments

- **`draw_component_type`** (*Type<DrawComponent>*) – The DrawComponent type which will be tracked by the EntityRenderer.
- **`component_type_args`** (*Type<Component>[]*) – The other Component types which are required to be present for the EntityRenderer to track the Entity.

Abstracts

`render`(*entity*)

This callback is triggered by the World *draw* function. Place your drawing code inside this function.

Arguments

- **`entity`** (*Entity*) – An Entity which has all `component_types` and the `draw_component_type`.

Example

```
import { Entity, EntityRenderer } from "encompass-ecs";
import { CanvasComponent } from "game/components/canvas";
import { PositionComponent } from "game/components/position";

@Renders(CanvasComponent, PositionComponent)
export class CanvasRenderer extends EntityRenderer {
    public render(entity: Entity) {
        const position_component = entity.get_component(PositionComponent);
        const canvas_component = entity.get_component(CanvasComponent);

        const canvas = canvas_component.canvas;

        love.graphics.draw(
            canvas,
            position_component.x,
            position_component.y,
            0,
            canvas_component.x_scale,
            canvas_component.y_scale,
            canvas.getWidth() * 0.5,
            canvas.getHeight() * 0.5,
        );
```

(continues on next page)

(continued from previous page)

```
    }
}
```

GeneralRenderer

```
import { GeneralRenderer } from "encompass-ecs";
```

An GeneralRenderer is an Engine which is responsible for reading the game state in order to draw elements to the screen.

It also requires a layer, which represents the order in which it will draw to the screen.

It is an anti-pattern to modify Entities or Components from within a GeneralRenderer.

Properties

layer
The layer at which the GeneralRenderer will be drawn. Lower numbers mean it is drawn earlier.

Abstracts

render()
This callback is triggered by the World *draw* function. Place your drawing code inside this function.

Example

```
import { GeneralRenderer } from "encompass-ecs";
import { SceneComponent } from "../components/scene";

export class SceneRenderer extends GeneralRenderer {
    public layer = 1;

    public render() {
        for (const scene_component of this.read_components(
            SceneComponent
        ).values()) {
            scene_component.scene.render();
        }
    }
}
```

Index