Archetype ECS lib Documentation

La gestion de contenu intégrée

PirateJL

Table of contents

coverage 99.22%

1. Archetype ECS Lib

A tiny **archetype based ECS** (Entity Component System) for TypeScript.

This documentation is split into 4 parts :

- **Explanation** of the general operation of the library
- Find information in the **Reference**
- Target a specific goal using the **How-To Guides**
- Learn through the **Tutorials**: step-by-step guidance

1.1 Install

NPM package available here

```
npm i archetype-ecs-lib
```

1.2 Quick start

```
import { World, Schedule } from "archetype-ecs-lib";

class Position { constructor(public x = 0, public y = 0) {} }
class Velocity { constructor(public x = 0, public y = 0) {} }

const world = new World();

// Spawn immediately
const e = world.spawn();
world.add(e, Position, new Position(0, 0));
world.add(e, Velocity, new Velocity(1, 0));

// A simple system
world.addSystem((w) => {
  for (const { e, c1: pos, c2: vel } of w.query(Position, Velocity)) {
    pos.x += vel.x * dt;
    pos.y += vel.y * dt;

    // Defer structural changes safely
    if (pos.x > 10) w.cmd().despawn(e);
  }
});

world.update(1 / 60);
```

Note: `SystemFn` is typed as `(world: WorldApi, dt) => void`.
Checkout the tutorials for more!

1.3 Notes & limitations

- This is intentionally minimal: **no parallelism**, no borrow-checking, no automatic conflict detection.
- Query results use `c1/c2/...` fields for stability and speed; you can wrap this in helpers if you prefer tuple returns.
- `TypeId` assignment is process-local and based on constructor identity (`WeakMap`).

1.4 License

This code is distributed under the terms and conditions of the MIT license.

January 6, 2026

January 3, 2026

2. Explanation

2.1 ECS and the game loop

ECS is best understood as the **way you organize game state and game logic**, not as the thing that *does everything*. In a typical game, the loop still has input, rendering, audio, physics, networking, etc. ECS provides a **consistent place** for runtime data (components) and behavior (systems), plus a **schedule** that defines when that behavior runs. This library already models this explicitly with `World.update(dt)` and with a phase-based `Schedule` that flushes between phases.

2.1.1 Frame phases

A "frame" is rarely just "update then draw". Most games are structured in phases, even if informally. A common conceptual breakdown:

1. **Input**: read devices/events, translate into game intent
2. **Simulation**: movement, AI, gameplay rules, timers
3. **Physics** (optional separate step): integrate, solve collisions, constraints
4. **Post-sim**: resolve gameplay outcomes, spawn/despawn, apply state transitions
5. **Render prep**: build renderable data, sort, cull
6. **Render**: submit to GPU / engine renderer
7. **End-of-frame**: cleanup, present frame, etc.

The `Schedule` is designed exactly for this idea: you define phases (strings) and run them in order, with `flush()` after each phase.

2.1.2 Where ECS fits

ECS typically fits in the **simulation and render-prep** parts of the loop:

- **World** holds the mutable runtime state (entities + components)
- **Systems** implement the game logic by querying components and mutating them
- **Commands** allow safe structural changes during those systems (`cmd()` → `flush()`)
- **Schedule** provides deterministic ordering and safe mutation boundaries between phases

A useful mental model:

- Rendering engines want a *renderable snapshot* (meshes, transforms, materials, draw lists).
- Input systems produce *intent/state* (move left, fire, target position).
- Physics engines operate on *physical representations* (bodies, colliders).

ECS sits in the middle coordinating these, not replacing them.

A concrete mapping using this primitives

- **Input phase**: read input → write `InputState` component / resource → enqueue spawns/despawns if needed
- `flush()`
- **Sim phase**: run movement/AI/gameplay using queries → update `Position` , `Velocity` , etc.
- `flush()`
- **Render phase**: build lightweight render data (`RenderTransform` , `Visible` , etc.) → hand off to renderer

This is why “flush points” exist in an ECS schedule: they define when the world structure is allowed to change and when the next phase sees those changes.

2.1.3 Why ECS does not replace rendering, input, or physics engines

Rendering

A renderer is a specialized pipeline:

- GPU resources, shaders, batching, sorting, culling
- frame graph / render passes
- platform-specific backends

ECS is not a GPU pipeline. What ECS does well is:

- storing render-related data as components (`Transform` , `Renderable` , `MaterialRef` , etc.)
- running systems that prepare and synchronize data for the renderer

So ECS often produces a **render list** or updates engine scene objects, but the renderer still does the rendering.

Input

Input is inherently eventful and platform-driven:

- OS/window events
- device state polling
- mapping raw events to game actions

ECS can *store* input state (`InputAxis` , `ActionPressed` , etc.) and *process* it in systems, but it doesn’t replace the platform input layer. In practice:

- platform collects input
- ECS system transforms it into gameplay-friendly state

Physics

Physics engines are optimized solvers:

- broadphase / narrowphase collision detection
- integrators and constraint solvers
- continuous collision, joints, sleeping, etc.

ECS can represent physics **data** (mass, collider type, desired forces) and drive the physics engine, but the solver itself is a dedicated subsystem.

A common integration pattern:

- ECS → write forces/desired velocity into physics engine
- Physics step happens
- Physics results → write back transforms/velocities into ECS

2.1.4 The key idea: ECS is the *coordination model*

ECS shines when you treat it as:

- **a data model** for game state (components)
- **a behavior model** for game logic (systems)
- **an execution model** for ordering (schedule + phases + flush points)

But rendering/input/physics are specialized domains with their own constraints and pipelines. ECS coordinates them by being the “truth” for game state and by running the logic that translates between subsystems.

January 4, 2026

January 4, 2026

2.2 Integrating an ECS with Three.js

Three.js is a **rendering engine** (scene graph + GPU submission). This ECS is a **simulation architecture** (data in components, behavior in systems, ordered by a schedule, with safe structural changes via deferred commands + flush points). Integrating them well means **letting each do what it's good at**, and defining clean "hand-off" boundaries.

2.2.1 The mental model: ECS drives state, Three.js draws it

A practical split that scales:

- **ECS World** = authoritative game/sim state (position, velocity, health, selection, etc.)
- **Three.js Scene** = visual representation (Object3D transforms, meshes, materials, lights)

So the goal is not "put Three.js inside ECS", but:

Systems write simulation state → a render-sync step pushes that state into Three.js objects.

2.2.2 Where ECS fits in the Three.js render loop

Three.js typically runs:

1. update (your code)
2. `renderer.render(scene, camera)`

With ECS, your "update" becomes scheduled phases, e.g.:

- `input` (read DOM/input, write components/resources)
- `sim` (gameplay, movement, AI)
- `render` (sync ECS → Three.js, then render)

The `Schedule` already supports this exact idea and flushes commands between phases to make entity/component creation/removal deterministic.

2.2.3 Why flush points matter for Three.js integration

Spawning/despawning and add/remove are **structural changes** in this ECS and are expected to be deferred while iterating queries/systems.

That maps perfectly to Three.js object lifecycle:

- **During sim**: decide "this entity should appear/disappear" → enqueue ECS commands
- **At flush boundary**: ECS structure becomes stable
- **Render-sync phase**: create/remove corresponding `Object3D` safely, because you're no longer mid-iteration on archetype tables

This is the same reason this ECS has `cmd()` / `flush()` and why `Schedule` flushes between phases.

2.2.4 A clean integration pattern: “Renderable bridge” components

Common approach:

- A `Transform` component (position/rotation/scale) is owned by ECS.
- A `Renderable` component carries a reference/handle to what Three.js should draw (mesh id, model key, material key…).
- A render-sync system queries `(Transform, Renderable)` and applies changes to the corresponding `Object3D`.

Key idea: **ECS components store “what it is” and “where it is”**, while the actual `Mesh/Object3D` lives in Three.js.

This keeps:

- ECS portable (not tied to Three.js types everywhere)
- Three.js free to manage GPU resources

2.2.5 One-way vs two-way sync (pick a source of truth)

Integration gets messy when both ECS and Three.js “own” transforms.

A scalable default:

- **ECS is the source of truth** for gameplay transforms.
- Three.js `Object3D` is just the projection of that state.

Only do **two-way sync** when you truly need it (editor gizmos, drag interactions). Even then, treat it as a controlled input step:

- read Object3D change in `input` or `tools` phase
- write back to ECS components
- let sim proceed from ECS again

2.2.6 Why ECS does not replace Three.js (and shouldn’t try)

Even with a “full ECS” architecture, Three.js still owns:

- scene graph concerns (parenting, cameras, lights)
- GPU resource lifetimes (buffers, textures, materials)
- draw submission, sorting, batching, culling strategies

ECS complements that by making **simulation state and logic** scalable: archetype tables + queries + systems + scheduling.

2.2.7 Scaling tips (when entity counts grow)

When you have many similar visuals:

- prefer **InstancedMesh** in Three.js
- let ECS systems produce instance transforms (dense arrays) from queries
- upload those transforms once per frame

This aligns with why archetype ECS exists: tight iteration over dense component columns.

January 4, 2026

January 4, 2026

2.3 What people mean by a “full ECS”

“ECS” can mean **just a storage model** (entities + components in some container), or it can mean an **entire game/app architecture** where *most runtime state and behavior* flows through an ECS **world + schedule + systems**.

A “full ECS” is typically an architecture where:

- **Entities** are only IDs/handles (no behavior).
- **Components** are only data.
- **Systems** are where behavior lives (pure-ish functions operating on data).
- A **World** is the single source of truth for runtime state.
- A **Scheduler** (or “app loop”) defines *when* systems run and in what order.
- Structural changes are controlled (often via a **command buffer**) so iteration stays safe and fast.

This library already contains several “full ECS” building blocks: **archetype tables (SoA)**, **queries**, **deferred commands**, and a **phase-based schedule**.

What makes it “full” is less “do you use archetypes?” and more “does the ECS define the whole program’s execution model?”

2.3.1 ECS as architecture, not just storage

Storage-only ECS (not “full”)

This is common in small libs or quick implementations:

- Entities: IDs
- Components: data bags
- “Systems”: often just loops in user code
- Little/no scheduling model
- No consistent lifecycle for input → simulation → rendering
- Structural changes are ad-hoc

You *can* build a game with this, but the ECS isn’t the **organizing principle**—it’s a container.

Architecture ECS (“full ECS”)

Here, ECS is the **spine of the app**:

- There’s a **main schedule** (often phases like `input → sim → render`).
- Systems are registered, ordered, and executed consistently each tick.
- Cross-cutting state is handled intentionally (resources/singletons, events, time, config).
- Structural changes are made safe/deterministic (command buffers, flush points).
- You get a uniform pattern for new features: “add data + add system”.

The `Schedule` explicitly models *phase ordering + flush barriers*, which is a key “architecture ECS” ingredient.

2.3.2 Difference between a library ECS and an engine ECS

Library ECS

Goal: provide **core ECS mechanics**.

Typical traits:

- Focus on **storage + query performance** (archetypes/SoA)
- Minimal assumptions about the rest of the program
- Simple scheduling (or none), often single-threaded
- You (the user) integrate input, rendering, physics, assets, scenes, etc.

Engine ECS (Bevy / Unity DOTS / etc.)

Goal: ECS is the **entire runtime framework**.

Engine ECS usually includes (beyond a library):

- A full **app lifecycle** (startup, update, fixed update, shutdown)
- Integrated **input**, **rendering**, **audio**, **physics**, **animation**, **UI**
- Asset pipeline + hot reload + serialization
- Advanced scheduling: dependency graphs, system sets, run criteria, fixed timesteps
- Often **parallel execution** + conflict detection
- Tooling/editor integration

So: **library ECS = the “ECS core”**. **engine ECS = ECS core + everything around it**, with ECS as the central organizing model.

January 4, 2026

January 4, 2026

2.4 Why archetype ECS?

An **archetype ECS** organizes entities into **tables** where every entity in a table shares the same component set, stored in **SoA** form (one column per component). This library explicitly follows this model: "Archetypes (tables) store entities in a SoA layout… Queries iterate matching archetypes efficiently… Commands defer structural changes…"

The "why" is mostly about making the *common case* (systems that iterate lots of entities with the same components) extremely fast and predictable.

2.4.1 Cache locality

Most game/sim systems look like:

- "for all entities with `Position` and `Velocity`, update position"
- "for all entities with `Transform` and `Renderable`, build render data"

With archetypes, those entities live together in a table, and each component is a dense column:

- `Position[]` contiguous
- `Velocity[]` contiguous

So the CPU reads memory sequentially, which is what caches and prefetchers love. That's the practical meaning of **cache locality**: fewer cache misses, more work per nanosecond.

In the library, this is literally the storage promise: SoA archetype tables + queries over matching archetypes.

2.4.2 Branch elimination (and "no-join" iteration)

In many ECS designs, the core loop must constantly ask:

- "does this entity have Velocity?"
- "if yes, fetch it; if not, skip"

That creates branches and scattered memory access.

With archetypes, the *membership check is moved up*:

1. pick archetypes that already contain all required components
2. iterate their rows

Inside the inner loop, there's no per-entity "has component?" branching—every row is guaranteed to match. The API reflects that by querying required component types and yielding direct component references (`c1`, `c2`, …).

This is what people mean by **branch elimination** in archetype ECS: fewer conditional checks in the hot loop, more straight-line code.

2.4.3 Predictable iteration

Archetype iteration tends to be predictable because:

- You iterate dense arrays (rows/columns), not sparse IDs.
- Results are shaped consistently (`e` , `c1` , `c2` , … in argument order).
- Structural changes are controlled: this library emphasizes deferring structural changes via `cmd()` and applying them at `flush()` points.
- `Schedule` adds explicit “phase barriers” by flushing between phases, making the world structure stable during each phase’s iteration.

That predictability is less about “deterministic order of entities” and more about **deterministic rules for when the world can change shape**.

2.4.4 Comparison with sparse-set ECS

A **sparse-set ECS** typically stores each component type separately (often as a dense array + sparse index by entity id). It’s excellent for:

- fast lookup for a single component type (`Position` alone)
- cheap per-component iteration
- simple storage and often cheaper structural changes for *single* components

But when a system needs **multiple components** (`Position + Velocity + Mass + Forces`), sparse-set often needs some form of **join**:

- iterate one component pool, check membership in the others
- or intersect sets / hop through indirections

That can introduce:

- more branching (`if has(...)`)
- more random memory access (chasing indices across pools)

Archetypes flip that trade-off:

- multi-component iteration is the “happy path” (no join inside the hot loop)
- but structural changes can be more expensive because adding/removing a component may move an entity between tables.

Rule of thumb

- If your game spends most time in **systems that read/write several components per entity**, archetypes tend to shine.
- If your workload is lots of **single-component iteration** and **high churn** (constant add/remove), sparse-set can be simpler and sometimes cheaper.

2.4.5 The real trade-off (why it’s not “always archetypes”)

Archetype ECS wins by making the hot loops fast, but it pays for it with:

- **structural churn cost** (moving entities between tables on add/remove)
- **many archetypes** if you have lots of component combinations
- a stronger need for **command buffering + flush boundaries** to keep iteration safe.

That’s why a “full ECS” architecture often includes commands + scheduling: it’s the natural partner to archetype storage.

January 14, 2026

January 4, 2026

2.5 Why deferred commands exist in an archetype ECS

In an archetype ECS, **deferred commands** (a command buffer) are not a "nice-to-have". They exist because **the fastest storage model makes certain mutations unsafe during iteration**. The library API expresses this directly with `world.cmd()`, `world.flush()`, and `Schedule.run(...)/flush barriers`.

2.5.1 Archetypes are tables, and queries walk those tables

An archetype ECS stores entities in **tables**:

- one archetype = one *component set*
- one row = one entity
- one column per component type (SoA)

A query like `world.query(Position, Velocity)` does not "scan entities". It first selects archetypes that contain the required component columns, then iterates **dense rows** in those tables.

This density is where the performance comes from.

2.5.2 The core problem: structural changes move entities between tables

A **structural change** is anything that changes the component *set* of an entity:

- `spawn()`
- `despawn(e)`
- `add(e, Ctor, value)`
- `remove(e, Ctor)`

In an archetype ECS, `add/remove` usually means:

1. remove the entity's row from its current archetype table
2. insert a row into another archetype table
3. update internal bookkeeping (where the entity lives now)

That is fundamentally different from `set(e, Ctor, value)`, which just updates a value *inside the same row/column*.

So: **structural change = table move**.

2.5.3 Why it's unsafe to do structural changes during a query

When you iterate a query, you are conceptually doing:

- "for each matching archetype table"
- "for each row index in that table"
- "read columns at that row"

If you structurally change any entity during this loop, you can break the iteration invariants:

1) Swap-remove can invalidate the current row

Many archetype implementations remove rows with **swap-remove** (O(1)): the last row is swapped into the removed row index.

If you remove entity A at row `i`, entity B may be swapped into row `i`.

- If your loop then increments `i`, entity B might be **skipped**.
- Or processed twice depending on iteration strategy.

2) Moving entities changes which archetypes match

Adding/removing a component can move an entity into or out of the set of archetypes that the query is iterating.

If you mutate membership while iterating:

- you can end up iterating an archetype that didn’t exist in the matching set at the start
- or miss entities that moved into a matching archetype

3) Internal indices can become stale mid-loop

The library `World` tracks where an entity lives (which archetype + row). A structural change updates those indices. If you mutate while holding references from the iteration, you can end up with:

- stale row pointers
- stale bookkeeping
- inconsistent state if multiple mutations occur

Even if you “think it works”, it’s fragile and will eventually bite.

2.5.4 Deferred commands are the solution: separate “read/iterate” from “mutate structure”

A command buffer enforces a clean two-step model:

1. **During iteration**: read data, compute decisions, mutate *component values* (safe)
2. **At a safe boundary**: apply structural changes in a batch (safe)

That’s exactly what the library documents:

- `world.cmd()` enqueues structural operations
- `world.flush()` applies queued commands
- `world.update(dt)` runs systems, then flushes at frame end
- `Schedule.run(...)` flushes **between phases**, providing deterministic barriers

This is why deferred commands exist: they preserve **iteration correctness** without giving up **table-based performance**.

2.5.5 Why flushing in phases is architecturally important

The library `Schedule` explicitly flushes after each phase.

This is not just “nice ordering”. It creates **deterministic points** where the world’s structure is allowed to change.

Example mental model:

- **Input phase**: decide spawns/despawns based on input → enqueue commands
- **Flush**: apply those spawns so they exist for simulation
- **Simulation phase**: move things, detect collisions → enqueue structural changes
- **Flush**: apply spawns/despawns/removals before render
- **Render phase**: build render data from a stable world snapshot

That separation reduces “action at a distance” bugs and makes debugging easier:

- “why does entity exist in sim but not render?” → check which phase flushed it.

2.5.6 What you gain by deferring

Correctness

- No skipped entities
- No double-processing due to swap-remove effects
- Stable iteration semantics

Determinism

- Structural changes occur at explicit boundaries
- Easier to reason about ordering

Performance

- Keeps archetype iteration tight and cache-friendly
- Batching structural operations reduces churn

2.5.7 What to do inside a system

Inside a system (or a query loop), follow this rule:

- ✅ mutate component values directly (e.g. `pos.x += ...`)
- ✅ enqueue structural changes via `cmd()`
- ❌ don’t call structural `World` ops directly mid-iteration

2.5.8 Summary: the “why” in one sentence

Deferred commands exist because **archetype queries iterate dense tables**, and **structural changes move rows between tables**, which can invalidate iteration—so we **queue structural changes** and apply them at **safe flush boundaries** (`flush()` / schedule phases).

January 4, 2026

January 4, 2026

2.6 Why use Events in ECS?

2.6.1 Events solve a different problem than Components and Resources

ECS has three kinds of data:

- **Components**: persistent, per-entity state (Position, Velocity, Health)
- **Resources**: persistent, global state (Input, Time, Config, caches)
- **Events**: transient messages (Hit happened, Click happened, Play sound)

Trying to represent “something happened” as a component usually causes awkward designs:

- adding/removing “Event components” becomes structural churn
- you need cleanup systems to remove them
- multiple systems race to observe/remove them

Events avoid that by being explicitly transient.

2.6.2 Events reduce coupling between systems

Without events:

- `combatSystem` might call `audioSystem` directly
- or it might mutate a shared global array

With events:

- producers don’t know consumers exist
- consumers don’t know who produced the messages

This keeps systems reusable and easy to rearrange in `Schedule`.

2.6.3 Why double-buffering?

A common bug in event systems is “events appear while I’m iterating”.

Double-buffering prevents that:

- consumers read a stable snapshot (`read buffer`)
- producers write to a different buffer (`write buffer`)
- swap happens at deterministic boundaries

No surprises. No iterator invalidation. No mid-phase visibility.

2.6.4 Why phase-scoped delivery?

This ECS already has a concept of **phase boundaries**:

- structural changes are deferred via Commands
- `flush()` applies them between phases

Events align with the same boundary:

- `swapEvents()` delivers events between phases

This makes it easy to design pipelines:

- `input` produces actions → `beforeUpdate` consumes
- `update` produces gameplay events → `afterUpdate` consumes
- `render` produces UI/VFX events → `afterRender` consumes
- `audio` consumes sound events

2.6.5 Trade-offs (and the forwarding pattern)

With phase-scoped delivery, an event is visible in the **next phase only**. To deliver an event across multiple phases (e.g., from `update` to `audio`), you forward it by draining and re-emitting.

This is deliberate:

- it keeps pipelines explicit
- prevents “stale” events lingering through unrelated phases
- makes delivery deterministic and easy to debug

January 14, 2026

January 13, 2026

3. How To Guides

3.1 How to add InputState + AssetCache as Resources and use them in systems

3.1.1 Goal

Store **Input state** and an **Asset cache** as world **Resources**, then access them inside systems using `requireResource()`.

Example InputStateRes

```
export class InputStateRes
{
    public keysDown = new Set<string>();
    public keysPressed = new Set<string>();   // pressed this frame
    public keysReleased = new Set<string>();  // released this frame

    public mouseX = 0;
    public mouseY = 0;
    public mouseButtonsDown = new Set<number>();
    public mousePressed = new Set<number>();   // pressed this frame
    public mouseReleased = new Set<number>();  // released this frame
    public wheelDeltaY = 0;

    beginFrame(): void
    {
        this.keysPressed.clear();
        this.keysReleased.clear();
        this.mousePressed.clear();
        this.mouseReleased.clear();
        this.wheelDeltaY = 0;
    }

    keyDown(code: string): void
    {
        if (!this.keysDown.has(code)) this.keysPressed.add(code);
        this.keysDown.add(code);
    }

    keyUp(code: string): void
    {
        if (this.keysDown.has(code)) this.keysReleased.add(code);
        this.keysDown.delete(code);
    }

    mouseMove(x: number, y: number): void {
        this.mouseX = x;
        this.mouseY = y;
    }

    mouseDown(btn: number): void
    {
        if (!this.mouseButtonsDown.has(btn)) this.mousePressed.add(btn);
        this.mouseButtonsDown.add(btn);
    }

    mouseUp(btn: number): void
    {
        if (this.mouseButtonsDown.has(btn)) this.mouseReleased.add(btn);
        this.mouseButtonsDown.delete(btn);
    }

    wheel(deltaY: number): void
    {
        this.wheelDeltaY += deltaY;
    }
}
```

Example AssetCacheRes

```
export class AssetCacheRes
{
    private images = new Map<string, HTMLImageElement>();
    private pending = new Map<string, Promise<HTMLImageElement>>();

    /** Loads once, dedupes concurrent calls, returns the same instance thereafter. */
    public getImage(url: string): Promise<HTMLImageElement>
    {
        const ready = this.images.get(url);
        if (ready) return Promise.resolve(ready);

        const p = this.pending.get(url);
        if (p) return p;

        const promise = new Promise<HTMLImageElement>((resolve, reject) => {
            const img = new Image();
            img.onload = () => {
                this.images.set(url, img);
                this.pending.delete(url);
                resolve(img);
            };
            img.onerror = (e) => {
                this.pending.delete(url);
                reject(e);
            };
            img.src = url;
        });

        this.pending.set(url, promise);
        return promise;
    }

    /** Returns the image if already loaded; otherwise undefined. */
    public peekImage(url: string): HTMLImageElement | undefined {
        return this.images.get(url);
    }
}
```

3.1.2 1) Register the resources at startup

```
world.initResource(InputStateRes, () => new InputStateRes());
world.initResource(AssetCacheRes, () => new AssetCacheRes());
```

That’s the only “required” setup. Everything else assumes these exist.

3.1.3 2) Wire DOM events into `InputStateRes`

Attach listeners once:

```
export function attachInput(world: WorldApi): void
{
    const input = world.requireResource(InputStateRes);

    window.addEventListener("keydown", e => input.keyDown(e.code));
    window.addEventListener("keyup",   e => input.keyUp(e.code));
    window.addEventListener("mousemove", e => input.mouseMove(e.clientX, e.clientY));
    window.addEventListener("mousedown", e => input.mouseDown(e.button));
    window.addEventListener("mouseup",   e => input.mouseUp(e.button));
    window.addEventListener("wheel",     e => input.wheel(e.deltaY), { passive: true });
}
```

Call it after `initResource(...)`.

3.1.4 3) Reset “pressed/released” flags once per frame

Add a phase/system that runs before gameplay update:

```
export function beginFrameSystem(w: WorldApi, _dt: number): void
{
    w.requireResource(InputStateRes).beginFrame();
}
```

3.1.5 4) Read input from systems

Example “move player” system:

```
export function playerMoveSystem(w: WorldApi, dt: number): void
{
    const input = w.requireResource(InputStateRes);

    let dx = 0, dy = 0;
    if (input.keysDown.has("KeyW")) dy -= 1;
    if (input.keysDown.has("KeyS")) dy += 1;
    if (input.keysDown.has("KeyA")) dx -= 1;
    if (input.keysDown.has("KeyD")) dx += 1;

    const speed = 220;

    for (const { c1: tr } of w.query(Transform, PlayerTag)) {
        tr.x += dx * speed * dt;
        tr.y += dy * speed * dt;
    }
}
```

3.1.6 5) Use `AssetCacheRes` in a render system (deduped async loads)

```
export function renderSpritesSystem(ctx: CanvasRenderingContext2D)
{
    return (w: WorldApi, _dt: number): void => {
        const assets = w.requireResource(AssetCacheRes);

        for (const { c1: tr, c2: sp } of w.query(Transform, Sprite)) {
            assets.getImage(sp.url).catch(() => {});
            const img = assets.peekImage(sp.url);
            if (!img) continue;

            ctx.drawImage(img, tr.x, tr.y, sp.w, sp.h);
        }
    };
}
```

3.1.7 6) Run phases in order

Minimal schedule:

```
sched.add("beginFrame", beginFrameSystem);
sched.add("update", playerMoveSystem);
sched.add("render", renderSpritesSystem(ctx));
```

Game loop:

```
sched.run(world, dt, ["beginFrame", "update", "render"]);
```

3.1.8 Common variations

Optional resource usage

If a resource is optional (debug/editor), use:

```
const dbg = w.getResource(DebugRes);
if (dbg) dbg.enabled = true;
```

Preload assets (menu/loading screen)

```
await Promise.all(urls.map(u => world.requireResource(AssetCacheRes).getImage(u)));
```

January 14, 2026

January 9, 2026

3.2 How to add/remove components at runtime

1. Define your component types (classes):

```
class Position { constructor(public x = 0, public y = 0) {} }
class Velocity { constructor(public x = 0, public y = 0) {} }
```

1. Add/remove **immediately** when you are **not** iterating a query:

```
const e = world.spawn();
world.add(e, Position, new Position(0, 0));
world.add(e, Velocity, new Velocity(1, 0));

// Or add many at once
const z = world.spawn();
world.addMany(z, [new Position(0, 0), new Velocity(1, 0)])

world.remove(e, Velocity);
```

1. Add/remove **during a query/system** using **deferred commands**:

```
world.addSystem((w: any) => {
  for (const { e, c1: pos } of w.query(Position)) {
    if (pos.x > 10) w.cmd().add(e, Velocity, new Velocity(1, 0));
    if (pos.x < 0)  w.cmd().remove(e, Velocity);
  }
});

// Or remove many at once
world.addSystem((w: any) => {
  for (const { e, c1: pos } of w.query(Position, Velocity)) {
    if (pos.x < 0)  w.cmd().removeMny(e, Position, Velocity);
  }
});

// apply queued structural changes
world.flush();
```

January 8, 2026

January 4, 2026

3.3 How to despawn entities safely

1. Despawn **immediately** when not iterating:

```
world.despawn(e);
```

1. Despawn **during a query/system** via `cmd()`:

```
world.addSystem((w: any) => {
  for (const { e, c1: pos } of w.query(Position)) {
    if (pos.x > 10) w.cmd().despawn(e);
  }
});

// apply despawns
world.flush();
```

1. Or rely on end-of-frame flush:

```
world.update(dt); // runs systems, then flushes
```

January 4, 2026

January 4, 2026

3.4 How to have multiple Worlds (globe vs ground simulation)

1. Create two worlds:

```
const globeWorld = new World();
const groundWorld = new World();
```

1. Give each one its own schedule (recommended):

```
const globeSched  = new Schedule();
const groundSched = new Schedule();
```

1. Run both each frame (same `dt`):

```
globeSched.run(globeWorld, dt, ["input", "sim", "render"]);
groundSched.run(groundWorld, dt, ["input", "sim", "render"]);
```

1. Share data **explicitly** between worlds (pick one):
2. copy values at a known point (end of `sim`, start of other `sim`)
3. or have a “bridge” step in your outer loop that reads from one world and writes into the other (via normal `add/set` or via `cmd()` + `flush()`)

January 14, 2026

January 4, 2026

3.5 How to integrate ECS into a game loop

3.5.1 Option A — Use `world.update(dt)`

1. Register systems with `addSystem(...)`
2. In your loop call:

```
function tick(dt: number) {
  world.update(dt); // runs systems, then flushes
}
```

3.5.2 Option B — Use `Schedule` phases (recommended for games)

1. Build a schedule (`input` , `sim` , `render`)
2. In `requestAnimationFrame` :

```
let last = performance.now();

function frame(now: number) {
  const dt = (now - last) / 1000;
  last = now;

  sched.run(world, dt, ["input", "sim", "render"]); // flush between phases
  renderer.render(scene, camera);

  requestAnimationFrame(frame);
}

requestAnimationFrame(frame);
```

January 4, 2026

January 4, 2026

3.6 How to run logic conditionally

3.6.1 Option A — Guard inside the system (simple)

1. Put a condition at the top:

```
let paused = false;

world.addSystem((w: any, dt: number) => {
  if (paused) return;
  for (const { c1: pos, c2: vel } of w.query(Position, Velocity)) {
    pos.x += vel.x * dt;
  }
});
```

3.6.2 Option B — Conditional phases (skip whole groups)

1. Maintain your phase list dynamically:

```
const base = ["input", "sim", "render"];

function getPhases(paused: boolean) {
  return paused ? ["input", "render"] : base;
}

sched.run(world, dt, getPhases(paused));
```

3.6.3 Option C — Wrap systems (reuse predicates)

1. Make a helper:

```
const runIf = (pred: () => boolean, fn: (w: any, dt: number) => void) =>
  (w: any, dt: number) => { if (pred()) fn(w, dt); };

world.addSystem(runIf(() => !paused, (w, dt) => {
  for (const { c1: pos, c2: vel } of w.query(Position, Velocity)) {
    pos.x += vel.x * dt;
  }
}));
```

January 4, 2026

January 4, 2026

3.7 How to split logic into multiple system phases

1. Create a `Schedule` and register systems by phase name:

```
const sched = new Schedule();

sched
  .add("input", (w: any) => { /* ... */ })
  .add("sim",   (w: any, dt: number) => { /* ... */ })
  .add("render",(w: any) => { /* ... */ });
```

1. Define phase order:

```
const phases = ["input", "sim", "render"];
```

1. Run it each tick (flush happens after each phase):

```
sched.run(world, dt, phases);
```

January 4, 2026

January 4, 2026

3.8 How to use ECS alongside Three.js

3.8.1 Pattern: ECS owns state, Three.js owns objects

1. Keep Three.js objects in a map (outside ECS):

```
const meshes = new Map<number, THREE.Object3D>(); // key = entity.id
```

1. Add components for simulation and "render tag":

```
class Position { constructor(public x=0, public y=0, public z=0) {} }
class Renderable { constructor(public kind: "cube" | "ship" = "cube") {} }
```

1. Spawn entities in ECS:

```
const e = world.spawnMany(
  new Position(0, 0, 0),
  new Renderable("cube")
)
```

1. Create a **render-sync** system in a `render` phase:
2. create missing meshes
3. update transforms
4. remove meshes for despawned entities (see step 5)

```
sched.add("render", (w: any) => {
  for (const { e, c1: pos, c2: rend } of w.query(Position, Renderable)) {
    let obj = meshes.get(e.id);
    if (!obj) {
      obj = makeObjectFromKind(rend.kind); // your factory
      scene.add(obj);
      meshes.set(e.id, obj);
    }
    obj.position.set(pos.x, pos.y, pos.z);
  }
});
```

1. Despawn visually **after flush**:
2. despawn in ECS via `cmd().despawn(e)`
3. after the flush boundary, remove from `meshes` if it's gone

   A simple cleanup pass each frame:

```
for (const [id, obj] of meshes) {
  // if you track alive entities externally, remove when not alive anymore.
  // (One common approach: record seen IDs during the render query and remove the rest.)
}
```

January 8, 2026

January 4, 2026

3.9 How to use Events to decouple systems across phases

3.9.1 Goal

Emit events in one phase and consume them in a later phase, without coupling systems directly.

This guide assumes you already have a `Schedule` with multiple phases and that the schedule swaps events between phases.

3.9.2 1) Define event types

Use classes (recommended) or token keys.

```
export class DamageEvent {
    constructor(public target: Entity, public amount: number) {}
}

export class PlaySoundEvent {
    constructor(public id: string) {}
}
```

3.9.3 2) Emit events from a producer system

Example: gameplay system emits damage + sound.

```
function combatSystem(w: WorldApi, _dt: number) {
    // ... detect hit
    w.emit(DamageEvent, new DamageEvent(target, 10));
    w.emit(PlaySoundEvent, new PlaySoundEvent("hit"));
}
```

3.9.4 3) Consume events in the next phase

Place a consumer in the **next** phase (phase-scoped delivery):

```
function applyDamageSystem(w: WorldApi, _dt: number) {
    w.drainEvents(DamageEvent, (ev) => {
        const hp = w.get(ev.target, Health);
        if (!hp) return;
        hp.value -= ev.amount;
    });
}
```

Schedule order:

```
schedule.add("update", combatSystem);
schedule.add("afterUpdate", applyDamageSystem);
```

3.9.5 4) Deliver events to late phases (forwarding pattern)

With phase-scoped delivery, an event emitted in `update` is visible in `afterUpdate`. If you want it to reach `audio` several phases later, forward it:

```
function forwardSoundSystem(w: WorldApi, _dt: number) {
    w.drainEvents(PlaySoundEvent, (ev) => {
        w.emit(PlaySoundEvent, ev); // re-emit for the next phase
    });
}

function audioSystem(w: WorldApi, _dt: number) {
    w.drainEvents(PlaySoundEvent, (ev) => {
        console.log("[audio] play:", ev.id);
    });
}
```

Example pipeline:

```
schedule.add("update", combatSystem);            // emits PlaySoundEvent
schedule.add("afterUpdate", forwardSoundSystem); // forwards -> render
schedule.add("afterRender", forwardSoundSystem); // forwards -> audio
schedule.add("audio", audioSystem);              // consumes
```

3.9.6 5) Use `events(key).values()` for read-only inspection

If you need to check what's readable without consuming it:

```
const pending = w.events(DamageEvent).values();
if (pending.length > 0) {
    // inspect (do not store array reference)
}
```

Prefer `drainEvents` for typical processing.

3.9.7 6) Clear events when resetting state

To clear one type:

```
w.clearEvents(DamageEvent);
```

To clear all readable event buffers:

```
w.clearEvents();
```

January 13, 2026

January 13, 2026

4. Reference

4.1 Archetypes

4.1.1 Purpose

An **archetype** is an internal storage “table” that groups together all entities sharing the **same set of component types**. Archetypes are the core performance mechanism of this ECS: queries match archetypes first, then iterate rows inside them.

4.1.2 Storage model

Table layout (SoA)

Archetypes store component data in **Structure of Arrays (SoA)** form:

- **one column per component type**
- each entity occupies a **row** across all columns

This is the reason queries are efficient: iteration is over dense arrays rather than scattered objects.

4.1.3 Archetype membership

Structural changes move entities between archetypes

When an entity’s component *set* changes, the entity moves to a different archetype:

- `add(e, Ctor, value)` is **structural** and *may move* the entity to another archetype
- `remove(e, Ctor)` is **structural** and *may move* the entity to another archetype

Non-structural updates do not change archetype membership:

- `set(e, Ctor, value)` updates the value but does not change the component set

4.1.4 Queries and archetypes

Archetype filtering

`query(...ctors)` only iterates archetypes that contain *all* required component columns, then yields matching entity rows.

Query row shape

For `query(A, B, C)`, the yielded row contains:

- `e` (entity handle)
- `c1`, `c2`, `c3` component values in the same order as the ctor arguments

4.1.5 Safety constraints

Structural changes during iteration

While iterating queries (and generally while systems run), doing structural changes directly can throw. The recommended pattern is:

- enqueue structural changes via `world.cmd()`
- apply them via `world.flush()` (or at the end of `world.update(dt)`)

This matters because structural changes imply archetype moves.

4.1.6 Visibility / Public API

Archetypes are an **internal mechanism** (the public exports are `Types`, `TypeRegistry`, `Commands`, `World`, `Schedule`). Users interact with archetypes only indirectly through `World` operations and `query()`.

January 4, 2026

January 4, 2026

4.2 Commands

4.2.1 Purpose

`Commands` is a **deferred structural change buffer**. It lets you enqueue structural operations (spawn/despawn/add/remove) while iterating queries or running systems, then apply them later via `world.flush()` (or at the end of `world.update(dt)`).

4.2.2 How to obtain a `Commands` buffer

`world.cmd(): Commands`

`World.cmd()` returns a `Commands` instance you can use to enqueue operations.

Typical usage:

```
const cmd = world.cmd();

cmd.spawn((e) => {
    cmd.add(e, Position, new Position(0, 0));
});

cmd.add(entity, Velocity, new Velocity(1, 0));
cmd.remove(entity, Velocity);
cmd.despawn(entity);

world.flush();
```

4.2.3 Supported operations

The command buffer supports these operations (as documented by the project):

`spawn(init?)`

Enqueues creation of a new entity.

- `init?: (e: Entity) => void` is an optional callback invoked with the spawned entity, typically used to enqueue `add()` calls for initial components.

`spawnBundle(...items: ComponentCtorBundleItem[])`

Queues the creation of a new entity, along with its initial components, and applies everything on the next flush (within the same flush cycle).

- `...items: ComponentCtorBundleItem[]` is the list of components to add to the newly created entity.
- Internally, it iterates over the items and calls `add(e, ctor, value)` for each component.

`despawn(e: Entity)`

Enqueues removal of an entity.

`despawnBundle(entities: Entity[])`

Enqueues the destruction of multiple entities. The actual removals are applied when commands are flushed.

- `entities: Entity[]` is the list of entities to despawn.
- Internally, it iterates over the array and calls `despawn(e)` for each entity.

`add(e, ctor, value)`

Enqueues adding a component to an entity. This is a **structural** change (it may move the entity between archetypes), which is why it is commonly deferred.

`addBundle(e: Entity, ...items: ComponentCtorBundleItem[])`

Enqueues adding multiple components to an existing entity. All component adds are applied on flush.

- `e: Entity` is the target entity.
- `...items: ComponentCtorBundleItem[]` is the list of components to add.
- Internally, it loops through the items and calls `add(e, ctor, value)` for each component.

`remove(e, ctor)`

Enqueues removing a component from an entity. This is also a **structural** change.

`removeBundle(e: Entity, ...ctors: ComponentCtor<any>[])`

Enqueues removal of multiple component types from an entity. The removals are applied on flush.

- `e: Entity` is the target entity.
- `...ctors: ComponentCtor<any>[]` is the list of component constructors (types) to remove.
- Internally, it loops through the ctors and calls `remove(e, ctor)` for each one.

4.2.4 Applying commands

`world.flush(): void`

Applies all queued commands. `World.update(dt)` also flushes automatically at the end of the frame.

With `Schedule`

When using `Schedule`, `world.flush()` is called **after each phase**, creating deterministic “phase barriers” for command application.

4.2.5 Safety rule

Direct structural operations can throw while iterating queries or running systems. The intended pattern is:

- enqueue structural changes with `world.cmd()`
- apply them with `world.flush()` (or let `update()` do it)

January 8, 2026

January 4, 2026

4.3 Components

4.3.1 Purpose

A **component** is a unit of data attached to an `Entity`. In this ECS, components are stored in **archetypes (tables)** using a **Structure-of-Arrays (SoA)** layout: **one column per component type**.

4.3.2 Component "type" (key)

A component type is identified by a **constructor** (typically a class):

```
class Position { constructor(public x = 0, public y = 0) {} }
class Velocity { constructor(public x = 0, public y = 0) {} }
```

Any class used as a type key is considered a valid component type.

TypeId mapping

Internally, component constructors are mapped to a stable numeric **`TypeId`** via `typeId()`. `TypeId` assignment is **process-local** and based on **constructor identity** (via `WeakMap`).

4.3.3 Component "value"

The component value is the actual instance stored in the archetype column (e.g. `new Position(1,2)`).

- Values are stored per-archetype, per-column (SoA)
- Queries return **direct references** to these values (you mutate them in place)

4.3.4 World operations on components

All component operations are done through `World` using the component constructor as the key.

Presence / access

- `has(e, Ctor): boolean`
- `get(e, Ctor): T | undefined`

Update (non-structural)

- `set(e, Ctor, value): void` Requires the component to exist; otherwise throws.

Structural changes

These may **move the entity between archetypes**:

- `add(e, Ctor, value): void`
- `remove(e, Ctor): void`

4.3.5 Queries and component ordering

`world.query(A, B, C)` yields rows shaped like:

- `e` : the entity
- `c1` , `c2` , `c3` : component values in the **same order** as the ctor arguments

Example:

```
for (const { e, c1: pos, c2: vel } of world.query(Position, Velocity)) { }
```

4.3.6 Safety rules during iteration

While iterating a query (or while systems are running), **direct structural changes can throw**. Use deferred commands instead:

- enqueue via `world.cmd()`
- apply via `world.flush()`

January 4, 2026

January 4, 2026

4.4 Entity

4.4.1 Purpose

An **Entity** is a lightweight, opaque handle used to reference rows stored inside archetypes. It is **not** the data itself (components hold the data).

4.4.2 Type

```
type Entity = { id: number; gen: number };
```

- `id` : stable numeric slot identifier
- `gen` : **generation counter** used to detect stale handles after despawn / reuse

4.4.3 Semantics

Identity

An entity handle is considered valid only if **both**:

- the `id` refers to an allocated slot
- the `gen` matches the current generation for that slot

Stale handles

If an entity is despawned and the `id` is later reused, the `gen` will differ. This prevents accidentally operating on "the new entity that reused the same id".

4.4.4 Where entities come from

- `world.spawn()` returns an `Entity` handle
- `world.query(...)` yields rows that include `e: Entity`

4.4.5 Where entities are used

Entities are passed into World operations (examples):

- lifecycle: `despawn(e)`
- components: `add(e, Ctor, value)` , `remove(e, Ctor)` , `get(e, Ctor)` , `set(e, Ctor, value)`
- commands (deferred): `cmd.despawn(e)` , `cmd.add(e, ...)` , `cmd.remove(e, ...)`

4.4.6 Related behavior

Safety during iteration

When iterating query results (which contain `e: Entity`), structural changes should be deferred via commands and applied with `flush()` .

January 4, 2026

January 4, 2026

4.5 Reference: Events API

4.5.1 Overview

Events are **typed, transient messages** used to decouple systems. They are stored per event type in **double-buffered channels**:

- `emit()` appends to the **write buffer** (current phase)
- `drain()` / `values()` read from the **read buffer** (previous phase)
- At each phase boundary, `world.swapEvents()` swaps buffers so events become visible to the next phase

Key type

Event channels are keyed by `ComponentCtor<T>` (same as components/resources). Keys are compared by identity.

4.5.2 `EventChannel<T>` (Events.ts)

`emit(ev: T): void`

Appends an event to the **write buffer** for the current phase.

Notes

- Emitted events are **not readable in the same phase**
- They become readable after the next `swapBuffers()` / `world.swapEvents()`

`drain(fn: (ev: T) => void): void`

Iterates all **readable events** (read buffer) and then **clears** that buffer.

Semantics

- Reads only events emitted in the **previous phase**
- After `drain`, `count()` becomes `0`

Performance

- No iterator allocations; uses indexed loop
- Clears with `length = 0`

`values(): readonly T[]`

Returns a read-only view of the **read buffer**.

Semantics

- Snapshot is valid until the next boundary swap
- Do not store the returned array long-term

`count(): number`

Returns the number of readable events currently in the **read buffer**.

`clear(): void`

Clears the **read buffer** only.

`clearAll(): void`

Clears both **read** and **write** buffers.

`swapBuffers(): void` **(internal)**

Swaps read/write buffers and clears the new write buffer.

Semantics

- Makes events emitted in the previous phase readable now
- Drops any undrained events from the prior read buffer at the next swap (phase-scoped delivery)

4.5.3 Delivery model summary (phase-scoped)

If you run phases:

`A -> B -> C`

Events emitted in **A** are readable in **B**. If not drained in **B**, they are dropped at `B -> C` swap.

January 13, 2026

January 13, 2026

4.6 Non goals

January 4, 2026

January 4, 2026

4.7 Query — Reference

4.7.1 Purpose

A **Query** iterates all entities that have **all required component types**, efficiently by scanning only the **matching archetypes (tables)**.

4.7.2 API

`world.query(...ctors): Iterable<any>`

`ctors` is a list of component constructors (types) you want to require.

```
for (const row of world.query(Position, Velocity)) {
  // ...
}
```

Queries yield rows shaped like:

- `e` : the `Entity`
- `c1` , `c2` , `c3` , ...: component values **in the same order** as the `ctors` arguments

So `query(A, B, C)` yields `{ e, c1: A, c2: B, c3: C }` .

4.7.3 Row mapping and ordering

Deterministic component fields

The mapping is positional:

- `query(A)` → `{ e, c1 }`
- `query(A, B)` → `{ e, c1, c2 }`
- `query(A, B, C)` → `{ e, c1, c2, c3 }`

And `cN` always corresponds to the Nth constructor you passed.

4.7.4 Safety rules during iteration

While iterating a query (or while systems are running), **structural changes** (spawn/despawn/add/remove) can throw.

Use:

- `world.cmd()` to defer changes
- `world.flush()` (or `world.update()`) to apply them safely

4.7.5 Example

```
for (const { e, c1: pos, c2: vel } of world.query(Position, Velocity)) {
    pos.x += vel.x;
    pos.y += vel.y;

    // Safe structural change: defer it
    if (pos.x > 10) world.cmd().despawn(e);
}
```

This pattern is recommended explicitly for queries.

January 6, 2026

January 4, 2026

4.8 Resources (Singletons / World Globals)

Resources are **typed singleton values stored on the `World`**, keyed by a `ComponentCtor<T>` (same "key shape" as components). They are **not attached to entities**.

They're ideal for global state like **Time**, **Input**, **Asset caches**, **Config**, **RNG**, **Selection**, etc.

4.8.1 Concepts

What is a Resource?

A resource is a **single instance of data** stored globally in the ECS `World`.

- **Components** → many per world, attached to entities
- **Resources** → one per key, stored in the world

Key type: `ComponentCtor<T>`

All resource APIs use:

```
ComponentCtor<T>
```

This usually means:

- a **class constructor** (e.g. `class TimeRes { ... }`)
- or a **token function** (unique function used as a key)

Keys are compared by **identity** (reference equality), not by name.

4.8.2 API summary

All methods live on `World` / `WorldApi`.

```
setResource<T>(key: ComponentCtor<T>, value: T): void
getResource<T>(key: ComponentCtor<T>): T | undefined
requireResource<T>(key: ComponentCtor<T>): T
hasResource<T>(key: ComponentCtor<T>): boolean
removeResource<T>(key: ComponentCtor<T>): boolean
initResource<T>(key: ComponentCtor<T>, factory: () => T): T
```

Structural safety: resource operations are **not structural changes** (unlike spawn/despawn/add/remove). They do not require flushing and are safe to call during system execution.

4.8.3 Method reference

`setResource<T>(key, value): void`

Stores (or replaces) the resource value for `key`.

Behavior

- Overwrites any existing value.
- Does not flush and does not affect archetypes.

Example

```
class ConfigRes { constructor(public difficulty: "easy" | "hard") {} }

world.setResource(ConfigRes, new ConfigRes("hard"));
```

`getResource<T>(key): T | undefined`

Returns the resource value if present, otherwise `undefined`.

Use when

- the resource is **optional** (debug tools, plugins, editor-only state)

Important note

- If you **explicitly store** `undefined` as the value, this also returns `undefined`.
- Use `hasResource(key)` to distinguish:
- “missing”
- vs “present but undefined”

Example

```
const debug = world.getResource(DebugRes);
if (debug) debug.enabled = true;
```

`requireResource<T>(key): T`

Returns the resource value if present, otherwise **throws**.

Use when

- the resource is **required** for correct operation (Time, Input, AssetCache, Config)

Throws

- `Error` if missing

Example

```
const input = w.requireResource(InputStateRes);
if (input.keysDown.has("KeyW")) { /* ... */ }
```

`hasResource<T>(key): boolean`

Checks whether an entry exists for `key`.

Use when

- you need to distinguish missing vs present-but-undefined
- you want conditional initialization

Example

```
if (!world.hasResource(TimeRes)) {
  world.setResource(TimeRes, new TimeRes());
}
```

`removeResource<T>(key): boolean`

Removes the resource entry for `key`.

Returns

- `true` if the entry existed and was removed
- `false` otherwise

Example

```
world.removeResource(DebugRes);
```

`initResource<T>(key, factory): T`

Insert-once helper.

Behavior

- If resource exists → returns existing value (factory is not called)
- If missing → calls `factory()`, stores, returns the new value

Use when

- bootstrapping default resources without double-init

Example

```
class TimeRes { dt = 0; elapsed = 0; }

world.initResource(TimeRes, () => new TimeRes());
```

4.8.4 Usage patterns

Pattern: “bootstrap required resources once”

```
class TimeRes { dt = 0; elapsed = 0; }
class InputStateRes { keysDown = new Set<string>(); }

world.initResource(TimeRes, () => new TimeRes());
world.initResource(InputStateRes, () => new InputStateRes());
```

Pattern: “systems read required resources”

```
function timeSystem(w: WorldApi, dt: number) {
  const time = w.requireResource(TimeRes);
  time.dt = dt;
  time.elapsed += dt;
}
```

Pattern: “asset cache resource”

```
class AssetCacheRes {
  images = new Map<string, HTMLImageElement>();
}

world.initResource(AssetCacheRes, () => new AssetCacheRes());
```

4.8.5 Gotchas

1) Keys must be stable and unique

Because keys are identity-based:

- ✅ `class TimeRes {}` used as key is stable
- ✅ a top-level `const TOKEN = (() => {}) as ComponentCtor<T>` is stable
- ❌ creating a new token function inline each time won’t match previous entries

2) Prefer `requireResource()` in gameplay systems

It keeps systems clean and fails fast when initialization is missing.

3) Resources are not entities

Do not use resources for data that should exist per-entity (that’s components).

January 14, 2026

January 9, 2026

4.9 Schedule

4.9.1 Purpose

`Schedule` is a **phase runner**: it groups systems under named phases, then runs those phases in a chosen order, calling `world.flush()` **between phases** to apply deferred structural commands deterministically.

4.9.2 Construction

```
const sched = new Schedule();
```

`Schedule` is independent from `World`; you pass the `World` (or compatible object) at run time.

4.9.3 Adding systems to phases

`add(phase: string, fn: SystemFn): this`

Registers a system function under a phase name.

- You can register multiple systems under the same phase.

Example:

```
sched
  .add("input",  (w: any) => { /* ... */ })
  .add("sim",    (w: any, dt) => { /* ... */ })
  .add("render", (w: any) => { /* ... */ });
```

4.9.4 Running phases

`run(world: WorldLike, dt: number, phases: string[]): void`

Runs the schedule for a single tick:

- Executes phases **in the exact order** provided by `phases`.
- Calls `world.flush()` **after each phase** (phase barrier).

Example:

```
const phases = ["input", "sim", "render"];
sched.run(world, 1/60, phases);
```

4.9.5 Flush semantics

`Schedule` relies on `world.flush()` to apply deferred structural changes queued via commands, enabling safe structural edits while systems and queries run.

4.9.6 Relationship to `World.update(dt)`

- `world.update(dt)` runs the world's own registered systems and flushes at the end.
- `Schedule` is used when you want **explicit phase ordering** and **flush points between groups of systems** rather than only at frame end.

January 4, 2026

January 4, 2026

4.10 Systems

4.10.1 Purpose

A **system** is a function executed by the ECS to update simulation state (usually by iterating queries and mutating component values). Systems are registered on the `World`, and executed during `world.update(dt)`.

4.10.2 System function type

`SystemFn`

A system is a function with the signature:

- `(world: WorldApi, dt: number) => void`

In practice, examples call `query()` and `cmd()` inside systems, which are available through `WorldApi`.

4.10.3 Registering systems

`world.addSystem(fn): this`

Adds a system to the world.

- Systems run in the **order they were added** (as described by "runs systems in order").

Example:

```
world.addSystem((w: any, dt: number) => {
    for (const { e, c1: pos, c2: vel } of w.query(Position, Velocity)) {
        pos.x += vel.x * dt;
        pos.y += vel.y * dt;

        if (pos.x > 10) w.cmd().despawn(e);
    }
});
```

4.10.4 Running systems (frame execution)

`world.update(dt): void`

Runs one ECS frame:

1. Runs all registered systems (in order)
2. Flushes queued commands at the end

The reference summary explicitly lists:

- `addSystem(fn): this`
- `update(dt): void` *(runs systems in order, then flushes)*

4.10.5 Structural changes inside systems

While systems are running (and while iterating queries), doing structural changes directly can throw. The recommended pattern is:

- enqueue structural changes with `world.cmd()`
- apply them with `world.flush()` (or let `update()` do it at the end)

4.10.6 Systems in phases (Schedule)

If you need explicit ordering across *groups* of systems, use `Schedule`:

- `sched.add(phase, systemFn)`
- `sched.run(world, dt, phases)` runs phases in order and calls `world.flush()` after each phase

This provides deterministic “phase barriers” where deferred commands are applied.

January 6, 2026

January 4, 2026

4.11 World

4.11.1 Purpose

`World` is the **central authority** of the ECS. It owns and coordinates:

- entity lifecycle
- archetypes and component storage
- queries
- deferred structural commands
- system execution

There is **exactly one `World` instance per ECS context**.

4.11.2 Construction

```
const world = new World();
```

Side effects

- Initializes an empty entity pool
- Initializes archetype storage
- Initializes command buffer
- Initializes system list

4.11.3 Entity Lifecycle API

`spawn(): Entity`

Creates a new entity immediately.

- Allocates a new entity id
- Marks entity as alive
- Places entity in the empty archetype

```
const e = world.spawn();
```

`spawnMany(...items: ComponentCtorBundleItem[]): Entity`

Creates a new entity along with its initial components immediately.

- `...items: ComponentCtorBundleItem[]` is the list of components to add to the newly created entity.
- Internally, it iterates over the items and calls `add` for each component.

`despawn(e: Entity): void`

Immediately removes an entity.

- Invalidates the entity handle (`gen` mismatch)
- Removes the entity from its archetype
- Frees the slot for reuse

Throws if:

- entity is stale or not alive

`despawnMany(entities: Entity[]): void`

Immediately removes multiple entities.

- `entities: Entity[]` is the list of entities to despawn.
- Internally, it iterates over the array and calls `despawn(e)` for each entity.

`isAlive(e: Entity): boolean`

Checks whether an entity handle is still valid.

```
if (world.isAlive(e)) { ... }
```

4.11.4 Component API

All component types are identified by **constructor identity**.

`has<T>(e: Entity, ctor: ComponentCtor<T>): boolean`

Checks if an entity has a component.

`get<T>(e: Entity, ctor: ComponentCtor<T>): T | undefined`

Returns the component value or `undefined`.

- Does **not** throw if missing
- Returns `undefined` for stale entities

`add<T>(e: Entity, ctor: ComponentCtor<T>, value: T): void`

Adds a component to an entity.

- **Structural change**
- Moves the entity to a different archetype

Throws if:

- entity is stale
- component already exists
- structural changes are forbidden (see iteration rules)

`addMany(e: Entity, ...items: ComponentCtorBundleItem[]): void`

Adding multiple components to an existing entity.

- `e: Entity` is the target entity.
- `...items: ComponentCtorBundleItem[]` is the list of components to add.
- Internally, it loops through the items and calls `add` for each component.

`remove<T>(e: Entity, ctor: ComponentCtor<T>): void`

Removes a component.

- **Structural change**
- Moves the entity to a different archetype

Throws if:

- entity is stale
- component does not exist
- structural changes are forbidden

`removeMany(e: Entity, ...ctors: ComponentCtor<any>[]): void`

Removes multiple component types from an entity.

- `e: Entity` is the target entity.
- `...ctors: ComponentCtor<any>[]` is the list of component constructors (types) to remove.
- Internally, it loops through the ctors and calls `remove` for each one.

`set<T>(e: Entity, ctor: ComponentCtor<T>, value: T): void`

Updates an existing component value.

- **Non-structural**
- Does not change archetypes

Throws if:

- entity is stale
- component does not exist

4.11.5 Query API

`query(...ctors): Iterable<QueryRow>`

Iterates entities that contain **all requested components**.

```
for (const { e, c1, c2 } of world.query(A, B)) {
    // e  -> Entity
    // c1 -> A
    // c2 -> B
}
```

PROPERTIES

- Iterates archetypes, not entities
- Components are returned as `c1`, `c2`, … in **argument order**
- Query iteration **locks structural changes**

4.11.6 Structural Change Rules

While iterating a query or running systems:

- ❌ `spawn`, `despawn`, `add`, `remove` are forbidden
- ✔️ `get`, `set`, `has` are allowed

Violations throw a runtime error.

4.11.7 Command Buffer API

`cmd(): Commands`

Returns a command buffer for **deferred structural changes**.

```
world.cmd().despawn(e);
```

Commands are **queued**, not applied immediately.

`flush(): void`

Applies all queued commands.

- Safe to call after queries
- Automatically called by `update()` and `Schedule`

4.11.8 System API

`addSystem(fn: SystemFn): this`

Registers a system.

```
world.addSystem((w, dt) => { ... });
```

Systems are executed **in insertion order**.

`update(dt: number): void`

Runs one ECS frame.

Execution order:

1. Run all systems
2. Flush deferred commands

```
world.update(1 / 60);
```

4.11.9 Events API

`emit<T>(key: ComponentCtor<T>, ev: T): void`

Emits an event of type `T` into the current phase write buffer.

`events<T>(key: ComponentCtor<T>): EventChannel<T>`

Returns the event channel for `key`, creating it if missing.

`drainEvents<T>(key: ComponentCtor<T>, fn: (ev: T) => void): void`

Drains readable events for the given type.

Behavior

- If the channel doesn't exist yet, it's a **no-op** (does not allocate/create)

`clearEvents<T>(key?: ComponentCtor<T>): void`

Clears readable events.

- If `key` is provided: clears that event type's **read buffer**
- If omitted: clears the **read buffers of all** event types

`swapEvents(): void` **(internal / schedule boundary)**

Swaps all event channels' buffers. Called by `Schedule` at phase boundaries.

Required schedule behavior At each phase boundary:

```
world.flush();
world.swapEvents();
```

4.11.10 Internal Guarantees

- Archetypes use **Structure of Arrays (SoA)**
- Entity handles are generation-safe
- Component lookups are O(1) per archetype row
- Queries are archetype-filtered, not entity-scanned

4.11.11 Error Conditions (Summary)

Operation	Error Condition
add / remove	during query iteration
add	component already exists
remove	component missing
set	component missing
any	stale entity

4.11.12 Design Constraints

- Single-threaded
- No automatic conflict detection
- No parallel systems
- No borrowing model

These are **intentional** for simplicity and predictability.

January 13, 2026

January 4, 2026

5. Tutorials

5.1 Tutorial 1 — Your first ECS World

Outcome: you'll run a tiny simulation loop where entities with `Position` + `Velocity` move over time, using `World`, `spawn`, `add`, `query`, `addSystem`, and `update(dt)`.

5.1.1 1) What is an ECS? (one sentence)

ECS is a way to build simulations where **entities are IDs**, **components are data**, and **systems are functions that iterate entities with specific components**.

5.1.2 2) Create a tiny project

```
mkdir ecs-tutorial-1
cd ecs-tutorial-1
npm init -y
npm i archetype-ecs-lib
npm i -D typescript tsx
```

Install is `npm i archetype-ecs-lib`.

5.1.3 3) Create `tutorial1.ts`

Create a file named `tutorial1.ts` with this code:

```
import { World, WorldApi } from "archetype-ecs-lib";

// 1) Components = data (any class can be a component type)
class Position { constructor(public x = 0, public y = 0) {} }
class Velocity { constructor(public x = 0, public y = 0) {} }

// 2) Create a World (owns entities, components, systems)
const world = new World();

// 3) Spawn an entity and add components
const e = world.spawnMany(
  new Position(0, 0, 0),
  new Velocity(2, 0)// 2 units/sec along x
)

// 4) Add a system (runs each update)
world.addSystem((w, dt) => {
  for (const { e, c1: pos, c2: vel } of w.query(Position, Velocity)) {
    pos.x += vel.x * dt;
    pos.y += vel.y * dt;
  }
});

// 5) Run a small simulation loop (60 frames)
const dt = 1 / 60;

for (let frame = 1; frame <= 60; frame++) {
  world.update(dt);

  // Read back Position and print it
  const pos = world.get(e, Position)!;
  if (frame % 10 === 0) {
    console.log(`frame ${frame}: x=${pos.x.toFixed(2)} y=${pos.y.toFixed(2)}`);
  }
}
```

This uses the documented API:

- `spawn()` , `add(e, Ctor, value)`
- `addSystem(fn)`
- `query(Position, Velocity)` yielding `{ e, c1, c2 }`
- `update(dt)` to run systems each tick

5.1.4 4) Run it

```
npx tsx tutorial1.ts
```

You should see something like:

- `frame 10: x=0.33 ...`
- `frame 60: x=2.00 ...`

(Your exact decimals may differ slightly depending on rounding.)

5.1.5 5) You've built the core loop

You now have:

- a `World`
- entities created with `spawn()`
- components added with `add()`
- a system iterating `query(...)`
- a running simulation driven by `update(dt)`

January 8, 2026

January 4, 2026

5.2 Tutorial 2 — Components & archetypes

Outcome: you'll *see* how component sets automatically form **archetypes (tables)**, and how entities "move" between them when you `add()` / `remove()` components—without digging into internals. Archetypes store data in **SoA** (one column per component type).

5.2.1 1) Define a few component types

Create `tutorial2.ts`:

```
import { World } from "archetype-ecs-lib";

// Components are just data classes
class Position { constructor(public x = 0, public y = 0) {} }
class Velocity { constructor(public x = 0, public y = 0) {} }
class Health   { constructor(public hp = 100) {} }
```

The ECS uses component constructors as the "type key", and archetypes store entities in SoA tables.

5.2.2 2) Create a World and spawn entities with different component sets

```
const world = new World();

// e1 has: Position
const e1 = world.spawn();
world.add(e1, Position, new Position(1, 1));

// e2 has: Position + Velocity
const e2 = world.spawn();
world.add(e2, Position, new Position(0, 0));
world.add(e2, Velocity, new Velocity(1, 0));

// e3 has: Health
const e3 = world.spawn();
world.add(e3, Health, new Health(50));
```

5.2.3 3) Add a tiny helper to "see" matches

We can't (and don't need to) access archetype tables directly. Instead, we observe *which queries match*, before and after structural changes.

```
function ids(iter: Iterable<{ e: { id: number } }>): number[] {
  const out: number[] = [];
  for (const row of iter) out.push(row.e.id);
  return out.sort((a, b) => a - b);
}

function dump(label: string) {
  console.log(`\n=== ${label} ===`);
  console.log("Position:", ids(world.query(Position)));
  console.log("Velocity:", ids(world.query(Velocity)));
  console.log("Health:  ", ids(world.query(Health)));
  console.log("Pos+Vel: ", ids(world.query(Position, Velocity)));
  console.log("Pos+HP:  ", ids(world.query(Position, Health)));
}
```

The query API yields `{ e, c1, c2, ... }` rows in the order you request components.

5.2.4 4) Observe the “automatic archetypes” effect

Add this and run once:

```
dump("initial");
```

You’ll see (by IDs) that:

- `e1` matches `Position` only
- `e2` matches both `Position` and `Pos+Vel`
- `e3` matches `Health` only

What this demonstrates: entities with the **same component set** are stored together (same archetype). Archetypes are created implicitly as you introduce new component combinations.

5.2.5 5) Make an entity “move” between archetypes (add)

Now **add** a component to `e1`:

```
world.add(e1, Velocity, new Velocity(0, 2));
dump("after: add Velocity to e1");
```

You should see:

- `e1` now appears in `Velocity`
- and also in `Pos+Vel`

Why: `add()` is a **structural change** that can move an entity into a different archetype table (because its component set changed).

5.2.6 6) Make an entity “move” between archetypes (remove)

Now **remove** `Position` from `e2`:

```
world.remove(e2, Position);
dump("after: remove Position from e2");
```

You should see:

- `e2` disappears from `Position` and `Pos+Vel`
- `e2` still appears in `Velocity`

Again: `remove()` is structural and can move the entity to a new archetype.

5.2.7 7) Run it

```
npx tsx tutorial2.ts
```

5.2.8 What you just learned (by doing)

- Components are plain data types (classes).
- Archetypes (tables) are created automatically for each distinct component set, stored in **SoA** layout.

- When you `add()` / `remove()` components, entities "move" because their component set changes (structural change).

Note for later tutorials: structural changes can be unsafe while iterating; that's why `cmd()` + `flush()` exist.

January 14, 2026

January 4, 2026

5.3 Tutorial 3 — Deferred structural changes

Outcome: you'll learn the one rule that prevents most ECS bugs: **don't change entity structure while iterating**. You'll reproduce the problem safely, then fix it using **Commands** and **flush points** (via `Schedule`). The library explicitly supports this workflow: defer structural operations with `world.cmd()` and apply them with `world.flush()` / `Schedule` phase boundaries.

5.3.1 1) Create `tutorial4.ts`

```
import { World, WorldApi, Schedule } from "archetype-ecs-lib";
```

5.3.2 2) Define simple components

```
class Position { constructor(public x = 0) {} }
class Velocity { constructor(public x = 0) {} }
```

5.3.3 3) Setup: spawn a few movers

```
const world = new World();

function spawnMover(x: number, vx: number) {
  const e = world.spawn();
  world.add(e, Position, new Position(x));
  world.add(e, Velocity, new Velocity(vx));
  return e;
}

spawnMover(0,  2);
spawnMover(5, -3);
spawnMover(9,  1);
```

This is standard structural usage: `spawn()` + `add()`.

5.3.4 4) The unsafe thing (don't do this)

Add this function:

```
const unsafeDespawnInsideQuery: SystemFn = (w: WorldApi) => {
  for (const { e, c1: pos } of w.query(Position)) {
    if (pos.x > 8) {
      // ❌ Structural change during iteration (may throw)
      w.despawn(e);
    }
  }
}
```

Now call it once (inside a try/catch so the tutorial keeps going):

```
try {
  unsafeDespawnInsideQuery(world);
  console.log("unsafe: no error (but still not safe)");
} catch (err: any) {
  console.log("unsafe: error as expected ->", String(err.message ?? err));
}
```

The lib will warn that structural changes during query iteration can throw and instructs to use `cmd()` + `flush()` instead.

5.3.5 5) The safe fix: use Commands

Replace the unsafe function with a safe one:

```
const safeDespawnInsideQuery: SystemFn = (w: WorldApi) => {
  for (const { e, c1: pos } of w.query(RenderContextComponent)) {
    if (pos.x > 8) {
      // ✅ Defer structural change
      w.cmd().despawn(e);
    }
  }
}
```

Commands let you queue:

- `spawn`, `despawn`, `add`, `remove`

5.3.6 6) Apply commands at a flush point

Option A — Manual flush

```
safeDespawnInsideQuery(world);
world.flush(); // apply queued despawns
```

`flush()` applies queued commands (and `update()` also flushes automatically at the end).

Option B — Flush at phase boundaries (recommended)

Use `Schedule`, which flushes after each phase:

```
const sched = new Schedule();

sched.add("sim", (w: WorldApi) => {
  // move
  for (const { c1: pos, c2: vel } of w.query(Position, Velocity)) {
    pos.x += vel.x;
  }
});

sched.add("cleanup", (w: WorldApi) => {
  // safely despawn based on updated positions
  safeDespawnInsideQuery(w);
});

// Flush happens after each phase automatically
const phases = ["sim", "cleanup"];
```

`Schedule.run(world, dt, phases)` runs phases and calls `world.flush()` after each phase.

5.3.7 7) Run a few ticks and print what's left

Add a small logger:

```
function logPositions(w: WorldApi, label: string) {
  const items: string[] = [];
  for (const { e, c1: pos } of w.query(Position)) {
    items.push(`e${e.id}:${pos.x.toFixed(1)}`);
  }
  console.log(label, items.join(" | ") || "(none)");
}
```

Now run:

```
logPositions(world, "before");

for (let i = 0; i < 5; i++) {
  sched.run(world, 0, phases);
  logPositions(world, `after tick ${i + 1}`);
}
```

5.3.8 8) Full file (copy/paste)

```
import { World, Schedule } from "archetype-ecs-lib";

class Position { constructor(public x = 0) {} }
class Velocity { constructor(public x = 0) {} }

const world = new World();

function spawnMover(x: number, vx: number) {
  const e = world.spawn();
  world.add(e, Position, new Position(x));
  world.add(e, Velocity, new Velocity(vx));
  return e;
}

spawnMover(0,  2);
spawnMover(5, -3);
spawnMover(9,  1);

const unsafeDespawnInsideQuery: SystemFn = (w) => {
  for (const { e, c1: pos } of w.query(Position)) {
    if (pos.x > 8) {
      w.despawn(e); // ❌ may throw
    }
  }
}

try {
  unsafeDespawnInsideQuery(world as any);
  console.log("unsafe: no error (but still not safe)");
} catch (err: any) {
  console.log("unsafe: error as expected ->", String(err.message ?? err));
}

const safeDespawnInsideQuery: SystemFn = (w) => {
  for (const { e, c1: pos } of w.query(Position)) {
    if (pos.x > 8) w.cmd().despawn(e); // ✅ deferred
  }
}

function logPositions(w: WorldApi, label: string) {
  const items: string[] = [];
  for (const { e, c1: pos } of w.query(Position)) {
    items.push(`e${e.id}:${pos.x.toFixed(1)}`);
  }
  console.log(label, items.join(" | ") || "(none)");
}

const sched = new Schedule();

sched.add("sim", (w: WorldApi) => {
  for (const { c1: pos, c2: vel } of w.query(Position, Velocity)) {
    pos.x += vel.x;
  }
});

sched.add("cleanup", (w: WorldApi) => {
  safeDespawnInsideQuery(w);
});

const phases = ["sim", "cleanup"];

logPositions(world, "before");
for (let i = 0; i < 5; i++) {
  sched.run(world, 0, phases); // flush after each phase
  logPositions(world, `after tick ${i + 1}`);
}
```

5.3.9 9) Run it

```
npx tsx tutorial4.ts
```

You’ll see:

- the unsafe version may throw (depending on timing/guarding)
- the safe version consistently despawns entities after they cross the threshold
- phase flush points make the timing predictable

January 14, 2026

January 4, 2026

5.4 Tutorial 4 — Writing systems

Outcome: you'll write real gameplay logic as **systems**: query components, mutate data safely, and run everything through a **Schedule** (`input → sim → cleanup`) with automatic `flush()` between phases.

5.4.1 1) Create `tutorial3.ts`

```
import { World, WorldApi, Schedule, SystemFn } from "archetype-ecs-lib";
```

The lib exports `World` and `Schedule`.

5.4.2 2) Define components (data only)

```
class Position { constructor(public x = 0, public y = 0) {} }
class Velocity { constructor(public x = 0, public y = 0) {} }
class Lifetime { constructor(public seconds = 1.0) {} } // despawn when <= 0
```

5.4.3 3) Create a World and spawn a few entities

```
const world = new World();

function spawnMover(x: number, y: number, vx: number, vy: number, life = 2.0) {
  const e = world.spawn();
  world.add(e, Position, new Position(x, y));
  world.add(e, Velocity, new Velocity(vx, vy));
  world.add(e, Lifetime, new Lifetime(life));
  return e;
}

spawnMover(0, 0,  2, 0, 1.2);
spawnMover(0, 1,  1, 0, 2.5);
spawnMover(0, 2, -1, 0, 0.8);
```

This uses the documented structural ops: `spawn()` and `add()`.

5.4.4 4) System function signature (what you write)

A system is a function called like:

- `(world, dt) => void`

Systems are added using `world.addSystem()` like `world.addSystem((w: WorldApi, dt: number) => ...)`.

In this tutorial we'll register systems on a `Schedule` (phases), but the function shape is the same.

5.4.5 5) Write your first real system: movement

This system queries `Position + Velocity` and updates positions.

```
const movementSystem: SystemFn = (w: WorldApi, dt: number) => {
  for (const { c1: pos, c2: vel } of w.query(Position, Velocity)) {
    pos.x += vel.x * dt;
    pos.y += vel.y * dt;
  }
}
```

Query rows provide `{ e, c1, c2, ... }` in the same order as the query arguments.

5.4.6 6) Mutating data safely: despawn using commands

Despawning is a **structural change**, so do it through `cmd()` inside systems.

```
const lifetimeSystem: SystemFn = (w: WorldApi, dt: number) => {
  for (const { e, c1: life } of w.query(Lifetime)) {
    life.seconds -= dt;
    if (life.seconds <= 0) {
      w.cmd().despawn(e); // safe: deferred
    }
  }
}
```

5.4.7 7) Add a small “cleanup / log” system

We’ll print positions so you can see it running. This does not do structural changes.

```
const logSystem: SystemFn = (w: WorldApi, dt: number) => {
  const lines: string[] = [];
  for (const { e, c1: pos } of w.query(Position)) {
    lines.push(`e${e.id} @ (${pos.x.toFixed(2)}, ${pos.y.toFixed(2)})`);
  }
  console.log(`frame ${frame}: ${lines.join(" | ")}`);
}
```

5.4.8 8) Run systems via Schedule (phases)

1. Create a schedule
2. Register systems under phases
3. Run phases each tick

```
const sched = new Schedule();

sched.add("sim", movementSystem);
sched.add("sim", lifetimeSystem);

// log in a separate phase so structural changes are already flushed
let frameNo = 0;
sched.add("cleanup", (w: WorldApi) => {
  frameNo++;
  logSystem(w, frameNo);
});

const phases = ["sim", "cleanup"];
```

`Schedule.run(world, dt, phases)` runs phases in order and calls `world.flush()` after each phase.

5.4.9 9) Run the loop

```
const dt = 1 / 10; // bigger dt so it’s easy to see
for (let i = 0; i < 20; i++) {
  sched.run(world, dt, phases);
}
```

5.4.10 10) Full file (copy/paste)

```
import { World, WorldApi Schedule, SystemFn } from "archetype-ecs-lib";

class Position { constructor(public x = 0, public y = 0) {} }
class Velocity { constructor(public x = 0, public y = 0) {} }
class Lifetime { constructor(public seconds = 1.0) {} }

const world = new World();

function spawnMover(x: number, y: number, vx: number, vy: number, life = 2.0) {
  const e = world.spawn();
  world.add(e, Position, new Position(x, y));
  world.add(e, Velocity, new Velocity(vx, vy));
  world.add(e, Lifetime, new Lifetime(life));
  return e;
}

spawnMover(0, 0,  2, 0, 1.2);
spawnMover(0, 1,  1, 0, 2.5);
spawnMover(0, 2, -1, 0, 0.8);

const movementSystem: SystemFn = (w: WorldApi, dt: number) => {
  for (const { c1: pos, c2: vel } of w.query(Position, Velocity)) {
    pos.x += vel.x * dt;
    pos.y += vel.y * dt;
  }
}

const lifetimeSystem: SystemFn = (w: WorldApi, dt: number) => {
  for (const { e, c1: life } of w.query(Lifetime)) {
    life.seconds -= dt;
    if (life.seconds <= 0) w.cmd().despawn(e);
  }
}

const logSystem: SystemFn = (w: WorldApi, dt: number) => {
  const lines: string[] = [];
  for (const { e, c1: pos } of w.query(Position)) {
    lines.push(`e${e.id} @ (${pos.x.toFixed(2)}, ${pos.y.toFixed(2)})`);
  }
  console.log(`frame ${frame}: ${lines.join(" | ")}`);
}

const sched = new Schedule();
sched.add("sim", movementSystem);
sched.add("sim", lifetimeSystem);

let frameNo = 0;
sched.add("cleanup", (w: WorldApi) => {
  frameNo++;
  logSystem(w, frameNo);
});

const phases = ["sim", "cleanup"];

const dt = 1 / 10;
for (let i = 0; i < 20; i++) {
  sched.run(world, dt, phases);
}
```

5.4.11 11) Run it

```
npx tsx tutorial3.ts
```

You'll see entities moving, then disappearing as their `Lifetime` reaches 0 (despawned safely via commands + phase flush).

January 14, 2026

January 4, 2026

5.5 Tutorial 5 — ECS + Three.js (render-sync + safe spawn/despawn)

Outcome: you'll see **moving cubes** in Three.js. You'll also **spawn** new cubes on click and **despawn** them safely using `cmd()` + **phase flush boundaries** (via `Schedule`).

5.5.1 1) Create a new project

```
mkdir ecs-threejs-tutorial
cd ecs-threejs-tutorial
npm init -y

npm i archetype-ecs-lib three
npm i -D vite typescript
```

The ECS package is installed as `archetype-ecs-lib`.

5.5.2 2) Add `index.html`

Create `index.html`:

```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>ECS + Three.js Tutorial</title>
    <style>
      html, body { margin: 0; height: 100%; overflow: hidden; }
      #hud {
        position: fixed; left: 12px; top: 12px;
        padding: 8px 10px; border-radius: 8px;
        background: rgba(0,0,0,0.55); color: #fff;
        font-family: system-ui, sans-serif; font-size: 13px;
        user-select: none;
      }
    </style>
  </head>
  <body>
    <div id="hud">Click to spawn cubes</div>
    <script type="module" src="/src/main.ts"></script>
  </body>
</html>
```

5.5.3 3) Add `src/main.ts`

Create `src/main.ts`: