EventBus Pro — Decoupled Event System

Overview

Thank you for downloading EventBus Pro!
This system provides a modular, decoupled, and highly extensible event-driven architecture for Unity projects. It enables robust communication between game objects and systems, with full runtime and editor support, and comes with a suite of sample scenes demonstrating best practices and advanced use cases.

Support

For support, visit: https://zdev-studios.com/contact or use the Unity Asset Store portal.

Table of Contents

Requirements

- Unity **2021.3 LTS** or newer (compatible with Unity 6+)
- No external dependencies

Quick Start

1. Import the Package

Import the EventBus asset into your project via the Unity Package Manager or by dragging the folder into your Assets directory.

2. Add the EventBus to Your Scene

- The EventBus is a static system and does not require a GameObject by default.
- To use, simply call:

```
// Define your event type
public struct MyEvent : IEvent { public int value; }
```

```
// Subscribe
EventBus.Subscribe<MyEvent>(OnMyEvent);

// Publish
EventBus.Publish(new MyEvent { value = 42 });

// Handler
void OnMyEvent(MyEvent evt) { Debug.Log(evt.value); }
```

- Unsubscribe when no longer needed:

```
EventBus.Unsubscribe<MyEvent>(OnMyEvent);
```

3. Explore the Samples

Open any sample scene from `Assets/ZDev-Assets/EventBus/Samples/` and press Play to see the system in action:

- **ArenaDemo.unity** — Advanced gameplay event flow
- **BasicEvents.unity** — Minimal event usage
- **StressTest.unity** — High-frequency event stress test
- **ThreadedEvents.unity** — Multithreaded event dispatch

Samples Overview

Each sample demonstrates a different use case:

- **Arena**: Event-driven player, drone, and HUD logic. Shows decoupled gameplay systems and prefab event handling.
- **Basic**: Simple publisher/subscriber pattern for learning fundamentals.
- **StressTest**: Simulates heavy event traffic to test performance and stability.
- **Threaded**: Demonstrates safe event dispatch from background threads.

All samples include full source code and are ready to run.

Feature Set

- **Decoupled Communication**: Publishers and subscribers do not need references to each other.
- **Type-Safe Events**: Strongly-typed event structs/classes via the `IEvent` interface.
- **Centralized Management**: All event flow is routed through a single, static bus.
- **Editor Tooling**: Includes a Debug Window for live event monitoring and diagnostics.
- **Thread-Safe Option**: Supports safe event dispatch from background threads (see Threaded sample).
- **Minimal Boilerplate**: Simple API for subscribing, publishing, and unsubscribing.
- **Assembly Definitions**: Modular asmdef files for fast compile times and clean project structure.
- **Full Source**: 100% C# source code

Advanced Usage

Custom Event Queues

Use `EventQueue<T>` for deferred, thread-safe, or async event processing. Events enqueued from any thread are dispatched to subscribers on the main thread next frame.

```
// Enqueue from a background thread
await Task.Run(() => EventQueue<DataReadyEvent>.Enqueue(new DataReadyEvent { data = result

// Subscribe a persistent handler with an optional filter
EventQueue<DataReadyEvent>.Subscribe(OnDataReady, filter: (in DataReadyEvent e) => e.IsVal

// Async-await the next matching event on the main thread
var evt = await EventQueue<DataReadyEvent>.WaitForAsync(destroyCancellationToken);
```

Cross-Thread Publish

Publish events to the main-thread bus from any background thread using `PublishFromThread`. Events are buffered and dispatched during the next frame's drain step.

```
// Safe to call from Task.Run, background threads, or Jobs
await Task.Run(() => EventBus<MyEvent>.PublishFromThread(new MyEvent { value = 99 }));
```

Owner-Tracked Subscriptions

Pass an owner reference to auto-remove a binding when the owner is garbage-collected, preventing stale handlers and memory leaks without an explicit Unsubscribe call.

```
// Binding is automatically removed when 'this' is GC'd
EventBus<PlayerDamagedEvent>.Subscribe(
    owner:   this,
    handler: OnDamaged,
    filter:  (in PlayerDamagedEvent e) => e.playerId == myId
);
```

Lifecycle Hooks

`EventBusLifecycle` is fully automatic — it injects a sweep step into Unity's PlayerLoop at startup (`BeforeSceneLoad`) to remove dead bindings and drain cross-thread queues every frame. You do not need to reference it in user code.

```
// No setup required — lifecycle is initialized automatically:
// [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
// The sweep runs each Update frame, cleaning up GC'd owner bindings.
```

Editor Integration

Open the Debug Window at any time via **Tools → Event Bus Debugger** in the Unity Editor. It displays all discovered `IEvent` types, live subscriber counts, and a rolling publish log (Editor and Development builds only).

```
// The window is opened from the menu — no code required:
// Tools → Event Bus Debugger
//
```

```
// To bridge editor-side events, subscribe in an Editor script:
EventBus<MyEditorEvent>.Subscribe(OnEditorEvent);
```

Arena Sample

The Arena sample demonstrates advanced patterns including decoupled player, drone, and HUD systems communicating entirely through events — no direct component references required. Open `Samples/Arena/ArenaDemo.unity` and study `ArenaEvents.cs` as a starting point for organizing your own event types.

Troubleshooting

No events received

- Ensure you are subscribing **before** publishing — there is no event history replay.
- Confirm the event struct type matches exactly between publisher and subscriber. Generic bus instances are per-type (`EventBus<MyEvent>` and `EventBus<OtherEvent>` are independent).
- If using an `EventFilter`, verify the filter predicate is not excluding all events.
- Check that `Unsubscribe` has not already been called before the event is published.

Handler fires after the subscribing object is destroyed

- Use the owner-tracked overload: `EventBus<T>.Subscribe(owner: this, handler: OnEvent)`. The binding is automatically removed when the owner is garbage-collected.
- Alternatively, call `EventBus<T>.Unsubscribe(binding)` explicitly in `OnDestroy` or `OnDisable`.

Cross-thread or queued events are not dispatched

- `EventBusLifecycle` injects the drain step automatically at startup. If you see a warning in the Console (`Could not inject sweep system into PlayerLoop`), the drain step did not register — ensure no code is stripping the PlayerLoop before `BeforeSceneLoad`.
- Events published with `PublishFromThread` or enqueued with `EventQueue<T>.Enqueue` are dispatched on the **next frame**, not immediately. Do not expect synchronous delivery.
- Use the Threaded sample (`Samples/Threaded/ThreadedEvents.unity`) as a reference for correct usage.

Editor Debug Window not showing events

- Open via **Tools → Event Bus Debugger** in the Unity Editor menu.
- The rolling publish log is only populated in **Editor and Development builds**. It will be empty in Release/Shipping builds by design.
- If subscriber counts show 0 for all types, no events have been subscribed yet in this session — press Play and interact with the scene.

Memory leaks / subscriber count keeps growing

- Avoid subscribing in `Update` or other frequently-called methods without a matching unsubscribe.
- Use owner-tracked bindings for MonoBehaviours so the sweep system can automatically clean up destroyed objects.
- Monitor live subscriber counts in the **Event Bus Debugger** window to identify leaking buses.

For more details, see the inline XML documentation in each script or contact support for advanced integration help.