6 · The Hosted Worker

For two volumes the outbox has been a table that fills and never empties. Volume II built its write half with care — every domain event committed in the same transaction as the state that raised it, so that a crash can neither lose an event nor invent one — and Part I of this volume proved that half correct under a race, an adversarial ordering, and a modelled crash. What has never existed is the thing that reads the table and carries its events onward. This chapter builds it: a hosted worker that runs for the life of the process, wakes on an interval, hands every undelivered row to the in-process publisher in the order the events occurred, and marks each done. It is deliberately the skeleton and not the whole animal. It does not yet retry a failed delivery or set aside a message that cannot be delivered — that is Chapter 7 — and it does not yet wire the events to any real subscriber — that is Chapter 8. What it establishes is the loop itself, correct in its lifecycle and its ordering, and proven on the harness Part I just finished.

What a hosted worker is

.NET's generic host runs a set of long-lived services alongside the application. A `BackgroundService` is the simplest of them: the host calls its `ExecuteAsync` once at startup and hands it a cancellation token that is triggered at shutdown, and the service is expected to loop until that token fires and then return promptly so the host can stop cleanly. The dispatcher is such a service. One subtlety governs its whole design and is the source of the most common bug in the pattern: a hosted service is a singleton, created once and living forever, while a `DbContext` is scoped, meant to live for one unit of work. A singleton may never hold a scoped dependency, so the dispatcher takes an `IServiceScopeFactory` and opens a fresh scope on every pass, resolving its database and publisher from that scope and letting them be disposed when the pass ends.

```
namespace Nexus.Infrastructure.Outbox;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.DependencyInjection;

/// <summary>The drain half Volume II described and did not build. It
/// wakes on an interval, hands every undelivered outbox row to the
/// in-process publisher in occurrence order, and marks it done.
/// Retry and backoff are Chapter 7; real subscribers, Chapter 8.</summary>
public sealed class OutboxDispatcher(
    IServiceScopeFactory scopes,
    ILogger<OutboxDispatcher> log) : BackgroundService
{
    private static readonly TimeSpan Interval =
        TimeSpan.FromSeconds(1);
    private const int BatchSize = 64;

    protected override async Task ExecuteAsync(CancellationToken stop)
    {
        while (!stop.IsCancellationRequested)
        {
            try { await DrainOnce(stop); }
            catch (Exception ex) { log.LogError(ex, "drain failed"); }

            try { await Task.Delay(Interval, stop); }
            catch (OperationCanceledException) { break; } // shutdown
        }
    }
}
```

Listing 6.1 — OutboxDispatcher.cs, the loop. The try/catch around DrainOnce keeps one bad pass from killing the worker — Chapter 7 makes that failure handling precise rather than blunt — and the catch on the delay turns shutdown cancellation into a clean exit rather than a logged error.

Draining a batch, in order

Each pass opens a scope, reads a bounded batch of undelivered rows in the order they occurred, and delivers them one at a time. The order is not incidental: an event that supersedes a policy must not reach a subscriber before the event that activated it, so the outbox is drained by the monotonic sequence Volume II stamped on each row. And the two steps within a row are deliberately ordered publish-then-mark, each row saved on its own, so that a crash can at worst redeliver a row, never lose one — a choice the Deep Dive returns to.

```
    private async Task DrainOnce(CancellationToken ct)
    {
        using var scope = scopes.CreateScope();
        var db = scope.ServiceProvider
            .GetRequiredService<NexusDbContext>();
        var publisher = scope.ServiceProvider
            .GetRequiredService<IOutboxPublisher>();

        var batch = await db.Outbox
            .Where(m => !m.Delivered)
            .OrderBy(m => m.Sequence)      // the order they occurred
            .Take(BatchSize)
            .ToListAsync(ct);

        foreach (var message in batch)
        {
            await publisher.Publish(message, ct);      // deliver, then
            message.MarkDelivered(DateTimeOffset.UtcNow);
            await db.SaveChangesAsync(ct);             // record it
        }
    }
```

Listing 6.2 — DrainOnce, complete. A scope per pass keeps the singleton worker clear of the scoped DbContext; the ordered query drains events as they happened; and one SaveChanges per row bounds any crash to a single redelivery. In this chapter IOutboxPublisher deserializes the event and hands it to the in-process dispatcher, which — until Chapter 8 — has no subscribers registered, so “delivered” means “handed over,” not “reacted to.”

A host to run it

In production the dispatcher runs in its own host — the Worker project this chapter adds — so it can be deployed and scaled apart from the API. The host is small, because all the wiring it needs is the Infrastructure registration Volume II already wrote:

```
// Nexus.Worker/Program.cs -- a dedicated host for the dispatcher.
var builder = Host.CreateApplicationBuilder(args);
builder.Services.AddNexusInfrastructure(builder.Configuration);
builder.Services.AddHostedService<OutboxDispatcher>();
builder.Build().Run();
```

Listing 6.3 — Nexus.Worker/Program.cs, complete. Nothing in the existing layers changes; the Worker is new outer-ring code that composes the same Infrastructure and adds one hosted service. The harness runs the very same dispatcher in process, which is how the next listing proves it.

Proven on the harness

The dispatcher is proven with the instrument Part I built. A real command through the API — acknowledging an alarm — writes one outbox row; the dispatcher drains it; the row is delivered, in order, and stamped. Two tests establish it: one drives a single pass deterministically, and one starts the real hosted loop and waits for it to drain without being asked.

```
public sealed class OutboxDispatcherTests(SqlServerFixture sql)
    : ApiTest(sql)
{
    [Fact]
    public async Task A_written_event_is_drained_and_marked_done()
    {
        var okafor  = await SeedOperator("A. Okafor",
                                         Roles.ShiftSupervisor);
        var alarmId = await ArrangeRaisedAlarm("unit-2",
                                               Severity.Critical);
        await new Night(Client).Acknowledge(   // writes one outbox row
            alarmId, TokenService.Issue(okafor), Key());

        await Sql.DrainOutboxOnce();   // one deterministic pass

        await using var db = Sql.NewContext();
        var row = await db.Outbox.SingleAsync();
        Assert.True(row.Delivered);
        Assert.NotNull(row.DeliveredOn);
    }

    [Fact]
    public async Task The_hosted_loop_drains_without_being_asked()
    {
        await using var host = Sql.StartWorkerHost(); // real loop
        var okafor  = await SeedOperator("A. Okafor",
                                         Roles.ShiftSupervisor);
        var alarmId = await ArrangeRaisedAlarm("unit-2",
                                               Severity.Critical);
        await new Night(Client).Acknowledge(
            alarmId, TokenService.Issue(okafor), Key());

        await Eventually(async () =>          // poll up to a timeout
        {
            await using var db = Sql.NewContext();
            return await db.Outbox.AllAsync(m => m.Delivered);
        });
    }
}
```

Listing 6.4 — The dispatcher, proven two ways. DrainOutboxOnce resolves the worker and runs a single pass, so the first test is deterministic; StartWorkerHost runs the real ExecuteAsync loop over the container, so the second proves the loop itself wakes and drains. DrainOnce is exposed to the harness through InternalsVisibleTo, not made public.

Run the suite:

```
$ dotnet test tests/Nexus.IntegrationTests
Passed! - Failed: 0, Passed: 36, Skipped: 0, Duration: 48.7 s
```

Thirty-six integration tests, six of them the dispatcher's first. The full solution — now including the Worker project — reports 150 green: 114 from Volume II, untouched, plus the thirty-six the harness has earned.

HONEST BOUNDARY

Four deferrals shaped this chapter. The worker delivers, but it does not yet handle a delivery that fails: a handler that threw would fall to the loop's blunt catch and be retried on the next pass forever — Chapter 7 adds real retry with backoff and quarantines the poison message that can never succeed. There are no real subscribers yet: publishing reaches an in-process dispatcher with nothing registered, so “delivered” means “handed over”; the reactions the Domain has described since Volume I are wired in Chapter 8, where Volume I's waiting rule finally goes live. Exactly-once under crash is not proven here: this chapter delivers at-least-once, and making that exactly-once when the dispatcher itself dies mid-delivery is Chapter 9. And a single worker drains the outbox; running several in parallel would need rows claimed so two workers never process one event, which is a horizontal-scaling concern named and left to a later phase, if ever — the demonstrator ships one.

The outbox is no longer a table that only fills. A hosted worker drains it, in order, for the life of the process, and stops cleanly when the host does — all of it proven on Part I's harness. What the worker cannot yet do is fail well. The next chapter makes it robust: a delivery that throws is retried on an exponential backoff rather than hammered, and a message that can never be delivered is set aside so that one poison row cannot wedge the whole queue behind it.

The Loop That Never Ends: Scopes, Lifecycle, and Delivery Order

WHERE WE ARE IN THE SOLUTION

```
Nexus.sln
 |- src/Nexus.Infrastructure
>|     Outbox/OutboxDispatcher.cs        NEW
>|     Outbox/IOutboxPublisher.cs        NEW
 |- src/Nexus.Worker                     NEW
 |- tests/Nexus.IntegrationTests
>|     OutboxDispatcherTests             NEW

Tests green: 150 (144 inherited · 6 new)
Outbox: drained in order · Reactions: none yet (Ch. 8)
```

The first chapter of Part II adds a worker, not a feature users can see. The outbox drains; nothing reacts yet. That gap is intentional — lifecycle and ordering are hard enough to earn a chapter of their own, and wiring real subscribers onto an unproven loop would confuse two problems. Part I's harness runs the loop in process, so the worker is tested the same way everything else is.

The host lifecycle, in plain language

A background worker is not a thread you start and forget; it is a citizen of the host's lifecycle. At startup the host calls `ExecuteAsync` and moves on, letting the loop run. At shutdown — a Ctrl-C, a container stop, a deployment — the host cancels the stopping token and then waits, up to a shutdown timeout, for `ExecuteAsync` to return. A worker that ignored the token would be killed when the timeout elapsed, possibly mid-delivery; a worker that honors it, as the dispatcher does by delaying on the token and breaking when it cancels, stops between passes, cleanly, every time. Figure D6.1 is that lifecycle.

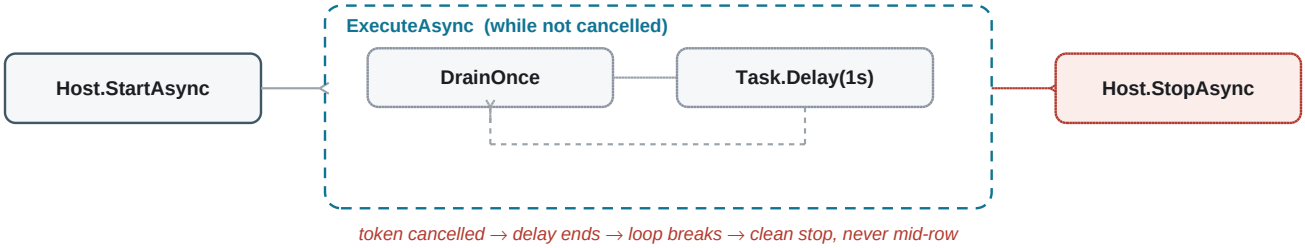


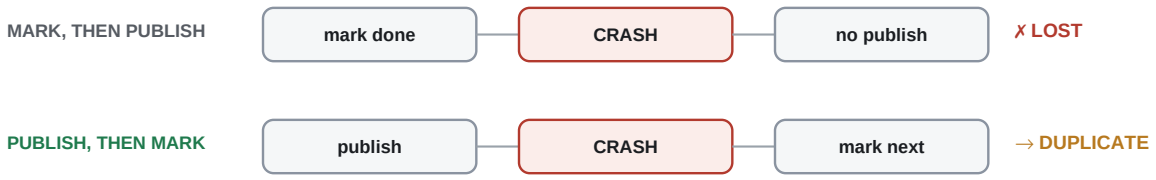
Figure D6.1 — The dispatcher across the host's life. Between start and stop it loops: drain, then delay. Cancellation ends the delay, the loop breaks, and the host completes its shutdown while the worker is at rest, never mid-row.

Why a singleton worker opens a scope every pass

This is the bug the pattern is famous for, and it is worth stating plainly because it compiles and runs before it fails. A `BackgroundService` is registered as a singleton: one instance for the whole life of the process. A `DbContext` is registered as scoped: one instance per unit of work, tracking changes and holding a connection, meant to be short-lived. Inject the context straight into the worker and you capture a single `DbContext` forever — its change-tracker grows without bound, its connection is held open, and stale entities accumulate until something breaks in production at three in the morning. The cure is the `IServiceScopeFactory` : the worker holds the factory, not the context, and opens a fresh scope on every pass, so each drain gets a clean context that is disposed when the pass ends — exactly the lifetime a unit of work is supposed to have.

Deliver, then mark: at-least-once by construction

Within a single row the dispatcher does two things — publish the event, and mark the row delivered — and the order it does them in is a decision, not a detail. Mark first and publish second, and a crash in the gap loses the event forever: the row says delivered, so no future pass will ever pick it up, yet no subscriber ever heard it. Publish first and mark second, and a crash in the gap merely redelivers: the row is still undelivered, so the next pass publishes it again. The first failure is silent data loss; the second is a duplicate. Between a lost event and a duplicated one there is no contest in a system that must react correctly — a duplicate can be recognized and ignored, a loss cannot be recovered — so the dispatcher publishes then marks, and accepts that it delivers at-least-once. Figure D6.2 sets the two orders against the same crash.



the dispatcher chooses the recoverable failure; Chapter 9 removes the duplicate.

Figure D6.2 — The same crash, two orderings. Mark-then-publish can lose the event; publish-then-mark can only duplicate it. The dispatcher chooses the recoverable failure and leaves the duplicate for Chapter 9 to remove.

Ordering and the cost of polling

Two smaller decisions deserve naming. The outbox is drained by a monotonic sequence rather than by timestamp, because timestamps collide and clocks drift, and an event that logically follows another must be delivered after it even if they were stamped in the same millisecond. And the worker polls on a fixed interval, which is the simplest correct design and also a compromise: a short interval means low delivery latency and many idle queries against a usually-empty table; a long one means fewer queries and later reactions. A more sophisticated dispatcher would be signalled the instant a row is written and poll only as a backstop, and that refinement is real — but it is an optimization of a correct loop, not a correctness fix, and the demonstrator keeps the honest, simple version and names the faster one here.

Where Chapter 7 takes this

This chapter's worker delivers when everything cooperates. Chapter 7 confronts the world in which things do not: a subscriber that throws, a transient failure that will pass on a retry, and a message that is malformed or forever unacceptable and will never succeed no matter how often it is tried. The blunt catch of Listing 6.1 becomes a real policy — retry a failed delivery on an exponential backoff so a flapping subscriber is not hammered, count the attempts, and after a threshold move the poison message to a quarantine so that one undeliverable row cannot wedge every good row queued behind it. Only once delivery can fail gracefully is it safe, in Chapter 8, to put real reactions on the other end of it.