ZMA PRACTICAL GUIDE: FROM SMALL APP → MODULAR MONOLITH → MICROSERVICES

August 11, 2025

(Complete, step-by-step handbook to build, scale, and operate ZMA — Zohaib Modular Architecture)

Short version: start tiny, keep clear boundaries, make the monolith **module-first**, implement the **Outbox** early, and extract one module at a time. This guide gives you the exact steps, folder layouts, commands, and patterns to do that without a painful rewrite.

ASSUMPTIONS & PREREQUISITES

- You use **.NET (ASP.NET Core)**, EF Core, Docker for local testing.
- Dev environment: dotnet CLI, PowerShell (or bash), Git.
- Basic knowledge: DI, DbContext, repo pattern, HTTP / message brokers.
- Goal: **testable**, **fast**, **transactionally safe**, **microservice-extractable**.

OVERVIEW

1. Goals & high-level strategy

1. **Start small** — single deployable modular monolith.
2. **Module-first** — structure application by features (Catalog, Orders, Payments).
3. **Infrastructure modularized** — per-module persistence code (DbContext, migrations) even if they use the same DB initially.
4. **Outbox & events** — implement reliable event publishing from day one.
5. **Extract safely** — copy module folder → new repo → its own DB → switch calls to HTTP/events.
6. **Operate** — containers, CI/CD, observability and contract tests.

2. Small app (quick start)

When to use: MVP, PoC, small e-commerce.

Minimal folder structure (single solution)

```
src/
  Presentation/ Web API
  Application/
    ├─ DTOs/
    ├─ Interfaces/
    ├─ Services/
    └─ Validators/
```

```
Domain/
  ├─ Entities/
  └─ ValueObjects/
Infrastructure/
  ├─ Persistence/     # AppDbContext, migrations
  └─ Repositories/
Shared/             # Utils, Result, Exceptions
```

What to implement (essentials)

- **Domain: Product, Order entities and business rules.**
- **Application: IProductService, ProductService — orchestration of domain + repository interfaces.**
- **Infrastructure: AppDbContext, ProductRepository implementing IProductRepository.**
- **Presentation: ProductsController calling IProductService.**

Quick commands (scaffold)

- **dotnet new sln -n ZMA.Small**
- **dotnet new webapi -n Presentation -o src/Presentation**
- **dotnet new classlib -n Application -o src/Application**
- **dotnet new classlib -n Domain -o src/Domain**
- **dotnet new classlib -n Infrastructure -o src/Infrastructure**
- **dotnet sln add src/**/*.csproj**

Wire project references: Presentation → Application; Application → Domain; Infrastructure → Application (implements interfaces).

Keep it tidy:

- Use DTOs to decouple API from Domain.
- Unit tests for Domain and Application layers (mock repositories).

3. Evolve to Modular Monolith (medium scale)

Goal: isolate modules so extraction is simple; keep single deploy.

What changes

- *Split Application into feature modules:*

1. *Application/*
    a. *CatalogModule/*
    b. *OrdersModule/*
    c. *Shared/*
2. *Infrastructure/*
    a. *Persistence/*

        - ➢ *CatalogPersistence/   # CatalogDbContext (class), migrations*
        - ➢ *OrdersPersistence/   # OrderDbContext (class), migrations*
    3. *Repositories/*
- ***Important:*** *Create per-module DbContext classes now — they may use the same DB connection string at first, but code is already isolated.*

Why per-module DbContext now?

1. Keeps EF mappings & migrations per module.
2. When extracting later, you just change connection string and move the folder.

Steps to modularize:

1. *Create module projects or folders: Modules/Catalog/{Domain,Application,Infrastructure,Presentation} (or keep inside solution as projects).*
2. *Move Catalog entities → Catalog.Domain.*
3. *Create CatalogDbContext in Catalog.Infrastructure/Persistence.*
4. *Add DI extension AddCatalogModule(IServiceCollection, IConfiguration).*
5. *Repeat for Orders.*

Migrations & DbContexts (still one DB at this stage)

Run migrations per context:

1. dotnet ef migrations add InitCatalog --context CatalogDbContext --project src/Modules/Catalog/Infrastructure --startup-project src/Presentation

2. dotnet ef database update --context CatalogDbContext --project src/Modules/Catalog/Infrastructure --startup-project src/Presentation

Replace direct cross-module calls with interfaces

If Orders needs product info, call ICatalogClient (an interface). Implementation in the monolith calls the Catalog module directly; when extracted, implement ICatalogClient to call CatalogService API.

4. Implement Outbox pattern & events

Goal: reliable, transactional publish of integration events — critical for safe extraction.

Outbox idea:

1. *Write domain changes and integration event(s) into the database in the same DB transaction (Outbox table).*
2. *A background publisher reads Outbox rows and publishes to message broker, marking them processed.*

Outbox schema

SQL: `CREATE TABLE OutboxMessages (

Id uuid PRIMARY KEY,

OccurredOn timestamptz,

Type varchar,

Payload text,

Processed boolean DEFAULT false

);`

Pseudocode for saving within transaction:

```
using var tx = await dbContext.Database.BeginTransactionAsync();
dbContext.Orders.Add(order);
dbContext.OutboxMessages.Add(new OutboxMessage{ Type="OrderCreated",
Payload=json });
await dbContext.SaveChangesAsync();
await tx.CommitAsync();
```

Background publisher (HostedService):

1. *Poll OutboxMessages where Processed = false, publish to RabbitMQ/Kafka, set Processed = true.*
2. *Ensure idempotency on consumers.*

5. Extract a module into a microservice (step-by-step)

The project will affect the finance department, IT department, and external stakeholders such as investors and regulatory bodies.

1. **Prepare in monolith**
   - Ensure Catalog code is self-contained (Domain, Application, Infrastructure, Presentation folders).
   - Ensure Catalog has its own DbContext, its own migrations folder in Infrastructure.
   - Ensure Catalog publishes events to Outbox and has contracts in SharedKernel.
2. **Create new repo/service**
   - Create CatalogService solution.
   - Copy Catalog folders (Domain, Application, Infrastructure, Presentation).
   - Add SharedKernel as package or reference (for events/contracts).
   - Set CatalogDbContext connection string to new DB.
3. **Data migration**

   Options:
   - **ETL dump**: export catalog tables from monolith DB → import into new Catalog DB.
   - **Replay events**: if monolith has event history, replay ProductCreated events to rebuild.
   - **Dual-write** (cutover): for a short window, write to both monolith DB and Catalog DB until cutover.
4. **Run migrations & start service**

   ```
   dotnet ef database update --project CatalogService.Infrastructure --startup-project
   CatalogService.Presentation
   ```
5. **Replace in consumer modules**
   - Replace internal direct calls to Catalog module with an ICatalogClient implementation that calls the new CatalogService API (via HttpClientFactory/gRPC).
   - Or keep using events (subscribers) for eventual consistency.
6. **Cutover & validate**

- Run tests, staging validations.
- Switch traffic gradually (feature flags, API Gateway routing).
- Monitor closely.

6. Full microservices ops (what to add)

- **Containerize** every service: Dockerfile per service.
- **Local compose** for dev (Postgres per service + RabbitMQ).
- **CI/CD**: pipelines per repo → build/test/image push → deploy.
- **Kubernetes** for production (Helm charts).
- **API Gateway** (YARP/Ocelot) for routing & auth.
- **Observability**: OpenTelemetry traces, Prometheus metrics, Grafana dashboards, ELK/Seq logs.
- **Secrets**: Vault or Azure Key Vault / Kubernetes Secrets.

7. Testing strategy

- **Unit tests**: Domain logic only — fast, no framework.
- **Application tests**: Use mocks for ports/interfaces.
- **Integration tests**: Use Testcontainers (Postgres, RabbitMQ) or docker-compose.
- **Contract tests**: Provider/consumer tests for services communicating via HTTP/events (Pact or equivalent).
- **E2E**: Few smoke flows in CI against staging stack.

8. Common code patterns & snippets

DI extension (module registration)

```
 public static class CatalogModuleExtensions
 {
public static IServiceCollection AddCatalogModule(this IServiceCollection services, IConfiguration config)
{
      services.AddDbContext<CatalogDbContext>(opts => opts.UseNpgsql(config.GetConnectionString("CatalogDb")));
      services.AddScoped<IProductRepository, ProductRepository>();
      services.AddScoped<IProductService, ProductService>();
      return services;
  }
}
```

Outbox publisher (simplified)

```
public class OutboxPublisher : BackgroundService
{
    private readonly IServiceProvider _sp;
    public OutboxPublisher(IServiceProvider sp) => _sp = sp;
    protected override async Task ExecuteAsync(CancellationToken ct)
    {
        while (!ct.IsCancellationRequested)
        {
            using var scope = _sp.CreateScope();
            var db = scope.ServiceProvider.GetRequiredService<CatalogDbContext>();
            var messages = await db.OutboxMessages.Where(m => !m.Processed).Take(50).ToListAsync(ct);
            foreach(var m in messages)
            {
                // publish to broker (Rabbit/Kafka)
                // mark processed
                m.Processed = true;
            }
            await db.SaveChangesAsync(ct);
            await Task.Delay(1000, ct);
        }
    }
}
```

Replace internal call with HttpClient (while in monolith you can keep direct)

```
public class CatalogHttpClient : ICatalogClient
{
 private readonly HttpClient _http;
public CatalogHttpClient(HttpClient http) => _http = http;
public Task<ProductDto> GetProduct(Guid id) =>
_http.GetFromJsonAsync<ProductDto>($"/api/products/{id}");
}
```

9. Data ownership & transactions

- ***Single-module writes****: local DB transaction using EF Core (ACID).*
- ***Cross-module transactions****: use* ***Saga / Process Manager*** *+ events; avoid distributed 2PC.*
- ***Payment or external calls****: mark intermediate states and publish events (eventual consistency).*

10. Best practices & rules of the road

- *Keep SharedKernel minimal: only base types, contracts, event DTOs — nothing business-specific.*
- *Module boundary rule: a module must not directly access another module's DbContext or entities.*
- *Version events: include Version in event payloads for backward compatibility.*
- *Idempotency: consumers must handle duplicate events.*
- *Migrations per DbContext: keep them scoped to project.*
- *Automate: CI scripts for migrations, docker builds, contract tests.*
- *Document: keep a short README per module explaining responsibilities and contracts.*

11. Typical migration checklist (middle → microservice for one module)

- *Ensure module isolation (own DbContext + migration).*
- *Extract module folder to new repo/service.*
- *Create new DB and run migrations.*
- *Migrate data (ETL/replay).*
- *Implement API client in monolith for other modules.*
- *Update consumers to use API or events.*
- *Test end-to-end in staging.*
- *Switch routing; deprecate monolith module code after stable.*

12. Troubleshooting — common pain points

- *Tight coupling: fix by adding interface layer and switching consumers to interfaces.*
- *Shared entities used everywhere: replace cross-module entity usage with DTOs or contracts.*
- *Data drift during migration: use slow double-write and verify with checksums.*
- *Event schema incompatible: maintain event versioning and backward-compatible consumers.*

13. Appendix — canonical folder structures

Small app (single monolith)

src/

- *Presentation/*
- *Application/*
    - *DTOs/*
- *Interfaces/*
    - *Services/*
- *Domain/*
    - *Entities/*
- *Infrastructure/*
    - *Persistence/*
    - *Repositories/*
- *Shared/*

    - *Common/*

Modular monolith (medium)

src/

- ***Modules/***
    - ***Catalog***
        - ***Domain***
        - ***Aplication***
        - ***Infrastructure***
        - ***Presentation***
    - ***Orders***
        - ***Domain***
        - ***Aplication***
        - ***Infrastructure***
        - ***Presentation***
    - ***Payment***
        - ***Domain***
        - ***Aplication***
        - ***Infrastructure***
        - ***Presentation***
- ***SharedKernel/***
- ***Presentation/***

Microservices (large)

- ***Services/***
    - ***CatalogService***
        - ***Domain***
        - ***Aplication***
        - ***Infrastructure***
        - ***Presentation***
    - ***OrderService***
        - ***Domain***
        - ***Aplication***
        - ***Infrastructure***
        - ***Presentation***
    - ***PaymentService***
        - ***Domain***
        - ***Aplication***
        - ***Infrastructure***
        - ***Presentation***
- ***SharedKernel/***

- *Gateways/*
    - *ApiGateway/*
    - *AuthService/*

14. Quick starter checklist

- ***Scaffold solution + projects (module-first)***
- ***Create domain entities & unit tests***
- ***Implement per-module DbContext (even if same DB)***
- ***Implement Outbox and background publisher***
- ***Write one cross-module contract in SharedKernel***
- ***Add integration tests using docker-compose/Testcontainers***
- ***Prepare migration playbook for first module extraction***

.