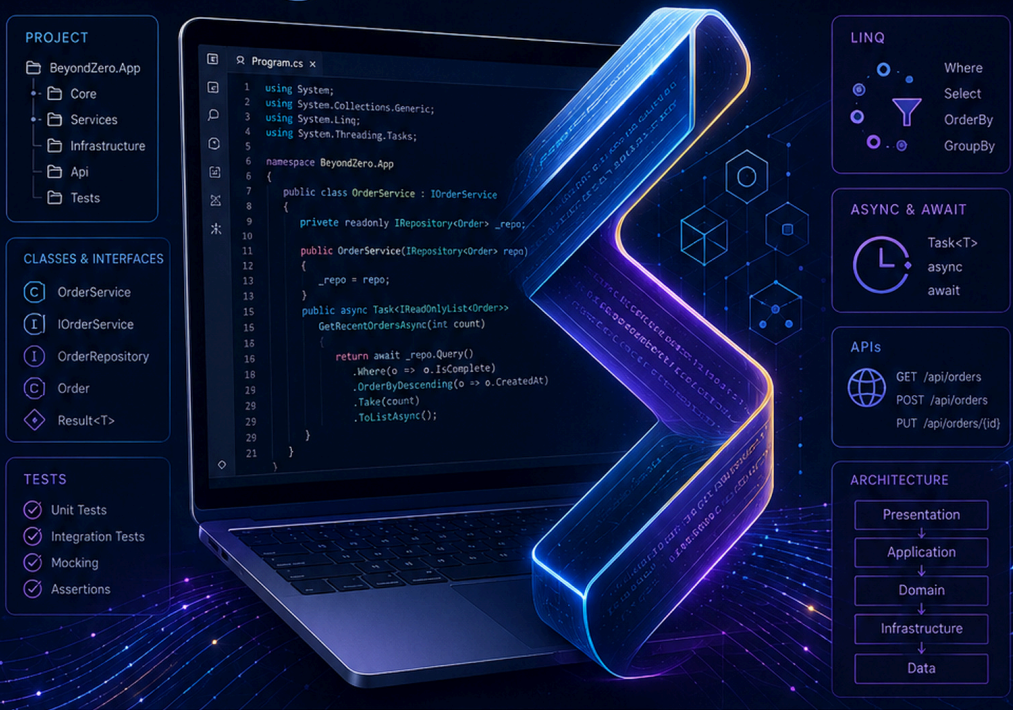
BEYOND ZERO SERIES

C#
Beyond Zero



AN INTERMEDIATE GUIDE TO

PRACTICAL C# AND .NET DEVELOPMENT

LANCASTER SOLUTIONS LLC

C#

BEYOND ZERO

Intermediate .NET Application Engineering, APIs, Data, and Operations

Expanded Professional Edition

Architecture • maintainability • testing • security • performance • deployment • guided labs • production capstone

Curtis Wayne Lancaster II

Published by Lancaster Solutions LLC

Beyond Zero Series • Practical technology education for real-world builders

About This Expanded Edition

> **AUDIENCE:** Readers who completed C# From Zero or can create classes, collections, LINQ queries, async methods, and basic tests.

This revised edition moves from language fluency into application boundaries, data integrity, HTTP contracts, multi-tenant security, background work, diagnostics, and repeatable delivery.

Technical baseline

C# 14 on .NET 10, with stable architectural practices and explicit notes where frameworks or deployment modes impose additional constraints.

What “intermediate” means in this series

You can already write small working programs or pages. This book focuses on what happens when the work must be changed safely, tested repeatedly, deployed predictably, and understood by someone other than its original author.

The objective is not to memorize every feature. It is to build judgment: choosing boundaries, stating contracts, designing failure behavior, gathering evidence, and balancing complexity against the real needs of a project.

Examples are intentionally compact. The labs, projects, and capstone ask you to supply missing production details such as validation, logging, authorization, migration, accessibility, or rollback. Those omissions are part of the learning design rather than permission to ignore them.

How to use the book

- Read the chapter thesis first and write down what you expect the technique to improve.
- Run or reproduce the example in a small, disposable project before transplanting it into a larger codebase.
- Complete the guided lab with tests and notes. Then attempt the independent challenge without copying the lab structure mechanically.
- Keep an engineering notebook containing assumptions, decisions, failed experiments, measurements, and follow-up work.
- Treat the capstone as a portfolio-quality delivery: source code is only one deliverable alongside evidence, documentation, and an operational plan.

Publication and educational use

Contents and Learning Roadmap

The roadmap is organized into four stages. Each stage introduces decisions that the next stage assumes you can explain and verify.

Stage	Focus	Coverage	Core topics
Part 1	Structure, Models, and Contracts	Chapters 1–4	Solution Architecture and the .NET Build Model, Nullable Reference Types, Records, and Pattern-Oriented Modeling, Generics, Constraints, and Reusable Contracts, LINQ, Query Semantics, and Deferred Execution
Part 2	Boundaries, Data, and Collaboration	Chapters 5–8	Asynchronous Programming, Cancellation, and Concurrency, Dependency Injection, Configuration, and Logging, Entity Framework Core, Transactions, and Data Access, ASP.NET Core APIs, Validation, and HTTP Contracts
Part 3	Reliability, Scale, and Verification	Chapters 9–11	Authentication, Authorization, and Multi-Tenant Boundaries, Automated Testing, Testcontainers, and Contract Evidence, HTTP Resilience, Background Services, and Messaging
Part 4	Delivery, Operations, and Evolution	Chapters 12–14	Performance, Allocation, and Diagnostics, Architecture Patterns, Events, and Maintainable Boundaries, Containers, CI/CD, Observability, and Production Readiness

Projects, capstone, and references

PART I

Structure, Models, and Contracts

Chapters 1–4

CHAPTER 1
Solution Architecture and the .NET Build Model

> **CHAPTER THESIS:** Intermediate C# begins by making project boundaries, target frameworks, dependencies, analyzers, and artifacts explicit across a solution.

Learning outcomes

- Explain how projects and solutions shapes boundaries, testability, and failure behavior in a maintainable implementation.
- Apply SDK-style projects in a focused implementation and identify the contract it creates for callers.
- Compare an implementation that uses reference direction with a simpler alternative, then justify the added complexity.
- Evaluate the operational and maintenance consequences of build evidence under realistic change and failure.
- Evaluate the solution Layout example, identify assumptions about projects and solutions and SDK-style projects, and justify one viable alternative.
- For “Solution Architecture and the .NET Build Model”, produce evidence using compiler warnings, automated tests, HTTP contract examples, database migrations, and observable service behavior; state what the evidence proves and what remains uncertain.

The intermediate shift

The intermediate challenge in “Solution Architecture and the .NET Build Model” is not learning one more API; it is deciding where the behavior belongs and what must remain true when the surrounding system changes.

Projects and solutions, SDK-style projects, Reference direction, and Build evidence reinforce one another, but they optimize different risks. The chapter asks you to decide which concern owns each rule, where translation occurs, and how the design behaves when one assumption changes.

Credible evidence for “Solution Architecture and the .NET Build Model” includes compiler warnings, automated tests, HTTP contract examples, database migrations, and observable service behavior. Capture the setup and expected result so another reader can reproduce the claim instead of trusting a successful demonstration.

Projects and solutions

A solution groups projects for development, while each .csproj defines an independently buildable target and dependency graph.

Implementation guidance. Separate domain, application, infrastructure, API, and test projects only when the boundaries reduce coupling; avoid creating projects that contain one class and no independent reason to change.

For Projects and solutions, the design payoff is controlled change: callers depend on the contract while the implementation can evolve behind it.

SDK-style projects

Modern SDK projects infer source files and use concise properties for target frameworks, nullable analysis, implicit usings, and package references.

Implementation guidance. Centralize common settings carefully and keep project-specific requirements visible.

Operationally, SDK-style projects must define failure ownership, retained context, retry safety, and the signal shown to users or operators.

Reference direction

Project references should point from outer adapters toward inner policy. Cycles make the solution difficult to build, test, and reason about.

Implementation guidance. Use interfaces in the consuming layer and implement them in infrastructure, then connect dependencies in the host.

Verify Reference direction through its narrowest public contract, then add adversarial cases that exercise malformed input and dependency failure.

Build evidence

A successful IDE run does not prove a clean command-line build, publish, or test artifact.

Implementation guidance. Use dotnet restore, build, test, and publish from a clean environment and preserve the resulting artifacts.

For maintenance, Build evidence should communicate ownership through names, types, modules, and documentation rather than institutional memory.

Worked pattern

SOLUTION LAYOUT

```
src/
  Study.Domain/
  Study.Application/
  Study.Infrastructure/
```

```
  Study.Api/
tests/
  Study.Domain.Tests/
  Study.Infrastructure.Tests/
Study.slnx
```

Walkthrough

1. The domain project contains rules and value types.
2. Application coordinates use cases through abstractions.
3. Infrastructure implements databases and external services; the API composes the application.

Design decisions to notice

- Projects and solutions: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- SDK-style projects: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- Reference direction: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- Build evidence: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- In the solution Layout, which decisions express the policy described in “Solution Architecture and the .NET Build Model”, and which are replaceable mechanisms behind the boundary?
- For “Solution Architecture and the .NET Build Model”, which guarantees are supplied by C#, .NET, and ASP.NET Core, and which still require explicit validation, cleanup, monitoring, or documentation?

Failure modes and diagnostic thinking

- A domain project referencing ASP.NET Core or Entity Framework.
- Circular project references hidden by moving types into a shared dumping-ground project.
- Different nullable settings across projects causing inconsistent guarantees.
- CI building only the startup project and missing test or package failures.

> **DIAGNOSTIC EXERCISE:** Reproduce this risk in the smallest useful example: A domain project referencing ASP.NET Core or Entity Framework. Identify the first observable symptom, the earliest detection boundary, one preventive control, and one operational signal that would distinguish recurrence from a different failure.

Guided lab

> **OBJECTIVE:** Split a console application into domain, application, infrastructure, and host projects.

Phase 1: Clarify the contract

Turn the objective into acceptance criteria: Split a console application into domain, application, infrastructure, and host projects. Name trusted and untrusted inputs, explicit non-goals, and the result a reviewer can observe.

Phase 2: Create a baseline

Capture a baseline for “Solution Architecture and the .NET Build Model” using compiler warnings, automated tests, HTTP contract examples, database migrations, and observable service behavior. Preserve the command, fixture, environment, or screenshot needed to repeat it.

Phase 3: Implement the smallest safe change

Implement the smallest coherent change that advances the objective. Keep projects and solutions behind a named boundary and verify each increment before adding the next.

Phase 4: Verify normal and hostile conditions

Exercise the normal case plus malformed input, boundary values, and a realistic failure involving SDK-style projects. Include cancellation, timeout, rollback, fallback, or recovery where the chapter makes it relevant.

Phase 5: Review and communicate

Review the evidence with the objective in view. Record the chosen design, the rejected alternative, remaining risk, and the signal that would justify revisiting reference direction.

Independent challenge

Add analyzers and treat selected warnings as errors without suppressing legitimate findings.

> **CONSTRAINT:** Complete the independent challenge with a different dependency lifetime from the worked pattern. Explain how the change affects failure behavior, testing evidence, and maintenance cost.

Stretch challenge

Create a publish smoke test that launches the produced artifact with production-like configuration.

Chapter review

1. What problem does projects and solutions solve, and what new complexity can it introduce when overused?
2. What problem does SDK-style projects solve, and what new complexity can it introduce when overused?

3. What problem does reference direction solve, and what new complexity can it introduce when overused?
4. What problem does build evidence solve, and what new complexity can it introduce when overused?
5. Which evidence would demonstrate that projects and solutions remains correct under failure, and which part still requires representative measurement or human review?
6. Under what project constraints would a simpler alternative to build evidence be the more responsible choice?

Definition of done

- The objective is demonstrated for the normal path and failure cases relevant to projects and solutions.
- The contract around SDK-style projects is explicit, smaller than its implementation, and exercised by evidence.
- The verification does not depend on hidden secrets, machine-specific paths, or uncontrolled state while testing reference direction.
- A reviewer can trace the decision about build evidence from requirement to implementation and evidence.
- Known limitations, operational signals, and follow-up work for “Solution Architecture and the .NET Build Model” are recorded with an owner or review trigger.