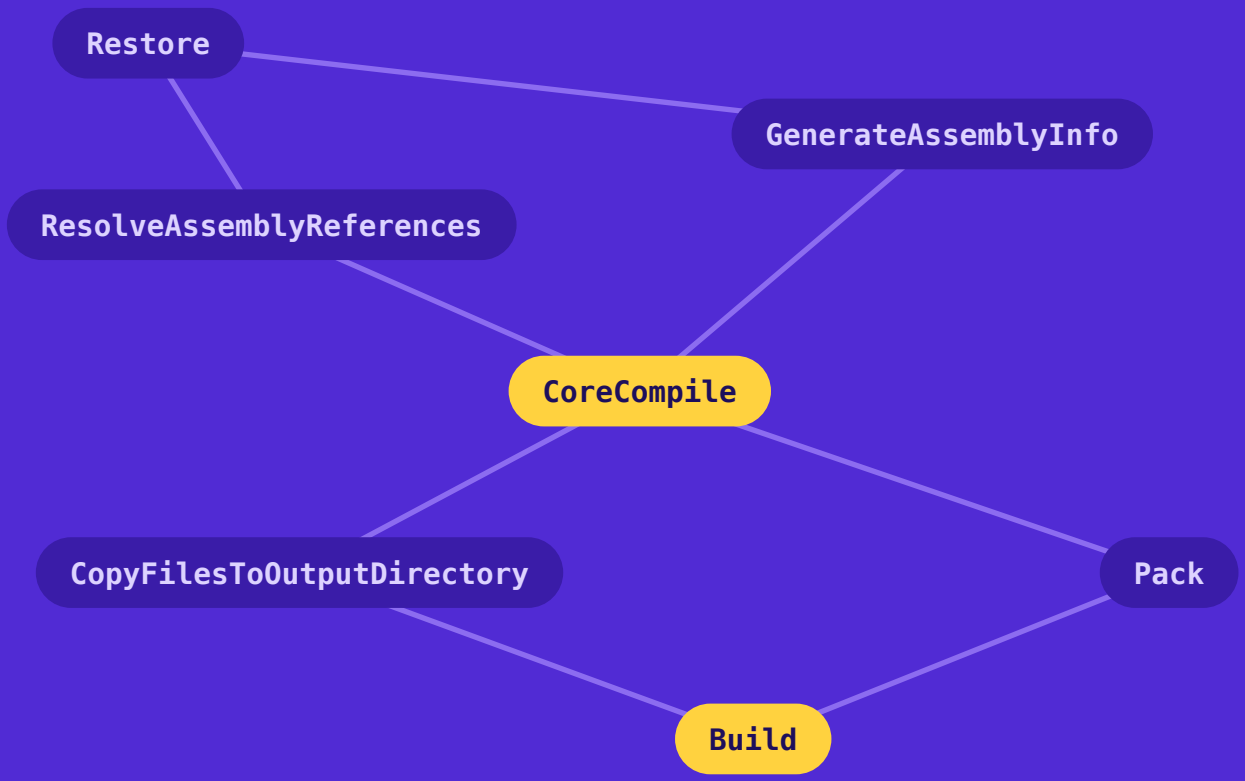
THE .NET BUILD ENGINE PREVIEW

MSBuild In Depth

Building, Extending, and Debugging .NET Builds



Gérald Barré

MICROSOFT MVP · MEZIANTOU.NET

Copyright © 2026 Gérald Barré

All rights reserved. No part of this publication may be reproduced, distributed, transmitted, stored in a retrieval system, or otherwise used in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the copyright holder, except as permitted by applicable copyright law.

Published by Gérald Barré
Montréal, Québec, Canada

First edition, 2026

Legal deposit, Bibliothèque et Archives nationales du Québec, 2026
Legal deposit, Library and Archives Canada, 2026

The information in this book is provided on an "as is" basis, without warranties of any kind, express or implied. Although the author has taken reasonable care in preparing this work, the author makes no representations or warranties regarding the accuracy, completeness, or suitability of its contents. To the fullest extent permitted by applicable law, the author shall not be liable for any loss or damage arising directly or indirectly from the use of, or reliance on, the information contained in this book.

Product names, company names, logos, brands, and other trademarks mentioned in this book are the property of their respective owners. Their use does not imply affiliation with, sponsorship by, or endorsement by the respective trademark owners.

Unless otherwise indicated, source-code examples included in this book are provided for educational purposes and may be used and adapted in your own projects. The source code is provided without warranty of any kind.

Contents

A free preview — the introduction and 2 chapters. The full edition has 31 chapters and 11 appendices.

Introduction in this preview
- What you will be able to do
- Who this book is for
- How this book is organized
- Conventions
- The sample projects

Part I — Foundations

Chapter 1 — What Is MSBuild? in this preview
- MSBuild is a build engine, not a compiler
- Where MSBuild lives
- The cast of characters
- A ten-minute "hello build"
- Your first binary log
- Exercises

Chapter 2 — Project File Anatomy in this preview
- The elements you will see
- A minimal SDK-style project
- What `Sdk="Microsoft.NET.Sdk"` does
- For contrast: the legacy (non-SDK) format
- Evaluation runs in passes
- Exercises

Chapter 3 — Properties
- Declaring and overriding: last assignment wins
- Conditions
- Where property values come from
- Property functions
- The distinction that explains everything: evaluation vs. execution
- Exercises

Chapter 4 — Items and Item Metadata
- Declaring items
- Well-known metadata
- Referencing custom metadata
- Transformations
- Item functions
- Batching with `%(...)`
- Globbing: Include, Exclude, Remove, Update
- The SDK's default item globs
- Under the hood: batching is expansion
- Exercises

Chapter 5 — Targets and the Build Process
- What a target is
- Ordering: three ways to say "run before this"
- Incremental build: skipping work that is already done
- Target batching: one up-to-date check per item
- Passing data between targets: Outputs and Returns
- The standard target graph
- Gotcha: AfterTargets vs. DependsOnTargets, and "why didn't my target run?"
- Exercises

Chapter 6 — Tasks
- What a task is
- A chain of tasks
- Capturing outputs with `<Output>`
- Copy, and doing real file work
- The `MSBuild` task: building other projects
- `Exec` vs. a real task
- How the engine runs a task
- Which shape of logic, and when
- Part I in one page
- Exercises

Part II — Seeing What MSBuild Did

Chapter 7 — Reading Build Output
- One build, one project
- Which logger am I looking at?
- Verbosity: the five levels
- File loggers: quiet console, complete log on disk
- The terminal logger: the modern default
- Warnings, errors, codes, and summaries
- Exercises

Chapter 8 — Binary Logs (binlog)
What a text log throws away
What a `.binlog` captures
Producing one
Embedding your own files
Property tracking: where a value came from
Replaying at any verbosity
Privacy and security: read this before you share one
Format stability and versioning
A binlog handling checklist
Pitfalls
Exercises

Chapter 9 — The MSBuild Structured Log Viewer
Installing and opening the viewer
The tree is the execution model
Search: a small query language to learn
Investigation 1: the wrong file was copied
Investigation 2: the target ran twice
Performance: the timeline and the slowest-first search
Diffing two builds
Exercises

Chapter 10 — Preprocessing and Evaluation Debugging
The sample: a project that is deeper than it looks
`-preprocess` (`-pp`): the effective project, flattened
`-targets`: what can I run?
Command-line evaluation queries
Evaluation vs. execution mistakes, and how to spot them
Debugging conditions that "should" be true
The evaluation profiler: `-profileEvaluation`
Putting it together
Exercises

Part III — Structuring Real Projects

Chapter 11 — Imports: .props and .targets Files
What `<Import>` does
The convention: .props early, .targets late
`$(MSBuildThisFileDirectory)`: locating siblings
Wildcard and conditional imports
Putting it together: a project that imports shared logic
Import position and override direction
Where the SDK does this for you
Where should shared build logic live?
Exercises

Chapter 12 — Directory.Build.props and Directory.Build.targets
Repository-wide defaults without editing every project
The discovery rule: walk up, first match wins, import exactly one
Timing: `.props` reacts to nothing, `.targets` reacts to everything
Chaining to a parent file
Central Package Management with Directory.Packages.props
The rest of the convention-named files
A nested solution, one shared property
Exercises

Chapter 13 — Multi-Project Solutions and the Build Graph
A three-project graph
`<ProjectReference>` drives build order
The solution: `.sln` and `.slnx`
`dotnet build` vs. `dotnet msbuild`
Parallel builds, node reuse, and determinism
The static graph build (`-graph`)
What to carry into Chapter 23
Exercises

Chapter 14 — Configurations, Platforms, and Multitargeting
Configuration and Platform: two ordinary properties
TargetFramework vs. TargetFrameworks: the inner and outer build
The gotcha that defines multitargeting: outer vs. inner properties
Runtime identifiers, self-contained, and AOT
Remember now, defer until publish
Exercises

Chapter 15 — Common SDK Properties and Build Customization
Where the build writes: output and intermediate paths
Language and quality knobs
Deterministic and reproducible builds
Assembly metadata and versioning
Case study: safe defaults for a repository
Exercises

Part IV — Extending MSBuild

Chapter 16 — Writing Inline Build Logic in Project Files
- The Part IV route map
- Inline logic: property and item groups inside a target
- Conditions and batching inside a target
- Failing on purpose, and surviving failure
- Four recipes: choosing the shape
- How the engine runs inline logic
- Exercises

Chapter 17 — Custom Tasks in C#
- Deriving from `Task` versus implementing `ITask`
- The task project
- Writing `Execute`
- Logging: `Message`, `Warning`, `Error`
- Registering and calling the task
- The assembly-loading problem
- Cancelable tasks
- Beyond one task: where to go next
- Exercises

Chapter 18 — Debugging Custom Tasks
- A task that misbehaves
- Reading the failure from a binlog
- Making failures diagnose themselves
- Attaching a debugger
- Engine diagnostics: `MSBuildDebugEngine` and failure dumps
- The ideal bug report
- Exercises

Chapter 19 — Code-Generation Builds
- The job, and the three things that make it hard
- Wiring it in: generate before `CoreCompile`
- Keeping generated code out of source control
- Incrementality that does not lie
- Timing gotchas, collected
- When to use this vs. a Roslyn source generator
- The safe code-generation checklist
- Exercises

Chapter 20 — Inline Tasks with RoslynCodeTaskFactory
- The shape of an inline task
- The Fragment form: a method body
- The Class form: a full ITask
- References and usings
- When to reach for an inline task, and when not to
- How the engine runs one
- Exercises

Chapter 21 — Shipping Build Logic in NuGet Packages
- The naming rule that triggers auto-import
- The three build folders
- Building the package
- What the entry points contain
- Direct consumption
- Transitive consumption
- Not leaking internal properties
- Multi-targeting: `buildMultitargeting/`
- Testing your build logic in isolation
- Where this fits
- Exercises

Chapter 22 — Authoring Project SDKs
- What a project SDK is
- Building your own SDK: the package layout
- Consuming a custom SDK
- The three forms of `Sdk` usage
- How MSBuild finds the SDK: resolvers
- When to build a custom SDK vs. a plain props/targets package
- Exercises

Part V — Performance and Production

Chapter 23 — Performance and Incrementality
- Triage order: the first hour
- When a build rebuilds work it should have skipped
- The fix: declare the output you produce
- The Visual Studio fast up-to-date check
- Evaluation cost and glob cost
- Parallelism, node reuse, and reference resolution
- Reference assemblies: why downstream projects skip compiling
- Build caching and the build server
- Exercises

Chapter 24 — Restore, NuGet, and MSBuild
- Restore is a separate build that runs before yours

What restore writes into obj/
Resolving versions: transitive graph, ranges, and nearest-wins
Lock files: making restore reproducible
Central Package Management, in depth
Reading common restore failures
Three more things restore does on .NET 10
Where this fits
Exercises

Chapter 25 — Packing and Publishing
What are you shipping?
`dotnet pack` and the Pack target
Controlling what goes into the package
Pack versus publish
`dotnet publish` and the Publish target
Customizing publish output with a post-publish step
Gotchas that only appear at ship time
Where this fits
Exercises

Chapter 26 — MSBuild in CI/CD
Pinning the SDK: `global.json`
Reproducible builds on the server
Locked restore
NuGet audit and `-warnaserror`
Always capture a binlog
The GitHub Actions workflow
The Azure Pipelines equivalent
Warnings-as-errors gating
Build annotations and problem matchers (GitHub)
The minimum CI build contract
Pitfalls
Exercises

Chapter 27 — Cross-Platform and Toolchain Concerns
Two MSBuilds: `dotnet` and `msbuild.exe`
Paths: the most common portability bug
Shell and quoting portability for `Exec`
Environment differences and runtime checks
Unsafe pattern, portable replacement
Exercises

Part VI — Advanced Patterns and Legacy

Chapter 28 — Hosting and Programming the MSBuild API
Evaluating versus building
`MSBuildLocator`: why order is everything
Evaluating a project
Building a project
Writing a custom logger
Running the tour
Use cases
Exercises

Chapter 29 — Advanced Evaluation and Metaprogramming
Property functions in depth
Path intrinsics
Item functions
Building items dynamically: cross-product vs. multi-batching
Recursion with the `MSBuild` task
Condition expression edge cases
Before you use any of this
Exercises

Chapter 30 — Anti-Patterns and Field Recipes
The field-guide index
Anti-patterns: how build logic rots
Field recipes
Exercises

Chapter 31 — Migrations and Legacy
The “before”: a real non-SDK library
The “after”: the same library, migrated
packages.config to PackageReference
Untangling an inherited “big ball of targets”
A migration checklist
Exercises

Where to Go Next
What you can do now that you could not before
Three habits and a bookmark
Where the knowledge goes stale, and where it does not
Going deeper
A last word

Appendix A — Command-Line Reference
Logging, verbosity, and diagnostics

Restore
Building: targets, properties, parallelism
Evaluation queries (no build)
Graph and isolated builds
Evaluation profiling
Warnings and errors

Appendix B — Reserved and Well-Known Properties and Item Metadata
Reserved properties: the project file
Reserved properties: the toolset and engine
Well-known properties (defaults you can override)
Common .NET SDK properties
Well-known item metadata

Appendix C — Property and Item Function Reference
Static .NET method calls
Instance string methods
`[MSBuild]::` intrinsics: arithmetic
`[MSBuild]::` intrinsics: bitwise
`[MSBuild]::` intrinsics: version comparison
`[MSBuild]::` intrinsics: paths
`[MSBuild]::` intrinsics: strings and encoding
`[MSBuild]::` intrinsics: OS and target framework
Item functions

Appendix D — Diagnostic Environment Variables
Engine diagnostics and debugging
Node reuse and process control
Logging detail
Terminal logger (build output)
Related context (not an MSBuild variable)
Notes on scope and safety

Appendix E — Binlog and Structured Log Viewer Search-Syntax Quick Reference
Basics
Node-kind keywords (`$`)
Field filters
Scope operators
Time, sorting, and duration
Purpose-built indexes
Everyday query recipes

Appendix F — Optional: Querying Binlogs with an MCP Server
Why hand a binlog to an AI assistant
What the binlog MCP server is
Installing the server
Query patterns that work well
When the log is too big
Reading binlogs programmatically
Automating regression detection in CI
Exercises

Appendix G — Multithreaded Task Execution
The model
Opting in: the attribute and the interface
`TaskEnvironment`: thread-safe stand-ins for process globals
Before and after
The subtle bugs
Turning it on, and how tasks are routed
Migrating an existing task safely

Appendix H — Symptom Index
Targets and ordering
Properties and evaluation
Files, items, and outputs
Restore, packages, and packing
Custom tasks and tooling
Performance
When none of the above fits

Appendix I — Glossary

Appendix J — Further Resources
Official MSBuild
Structured Log Viewer and binlog tooling
NuGet

Appendix K — Exercise Notes
Chapter 1 — What Is MSBuild?
Chapter 2 — Project File Anatomy
Chapter 3 — Properties
Chapter 4 — Items and Item Metadata
Chapter 5 — Targets and the Build Process
Chapter 6 — Tasks
Chapter 7 — Reading Build Output
Chapter 8 — Binary Logs (binlog)
Chapter 9 — The MSBuild Structured Log Viewer

Chapter 10 — Preprocessing and Evaluation Debugging
Chapter 11 — Imports: .props and .targets Files
Chapter 12 — Directory.Build.props and Directory.Build.targets
Chapter 13 — Multi-Project Solutions and the Build Graph
Chapter 14 — Configurations, Platforms, and Multitargeting
Chapter 15 — Common SDK Properties and Build Customization
Chapter 16 — Writing Inline Build Logic in Project Files
Chapter 17 — Custom Tasks in C#
Chapter 18 — Debugging Custom Tasks
Chapter 19 — Code-Generation Builds
Chapter 20 — Inline Tasks with RoslynCodeTaskFactory
Chapter 21 — Shipping Build Logic in NuGet Packages
Chapter 22 — Authoring Project SDKs
Chapter 23 — Performance and Incrementality
Chapter 24 — Restore, NuGet, and MSBuild
Chapter 25 — Packing and Publishing
Chapter 26 — MSBuild in CI/CD
Chapter 27 — Cross-Platform and Toolchain Concerns
Chapter 28 — Hosting and Programming the MSBuild API
Chapter 29 — Advanced Evaluation and Metaprogramming
Chapter 30 — Anti-Patterns and Field Recipes
Chapter 31 — Migrations and Legacy
Appendix F — Optional: Querying Binlogs with an MCP Server

Introduction

MSBuild is the engine underneath almost everything you do in .NET. Every `dotnet build`, every press of **F5** in Visual Studio, every `dotnet pack` that produces a NuGet package, every CI pipeline that turns a commit into a shippable artifact: all of it runs through MSBuild. And yet for most developers it stays a black box: a `.csproj` file you edit by copying snippets from the internet, and a build that either works or fails for reasons that are hard to see.

This book is about opening that box.

What you will be able to do

By the end, you will be able to:

- **Read any project file** (`.csproj`, `.props`, `.targets`) with enough confidence to predict the common cases, and know which tool to reach for when an import, an SDK target, or a global property changes the answer.
- **Explain the two phases**, evaluation and execution, well enough to predict the engine's behavior instead of guessing at it.
- **Structure real repositories** with `Directory.Build.props`, Central Package Management, multitargeting, and shared build logic.
- **Extend the build** with your own targets, custom tasks written in C#, and build logic packaged in NuGet.
- **Debug builds from evidence**: capture a binary log, read it in the MSBuild Structured Log Viewer, query evaluated properties and items from the command line, and answer questions like "why did this rebuild?" or "which target set that property?" in seconds. Newer tools, such as an MCP (Model Context Protocol) server that queries a binlog for you, are useful when they help; the skill underneath them is what lasts.
- **Ship it in CI** reproducibly, with the SDK pinned and a binlog captured on every run.

The through-line of the book is **observability**. MSBuild is far less mysterious once you can *see* what it did, so the debugging tools are not tucked into a final chapter. They are Part II, right after the foundations, and they are used everywhere after that.

One consequence belongs here rather than in the last chapter. **The model is stable; the tooling is not.** Evaluation before execution, properties and items, target ordering, incrementality: these have held for two decades and will outlive this edition. Tool surfaces move much faster: command-line switches, viewer menus, and above all the AI-assisted tooling in Appendix F, which carries its date openly. Where the book teaches something that ages, it says so. Learn the model; treat the tools as replaceable.

Who this book is for

This book is written for the **intermediate .NET developer** who is comfortable writing C# and using the command line, edits project files occasionally, but has never been taught how MSBuild works. If you have ever pasted a `<Target>` into a `.csproj` and hoped, this book is for you.

It is also useful for **build and DevOps engineers** maintaining large or slow builds, and for **library and SDK authors** who ship MSBuild logic (props and targets) to other people inside NuGet packages.

You do **not** need any prior knowledge of MSBuild internals. You do need C# basics, familiarity with a terminal, and a rough idea of what NuGet is.

How this book is organized

The book is in six parts:

- **Part I, Foundations,** builds the mental model: what MSBuild is, and the four core concepts (properties, items, targets, tasks) that everything else is made of.
- **Part II, Seeing What MSBuild Did,** is the heart of the book, and it comes early by design: reading build output, binary logs, the Structured Log Viewer, and the preprocessing and query switches that print what evaluation decided. Everything after it is safer because you can see it.
- **Part III, Structuring Real Projects,** scales the model up to multi-project repositories: imports, repository-wide defaults, solutions, multitargeting.

- **Part IV, Extending MSBuild,** teaches you to add your own behavior: inline targets, custom C# tasks and how to debug them, code generation, and shipping build logic in packages and SDKs.
- **Part V, Performance and Production,** covers incrementality and build performance, restore, packing, publishing, CI/CD, and cross-platform concerns.
- **Part VI, Advanced Patterns and Legacy,** finishes with the MSBuild programming API, advanced evaluation, a field guide of patterns and anti-patterns, and migrating legacy projects.

Each part opens with a page of its own, naming what you already hold, what the part adds, and what to watch for. Those pages take a minute, and they say plainly what the chapters only imply.

Read the chapters in order the first time, because later ones lean on the vocabulary and the debugging tools established earlier. After that, each stands on its own as a reference.

Reading paths

If a build is broken right now: go to Appendix H, the symptom index (symptom, first command, chapter), and to Part II (Chapters 7–10), which is the whole diagnostic toolkit in four chapters. Come back to the rest afterward; nothing there needs the chapters in between.

Otherwise, not everyone arrives with the same problem. If you cannot read straight through, these three routes each end somewhere useful. All three include Part I and Part II in full: every later chapter assumes both.

If you are…	Read
A build engineer: the repository builds slowly, or inconsistently, and it is your job to fix that	Part I (1–6) for the model and Part II (7–10), then 11, 12, 13, 23, 24, 26, and 30. The center of gravity is Part II: you cannot fix what you cannot see, and Chapter 23's triage order is the one to learn well.
A library or SDK author: you ship build logic to other people	Part I (1–6) and Part II (7–10), then 11, 12, 14, 15, 16, 17, 18, 21, 22, 24, 25, and 30. Chapter 21 is the one that matters most; Chapter 16 comes first because a package's `.targets` file is inline build logic, and Chapter 18 becomes urgent the first time somebody files a bug against your task.
A CI maintainer: the pipeline is yours and it must be reproducible	Part I (1–6) and Part II (7–10), then 12, 13, 15, 24, 26, 27, and Appendix H. Chapters 24 and 26 are the core of that path: locked restore and a pinned SDK are what make "the same commit" mean something, and Chapter 26 builds on the repository-wide settings of Chapters 12 and 15.

Chapters 3, 4, and 5 repay careful reading. Most later surprises trace back to them, and skimming them is the one shortcut that costs more than it saves.

Conventions

- **Version.** Everything targets the **.NET 10 SDK**. The book teaches the cross-platform `dotnet` MSBuild; where .NET Framework `msbuild.exe` or Visual Studio behaves differently, a **Version note** calls it out.
- **Command lines** are shown for `dotnet` / `dotnet msbuild` and work on Windows, macOS, and Linux. A handful of experiments need something the shell provides rather than MSBuild (setting an environment variable for one command, or touching a file's timestamp), and those are labeled **POSIX shell** or **PowerShell** and given in both.
- **A command too wide for the page wraps at a space**, with the continuation indented. It is still one command line: a switch is never split across two lines, so anything starting with `-` at the left edge of a listing belongs to the line above it.
- **Callouts** flag things worth stopping for: **Gotcha** (a common mistake), **Under the hood** (how the engine really does it), **Version note** (behavior that depends on a version), **Performance**, and **Try it** (a suggested experiment).

- **Two names are shortened after their first use.** The **MSBuild Structured Log Viewer**, the desktop application Chapter 9 is about, is "the Structured Log Viewer," or "the viewer," everywhere after Chapter 1 introduces it. A **binary log** is a "binlog" after Chapter 1, which is also what its file extension and every switch that produces one call it.
- Each chapter ends with **Exercises**: small, hands-on tasks against the sample projects. They are optional, but MSBuild rewards experimentation more than most subjects. **Appendix K is the answer key:** one note per exercise giving the expected result and the reason for it, so an exercise that asks you to predict something has somewhere to check the prediction.

The sample projects

Every code listing in this book is *read from a real file* when the book is compiled. Nothing is transcribed, and nothing can drift: a listing pointing at a file that does not exist is a build error, and every buildable project is compiled before the book ships. What you see is what ran.

Unless a chapter says otherwise, that rule holds everywhere: listings are read from the sample archive at compile time, buildable projects are compiled by CI, and captured console output is the real output of the command shown next to it, trimmed to the relevant lines, with absolute paths shortened to archive-root-relative ones and home paths to ~. The chapters do not repeat this. Where one *departs* from it (a file under `_nobuild/` that is meant to be read rather than built, a pre-captured binlog, a pipeline definition no local build can run), it says so at that point, and only there.

Those files ship as a single sample archive, and the exercises assume you have it.

Get it, then check it. Four steps, once:

1. Download the archive, a single `msbuild-in-depth-samples.zip`, from `https://goto.meziantou.net/msbuild-in-depth-samples.zip`
2. Unzip it anywhere. The folder it creates is the **archive root**: it holds `README.md`, `global.json`, and a `samples/` directory, and it is a Git repository with everything committed, so `git status` shows what you have changed.
3. Open a terminal *there*. Every command in this book is written relative to that folder.
4. Run the smoke test below. If both commands succeed, you are set up.

Worth knowing before you start:

- **Run commands from the archive root**, the folder holding the `README.md` and `global.json`, not from inside a project directory. Every path printed in this book is relative to that root, which is why the commands read `dotnet build samples/ch05-targets/…` rather than `dotnet build`. **When a command is meant to be run from somewhere else, the chapter says so on the line before it**, so an unlabeled command is always an archive-root command.
- **The SDK is pinned.** A `global.json` travels with the samples and constrains the .NET SDK version, so a sample evaluates identically for you and for the machine that validated it. It pins a *feature band* rather than one exact build: any .NET 10 SDK from the 10.0.4xx band at or above the version named will do, so `dotnet --version` need not match it digit for digit, but it must fall inside it. If it does not, fix that first. Chapter 26 explains the file, its `rollForward` policies, and why a pinned SDK is the first rule of a reproducible build.
- **The console uses the classic logger.** In an interactive terminal, `dotnet build` and `dotnet msbuild` normally use the *terminal logger*, a live progress display that hides most of the `Message` output the listings in this book print. The archive therefore ships `samples/Directory.Build.rsp`, a response file containing `-tl:off`. MSBuild reads the nearest such file above each project it builds, so every sample prints the classic output shown in the book. Add `-tl:on` to a command to see the terminal logger instead; Chapter 7 covers both, and your own repositories get the terminal logger until you opt out.
- **`_nobuild/` is intentional.** Folders with that name hold files meant to be *read, not built*: legacy non-SDK projects, fragments that are not whole projects, and builds that are intentionally broken so a chapter can show the error they produce. They are excluded from the build-everything command. A failure there is the point, not a problem.
- **What may differ.** The `*-output.txt` files hold the captures described above. Yours should match apart from absolute paths, timings, machine names, the SDK patch version (the last digits of `10.0.4xx`, and the MSBuild

version that comes with it), and the console format if you turn the terminal logger back on. If anything else differs, that difference is worth understanding.
- **`fixtures/` holds pre-captured inputs**, mostly `.binlog` files, so the debugging chapters can be followed without first reproducing a specific build.

Before the first exercise, do one smoke test from the archive root:

```
dotnet --version
dotnet build samples/ch01-intro/HelloBuild/HelloBuild.csproj
```

The first command should print the SDK selected by `global.json`; the second proves your shell is in the right folder and the sample archive is intact. Some exercises ask you to break or edit a sample. Because the archive is a Git repository, putting it back is one command: each chapter's closing reset note gives the `git checkout` and `git clean` lines for its own folder, so the next exercise starts from the intended files.

The archive's own `README.md` carries a chapter map (which folder belongs to which chapter) and the exercise loop in full.

Let's open the box. Turn to Chapter 1.

Chapter 1 — What Is MSBuild?

Most .NET developers meet MSBuild without noticing. You press **F5** in Visual Studio, or you type `dotnet build`, a few seconds pass, and a DLL appears in a `bin` folder. The thing that turned your source files into that DLL — deciding *what* to compile, in *what order*, with *which* references, and *whether* any of it even needed to run again: that is MSBuild.

This chapter builds the mental model the rest of the book depends on. We will not write anything complicated yet. The goal is to answer three questions precisely: what MSBuild *is*, where it *lives*, and what happens when you build.

MSBuild is a build engine, not a compiler

The most useful thing to learn early: **MSBuild does not compile C#.** The C# compiler (Roslyn, `csc`) compiles C#. MSBuild is the *engine* that *decides to invoke* the compiler, after it has figured out the inputs, resolved the references, and confirmed the work is not already up to date.

More precisely, MSBuild is a general-purpose engine that:

- reads an XML **project file** describing a build,
- evaluates it into a set of **properties** (scalar strings) and **items** (lists with metadata),
- and executes **targets** (ordered units of work), which in turn run **tasks**, the small pieces of .NET code that do the work (compile, copy, delete, run a tool, and so on).

Compiling a C# project is the most common *configuration* of that engine. The same engine builds F#, C++ (via a different toolset), NuGet packages, installers, and entirely custom pipelines. Once you see MSBuild as "a program that runs a graph of tasks driven by declarative XML," the `.csproj` file stops being magic and becomes something you can read.

Where MSBuild lives

There is not one MSBuild on your machine. There are usually several, and knowing which one runs matters when behavior differs.

- **The .NET SDK MSBuild.** When you run `dotnet build`, `dotnet msbuild`, or `dotnet pack`, you are using the MSBuild that ships *inside* the .NET SDK. It runs on .NET (cross-platform: Windows, macOS, Linux). **This is the MSBuild this book teaches**, pinned to the .NET 10 SDK.
- **Visual Studio's MSBuild.** Visual Studio ships its own copy that runs on .NET Framework. Building in the IDE uses it, and it also drives *design-time builds* that power IntelliSense.
- **`msbuild.exe` (.NET Framework).** The standalone Windows executable, also .NET Framework based, installed with Visual Studio or the Build Tools.

They share the same engine and the same project files, but they can differ in bundled SDK version, default tasks, and available runtime APIs. When a distinction matters in this book, it is called out as a **Version note**; unless stated otherwise, assume the SDK's `dotnet` MSBuild.

The cast of characters

`dotnet build` looks like one tool but orchestrates several:

Component	Role
`dotnet`	The driver CLI. `dotnet build` is essentially a wrapper that runs MSBuild.
MSBuild	The engine. Evaluates the project and runs the target graph.
NuGet	Restores packages. Integrated as MSBuild targets (`Restore`).
Roslyn (`csc`)	The C# compiler, invoked by the `Csc` task inside the `CoreCompile` target.
The .NET SDK	The bundle that contains all of the above plus the props/targets that define *how* a modern project builds.

A build is these pieces cooperating. MSBuild is the conductor.

A ten-minute “hello build”

Here is a complete, minimal .NET project. Two files make up the whole thing.

The project file:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

And the program:

```
Console.WriteLine("Hello from MSBuild!");
```

Notice what is **not** here: no list of source files, no reference to the .NET runtime, no compiler invocation. The `Sdk="Microsoft.NET.Sdk"` attribute pulls in a large body of default properties, items, and targets that supply all of that. Chapter 2 opens up what that attribute brings in.

Build it:

```
dotnet build samples/ch01-intro/HelloBuild/HelloBuild.csproj
```

MSBuild evaluates the project, restores packages if needed, runs the compile, and copies the result to `bin/`. Run the output and it prints `Hello from MSBuild!`.

Your first binary log

Text output tells you what a build *said*. A **binary log** (binlog) records what a build *did*: every target, every task, every property and item, the environment, even the contents of files read. It is the single most valuable debugging artifact in MSBuild, and we will lean on it throughout the book.

Producing one is a single flag:

```
dotnet build samples/ch01-intro/HelloBuild/HelloBuild.csproj -bl
```

That writes `msbuild.binlog` in the folder you ran it from. You cannot read it in a text editor (it is structured and compressed), so here, right now, is roughly what you just captured:

msbuild.binlog — the first two levels

```
Build (succeeded)
├── Evaluation
│   └── HelloBuild.csproj       ← every property and item, settled
├── HelloBuild.csproj (Restore)
└── HelloBuild.csproj (Build)
    ├── CoreCompile
    │   └── Csc                 ← the task that ran the compiler
    └── CopyFilesToOutputDirectory
```

Build, then project, then target, then task. Hanging off the project node is every property and item the engine settled on. Nothing is thrown away.

Figure 1: The top of a binlog, as the MSBuild Structured Log Viewer shows it. You do not need to understand any of it yet; the point is that it is a *tree*, and that the tree has the shape of the build itself.

Four levels: the build, then each project, then each target inside it, then each task inside that. The properties and items on the project node are Chapters 3 and 4, targets are Chapter 5, and tasks are Chapter 6, so by the end of Part I every node in that tree has a name. That is why we open the log properly only in Chapter 9, in the Structured Log Viewer (or "the viewer"), once the words mean something. Chapter 8 also shows how to replay a binlog as ordinary text, and Appendix F how to hand one to an AI assistant.

For now the habit matters more than the tool: add `-bl` whenever a build does something you do not understand. **Capture the build while the evidence still exists.** A build you captured is a build you can investigate later; a build you only watched scroll past is gone.

Gotcha. A binlog is thorough, and that cuts both ways. It embeds your project files and the environment variables the build saw, so it is safe on your own disk and *not* safe to attach to a public issue unexamined. What Figure 1 shows is the top of that tree; Chapter 8 covers what is inside the rest of it and how to share one safely. Capture freely; think before you send.

Exercises

1. Build `HelloBuild` twice in a row and compare the two outputs. Then build it a third time with `-v:normal` added, and search that output for lines starting with `Skipping target`. Which targets did MSBuild decide it could skip? (We explain the mechanism, incremental build, in Chapter 5.)
2. Add `-v:detailed` to the build and skim the output. Do not try to understand all of it; notice how much is happening beneath `dotnet build`.
3. Capture a binlog with `-bl`. You are not expected to understand the file yet: the point is to start the habit of capturing evidence while the build is fresh. Keep it if you like; once Chapter 9 has you install the viewer, it opens this file like any other.

Checking your answers. Appendix K gives the expected result for every exercise in this chapter, and the reason behind it. Read it after your attempt, not instead of one.

Reset when you are done. From the archive root, `git status` shows what you changed, `git checkout -- samples/ch01-intro` puts this chapter's sources back, and `git clean -fdX samples/ch01-intro` clears the build output a half-finished experiment left behind.

Carry forward

- **MSBuild is an engine.** It runs a graph of tasks. Compiling C# is one configuration of that engine, not its definition.
- **Two phases.** First *evaluation* turns XML into properties and items. Then *execution* runs targets and tasks. Confusing the two is the source of most MSBuild surprises; Chapter 3 makes the distinction concrete.

- **The SDK supplies the defaults.** Your `.csproj` is short because `Microsoft.NET.Sdk` brings thousands of lines of props and targets with it.
- **Binlogs are ground truth.** When in doubt, capture with `-bl` and inspect. Every later chapter refines one of these four ideas.

Chapter 2 — Project File Anatomy

A `.csproj` file is XML that MSBuild reads. Once you know the handful of elements it can contain, any project file, however large, becomes navigable. This chapter names the parts, then shows what the modern one-line `Sdk` attribute is really doing.

The elements you will see

Every MSBuild project is a `<Project>` root containing some combination of:

Element	Purpose
`<PropertyGroup>`	Declares **properties**: scalar string values (Chapter 3).
`<ItemGroup>`	Declares **items**: lists with metadata (Chapter 4).
`<ItemDefinitionGroup>`	Declares default **metadata** for every item of a type (Chapter 4).
`<Target>`	A named, ordered unit of work (Chapter 5).
`<Import>`	Pulls in another `.props`/`.targets` file (Chapter 11).
`<UsingTask>`	Registers a custom **task** so targets can call it (Chapter 17).
`<Sdk>` / `Sdk="…"`	Shorthand for a pair of implicit imports (below).

That is essentially the whole vocabulary. The complexity of real builds comes not from more element *types* but from how many of these are combined and imported.

A minimal SDK-style project

Here is a small but complete project that exercises the two most common elements, a `PropertyGroup` and an `ItemGroup`:

```
<Project Sdk="Microsoft.NET.Sdk">

  <!-- Properties: scalar configuration for the build -->
  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <!-- Items: lists of things, here a file embedded into the assembly.
       Calculator.cs is NOT listed: the SDK includes *.cs automatically. -->
  <ItemGroup>
    <EmbeddedResource Include="banner.txt">
      <LogicalName>Banner</LogicalName>
    </EmbeddedResource>
  </ItemGroup>

</Project>
```

Two things are worth pausing on.

First, `Calculator.cs` is compiled even though it is **not** listed anywhere. The SDK contributes a default `ItemGroup` that globs `**/*.cs` into the `Compile` item. You only write `<Compile>` entries when you want to *change* that default. If you list a file that is already globbed, the build fails with a duplicate item error. Chapter 4 covers items and globbing in depth.

Second, the `EmbeddedResource` item carries a child element, `<LogicalName>`. That is **item metadata** — a name/value pair attached to one item. Metadata is what makes items richer than a plain list of strings.

What `Sdk="Microsoft.NET.Sdk"` does

The `Sdk` attribute is shorthand for two implicit `Import` statements that wrap your project:

- **`Sdk.props`** is imported at the very *top*, before your content. It sets up default properties and the default items (the `**/*.cs` glob among them).
- **`Sdk.targets`** is imported at the very *bottom*, after your content. It defines the targets that turn properties and items into a build, plus the defaults that depend on values you chose.

So a modern project is really a sandwich. Properties are evaluated in file order, so the order of the three layers decides who overrides whom:

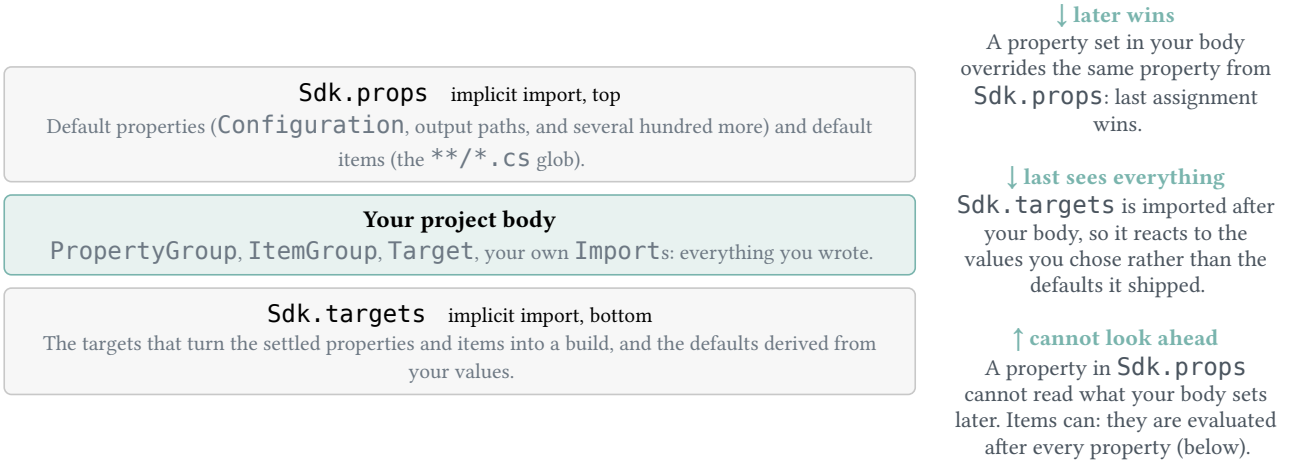


Figure 2: The import sandwich. Properties are evaluated top to bottom, so each layer can override what the layer above it set, and only the bottom layer sees every final value.

This ordering is intentional. Because `Sdk.props` runs *first*, it sets defaults that your project can *override* (you get the last word on properties). Because `Sdk.targets` runs *last*, the SDK's targets can *react* to the final values you chose. Keep this "props early / targets late" shape in mind: it is the same pattern you will use yourself in Chapters 11 and 12.

Most SDK defaults also carry a `Condition` that sets them only when the property is still empty. That guard is not there to protect against your project body, which comes later and wins anyway. It protects values set *earlier*: an environment variable, or a `Directory.Build.props` file, which `Sdk.props` imports before its own defaults (Chapter 12). Without the guard, the SDK would overwrite them.

> **Under the hood.** The SDK is resolved on disk by an *SDK resolver* before evaluation begins. `dotnet --info` shows the SDK in use, and `global.json` pins it; this repository pins the .NET 10 SDK so every sample evaluates identically. You can see the fully expanded sandwich with `dotnet msbuild -pp` (preprocess), which we use in Chapter 10.

For contrast: the legacy (non-SDK) format

Before 2017, project files looked very different. Nothing was implicit: every source file was listed, references were spelled out, and two `Import` statements had to be written by hand.

Do not try to decode this yet. You have not met properties, items, targets, or imports in detail, and nothing later in this chapter depends on understanding a single line of it. Skim it for *shape* (how much is written out) and read the four bullets underneath. That contrast is what matters here; the mechanics are Chapters 3 to 6, and Chapter 31 migrates a larger project of the same shape.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Legacy, non-SDK project format (pre-2017). Shown for contrast only.
     It targets .NET Framework and relies on Full-Framework MSBuild, so it is
     kept under _nobuild and is not compiled by the sample build on this SDK. -->
<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Common.props" />
  <PropertyGroup>
```

```
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <OutputType>Library</OutputType>
    <RootNamespace>Anatomy.Legacy</RootNamespace>
    <AssemblyName>Anatomy.Legacy</AssemblyName>
    <TargetFrameworkVersion>v4.8</TargetFrameworkVersion>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
  </ItemGroup>
  <!-- Every source file must be listed explicitly -->
  <ItemGroup>
    <Compile Include="Calculator.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
  </ItemGroup>
  <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
</Project>
```

Compare it to the minimal SDK project above and the SDK's value is obvious:

- Every `.cs` file is listed explicitly in a `<Compile>` item.
- The `Import` of `Microsoft.Common.props` (top) and `Microsoft.CSharp.targets` (bottom), the same props/targets sandwich, is written out by hand instead of being implied by `Sdk=`.
- Framework references are enumerated one by one.
- The root carries an `xmlns` namespace, which MSBuild required before version 15 (Visual Studio 2017), and a `ToolsVersion`, which modern MSBuild ignores. An SDK-style project needs neither.

You will still meet this format in older repositories, and Chapter 31 covers migrating it to SDK-style. This sample targets .NET Framework and needs .NET Framework MSBuild, so it lives under `_nobuild` and is shown for reading, not building.

Evaluation runs in passes

When MSBuild evaluates a project, it does not read it once from top to bottom. It makes **several passes** over the whole project, implicit imports included, each pass handling one kind of element:

1. **Properties** (and imports), in file order, across every file.
2. **Item definitions** (`<ItemDefinitionGroup>`).
3. **Items**, in file order.
4. **`UsingTask`** registrations.
5. **Targets**, which are only recorded here; they run later.

Within a pass, order matters: a property's final value depends on what came after it, and an import's position decides whether it can see, or be overridden by, the surrounding project. Across passes, order does not: every item is evaluated after *every* property is settled. This project reads `$(Audience)` twice, both times above the line that sets it:

```
<Project Sdk="Microsoft.NET.Sdk">

  <!-- Evaluation runs in passes: every property first (in file order, across
       all imports), then item definitions, then items, then targets. Both
       lines below read $(Audience) BEFORE the line that sets it. -->

  <PropertyGroup>
    <!-- Properties pass: Audience has not been assigned yet, so this is []. -->
    <EarlyCopy>[$(Audience)]</EarlyCopy>
  </PropertyGroup>

  <ItemGroup>
    <!-- Items pass: runs after EVERY property is settled, so it sees world. -->
    <Greeting Include="Hello-$(Audience)" />
```

```
  </ItemGroup>

  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Audience>world</Audience>
  </PropertyGroup>

  <Target Name="Show">
    <Message Importance="high" Text="Property read early: $(EarlyCopy)" />
    <Message Importance="high" Text="Item read early:     @(Greeting)" />
  </Target>

</Project>
```

```
dotnet msbuild samples/ch02-anatomy/Passes/Passes.csproj -t:Show
```

```
  Property read early: []
  Item read early:     Hello-world
```

The property saw nothing, because the properties pass had not reached `Audience` yet. The item saw `world`, because the items pass started only after the last property was assigned. That is also how the default `**/*.cs` glob can sit in `Sdk.props`, above your project, and still obey the `EnableDefaultCompileItems` switch you set in your body (Chapter 4). The next chapter builds on the properties pass.

> **Core now / later.** This chapter showed more than it expects you to hold. **Core now:** the root elements and what each is for, and the fact that evaluation settles every property before any item, with the SDK's props before your file and its targets after it. That is enough to read any `.csproj`. **Later:** the default item globs (Chapter 4), how the SDK is resolved on disk and how you would write your own (Chapter 22), and preprocessing (the `-pp` switch used in the exercises below), which gets its proper treatment in Chapter 10. Use them now as recipes; they become mechanisms in their own chapters.

Exercises

1. Run `dotnet msbuild samples/ch02-anatomy/Minimal/Minimal.csproj -pp:out.xml` and open `out.xml`. Find the implicit imports of `Sdk.props` and `Sdk.targets` and notice how much was added around your handful of lines.
2. In `Minimal.csproj`, add `<Compile Include="Calculator.cs" />` and build. Read the error, then remove it again — you have seen the default glob collide with an explicit include.
3. List the targets available to the minimal project with `dotnet msbuild samples/ch02-anatomy/Minimal/Minimal.csproj -targets`. Almost none of them are in your file — they came from the SDK.

> **Checking your answers.** Appendix K gives the expected result for every exercise in this chapter, and the reason behind it. Read it after your attempt, not instead of one.
>
> **Reset when you are done.** From the archive root, `git status` shows what you changed, `git checkout -- samples/ch02-anatomy` puts this chapter's sources back, and `git clean -fdX samples/ch02-anatomy` clears the build output a half-finished experiment left behind.

> **Carry forward**
>
> - A project file is XML with a small vocabulary: `<PropertyGroup>`, `<ItemGroup>`, `<Target>`, `<Import>`, `<UsingTask>`, and the occasional `<ItemDefinitionGroup>`. Everything else in this book is those elements at larger scale.

- **Evaluation runs in passes**: every property (in file order, across all imports), then item definitions, then items, then `UsingTask`s, then targets. An item always sees final property values; a property sees only what was assigned above it.
- **`Sdk="Microsoft.NET.Sdk"` is two imports**, not a mode. `Sdk.props` is spliced in above your content and `Sdk.targets` below it. Your project file is the filling in that sandwich.
- SDK-style projects are short because the SDK supplies the defaults and the item globs, not because they do less. The ceremony is still there; it lives somewhere you did not have to type.
- The legacy non-SDK format is the same engine with all of that ceremony written out by hand. Recognizing it is enough for now; Chapter 31 migrates one.
- When the shape of a project is not obvious from the file in front of you, `dotnet msbuild -pp` prints the whole sandwich flattened. Chapter 10 makes a habit of it.

End of preview

The full edition continues with the other 29 chapters and 11 appendices.

meziantou.net

MSBuild In Depth

Building, Extending, and Debugging .NET Builds

Every `dotnet build`, every press of F5, every NuGet package and CI pipeline runs through MSBuild — and for most developers it stays a black box. This book opens it. Starting from the engine's two phases and its four core concepts, it builds up to custom tasks, code generation, packaging, and shipping build logic — then goes deep on the part most guides skip: *seeing* what a build actually did, with binary logs, the Structured Log Viewer, and the binlog MCP server.

What you will learn

- Read any `.csproj`, `.props`, or `.targets` file
- Write custom tasks and targets in C#
- Debug builds with binlogs and the MCP server
- Ship build logic in NuGet packages
- Master evaluation vs. execution
- Structure large repos and multi-targeting
- Diagnose and fix broken incrementality
- Make builds reproducible in CI/CD

About the Author

Gérald Barré, known online as **meziantou**, is a software developer with more than ten years of industry experience and a recognized **Microsoft MVP**.

He works across the .NET stack: C#, ASP.NET Core, Blazor, Entity Framework Core, and the tooling around them, including MSBuild, the subject of this book. His interests span web development, software architecture, design patterns, and developer productivity.

Gérald writes regularly at **meziantou.net**, where he has published hundreds of articles on .NET and C#. He maintains several widely used open-source projects, including **Meziantou.Analyzer** (a set of Roslyn analyzers) and **Meziantou.Framework**.

Gérald Barré
Microsoft MVP · meziantou.net

Microsoft Most Valuable Professional