

copilot-instructions

Contents

GitHub Copilot Instructions for .NET Workshop	1
0) Workshop Mode	1
1) Workflow (TDD + Build Hygiene)	2
2) Solution & Project Architecture (Clean Architecture)	2
3) C# Coding Style (C#/.NET conventions)	2
4) DDD Modeling Rules	3
5) Testing Rules	3
6) Conventional Commits	3
7) Object Calisthenics (lightweight)	3
8) Documentation Organization	4
9) Practical Scaffolds & Prompts (use verbatim)	4
9.1 Generate a new Aggregate (Domain)	4
9.2 Minimal API endpoint (Api) + DI wiring	4
9.3 Unit test pattern (xUnit + FakeItEasy)	5
9.4 Conventional commit + PR helper	5
10) Guardrails (Workshop)	5
Appendix A — Expected Layout (example)	5
Appendix B — OpenTelemetry Observability	5

GitHub Copilot Instructions for .NET Workshop

These instructions are automatically applied to all GitHub Copilot suggestions in this repository.

0) Workshop Mode

- Assume **.NET 9**, **xUnit**, **FakeItEasy**, **Minimal API**, **ILogger**, **async/await** everywhere.
- Prefer **Clean Architecture** solution layout and **DDD** patterns.
- Always generate examples and code in **English**.

1) Workflow (TDD + Build Hygiene)

- **TDD first:** when asked to implement a feature, propose/emit tests before code.
 - After you output code, assume we run `dotnet build && dotnet test` and fix warnings/errors before committing.
 - When referencing rule sets, state what you followed (e.g., "Used: Clean Architecture, DDD, Tests").
-

2) Solution & Project Architecture (Clean Architecture)

Generate and maintain the following structure:

`[project].Domain`, `[project].Application`, `[project].Infrastructure`,
`[project].Api`, plus `tests/[project].UnitTests` and `tests/[project].IntegrationTests`.
Enforce dependencies:

- **Domain** → no deps
 - **Application** → Domain only
 - **Infrastructure** → Application + Domain
 - **Api** → Infrastructure only
- Prefer **feature-oriented folders** (e.g., `Order/`, `Customer/`) instead of technical groupings.

Implementation guidance:

- **Domain:** business logic, entities, value objects, domain events. No external deps.
 - **Application:** use cases, commands/queries, ports (interfaces).
 - **Infrastructure:** adapters, EF/database, external integrations.
 - **Api:** minimal endpoints + request/response mapping only.
 - Use DI; avoid circular deps; no mediator library for this workshop.
-

3) C# Coding Style (C#/.NET conventions)

- File-scoped namespaces; one type per file; 4-space indent; **PascalCase** for types/members, **camelCase** for locals/parameters; constants ALL_CAPS.
- Prefer clarity over brevity; use **nameof** in exceptions; **make classes sealed by default** unless inheritance is intentional.

See `.github/instructions/csharp.instructions.md` for full C#/.NET coding standards and best practices.

4) DDD Modeling Rules

- Model **Aggregates** with factory methods (no public constructors), encapsulate invariants, and avoid direct navigation to other aggregates.
- **Entities** live inside aggregates; no public setters; lifecycle managed by the root.
- **Value Objects** are immutable; implement value equality.
- Prefer **Strongly-Typed IDs** (e.g., `OrderId`) as value objects.
- **Repositories** are interfaces for aggregate roots with **business-intent method names** (no generic CRUD verbs in domain).

Method naming: favor ubiquitous language (e.g., `PlaceOrder`, `MarkAsShipped`) over `Create/Update/Delete/Get`.

5) Testing Rules

- **Test framework:** `xunit.v3`.
 - **Mocks:** `FakeItEasy`.
 - **Integration:** `Testcontainers` (and `Microcks` if doing contract tests).
 - Unit tests target **Domain + Application** only; Integration tests target **Infrastructure + Api**.
 - Organize tests by **feature** and, for complex types, by **class-per-method** folders.
 - Keep tests descriptive; run tests frequently (`dotnet test / dotnet watch`).
-

6) Conventional Commits

- Use `<type>([optional scope]): <description>` with 72-char subject limit.
- Types: `feat|fix|docs|style|refactor|perf|test|build|ci|chore|revert`.
- Keep one logical change per commit; use scope to denote layer/feature.

Examples

```
feat(api): add order endpoint
fix(domain): correct order validation logic
test(order): add unit tests for order creation
chore: update dependencies
```

7) Object Calisthenics (lightweight)

When refactoring, prefer these constraints to keep code small and intention-revealing:

- One level of indentation per method; avoid **else** with guard clauses ("fail fast").
 - Wrap primitives into meaningful types; prefer **first-class collections**.
 - Avoid long call chains ("one dot per line" guideline).
 - Don't abbreviate names; keep classes/methods small and focused.
 - Limit domain classes' setters; prefer factories and invariants.
-

8) Documentation Organization

- **All documentation** must be placed in the `docs/` directory at repository root
 - Project README.md stays at root, but detailed docs go in `docs/`
 - Documentation types and locations:
 - Architecture Decision Records (ADRs): `docs/adr/`
 - API documentation: `docs/api/`
 - User guides and tutorials: `docs/guides/`
 - Design documents: `docs/design/`
 - Lab exercises and workshop materials: `docs/labs/` or `docs/`
 - Use clear, descriptive filenames: `docs/api/authentication-guide.md` not `docs/auth.md`
 - Always link to docs from main README.md with relative paths
 - When generating documentation, ask: "Should this go in docs/ or is it the main README?"
-

9) Practical Scaffolds & Prompts (use verbatim)

9.1 Generate a new Aggregate (Domain)

Create a DDD aggregate in `[project].Domain/Order`:

- Strongly-typed IDs (`OrderId`, `CustomerId`)
- Private ctor + static factory `Order.Create(...)`
- Invariants: `quantity > 0`, `price > 0`
- Value object `Address { Street, City, Country }` as immutable record
- Business methods: `RegisterOrderItem`, `ChangeShippingAddress` (raise domain event)
- No navigation to other aggregates; no public setters

9.2 Minimal API endpoint (Api) + DI wiring

In `[project].Api` (Minimal API):

- Add endpoints: `GET /orders/{id}`, `POST /orders`
- Map requests to Application commands/queries; no business logic in Api
- Register DI for repositories/services in Infrastructure
- Use `ILogger`, `async/await`, proper 400/404/500 handling with `ProblemDetails`

9.3 Unit test pattern (xUnit + FakeItEasy)

Create tests in `tests/[project].UnitTests/OrdersTests/`:

- One test class per method (`CreateTests.cs`, `RegisterOrderItemTests.cs`)
- Use `FakeItEasy` for collaborator interfaces
- Use descriptive test names and cover invalid scenarios (guard clauses)

9.4 Conventional commit + PR helper

Write a Conventional Commit subject (≤ 72 chars) and a PR description with:

- Intent
 - Scope (layer/feature)
 - Risk/impact
 - Linked issue(s)
-

10) Guardrails (Workshop)

- Do **not** invent external dependencies without being asked.
 - Keep domain logic **out of Api/Infrastructure**.
 - Prefer small, composable methods; log meaningfully with `ILogger<T>`.
 - If a rule conflicts, **Clean Architecture boundaries win** (then DDD, then Style).
-

Appendix A — Expected Layout (example)

```
src/  
  Sales.Domain/  
  Sales.Application/  
  Sales.Infrastructure/  
  Sales.Api/  
tests/  
  Sales.UnitTests/  
  Sales.IntegrationTests/
```

Appendix B — OpenTelemetry Observability

- Use `OpenTelemetry` for distributed tracing in API layer
- Configure `ActivitySource` for custom spans in Application layer
- Add meaningful tags to activities (e.g., task ID, operation type)
- Use console exporter for workshop demos (no complex infrastructure needed)