ChatGPT

.NET Core CLI GitHub Repository Best Practices

Last Updated (UTC): 2025-05-27

Why Organization Matters

- **Clarity and Maintainability:** A well-structured repository makes it easy for developers to find code and understand project areas. Keeping different concerns (UI, business logic, data access, etc.) in separate modules or layers prevents messy tangling of functionality [1]. This separation of concerns leads to cleaner code boundaries and easier maintenance over time.
- **Scalability (Team and Codebase):** As projects and teams grow, a clear structure minimizes merge conflicts and coordination issues. A logical division (e.g. multiple projects in one solution or even multiple repos for distinct services) allows teams to work in parallel without “fighting” over the same files [2]. It also enables more granular permission control if needed for larger organizations.
- **Avoiding Entanglement:** A common anti-pattern is cramming all features into a single project. This often mixes unrelated functionality (for example, GUI code alongside database code) and makes it hard to locate or isolate changes [1]. Good organization, such as using separate class libraries or layers, ensures changes in one area have minimal impact on others, reducing the risk of regressions.
- **New Developer Onboarding:** A consistent structure helps newcomers. When folder names, project names, and namespaces follow predictable patterns, new contributors can quickly navigate the codebase. It reduces the time spent figuring out where to add new classes or features [3] [4]. In short, organization communicates *how* the system is put together, which accelerates understanding.

Directory & File Layout

A standard and consistent directory layout at the repository root is highly recommended. Typical top-level folders include:

- `src/` – Contains the source code of your application or services (often one subfolder per project or component) [5]. For example, a solution might have `src/YourApp.Web`, `src/YourApp.Core`, `src/YourApp.Data`, etc., each containing a `.csproj` for that module.
- `tests/` – Contains all test projects, mirroring the structure/names of the projects they test [6]. For example, `YourApp.Core.Tests` alongside `YourApp.Core` ensures test code is kept separate from production code. This separation prevents shipping test code with the app and keeps production assemblies lean [7].
- `docs/` – *(Optional)* Documentation files, such as additional guides, FAQs, or design docs [8]. This could include a `README.md` (if not in root) or more detailed developer guide, installation instructions, etc., organized for easy reference.
- `build/` – *(Optional)* Build scripts or CI/CD pipeline definitions. Some projects use a `build/` directory to hold scripts (PowerShell, Bash, YAML pipelines) or configuration for building the project (e.g. Docker files, cloud deployment scripts) [8]. Similarly, an `artifacts/` folder is often used to store build outputs or packages [8] (excluded via .gitignore).
- **Miscellaneous:** Other common files at the root include:

- `README.md` – A high-level overview of the project, build/run instructions, and status badges [9]. This is the first entry point for new users or contributors.
- `LICENSE` – The license file for open-source projects (e.g. MIT, Apache 2.0) [10].
- `.gitignore` – A standard git ignore file to exclude build output, packages, user-specific and IDE files (like `bin/`, `obj/`, `.vs/`, `*.user` files, etc.) [11]. Using a well-curated .gitignore (e.g. the official Visual Studio template) keeps the repository clean of transient files.
- `.editorconfig` – Coding style settings to ensure consistency across editors [11]. This helps maintain uniform formatting (tabs/spaces, naming, etc.) for all contributors.

By following a consistent layout (for instance, always using `src` and `tests` folders), developers can jump between projects with ease. It also allows automation (build or test scripts) to easily locate code and tests. The example below illustrates a simplified standard layout:

```
YourRepo.sln
├── src
│   ├── YourApp.MainProject/
│   │   └── YourApp.MainProject.csproj
│   ├── YourApp.CoreLibrary/
│   │   └── YourApp.CoreLibrary.csproj
│   └── YourApp.DataAccess/
│       └── YourApp.DataAccess.csproj
├── tests
│   ├── YourApp.MainProject.Tests/
│   │   └── YourApp.MainProject.Tests.csproj
│   └── YourApp.CoreLibrary.Tests/
│       └── YourApp.CoreLibrary.Tests.csproj
├── docs/ *(e.g. additional docs or guides)*
├── build/ *(e.g. build scripts, CI workflows)*
├── .editorconfig
├── .gitignore
├── LICENSE
└── README.md
```

This structure separates concerns: all production code under **src**, all test code under **tests**, and supporting materials (docs, scripts) in their own places [5] [6]. Consistent naming (matching test project names to the corresponding library names) further reinforces this clarity [12].

Solution (.sln) Best Practices

- **Single Solution File:** Keep a single `.sln` at the root of the repo that includes *all* projects (including test projects). This makes it easy to build or open the entire codebase at once in an IDE [13] [10]. In .NET Core, the CLI can work without a solution, but having one is beneficial for IDE integration and for running all tests together (e.g. `dotnet test` on the solution will run every test project) [14].
- **Commit the .sln to VCS:** Always include the solution file in source control. Whenever projects are added or removed, the `.sln` updates; keeping it versioned ensures that a fresh clone can be

opened and built without manual steps [15]. A committed solution file allows building historical versions of the repository easily, which is crucial for reproducibility.

- **Organization within Solution:** Use Solution Folders (virtual folders in Visual Studio) if needed to group projects logically (e.g. put all test projects under a "Tests" folder in Solution Explorer). This has no effect on build but helps developers navigate in Visual Studio. Keep the solution structure consistent with the folder structure – for example, if projects are under a `src` folder on disk, mirror that grouping in the solution for clarity.
- **Managing Solutions via CLI:** Leverage .NET CLI commands to manage the solution. For example, use `dotnet new sln` to create a solution, and `dotnet sln add <project>` to add projects to it [16]. This ensures the solution file stays updated. It's also a convenient way to script project setup or CI builds.
- **Multiple Solutions (Advanced):** In general one solution is sufficient. However, for very large repos, you might maintain multiple solution files (e.g. a lightweight solution with just core projects vs. a full solution) to improve IDE performance. If so, clearly document their intended use. In all cases, prefer consistency – avoid having some projects only in certain solutions by accident.

Project (.csproj) Best Practices

- **Use SDK-Style Projects:** Always use the modern SDK-style `.csproj` format for .NET Core/5+/6+ projects. This format is leaner (no verbose `<Compile>` entries for each file) and is now the default for all new .NET projects. There is virtually *no* reason to use the old legacy format – SDK-style should be the default for any new project [17]. It simplifies project files and enables new tooling features.
- **Consistent Naming:** Name project files and default namespaces according to their purpose and scope. Good naming makes the function of a project immediately clear [18]. For example, include descriptive terms like *"Web"*, *"API"*, *"Data"*, *"Tests"* in the project name. Consistently use a naming scheme (e.g. `Company.Product.Layer` or `Product.Feature`) across all projects in the solution [19]. This consistency should extend to assembly names and namespaces, which by default follow the project name.
- **Project References over Binary References:** Structure projects such that internal dependencies use project-to-project references (`<ProjectReference>`) rather than referencing compiled DLLs. This ensures that builds pick up the latest code and enables features like **transitive NuGet restore**. The SDK-style projects make adding references easy (`dotnet add reference`) and keep paths relative, so you don't need to hard-code output paths.
- **TargetFramework and Platforms:** In each `.csproj`, specify the appropriate `<TargetFramework>` (or plural `<TargetFrameworks>` for multi-targeting libraries). Use a specific TFM like `net8.0` rather than broad old frameworks unless necessary. If you multi-target, ensure the code is tested on all targets. Keep target frameworks consistent across projects when possible to avoid confusion.
- **Enable Nullable Reference Types:** Make sure `<Nullable>enable</Nullable>` is turned on in all project files (this is on by default for new .NET Core projects) [20]. This feature provides compile-time null-safety checks, catching potential `NullReferenceException` issues early. Embracing nullability is a best practice for modern .NET development to improve code quality.
- **Treat Warnings as Errors:** In production projects, consider treating compiler warnings as errors (`<TreatWarningsAsErrors>true</TreatWarningsAsErrors>` in the csproj) [21]. This forces the team to address potential issues promptly rather than letting warnings accumulate. It keeps the codebase tidy and prevents minor issues from slipping by. (You can also configure specific warning waves or codes to exclude if needed, via `<NoWarn>` or `<WarningsNotAsErrors>`.)

- **Analyzers and Code Style:** Leverage built-in .NET analyzers and code style settings. For example, set `<AnalysisLevel>latest</AnalysisLevel>` and `<AnalysisMode>All` to enable all recommended rules [22]. Pair this with an `.editorconfig` to define style conventions, and use `<EnforceCodeStyleInBuild>true</EnforceCodeStyleInBuild>` so that style/style-rule violations surface as build warnings or errors [23]. This enforces consistent coding standards across the project.
- **Centralize Common Settings:** Use a `Directory.Build.props` file at the repository root (next to the .sln) to define properties common to all projects (e.g. target framework version, company name, version numbers, Nullable enabled, analyzer settings) [24]. Centralizing these reduces duplication and ensures all projects use the same versions and settings. It's easier to update one file than many individual csproj files.
- **NuGet Package Management:** Use `<PackageReference>` in csproj for NuGet dependencies (this is default in SDK-style projects). Avoid committing binaries or packages into the repo; instead, use NuGet feeds to manage third-party libraries. If you must include a library that isn't on NuGet, consider a `lib/` folder for it [25], but this should be rare. Keep packages updated and consider tools like `dotnet outdated` or Dependabot for automated package updates to patch vulnerabilities or bugs.
- **Configurations and Secrets:** Do not hard-code environment-specific configurations in csproj. Use appsettings.json or user-secrets for sensitive data rather than storing them in project files. Keep the csproj focused on build configuration (target frameworks, output type, etc.). If you need different builds (Debug/Release), use the standard `<PropertyGroup Condition="...'">` constructs or multiple yaml pipelines rather than maintaining separate csproj for each config.

VS Code Workspace Best Practices

- **Include VS Code Configs:** If your team uses Visual Studio Code, check in the `.vscode` folder with useful configurations. Specifically, the **launch** and **tasks** files can be generated by VS Code (`.NET: Generate Assets for Build and Debug`) and should be added to source control [26]. This ensures that debugging and common tasks (like running the app or tests) are pre-configured for all contributors. For example, a `launch.json` might include configurations to run the web project and attach the debugger, and `tasks.json` might have a task to build the solution or run tests.
- **Workspace Recommendations:** Consider adding a `.vscode/extensions.json` file to recommend the official C# extension (and any other relevant extensions) to developers when they open the repository. This helps ensure everyone has the required tooling (OmniSharp, etc.) for a good experience. You can also include settings in `.vscode/settings.json` to tune editor behavior (though most code style settings should live in .editorconfig for multi-IDE consistency).
- **Open Folder (Repo) as Workspace:** Organize the repo so that developers can open the repository folder directly in VS Code. OmniSharp (the C# language server) will detect the solution or projects. Having the `.sln` at the root helps OmniSharp auto-load all projects [13] [27]. If the solution is not found automatically, developers can use the command palette to select the solution (`OmniSharp: Select Project` command) – but this is rarely needed if the structure is conventional.
- **Multi-Project Debugging:** If your repo has multiple startup projects (say an API and a separate frontend in the same repo), you can add a *compound* launch configuration in `launch.json` to start them together. Ensure documentation (or the README) explains how to use the provided launch configs. For example, a compound config can launch the API and then the frontend (or a test runner) simultaneously for integrated debugging.

- **Keep VS Code Settings in Sync:** Try to have VS Code settings align with other IDE settings. For instance, if using EditorConfig for formatting, VS Code will respect it. If there are workspace-specific nuances (like paths for local tools, or enabling/disabling specific IntelliSense features), document them or include them in the `.vscode/settings.json`. The goal is that a developer using VS Code can clone the repo, open it, and hit F5 to run, without manual setup beyond installing the .NET SDK and recommended extensions.

Onboarding Essentials

- **Comprehensive README:** Provide a clear `README.md` at the root with project description, prerequisites, build and run steps, and contribution guidelines. This is the first stop for new developers and potential open-source contributors. A well-written README helps readers understand the project and get started quickly [28]. It should cover how to build the project (e.g. "run `dotnet build` or open with Visual Studio"), how to run tests (`dotnet test` or other), and how to run the application. Include any setup needed (for example, environment variables, database migrations, etc.). Keeping the README concise but informative is key – if more detail is needed, link out to docs in the `/docs` folder or a wiki [29].
- **Contribution and Community Docs:** If the project is open to external contributors, include files like `CONTRIBUTING.md` (for guidelines on how to file issues, submit PRs, coding standards) and a `CODE_OF_CONDUCT.md` (especially for .NET Foundation projects) in the repository. These files make it explicit how to participate and the expected behavior in the community. Many established projects include these, and GitHub will surface them to newcomers.
- **License and Metadata:** Ensure a proper LICENSE file is present, especially for open source. This removes ambiguity about usage rights. The license type should also be mentioned in the README badge or NuGet package if applicable. Additionally, include project metadata like a `.gitignore` (as mentioned) and perhaps a `README` badge for build status (CI) and NuGet version if this is a library.
- **Environment Setup:** Document any required tools or SDKs. For .NET Core projects, indicate the target .NET SDK version. If a specific SDK is required, consider including a `global.json` file at the root to lock the expected SDK version [30]. This ensures that developers and CI use the correct .NET SDK, avoiding "works on my machine" issues when different SDKs are installed.
- **Quick Start Scripts:** For complex setups, provide scripts to streamline onboarding. For example, a `build.ps1` or `build.sh` at the root (or under `/build`) that restores, builds, and tests the solution with one command. You might also include a `run.ps1` script to launch the app with default settings. While not strictly necessary, these can significantly lower the bar for someone new to get the project running locally.
- **Continuous Integration:** Set up continuous integration (GitHub Actions, Azure DevOps, etc.) to automatically build and test pull requests. New contributors will appreciate seeing immediate feedback on their changes. Also consider adding a CI status badge in the README. A project with CI ensures that the main branch is always in a healthy state (build passes, tests green) for anyone pulling the latest code [31].
- **Issue Templates and Wiki:** For larger projects, using GitHub's ISSUE_TEMPLATE and PULL_REQUEST_TEMPLATE can guide contributors to provide necessary info. A project wiki or `/docs` folder can host more detailed architecture docs or design decisions history, which is invaluable for onboarding new team members or open-source contributors.
- **Developer Environment Consistency:** Encourage use of the same coding standards and tools. For instance, if everyone uses `dotnet format` or StyleCop Analyzers to enforce style, mention that. If there are recommended IDE settings (like enabling nullable, code analyzers) these should already be

part of the repo config as discussed (editorconfig, Directory.Build.props). The onboarding doc can mention these so newcomers install any necessary VS Code extensions or Visual Studio components.

(Onboarding is all about making the first experience with the repo smooth – clear documentation, automated checks, and consistency go a long way.)

Anti-Patterns to Avoid

Anti-Pattern	Why It's Problematic and Best Practice
Single Monolithic Project <br/>(Entire app in one .csproj)	Mixing all concerns in one project makes the codebase unwieldy and hard to navigate [1]. It's difficult for multiple developers to work on disparate areas without stepping on each other's toes. **Best Practice:** Split by layer or functionality into multiple projects (e.g., UI, Core, Data). This enforces separation of concerns and yields cleaner, more modular code.
Tests Embedded in Product Code <br/>(or no separate test project)	Including unit tests in the same assembly as production code means you might ship test classes with the app, and it bloats the output [7]. It also makes it harder to find tests vs. implementation in the same project. **Best Practice:** Keep tests in dedicated test projects (with a `.Tests` suffix) under a `/tests` directory. This way tests are not deployed and can reference the product code via project references, maintaining a clear divide [32].
Inconsistent or Vague Naming <br/>(Projects/folders named poorly)	If project names or namespace prefixes don't reflect their purpose, contributors get confused. For example, a project called "Utils" or "Misc" says nothing about its role. **Best Practice:** Use self-explanatory names and a consistent convention [19]. For instance, include keywords like *Api*, *Web*, *Data*, *Tests* in names. Keep folder structure in sync with namespaces for clarity [33]. Consistency here means less guesswork when navigating the repo.
Checking in Build Outputs <br/>(bin/, obj/, packages)	Committing compiler outputs or NuGet packages pollutes the repo and causes giant diffs. These files are generated or restored and should be git-ignored [11]. Including them risks platform-specific artifacts in version control. **Best Practice:** Rely on **.gitignore** to exclude `bin/`, `obj/`, and other build artifacts. Only source code, config, and necessary assets should be in the VCS. For third-party libs, use NuGet PackageReference; don't directly include DLLs unless absolutely unavoidable.
Lack of Documentation <br/>(No README or comments)	Repositories without a README (or with a very outdated one) deter newcomers – they have no guidance on what the project does or how to build it [28]. Similarly, poor inline documentation makes code hard to understand. **Best Practice:** Always maintain an up-to-date README.md at minimum, plus inline XML comments or markdown docs for key components. A well-documented repo attracts collaborators and reduces onboarding time [34].

Anti-Pattern	Why It's Problematic and Best Practice
No Automated Testing or CI <br/ >(Everything is manual)	Relying on developers to remember to run all tests or perform certain checks leads to human error. Absence of continuous integration means bugs or style violations can slip through until they're caught by someone manually. **Anti-Pattern:** "It works on my machine" syndrome. **Best Practice:** Set up CI pipelines to build and test on each push/PR [31]. Automate what you can – testing, linting, deployment – so the process is consistent and reliable. This ensures code quality and builds remain high, and any issues are flagged early.

Die-Hard Requirements

(Use this checklist to verify the repository meets the most critical standards.)

- [ ] **Builds Cleanly from Scratch:** A new clone of the repository should build and test successfully using a single command (e.g. `dotnet build && dotnet test`). All dependencies (NuGet packages, submodules) should restore automatically. No manual tweak should be required to get a green build.
- [ ] **Single Source of Truth for Versioning:** Use a single approach for version numbers (e.g. `VersionPrefix` in CSProj or Nerdbank.GitVersioning). This avoids inconsistencies in assembly versions or NuGet package versions across projects.
- [ ] **Nullable Reference Types Enabled:** All projects have nullable reference types turned on to prevent null bugs at compile time [20].
- [ ] **No Warnings in Build (Treat as Errors):** The project builds with zero warnings. Warnings are treated as errors in CI builds to prevent overlooking them [21].
- [ ] **Static Analysis and Style Enforcement:** Code analysis rules are enabled (at least the default Microsoft analyzers) and code style is enforced via EditorConfig. The build should fail or warn on major rule violations (e.g. using `<EnforceCodeStyleInBuild>true</EnforceCodeStyleInBuild>`) [23].
- [ ] **Consistent Code Formatting:** An `.editorconfig` is present at the root with rules for C# code style, and the code adheres to it (no huge diffs from running `dotnet format`). This ensures a uniform coding style across contributions [11].
- [ ] **Unit Tests Cover Critical Logic:** There is a test project (or several) covering the core logic of the application. Important functionality has corresponding tests, and tests are run as part of the CI pipeline on each PR.
- [ ] **Continuous Integration Implemented:** A CI workflow (e.g. GitHub Actions, Azure Pipelines) is configured to build the solution and run tests on every push/PR [31]. The status of these checks is visible to maintainers and contributors (e.g., via badges or the PR checks UI).
- [ ] **Proper Documentation is Provided:** The repository includes a README with getting-started info, a LICENSE file for open source, and if open for contributions, a CONTRIBUTING guide. All major parts of the system have at least high-level documentation (either in the README or a docs folder).
- [ ] **No Sensitive Data in Repo:** Connection strings, passwords, or secret keys are not hardcoded in the repository. (Use user-secrets, environment variables, or secure stores for development secrets.) The `.gitignore` should prevent common secret files (like .*user or* .pswd files) from being committed.

- [ ] **Adheres to .NET Project Conventions:** The project uses the standard .NET conventions – e.g., project naming, framework targeting, and deployment structure follow typical .NET patterns (so that any .NET developer finds the repo familiar). It also means following the framework's guidelines like disposing of `IDisposable` objects, using async appropriately, etc., to maintain code health.

Each item above is a non-negotiable aspect of a professional-grade .NET Core repository. Checking all boxes helps ensure the project is robust, maintainable, and welcoming to contributors.

Appendix A: Sample Northwind Solution Structure

To solidify these concepts, let's consider an example: a fictitious **Northwind Traders** .NET Core solution (a classic sample domain). Below is a possible layout following the best practices described, along with snippets from the solution and project files:

Directory Layout (Northwind Example)

```
Northwind.sln
├── src
│   ├── Northwind.Api/                  -- ASP.NET Core Web API project
│   │   ├── Northwind.Api.csproj
│   │   └── (Controllers, Startup.cs, etc.)
│   ├── Northwind.Core/                 -- Class library for core domain logic
│   │   ├── Northwind.Core.csproj
│   │   └── (Entities, Services, etc.)
│   └── Northwind.Data/                 -- Class library for data access (EF
Core)
│       ├── Northwind.Data.csproj
│       └── (DbContext, Migrations, etc.)
├── tests
│   ├── Northwind.Core.Tests/           -- xUnit test project for Core
│   │   ├── Northwind.Core.Tests.csproj
│   │   └── (Unit test classes for domain logic)
│   └── Northwind.Data.Tests/           -- xUnit test project for Data
│       ├── Northwind.Data.Tests.csproj
│       └── (Unit test classes for data logic, e.g. repository tests)
├── Northwind.sln                       -- Solution file referencing all above
projects
├── README.md                           -- Project overview and build
instructions
├── Northwind.postman_collection.json   -- (Example: API sample requests, if
applicable)
├── global.json                         -- (Pins .NET SDK version, if needed)
├── .editorconfig                       -- Coding style definitions
├── .gitignore                          -- Git ignore for Visual Studio/Dev
```

```
files
└── LICENSE                                -- License for the project (e.g. MIT)
```

Notes: In this layout, the `Northwind.sln` at the root ties together three projects under *src* and two under *tests*. Each test project is named after the project it tests (Core and Data respectively) [12]. The **Api** project would reference both **Core** and **Data** projects, and the test projects would reference their target projects. The separation ensures the API and data layers stay decoupled except through the Core (for example, `Northwind.Data` might only depend on `Northwind.Core` interfaces, following a clean architecture approach).

Snippet: Solution and Project References

Below is an illustrative excerpt from **Northwind.sln** showing how projects are listed, and a snippet from **Northwind.Api.csproj** demonstrating project references:

```
Project("{GUID1}") = "Northwind.Api", "src\Northwind.Api\Northwind.Api.csproj",
"{GUID1}"
Project("{GUID2}") = "Northwind.Core",
"src\Northwind.Core\Northwind.Core.csproj", "{GUID2}"
Project("{GUID3}") = "Northwind.Data",
"src\Northwind.Data\Northwind.Data.csproj", "{GUID3}"
Project("{GUID4}") = "Northwind.Core.Tests",
"tests\Northwind.Core.Tests\Northwind.Core.Tests.csproj", "{GUID4}"
Project("{GUID5}") = "Northwind.Data.Tests",
"tests\Northwind.Data.Tests\Northwind.Data.Tests.csproj", "{GUID5}"
EndProject
...
Global
    GlobalSection(SolutionConfigurationPlatforms) = preSolution
        Debug|Any CPU = Debug|Any CPU
        Release|Any CPU = Release|Any CPU
    EndGlobalSection
    ... (Other solution configuration sections)
EndGlobal
```

(The solution file lists each project with a unique GUID and relative path. In practice, Visual Studio or `dotnet sln add` *manages these entries.)*

And here is a partial **Northwind.Api.csproj** (SDK-style) highlighting key settings and references:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <AssemblyName>Northwind.Api</AssemblyName>
    <!-- Other settings like Version, ASP.NET launch profiles, etc. -->
  </PropertyGroup>

  <ItemGroup>
    <!-- Project references to Core and Data projects -->
    <ProjectReference Include="..\Northwind.Core\Northwind.Core.csproj" />
    <ProjectReference Include="..\Northwind.Data\Northwind.Data.csproj" />
  </ItemGroup>

  <ItemGroup>
    <!-- Example NuGet package references -->
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="7.0.5" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.5.0" />
  </ItemGroup>
</Project>
```

Key points in this `.csproj` snippet:

- It uses `Microsoft.NET.Sdk.Web` (appropriate for an ASP.NET Core project). Other class libraries use `Microsoft.NET.Sdk`. All are SDK-style, as evident by the `<Project Sdk="...">` top line.
- `<TargetFramework>net8.0</TargetFramework>` specifies .NET 8.0 – all projects target the same version for consistency.
- Nullable reference types and implicit `using` directives are enabled (these are default for new projects, reinforcing best practices).
- The **ProjectReference** entries link to the Core and Data projects, allowing the Api to use their internals. This replaces older techniques of sharing DLLs – the build will ensure Core/Data build first and the Api auto-references those outputs.
- Package references (e.g., EntityFrameworkCore, Swagger) are included via `<PackageReference>` tags. No `.packages` folder or checked-in binaries exist – NuGet will restore these on build.

By following this structure, the Northwind example solution remains clean, navigable, and adherent to .NET conventions. New developers can quickly understand the separation of layers: **Api** for web concerns, **Core** for domain logic, **Data** for database access, with tests validating each layer. The projects are loosely coupled: for instance, if a new UI (say, a Blazor frontend) is needed, it could be added as another project referencing Core (and perhaps Data), without touching the existing Api project – thanks to the clear boundaries.

References

1. Riccardo Rigutini, *"Common Practices In .NET Project Structure"*, C# Corner, 2021. – Describes common project organization patterns and naming conventions in .NET [1] [19].
2. Emanuele Bartolesi, *".NET Project Folders Structure"*, Dev.to, 2022. – Offers a standard folder layout for .NET projects (src, tests, docs, etc.) and explains the purpose of each item [8] [11].
3. Stack Overflow – *"Should a .sln be committed to source control?"*, answered by Arnkrishn, 2009. – Consensus is to always include the solution file in version control for consistency [15].

4. Microsoft Learn, *"Testing in .NET / Running tests"*, 2024. – .NET CLI allows running all tests in a solution with a single command (`dotnet test` on the .sln) [14].
5. Stack Overflow – *"Is SDK style project the recommended approach for new .NET projects?"*, comment by canton7, 2021. – Advises that SDK-style .csproj format should be used for all new .NET (including .NET Framework) projects [17].
6. Anton Martyniuk, *"Best Practices for Increasing Code Quality in .NET Projects"*, antondevtips.com, 2024. – Suggests enabling Nullable, treating warnings as errors, and using Directory.Build.props to enforce code quality rules across projects [35].
7. Jon Skeet, Stack Overflow answer to *"Do you put unit tests in the same project or another project?"*, 2008. – Recommends separating unit tests into their own assemblies, citing issues with shipping tests with product code and bloated assemblies [7].
8. Marcel.L, *"GitHub Repository Best Practices"*, Dev.to, 2024. – Covers general repo management tips; emphasizes clear README documentation [28] and automating workflows with CI (GitHub Actions) [31].
9. .NET Foundation, *"dotnet/new-repo Template"* (GitHub repository), 2023. – The template for new .NET projects, which includes standard files like README, LICENSE, .gitignore, and a checklist for prepping a repo for public release [36].
10. Aaron Bos, *"Debugging .NET in VS Code"*, Personal Blog, 2022. – Explains how VS Code sets up `launch.json` and `tasks.json` for .NET projects, and recommends committing these to share debugging configurations with the team [26].

---

[1] [3] [4] [6] [12] [18] [19] [33] Common Practices In .NET Project Structure
https://www.c-sharpcorner.com/article/common-practices-in-net-project-structure/

[2] [16] dotnet Core project structure best practices | by James Reátegui | Medium
https://medium.com/@jamesrf/dotnet-core-real-world-best-practices-per-james-p1-structure-93487f59519e

[5] [8] [9] [10] [11] [25] .NET Project Folders Structure - DEV Community
https://dev.to/kasuken/net-project-folders-structure-36o8

[7] [32] c# - Do you put unit tests in same project or another project? - Stack Overflow
https://stackoverflow.com/questions/347156/do-you-put-unit-tests-in-same-project-or-another-project

[13] .NET Core is Sexy — Building a Web API | by Jeremy Buisson | ITNEXT
https://itnext.io/net-core-is-sexy-building-a-web-api-cdb470cc8222

[14] Testing in .NET - .NET | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/core/testing/

[15] visual studio - Should a .sln be committed to source control? - Stack Overflow
https://stackoverflow.com/questions/1033809/should-a-sln-be-committed-to-source-control

[17] c# - Is SDK style project the recommended approach for new .NET Framework projects? - Stack Overflow
https://stackoverflow.com/questions/68083511/is-sdk-style-project-the-recommended-approach-for-new-net-framework-projects

[20] [21] [22] [23] [24] [35] Best Practices for Increasing Code Quality in .NET Projects
https://antondevtips.com/blog/best-practices-for-increasing-code-quality-in-dotnet-projects

[26] Debugging .NET in VS Code
https://aaronbos.dev/posts/debug-dotnet-vs-code

[27] C# - Visual Studio Marketplace
https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp

[28] [29] [31] [34] GitHub Repository Best Practices - DEV Community
https://dev.to/pwd9000/github-repository-best-practices-23ck

[30] global.json overview - .NET CLI - Learn Microsoft
https://learn.microsoft.com/en-us/dotnet/core/tools/global-json

[36] GitHub - dotnet/new-repo: Recommended template for new .NET Foundation repos
https://github.com/dotnet/new-repo