**Tender Management API Backend – Technical Specification and Development Guide**

**Alireza Nobakht**

This guide was written to help anyone working on the Tender Management API understand how it was built and why certain choices were made. It’s not just a technical manual — it’s meant to tell the story of how the system came together, step by step.

Whether you're setting it up, adding a new feature, or just curious how things work under the hood, this guide should give you a clear and friendly path through the backend. It covers everything from the main ideas behind the design to the smaller details that keep things running smoothly.

**Architectural Choices**

I began by sketching the architecture before writing any code because it felt like drawing a floor plan before building a house. A quick sketch costs nothing to adjust, but changing concrete walls later would be painful. By settling the big picture early, I gave myself a shared map that business people, testers, and future maintainers can all point to when they ask, “Where does this feature live?”

I chose a four-layer, Domain-Driven “Clean Architecture” because it keeps each piece of the codebase in its own drawer. The Presentation layer is the front door: it handles HTTP requests, checks JWT badges, and says “come in” or “try again.” The Application layer is the hallway switchboard: it validates input, coordinates work, and never touches database details directly. The Domain layer is the living room where the real business rules sit—rules about deadlines, bid limits, and status changes live here, protected from technical noise. Finally, the Infrastructure layer is the basement with all the plumbing: Entity Framework handles the heavy lifting for saves, Dapper delivers quick read-only queries, and services like email or logging plug in down here.

Several design choices flow naturally from that layout. I let EF Core handle writes because it manages change tracking and optimistic concurrency for me, while Dapper takes care of read-heavy endpoints so the API stays snappy under load. I keep tender and bid statuses in a lookup table instead of hard-coding words in an enum; when the business invents a new status like “Archived,” I can add a row in SQL instead of redeploying the whole API. Every call is async/await so the server doesn’t waste threads waiting on I/O. And each request carries a compact JWT token that states who the caller is and whether the caller is an Admin or Vendor, so restricted endpoints can check the badge instantly.

By fixing the architecture first, I created clear boundaries that make the code easier to test, quicker to reason about, and ready for tomorrow’s changes—whether that means splitting pieces into micro-services or plugging in a new mobile client.

**Initial Solution Scaffolding**

I began by opening Visual Studio and creating a blank solution named TenderManagement. A blank solution is only an empty container, but it lets me gather every future project in one place.

Inside that solution I added four projects that match the clean-architecture layers I planned earlier:

* Tender.Api — an ASP.NET Core Web API project
* Tender.Application — a class-library project
* Tender.Domain — a class-library project
* Tender.Infrastructure — a class-library project

Each project targets .NET 8, so the whole codebase builds on a consistent runtime.

I then set up project references so dependencies flow only inward. Tender.Api references *Application*, *Application* references *Domain*, and *Infrastructure* references both *Domain* and *Application*. This one-way path prevents accidental shortcuts that would break the layering.

To protect future changes I added an xUnit test project called Tender.Tests and referenced both *Domain* and *Application*, laying the groundwork for unit and integration tests. I also initialised a local Git repository and committed this skeleton, ensuring every new file is tracked from the start.

Finally, I made Tender.Api the startup project and ran the solution. Seeing the default weather-forecast endpoint return JSON confirmed the empty shell compiles and runs, so real feature work can now begin on solid ground.