**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.

**Creating the Domain Skeleton**

With the solution compiling, I turned to the Tender.Domain project and gave it some structure. First, I created two folders—Entities and ValueObjects—to separate full-blown domain objects from smaller, intrinsic types.

Inside Entities I added blank class files for the six core concepts the brief calls out: User, Vendor, Category, Status, Tender, and Bid. I also dropped in a BaseEntity class that every future entity will inherit from once I wire up common features like IDs, row-version tokens, and domain-event support.

Under ValueObjects I introduced two empty classes, Money and Deadline. These will later wrap primitive types to enforce rules such as “amount must be positive” or “deadline can’t be in the past,” but for now they are just placeholders.

After adding these files I built the solution again—everything still compiles, which proves the new scaffolding hasn’t broken anything. I committed this snapshot so the domain vocabulary is anchored in the repository before any behaviour gets added.

Database Design Overview

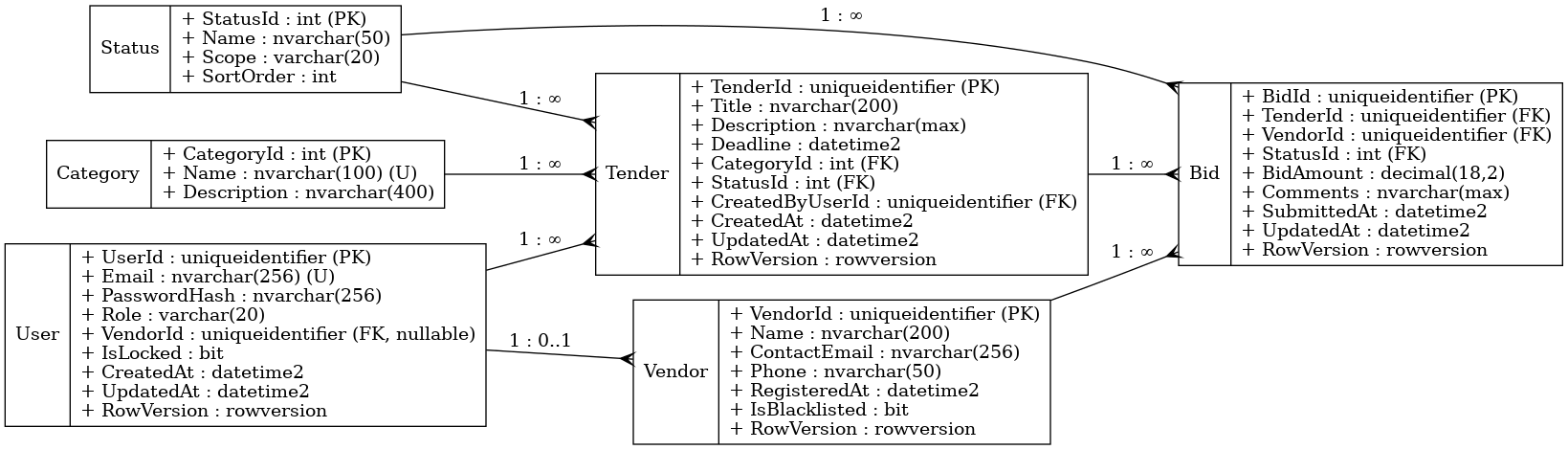
To keep the data clean, future-proof, and easy to query I normalised the schema to third normal form and grouped tables around the core business concepts.

Key design points

* Reference tables for lookup data – Status (shared by both Tender and Bid) and Category prevent duplicated text and let new values be added without code changes.
* Aggregate boundaries – Tender collects many Bid rows, while Vendor and User are separate roots; this lines up with the domain layers and the Unit-of-Work pattern.
* Integrity constraints – unique (TenderId, VendorId) on Bid stops duplicate bids; row-version columns enable optimistic concurrency; all monetary values use decimal(18,2) with a CHECK (BidAmount > 0).
* Audit and soft-delete ready – timestamp and RowVersion columns are in place, allowing soft-delete or full audit trails later without altering the structure.

Entity–relationship diagram

The diagram below shows tables, primary keys, and the main foreign-key links.



schema overview

* User — stores login credentials and role.
  + PK UserId (GUID)
  + Email (unique), PasswordHash, Role (“Admin” / “Vendor”), VendorId (nullable FK), audit fields, RowVersion.
* Vendor — company profile for bidding.
  + PK VendorId (GUID)
  + Name, ContactEmail, Phone, RegisteredAt, IsBlacklisted, RowVersion.
* Category — tender classification lookup.
  + PK CategoryId (int)
  + Name (unique), Description.
* Status — reusable status values for both tenders and bids.
  + PK StatusId (int)
  + Name, Scope (“Tender” or “Bid”), SortOrder.
* Tender — project open for bids.
  + PK TenderId (GUID)
  + Title, Description, Deadline, CategoryId (FK), StatusId (FK), CreatedByUserId (FK), audit fields, RowVersion.
* Bid — vendor proposal on a tender.
  + PK BidId (GUID)
  + TenderId (FK), VendorId (FK), StatusId (FK), BidAmount (decimal 18,2, CHECK > 0), Comments, timestamps, RowVersion.
  + Unique index on (TenderId, VendorId) prevents duplicate bids per vendor per tender.

This schema gives each layer of the application a solid, consistent backbone while staying flexible for future requirements like vendor blacklisting, multi-currency bids, or tender archival.

I began by listing the real-world things the system cares about—users, vendors, tenders, bids, statuses, and categories—and made each one its own table. That step alone satisfies first normal form: every column now holds a single value, and there are no repeating groups tucked inside one row.

Next, I checked for partial dependencies to hit second normal form. The classic culprit is a table with a multi-column key where some non-key data depends on only part of the key. In this design each table has a single-column surrogate primary key (either a uniqueidentifier or an int), so no column can drift and depend on “half a key.” For example, the bid amount and comments rely solely on BidId, not on TenderId or VendorId.

Finally, I removed transitive dependencies to reach third normal form. Descriptive data such as status names and category names live in their own lookup tables, Status and Category. The main tables hold only the foreign-key IDs, so a tender row no longer carries the text “Closed” or “Construction”—that information sits where it belongs and changes in exactly one place. The same logic pushed vendor contact details into Vendor instead of letting them leak into User; a user merely points at the vendor it represents. As a result every non-key column now depends on nothing but the key, the whole key, and nothing else.

By separating lookup values, using surrogate keys, and enforcing foreign-key constraints I avoided duplicate data, kept updates atomic, and prepared the schema for easy expansion—new statuses or categories can be inserted without touching any application code.

**Domain & Infrastructure**

I expanded the Tender.Domain project first.  
Inside the *Entities* folder I created seven classes:

* BaseEntity – holds Id, timestamps, RowVersion, and a tiny event list.
* User, Vendor, Category, Status, Tender, and Bid – each now carries the properties defined in the schema.

In the *ValueObjects* folder I added:

* Money – a positive-amount wrapper around decimal.
* Deadline – a future-only wrapper around DateTime.

To let entities raise notifications later I introduced IDomainEvent under *Tender.Domain.Events* and referenced it from BaseEntity.

Next I touched the Tender.Infrastructure project.  
Under a new *Persistence* folder I added TenderDbContext, the class that tells Entity Framework how to map every domain type to SQL and where to enforce things like the unique-bid index and decimal precision.

Finally, in Tender.Api I registered that context in Program.cs and pulled in the EF Core packages so UseSqlServer resolves.

With these additions the solution now contains real domain models, a working DbContext, and the API knows how to reach the database—all while the codebase still compiles cleanly.