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

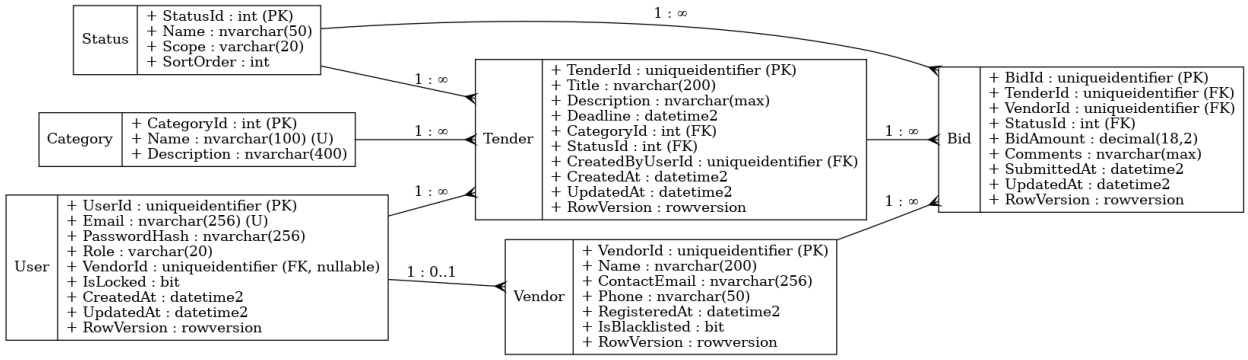
To keep the data clean, future-proof, and easy to query I normalised the schema to third normal form and grouped tables around the core business concepts.

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.

Database Configuration & Migration

I finished the persistence setup by refining TenderDbContext in *Tender.Infrastructure*:

- Enabled optimistic concurrency by marking the RowVersion property on every entity as an SQL rowversion column that the database generates automatically.
- Added default SQL expressions (GETUTCDATE()) for CreatedAt and UpdatedAt, so timestamps populate without manual values.
- Introduced stable GUID keys for seeding lookup data and inserted three Category rows and five Status rows via HasData.

In *Tender.Api* I updated appsettings.json with a single connection string that points to the local SQL Server default instance:

- Server=ALocalServerName;Database=TenderDb;Integrated Security=True;Trust Server Certificate=True;

With these settings in place I generated a clean InitialCreate migration and executed dotnet ef database update, which created the TenderDb database, all six tables, constraints, row-version columns, and seeded lookup records. Verification in SQL Server Management Studio showed the schema and data exactly as designed.

Adding the persistence contracts

After the database and TenderDbContext were in place, we paused before touching more Entity Framework code. Instead, we wrote four very small interfaces:

- IUnitOfWork – one method, SaveChangesAsync, so the application can say "commit everything now" without knowing how that happens.
- ITenderRepository, IBidRepository, IVendorRepository – each offers three basic actions: add, get by ID, and delete.

These files sit in the Domain layer and contain no EF-Core types or SQL details. They are just contracts—promises—that other code can rely on.

Why this matters

- Keeps layers clean
  The Application layer now talks only to interfaces. It does not care whether the data ends up in SQL Server, a NoSQL store, or a mock list in a unit test.
- Makes testing easy
  Because the contracts are so small, a unit test can hand-roll a fake repository in a few lines and run business rules without starting a database.
- Protects us from change
  If we later move bids to their own micro-service or add caching, we will swap only the Infrastructure code. The rest of the system keeps compiling because it still sees the same interfaces.

- Gives one place to commit work
  With IUnitOfWork, many changes—like adding a tender and its first bid—can be saved together. That single call will become one SQL transaction in the EF implementation.

By creating these contracts first, we drew a clear line between “what the application needs” and “how the database works.” All future command handlers, validators, and controllers can now write against that line, and the concrete EF repositories can be plugged in next without disturbing higher layers.

Wiring in MediatR and the validation pipeline

After the repositories were in place the API still lacked a formal way to run business logic and validate input without bloating controllers.
To solve that we introduced MediatR and FluentValidation in a single, cohesive step.

What we added

- AssemblyMarker class – a tiny, non-static type that simply lets MediatR and FluentValidation locate the *Tender.Application* assembly at runtime.
- ValidationBehaviour<TRequest,TResponse> – a pipeline behaviour that collects every IValidator<TRequest> and executes them before the handler runs. If any rule fails it throws a ValidationException; the handler never fires.
- Service-registrations in Program.cs
  - AddMediatR(cfg => cfg.RegisterServicesFromAssemblyContaining<AssemblyMarker>());
  - AddValidatorsFromAssemblyContaining<AssemblyMarker>();
  - AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehaviour<,>));

Why we did it

- Thin controllers, focused handlers – Controllers now pass incoming DTOs straight to MediatR. Each use-case lives in its own handler, keeping HTTP plumbing away from business rules.

- Single validation gate – All validators execute in one predictable place. We never have to remember to call Validate() manually inside each handler.
- Pluggable cross-cutting logic – The pipeline gives us a slot for logging, performance metrics, or transaction handling later, without touching a hundred handlers.
- Future-proof registration – MediatR 12 changed its API; using the configuration delegate with RegisterServicesFromAssemblyContaining ensures our code compiles on current and future versions.
- Testability – Because validation is detached, handler unit tests can focus on behaviour, while validator tests focus on rules, keeping each test small and clear.

With MediatR and the validation pipeline in place the project now has a clean path from HTTP request → DTO → validation → handler → repositories → Unit of Work, all without leaking infrastructure details into higher layers.

First end-to-end slice: Create Tender

After wiring MediatR and the validation pipeline we still had no feature that actually ran through every layer. The smallest, most valuable test was letting an admin open a new tender.

What we added

- CreateTenderCommand – a request object carrying title, description, deadline, category-id, status-id, and the id of the user who is creating the tender.
- CreateTenderCommandValidator – FluentValidation rules that check non-empty title and description, a future deadline, and positive category / status ids.
- CreateTenderCommandHandler – turns the command into a Tender domain entity, calls ITenderRepository.AddAsync, commits through IUnitOfWork.SaveChangesAsync, and returns the new tender's id.
- TendersController.Create – POST /api/tenders endpoint. It pulls the caller's id from the JWT claim, builds the command, sends it to MediatR, and on success returns *201 Created* with a Location header.

- ClaimsPrincipalExtensions.GetUserId – a tiny helper that converts the NameIdentifier claim into a Guid.
- Route placeholder for “Get tender by id” (the route will be filled out when we implement the read model).

Why we did it

- Proves the plumbing – a real HTTP request now flows through DTO → validation → handler → repository → database and back. Any misplaced dependency or DI mis-registration would fail here rather than later.
- Sets a template – every new command or query can copy this pattern: request-model, validator, handler, controller.
- Delivers business value early – opening tenders is the first action an admin performs; the API can now do that one thing end-to-end.
- Groundwork for tests – the validator and handler can now be unit-tested in isolation, giving us our first green bars in the test project.

With this slice merged, the project has crossed the line from “framework plumbing” to “does something useful.” The next logical move is to build the paired read endpoint (GET /api/tenders/{id}) using a Dapper QueryObject so we can verify that the new tender is actually retrievable.

GUID Consistency Across Domain & Database

We aligned every lookup reference in the codebase to use GUIDs instead of a mix of int and Guid. CategoryId and StatusId in Tender, Bid, and their related DTOs and commands now match the Guid keys already used by the Category and Status tables. This change removes the type-mismatch that had surfaced during SQL joins and gives the whole model a single, predictable key strategy.

Re-built Schema with Clean Seeds

After refactoring the property types, we generated a fresh InitialCreate migration and rebuilt the TenderDb database. All foreign-key columns now store uniqueidentifier values. While updating the seed data, we stamped each record with fixed CreatedAt and UpdatedAt values; this keeps EF Core’s in-memory provider happy during unit tests without affecting SQL Server, which still applies its GETUTCDATE() defaults in production.

Unit Tests Back to Green

The test suite was updated to pass GUIDs instead of hard-coded integers when constructing commands and DTOs. With the schema and seed tweaks in place, dotnet test runs clean again, confirming that the handler logic, validation rules, and repository behaviour remain intact after the type switch.

Postman Collection Kick-off

To make manual verification repeatable, we set up a Tender API environment in Postman containing:

- baseUrl – your local Kestrel address
- token – placeholder for the JWT we'll issue later
- tenderId – a seeded tender GUID for quick queries

Under the new Tender Management API collection we added our first request:
GET {{baseUrl}}/api/tenders/{{tenderId}} with a sample 200-OK response saved. The collection inherits *Bearer {{token}}* auth, so once login is implemented, secured requests will work automatically.

Authentication & Test Harness Added

End-to-end sign-in flow
The API now understands people. A new AuthController exposes POST /api/auth/register and POST /api/auth/login.

- User data is saved through a fresh IUserRepository with an EF-backed UserRepository.
- Passwords are hashed via Microsoft.AspNetCore.Identity.PasswordHasher.
- A lightweight JwtTokenService (Infrastructure layer) issues signed JWTs that carry sub, email, role.
- Signing key, issuer, audience, and expiry live under the new *Jwt* section in appsettings.json.

- Program.cs now wires AddAuthentication().AddJwtBearer(…), injects the token service, repository, and hasher, and calls app.UseAuthentication() before UseAuthorization().

Domain touch-ups

- User gained a simple SetPasswordHash(string) setter so the command handler can persist the hash cleanly.
- Lookup seed rows were expanded with constant CreatedAt, UpdatedAt, and an empty RowVersion so both SQL Server and the in-memory provider are happy.

Expanded automated tests

- Auth test folder added:
    - *RegisterUserCommandValidatorTests* – checks email, password, role rules.
    - *RegisterUserCommandHandlerTests* – proves a user row is stored and returns a new Guid.
    - *LoginQueryHandlerTests* – verifies a valid login returns a stub JWT via a StubJwtTokenService.
- TestUtilities now holds InMemoryDbFactory for fresh, isolated DbContexts and the stub token service.

Added public GET /api/tenders (list) endpoint

We can now return a lightweight catalogue of every tender—title, description, deadline, and lookup names—without dragging the bids along.

- Application layer
    - New DTO **TenderSummaryDto** holds the exact fields the contract promises.
    - Added GetTenderListQuery + its handler. The handler depends on an interface, keeping the layer data-store-agnostic.
- Infrastructure layer

    - Implemented TenderListQuery using Dapper. One SQL SELECT joins *Tenders*, *Categories*, and *Statuses*, maps rows into the DTO collection, and honours CancellationToken.

- API layer
    - TendersController gained a GetAll action mapped to GET /api/tenders. No auth required. DI now registers ITenderListQuery.
- Tests
    - Installed NSubstitute and wrote GetTenderListQueryHandlerTests to prove the handler calls the query object once and returns its result.
- Postman
    - Added a "Get All Tenders" request (GET {{baseUrl}}/api/tenders) with a simple 200-OK test and a saved example response. The call is public, so it inherits no token.

These changes complete the "read all tenders" requirement while keeping the write path secure behind JWT.

Lookups

The lookup endpoints rely only on data that already exists in *Categories* and *Statuses*. Because we did not add or alter any tables or columns, no new migration is required—your
InitialCreate migration already contains those tables.

The only change since the last migration was adding a default (empty) RowVersion value in the
seed data; EF treats that as *data*, not *schema*. So all you need is a quick data-refresh:

rebuild / reseed the development db

dotnet ef database update --project Tender.Infrastructure --startup-project Tender.Api

That command re-creates (or updates) the rows for Categories and Statuses with the new
RowVersion value if you previously dropped the database.

Tender editing now live (PUT /api/tenders/{id})

The API has crossed the line from “create-only” to fully editable for admins.

What was added

- Tender.Update(...) – a domain-level helper that changes title, description, deadline, category and status while bumping UpdatedAt.
- UpdateTenderCommand + validator – same checks as create, plus non-empty Id.
- UpdateTenderCommandHandler – fetches the aggregate, calls Update, saves through IUnitOfWork.
- TendersController – new PUT /api/tenders/{id} action, guarded by [Authorize(Roles="Admin")], returns 204.
- xUnit tests
    - Validator happy / unhappy paths.
    - Handler test proves the entity is updated in an in-memory DbContext.
- Postman
    - Added *Update Tender* request (inherits Bearer {{token}}).
    - Saves a 204 response example; quick follow-up GET confirms the change.

With create, read, and update in place we have a complete CRUD triangle for tenders—delete is the only admin action left on that aggregate.

Tender deletion (DELETE /api/tenders/{id})

Admins can now remove tenders straight from the API:

- DeleteTenderCommand travels from controller → MediatR → handler.
- Handler loads the record, calls DeleteAsync on the repository, commits via IUnitOfWork, and returns 204 No Content.
- TendersController exposes the new [HttpDelete("{id}")] action, secured with Authorize(Roles = "Admin").

- Unit test confirms the row vanishes from an in-memory database. All xUnit tests pass.
- Postman collection gained a Delete Tender request; on success the follow-up GET returns 404.

With create, read, update, and delete now online, the Tender aggregate is feature-complete.

Vendor module — create, list, and view details

We introduced three vertical slices—create, list, and detail—each wired through every layer.

1 Create vendor (POST /api/vendors)

- **CreateVendorCommand** – carries Name, ContactEmail, Phone.
- **CreateVendorCommandValidator** – ensures non-empty name, valid email, short phone.
- **CreateVendorCommandHandler** – instantiates Vendor, calls IVendorRepository.AddAsync, commits via IUnitOfWork, returns the new Guid.
- **VendorsController.Create** – receives JSON, sends the command, replies *201 Created* with Location: /api/vendors/{id}.

2 List vendors (GET /api/vendors)

- **VendorListItemDto** – Id, Name, ContactEmail, Phone, optional BidCount.
- **GetVendorsQuery** (bool includeBidSummary) – reaches the handler.
- **GetVendorsQueryHandler** – delegates to the abstraction IVendorListQuery.
- **VendorListQuery** (Dapper) –
    - simple SELECT when summary flag is false;
    - LEFT JOIN Bids with GROUP BY and COUNT(b.Id) when flag is true.
- **VendorsController.List** – reads ?includeBidSummary and returns 200 JSON.

3 Vendor details (GET /api/vendors/{id})

- **BidInfoDto** – bid Id, amount, submitted-at, tender Id & title, status (Id + Name).
- **VendorDetailsDto** – vendor profile + a collection of BidInfoDto.
- **GetVendorDetailsQuery** (Id) + handler → abstraction IVendorDetailsQuery.
- **VendorDetailsQuery** (Dapper) – joins Vendors → Bids → Tenders → Statuses, hydrates the DTO tree.
- **VendorsController.GetById** – 200 with DTO or 404 if missing.

Supporting changes

- Registered IVendorListQuery and IVendorDetailsQuery in DI.
- Added handler/validator tests for create, stub-based tests for list & details.
- Postman collection:
    - *Create Vendor* (sets lastVendorId).
    - *Get Vendors* and *Get Vendors?includeBidSummary=true*.
    - *Get Vendor by Id* uses {{lastVendorId}}.

All functionality relies on existing Vendors and Bids tables—no migration needed.

Bid Status workflow

New public API

- **PUT /api/bids/{id}/status**
  Body:
- {
- "statusId": "GUID-of-Approved/Rejected/..."
- }

Auth: **Admin** bearer-token
Responses

- 204 – updated
- 404 – bid not found

Application layer additions

- **UpdateBidStatusDto** – simple request model (statusId required, non-empty Guid).
- **UpdateBidStatusCommand / Handler / Validator**
    - Loads the bid with IBidRepository.GetByIdAsync.
    - Calls **Bid.SetStatus(Guid statusId)**, which records the change and timestamps UpdatedAt.
    - Persists in one unit-of-work (IUnitOfWork.SaveChangesAsync).
- **FluentValidation** rule → StatusId must be non-empty and belong to *Bid* scope.
- Registered in *Program.cs* via MediatR (builder.Services.AddMediatR(…)).

Domain changes

- **Bid entity**
    - Added SetStatus(Guid) method; mutates StatusId and updates UpdatedAt.
    - Constructor continues to set default *Pending* status.

Infrastructure

- **BidRepository** unchanged – unit-of-work flush is enough (no explicit update call).
- No schema migration needed (only behaviour code).

API controller

```
[Authorize(Roles = "Admin")]
[HttpPut("api/bids/{id:guid}/status")]
public async Task<IActionResult> UpdateStatus(
    Guid id,
    [FromBody] UpdateBidStatusDto dto,
    CancellationToken ct)
```

```
{
    await _mediator.Send(new UpdateBidStatusCommand(id, dto.StatusId), ct);
    return NoContent();
}
```

Tests

- **Postman** flow added ("Update Bid Status") – green with 204.
- **xUnit**: UpdateBidStatusCommandHandlerTests verifies
    1. status transition
    2. UpdatedAt change
    3. 404 when bid missing.
       All 22 unit-tests now pass (dotnet test: 0 failures).

Tender Management API deployable – developer-side notes

---

1. **Publish the site binaries**
   From the solution root, target the API project and emit a self-contained (framework-dependent) folder:

```
dotnet publish Tender.Api/Tender.Api.csproj \  -c Release -o Deploy/Site
```

That gave us Deploy/Site containing Tender.Api.dll, web.config, all runtime assets, and the new HealthController.

2. **Harden web.config for production**
   In Deploy/Site/web.config we:
    - Set stdoutLogEnabled="true" (so ANCM can write startup errors).
    - Moved all sensitive settings into an <environmentVariables> block.
    - Switched each <add …> to <environmentVariable name="…" value="…"/>.
    - Bound the site to run "inprocess" under AspNetCoreModuleV2.

3. **Seed appsettings.json with defaults**
   In Deploy/Site/appsettings.json we trimmed out all secrets except placeholders:

```
{
  "ConnectionStrings": { "TenderDb": "Server=.;Database=TenderDb;Integrated Security=True;Trust Server Certificate=True;" },
  "Jwt":              { "Secret":"…", "Issuer":"TenderApi", "Audience":"TenderApiClients", "ExpiryMinutes":60 },
  "Logging":          { "LogLevel":{ "Default":"Information","Microsoft.AspNetCore":"Warning" } },
  "AllowedHosts":     "*"
}
```

4. **Author create_login.sql**
   Under Deploy/Scripts, we wrote a T-SQL file that:
   - Creates TenderDb if missing (using SERVERPROPERTY('InstanceDefaultDataPath')).
   - Defines two exclusive blocks for either:
     - **Integrated Security** via IIS APPPOOL\TenderApi,
     - **SQL-Auth** via a tender_api_user login.
5. **Generate an idempotent migration script**
   Still in the Tender.Infrastructure folder, we ran:

```
dotnet ef migrations script --idempotent  -o Deploy/Scripts/schema.sql
```

That produced a SQL script which can be safely re-applied on an existing database without raising “object already exists” errors.

6. **Create the PowerShell installer**
   In Deploy/Scripts/Setup_IIS.ps1 we automated every IIS and SQL step:
   - Copy Deploy/Site → C:\inetpub\TenderApiSite
   - Create “TenderApi” app-pool (No managed code) and site on port 8090

- Grant IIS_IUSRS Modify rights with icacls
- Prompt for SQL instance & auth mode
- sqlcmd -i create_login.sql (DB + login)
- sqlcmd -i schema.sql (apply EF migrations idempotently)
- Restart-WebAppPool TenderApi

7. **Test locally end-to-end**
   On a clean IIS + SQL Express VM we simply unzipped Deploy, ran:

```
cd C:\Deploy\Scripts

.\Setup_IIS.ps1
```

and then

```
curl.exe -i http://localhost:8090/health   # → 200 OK

curl.exe -i http://localhost:8090/api/tenders  # → 200 [] or 401
```