Tender Management API – .NET C# Backend Evaluation Project

Overview

You are tasked with developing a **Tender Management API** for a company that oversees the publication of tenders and collection of vendor bids. Vendors can register and submit bids to open tenders, and administrators can review and update the bid statuses.

This project evaluates your skills in .NET backend development, database modeling, EF Core + Dapper integration, and overall software craftsmanship.

Objectives

You are expected to:

1. Design a normalized SQL Server schema (you define the table structures).
2. Use **Entity Framework Core (EF)** for **write operations** (e.g., POST, PUT, DELETE).
3. Use **Dapper** for **read operations**, especially where joins are involved.
4. Build a RESTful API using .NET (C#).
5. Implement **JWT-based authentication** for secure access.
6. Deploy the API to **IIS**.
7. Provide a Postman collection demonstrating all endpoints.

Requirements and Expectations

Your implementation should:

- Follow RESTful conventions.
- Implement proper error handling (e.g., 400, 404, 500).

- Include input validation (e.g., required fields).
- Use **Status** as a separate reference table (not inline text or enums).
- Return related data using nested object models (e.g., `Tender` includes `Bids` as an array).
- Use appropriate HTTP status codes and messages.
- Use `async/await` throughout.
- Ensure clear separation of concerns (e.g., services, repositories, controllers).

Authentication

- Implement **JWT-based authentication**.
- Provide endpoints for:
    - **Login** (`POST /api/auth/login`) — returns JWT token
    - **Register** (`POST /api/auth/register`) — creates a new user
- Users should have a **role** (e.g., "Admin", "Vendor").
- Protect endpoints that require authentication.
- Only Admins should be allowed to:
    - Approve or reject bids
    - Create, update, or delete tenders

Entities to Model

You are expected to design and implement the following entities:

- `User` — For authentication (including role).

- `Tender` — Projects open for bidding.
- `Category` — Tender classification.
- `Vendor` — A company that submits bids.
- `Bid` — A vendor’s proposal on a tender.
- `Status` — Reusable status values (used for `Tender` and `Bid`).

You are responsible for determining all table structures and relationships.

API Endpoints

Authentication

- `POST /api/auth/register`
  Register a new user (username, password, role)

- `POST /api/auth/login`
  Authenticate and return a JWT token

Tenders

- `GET /api/tenders`
  Return a list of tenders including:
    - Id, Title, Description, Deadline
    - Category (object: Id + Name)
    - Status (object: Id + Name)
      Bids should not be included.

- `GET /api/tenders/{id}`
  Return tender details including:
    - All fields
    - Category (object)
    - Status (object)
    - `Bids`: list of:
        - Bid Id, Amount, Submission Date
        - Vendor (object: Id + Name)
        - Status (object: Id + Name)

- `POST /api/tenders`
  Create a new tender
  Requires: Title, Description, Deadline, CategoryId, StatusId
  Admin only

- `PUT /api/tenders/{id}`
  Update an existing tender
  Admin only

- `DELETE /api/tenders/{id}`
  Delete a tender
  Admin only

Vendors

- `GET /api/vendors`
  List all vendors
  Optional: include summary of bids

- `GET /api/vendors/{id}`
  Show vendor details including:
    - Vendor info
    - List of bids with tender title and bid status

- `POST /api/vendors`
  Create a new vendor

Bids

- `POST /api/bids`
  Submit a new bid
  Requires: TenderId, VendorId, BidAmount, Comments
  Status should default to "Pending"

- `PUT /api/bids/{id}/status`
  Update the bid status (e.g., Approved, Rejected)
  Requires: StatusId
  Admin only

Lookups

- `GET /api/categories`
  List of tender categories

- `GET /api/statuses`
  List of all status values (e.g., Open, Closed, Pending)

Technology Guidelines

- Use **EF Core** for all create, update, and delete operations.
- Use **Dapper** for all read operations, especially those involving joins or aggregations.
- Use **SQL Server** as the database engine.
- Use **JWT for authentication**, and implement proper authorization logic per role.
- Use `appsettings.json` for configuration, including JWT secret and token expiration settings.

Deployment

- Host the API on **IIS**
- Include:
    - Deployment instructions
    - Authentication configuration notes

Postman Requirements

- Provide a **Postman collection** with:
    - All endpoints
    - Sample requests and responses

    - Environment support (e.g., base URL, token)
- Include a **README** for how to authenticate and use the secured routes

Deliverables

1. Source code (GitHub link or ZIP)
2. Database backup file (.bak format)
3. Postman collection
4. Deployment instructions (IIS)
5. README with setup and usage steps

Evaluation Criteria

- Correct and complete database schema
- Proper separation of EF and Dapper
- Code structure and maintainability
- API security with JWT
- RESTful design and correct use of HTTP verbs
- Handling of authentication and authorization
- Error handling and validation
- API usability via Postman
- Deployment completeness