CAPSTONE PROJECT

FINAL REPORT

DoConnect

A Collaborative Question and Answer Platform

(.NET + ANGULAR)

SUBMITTED BY

RAJULAPATI NAGA VENKATA ADILAKSHMI

WIPRO NGA - .Net Full Stack Angular - FY26 – C2

UNDER THE GUIDANCE OF

JYOTI S PATIL

Table of Contents

DoConnect — Problem Definition & Objectives

1. Brief overview of the problem statement

DoConnect is a collaborative Question & Answer platform designed for learners and technical communities who need a simple yet structured space to exchange knowledge. Users can register, log in, post questions, provide answers, and attach images such as screenshots, diagrams, or code snippets. Unlike larger platforms that may feel overwhelming, DoConnect emphasizes focused moderation through an admin module, enabling review, approval, and management of user contributions. Built with ASP.NET Core Web API, Angular, and SQL Server with EF Core, the system ensures secure access using JWT authentication and role-based control. With a responsive interface, API documentation via Swagger, and validation through unit testing, DoConnect delivers a lightweight, reliable, and scalable solution for effective knowledge sharing.

2. Problem definition

Users require a secure and searchable platform to post technical questions and receive reliable answers. Administrators need robust moderation tools, including content approval workflows and timely notifications for new submissions requiring review. The system must ensure high performance, strong security through JWT-based authentication, and extensibility with optional SignalR-powered real-time notifications. Image support is essential, with files stored securely on the server and referenced in the database for persistence.

Key Challenges Addressed:

- Absence of a trusted, moderated space for domain-specific technical discussions.
- Limited ability to attach and manage images within questions and answers.
- Lack of a structured approval workflow for publishing user-generated content.
- Delayed administrator awareness of new content requiring moderation.

3. Project goals and objectives

The **primary goal** is to deliver a Q&A platform with clear user and admin workflows, image attachments, search functionality, and moderated publishing.

Key Objectives :

1. **Authentication & Authorization** – Implement secure user and admin registration, login, and logout using JWT, with role-based access control to protect sensitive endpoints and views.
2. **Question & Answer Management** – Enable users to create, edit, and delete questions and answers with image attachments, while persisting content in SQL Server via EF Core and exposing CRUD operations through Web API.
3. **Search & Discovery** – Provide a search service for locating questions by title, content, or tags, integrated with a responsive frontend search UI.
4. **Image Handling** – Support secure image uploads and storage, with file paths stored in the database and images served safely through API endpoints or static middleware.
5. **Admin Moderation** – Deliver an admin dashboard for reviewing, approving, or rejecting user submissions, with optional SignalR integration for real-time moderation alerts.
6. **Documentation & Testing** – Provide comprehensive Swagger-based API documentation and ensure reliability through backend unit tests and basic frontend integration tests.

4. Success criteria

DoConnect platform will be measured against the following criteria:

- Users are able to securely register, log in, and create questions and answers, with the ability to attach images.
- Administrators can view, approve, reject, and delete user-generated content, ensuring that only approved content is visible to end users.
- The search functionality consistently returns relevant questions based on titles, content, or tags.
- Image uploads operate reliably, with secure storage and successful retrieval through

the application.

- All APIs are fully documented using Swagger and protected through JWT-based authentication.
- Optional SignalR-based notifications function reliably to alert administrators of new content requiring moderation.

5. Scope, constraints, and assumptions

Scope of DoConnect encompasses the feature set described above, delivered across three development sprints:

- Sprint I: Use case analysis and database design.
- Sprint II: Frontend implementation and API integration.
- Sprint III: Search functionality, real-time notifications, and Swagger-based API documentation.

The scope excludes advanced features such as voting mechanisms, reputation systems, or integration with full-text search engines (e.g., ElasticSearch) during the initial development phase.

Constraints & Assumptions

- SQL Server with Entity Framework Core will be used as the data persistence layer.
- Uploaded files will be stored in the server file system for Sprints I–III, rather than in cloud-based object storage.
- SignalR integration is considered optional and can be disabled in environments where real-time updates are not required.
- CI/CD pipelines and deployment automation are outside the scope of the initial sprints but will be planned for future iterations.

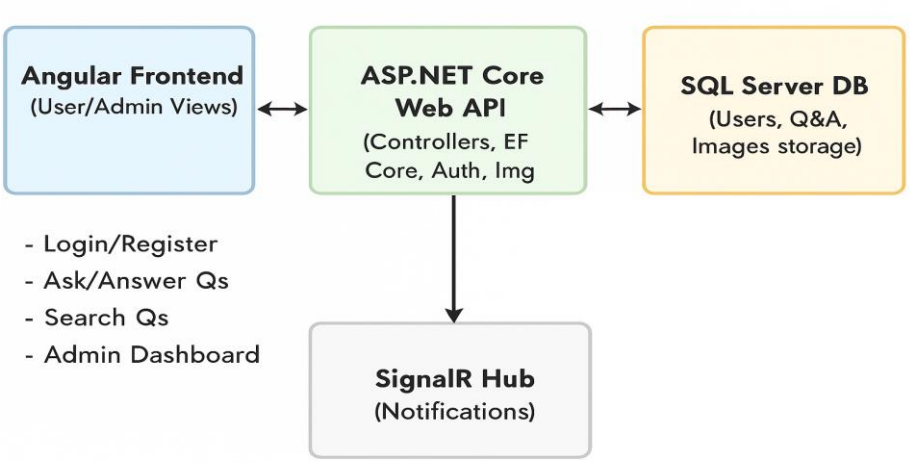
6. Frontend & Backend Architecture

6.1 Overview of chosen technology stack

- **Frontend**: Angular framework (TypeScript), utilizing Angular Router for navigation, HTTP Client for API communication, and optional state management through NgRx/Redux.
- **Backend**: ASP.NET Core MVC with Web API, implementing JWT-based authentication, Entity Framework Core as the ORM, and SignalR for optional real-time notifications.

- **Database**: Microsoft SQL Server with a normalized schema consisting of **Users**, **Questions**, **Answers**, and **Images** tables.
- **Image Handling**: Uploaded images are stored securely in the server file system, with file path references persisted in the database.
- **API Documentation**: Swagger UI is provided for interactive API documentation, testing, and exploration.

6.2 System design diagram (high-level)



7.Component Breakdown & API Design

7.1 Frontendcomponents(Angular)

1. Controllers

- AdminController.cs – Manages administrator-specific operations, including user management and the review or moderation of questions and answers.
- AnswersController.cs – Provides API endpoints for performing CRUD (Create, Read, Update, Delete) operations on answers.
- AuthController.cs – Handles authentication workflows such as user registration, login, password reset, and JWT token issuance.
- QuestionsController.cs – Exposes APIs for creating, editing, deleting, and retrieving questions.
- UsersController.cs – Manages user-related operations, including profile management and user listing.

2. Data

- DoConnectDbContext.cs – Defines the Entity Framework Core DbContext, responsible for managing database access, entity sets (Users, Questions, Answers, Images), and enforcing relationships, constraints, and configurations within the database schema.

3. DTOs (Data Transfer Objects)

Used to transfer data between layers and avoid exposing entity models directly.

- AnswerDto.cs – Represents answer-related data exchanged in API requests and responses.
- ForgotPasswordDto.cs – Encapsulates data required for processing forgot password requests.
- LoginDto.cs – Contains user credentials for login operations.
- QuestionDto.cs – Defines the structure for question-related data in API requests and responses.
- RegisterDto.cs – Handles data submitted during user registration.
- ResetPasswordDto.cs – Represents the data necessary to reset a user's password.

4. Models (Entities)

Represent database tables/entities.

- Answer.cs – Represents the Answers table; stores user-submitted answers and links to both users and questions.
- Image.cs – Represents the Images table; stores metadata and paths for uploaded images, associated with questions or answers.
- Question.cs – Represents the Questions table; stores user-posted questions along with relationships to users, answers, and images.
- User.cs – Represents the Users table; manages application user information, including authentication and associations with questions and answers.

5. Migrations

- Migrations Folder – Contains Entity Framework Core migration files, which define

versioned changes to the database schema. These files track schema evolution over time and allow consistent application of updates to the database structure.

6. Services (Business Logic Layer)

Encapsulating business logic within services ensures a clean separation of concerns between the controllers and core application logic.

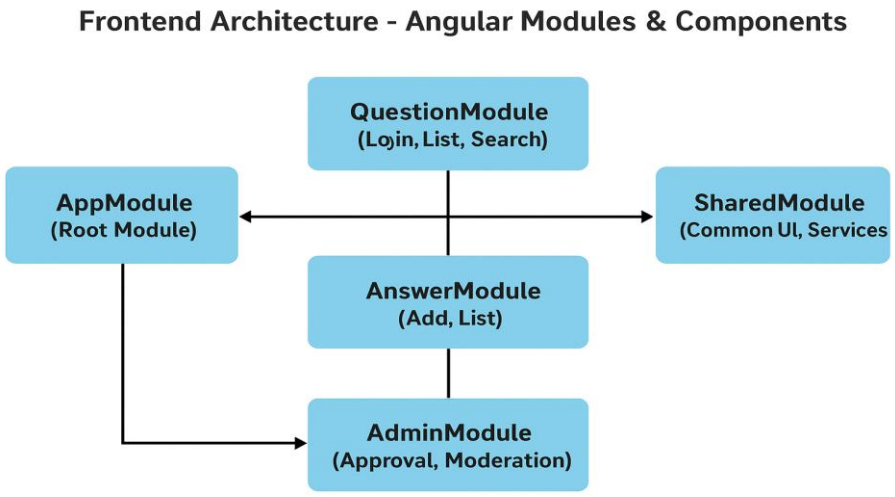
- AnswerService.cs – Implements business logic for creating, retrieving, updating, and deleting answers.
- AuthService.cs – Manages user authentication workflows, including registration, login, and credential validation.
- QuestionService.cs – Handles business rules and operations related to questions.
- TokenService.cs – Responsible for issuing, validating, and managing JWT tokens for secure authentication.

7. Configuration Files

- appsettings.json – Primary configuration file that defines core application settings, including database connection strings, JWT authentication parameters, and other global configurations.
- appsettings.Development.json – Environment-specific configuration file that overrides settings in appsettings.json for the development environment (e.g., local database connections, debug settings).

8. Root Files

- DoConnect.Api.csproj – Project file specifying dependencies, SDK version, and build configuration for the API project.
- DoConnect.Api.http – Contains sample HTTP requests for testing API endpoints directly within supported IDEs (e.g., Visual Studio, VS Code).
- Program.cs – The main application entry point responsible for configuring services, middleware, routing, and application startup logic.
- DoConnect.sln – Solution file that organizes all related projects within the *DoConnect* application for use in Visual Studio or VS Code.



7.2 API Design (ASP.NET Core Web API)

- **Authentication Endpoints:**
  - POST /api/auth/register → Register a new user or admin.
  - POST /api/auth/login → Authenticate a user and return a JWT token.
  - POST /api/auth/forgot-password → Initiate password reset process.
  - POST /api/auth/reset-password → Reset a user's password.

- **Question Endpoints:**
  - POST /api/questions → Create a new question (with optional image).
  - GET /api/questions → Retrieve all questions.
  - GET /api/questions/{id} → Retrieve question details by ID (with answers if available).
  - GET /api/questions/my-questions → Retrieve questions created by the logged-in user.

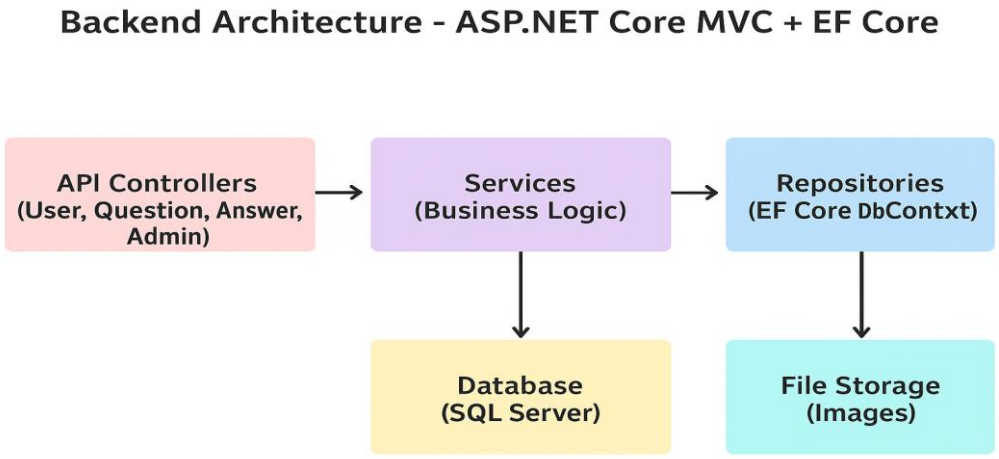
- **Answer Endpoints:**
  - POST /api/answers → Add a new answer to a question.
  - GET /api/answers/my-answers → Retrieve answers submitted by the logged-in user.

- **Admin Endpoints:**
  - PATCH /api/admin/questions/{id}/status → Update the status of a question (approve/reject).

- PATCH /api/admin/answers/{id}/status → Update the status of an answer (approve/reject).
- GET /api/admin/questions → Retrieve all questions (for moderation).
- GET /api/admin/answers → Retrieve all answers (for moderation).
- DELETE /api/admin/questions/{id} → Delete a question.
- DELETE /api/admin/answers/{id} → Delete an answer.
- GET /api/admin/users → Retrieve a list of users.
- PATCH /api/admin/users/{id}/role → Update a user's role.
- DELETE /api/admin/users/{id} → Delete a user.

- **User Endpoints:**
  - GET /api/users/me → Retrieve profile information of the logged-in user.

- **Authentication mechanism:** All secured endpoints require **JWT Bearer tokens**, passed in the request header:
  - Authorization: Bearer <token>



7 Database Design & Storage Optimization

7.2 Entity-Relationship Diagram (ERD)

1. Users → Questions

- **Type**: One-to-Many
- **Description**: A single user can post multiple questions.
- **Foreign Key**: `Questions.UserId` → references `Users.UserId`.

2. **Users → Answers**

- **Type**: One-to-Many
- **Description**: A single user can post multiple answers.
- **Foreign Key**: `Answers.UserId` → references `Users.UserId`.

3. **Questions → Answers**

- **Type**: One-to-Many
- **Description**: A question can have multiple answers.
- **Foreign Key**: `Answers.QuestionId` → references `Questions.QuestionId`.

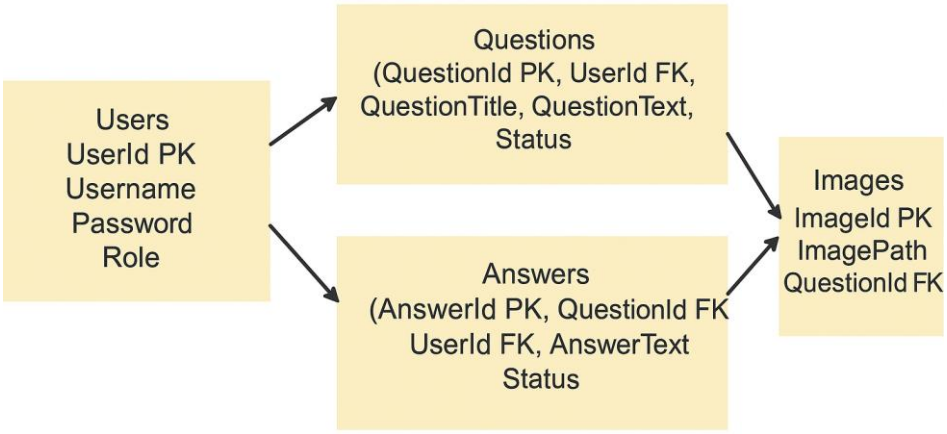
4. **Questions → Images**

- **Type**: One-to-Many (optional)
- **Description**: A question can have multiple images attached.
- **Foreign Key**: `Images.QuestionId` → references `Questions.QuestionId`.

5. **Answers → Images**

- **Type**: One-to-Many (optional)
- **Description**: An answer can have multiple images attached.
- **Foreign Key**: `Images.AnswerId` → references `Answers.AnswerId`.

Entity Relationship Diagram (ERD) - DoConnect Database



7.3 Storage optimization techniques

1. Indexing for Faster Queries
    - Clustered indexes are created on primary keys (`UserId`, `QuestionId`, `AnswerId`, `CommentId`).
    - Non-clustered indexes are applied to frequently searched fields, such as `Email` in **Users** and `CreatedAt` in **Questions**.
    - Covered indexes are added on foreign key columns to optimize JOIN operations.

2. Partitioning Large Tables
    - Large tables such as **Questions** and **Answers** are partitioned by `CreatedAt` date.
    - This improves performance for recent data retrieval by avoiding full table scans.

3. Normalization with Controlled Denormalization
    - The schema is normalized to **Third Normal Form (3NF)** to reduce redundancy.
    - Controlled denormalization is applied for high-frequency queries, such as caching the count of answers instead of recalculating on each request.

4. Query Optimization
    - Parameterized stored procedures are used for heavy queries to improve execution plan reuse.
    - Optimized JOINs replace nested queries where possible to reduce execution time.
    - Pagination (using `OFFSET-FETCH`) is implemented for large datasets to avoid returning excessive records at once.

5. Foreign Keys & Referential Integrity
    - Foreign key constraints ensure consistency between related tables.
    - Cascade delete and update rules are applied to prevent orphaned records

6. Caching Strategies
    - Frequently accessed metadata (e.g., categories, user roles) is cached in memory.
    - This reduces repetitive database hits and improves response times.
7. Connection Pooling
    - Connection pooling in Entity Framework Core is enabled to minimize the overhead of establishing new connections.
8. Use of Transactions
    - Multi-step operations (e.g., adding a question with an initial answer) are executed within transactions.
    - This guarantees atomicity and prevents partial updates.
9. Archiving Old Data
    - Inactive questions and answers older than a defined threshold are archived into separate tables.
    - This keeps active tables smaller, improving query performance.
10. Monitoring & Performance Tuning
    - Tools such as SQL Profiler and execution plans are used to identify and optimize slow queries.
    - Periodic index rebuilding and statistics updates are performed to sustain query speed