**DoConnect (Moderate Q&A Platform)**

*Capstone Project Submission*

Student Name: Pilli Veda Satya Suma

Batch & Instructor: Parth Shukla

# 

# Table of Contents

|  |  |
| --- | --- |
| ****Content**** | ****Page No**** |
| Abstract | 4 |
| Problem Definition and Objectives | 5 |
| Frontend & Backend Architecture | 7 |
| Component Breakdown & API Design | 11 |
| Database Design & Storage Optimization | 14 |
| Detailed Demonstration | 20 |
| Frontend Code | 23 |
| Backend Code | 23 |
| Database Schema | 23 |
| Deployment | 24 |
| Testing Strategy | 25 |
| Environment | 27 |
| Conclusion | 28 |

**List of Figures**

|  |  |
| --- | --- |
| ****Content**** | ****Page No**** |
| System Design Diagram | 10 |
| Entity Relationship Diagram | 15 |
| Process Flow Diagram | 16 |

**List of Tables**

|  |  |
| --- | --- |
| ****Content**** | ****Page No**** |
| Authentication & User APIs | 12 |
| Question APIs | 12 |
| Answer APIs | 12 |
| Admin APIs | 13 |
| Users Table | 17 |
| Questions Table | 17 |
| Answers Table | 18 |
| Images Table | 18 |

## ****Abstract****

DoConnect is a **moderated Question-and-Answer platform** that allows users to **post questions**, **provide answers**, and **attach images** for better clarity. An **admin role** ensures content quality and compliance by reviewing and approving posts. The system includes features such as **secure user authentication**, **role-based access control**, and **optimized database design** for efficient data management. It also uses **testing strategies** to maintain reliability and accuracy. This report presents the **problem statement**, **objectives**, **system design**, **database structure** and **testing method**. Future enhancements may include **real-time notifications**, **AI-driven moderation**, and **mobile application support**.

## Problem Definition and Objectives

* Introduction

Modern Q&A platforms play a critical role in **knowledge sharing and community engagement**. Platforms like **Stack Overflow** and **Quora** have set high standards, but they come with limitations such as **complex moderation mechanisms**, **lack of flexibility for custom workflows**, and **high infrastructure demands** for smaller organizations.  
**DoConnect** addresses these challenges by providing:

* A **moderated discussion environment**.
* **Secure authentication and authorization**.
* **Optimized database operations** for large datasets.
* A **user-friendly, responsive interface** built with Angular.
* An **API-driven architecture** for scalability and maintainability.

### ****Problem Definition****

Existing Q&A systems often lack:

* Strong moderation controls.
* Optimized database performance for large volumes of questions/answers.
* Secure role-based access with modern authentication methods.
* Flexible integration with **modern UI frameworks** and **Scalable APIs**.

**DoConnect** solves these issues by introducing:

* **JWT-based authentication**.
* **Role-based moderation** (Admins approve or reject content).
* **Image attachments for clarity**.
* **Search and pagination for efficiency**.
* **Unit testing with xUnit and Moq** for quality assurance.

## ****Existing System****

The existing widely-used Q&A systems (e.g., **Stack Overflow**, **Quora**) offer large-scale community features but come with **limitations**:

* **High infrastructure and hosting costs** for self-deployment.
* **Lack of fine-grained moderation workflows** for smaller organizations.
* **Complex architectures** that are difficult to maintain for small teams.
* **Performance issues** for niche implementations with limited resources.

## ****Proposed System****

**DoConnect** proposes a **secure, scalable, and cost-effective Q&A platform** that:

* Supports **image uploads** for questions and answers.
* Implements **search functionality with pagination** for performance optimization.
* Incorporates **xUnit and Moq** for unit and integration testing.

## Objectives

* JWT-based auth; role-based moderation.
* Ask/answer with images; search and pagination.
* Normalized schema; indexes; selective includes.
* Angular UI with routing/guards/interceptors.
* Unit/Integration tests for critical paths.

# Frontend & Backend Architecture

### ****1. Frontend Layer – Angular Single Page Application (SPA)****

The **UI layer** is built using **Angular**, which provides a responsive and dynamic interface for users.

**Key Features:**

* **Single Page Application (SPA):**
  + Only loads the HTML, CSS, and JavaScript once and dynamically updates the content using API calls.
* **Component-based Architecture:**
  + Each feature (Login, Dashboard, Questions, Answers) is implemented as an Angular component.
* **Routing and Navigation:**
  + Angular Router is used to handle client-side navigation without full page reload.
* **Authentication Handling:**
  + **JWT tokens** are stored in **local storage** or **session storage**.
  + Route Guards prevent unauthorized access to protected routes like /dashboard or /admin.
* **HTTP Interceptors:**
  + Automatically attach the **Authorization: Bearer** header to outgoing requests.
* **Styling:**
  + Uses **Bootstrap**, **Angular Material**, and **custom CSS** for responsive design.

**Frontend Responsibilities:**

* Register/Login users via API.
* Display all approved questions and answers.
* Provide a rich-text editor for posting questions and answers.
* Allow image uploads for better content clarity.
* Implement **search with pagination** for scalability.
* Notify users of status changes (Pending → Approved).

### ****2. Backend Layer – ASP.NET Core Web API****

The **backend** is developed using **ASP.NET Core 9.0** and follows a **Clean Architecture** pattern with **Controllers → Services → Repositories**.

**Core Responsibilities:**

* Expose RESTful API endpoints for all core functionalities:
  + **Authentication & Authorization** (Login, Register, Role Assignment).
  + **CRUD operations** for Questions, Answers, and Images.
  + **Admin actions** (Approve/Reject/Delete).
* **Business Logic Implementation:**
  + Enforce validation rules and role-based access.
  + Ensure a question must be approved before accepting answers.
* **Static File Management:**
  + Handles **file upload and storage** for question/answer images under:
  + wwwroot/uploads
  + Configures **StaticFiles middleware** for serving uploaded images.
* **Security Layer:**
  + Implements **JWT authentication** with role-based policies.
  + Hashes passwords using **ASP.NET Core Identity hashing algorithm**.
* **CORS Configuration:**
  + Allows requests from Angular’s development URL (http://localhost:4200) and production domain.
* **Logging & Exception Handling:**
  + Uses **Serilog** or **built-in logging providers**.
* **Performance Optimization:**
  + Implements **server-side pagination** and **filtering**.
  + Uses **LINQ queries with EF Core Includes** for efficient data retrieval.

### ****3. Database Layer – SQL Server with EF Core ORM****

The database is implemented in **Microsoft SQL Server** with **Entity Framework Core** as the ORM for handling data persistence.

**Database Features:**

* **Normalized Schema:**
  + Entities: Users, Questions, Answers, Images.
* **Relationships:**
  + One-to-many between Users → Questions.
  + One-to-many between Questions → Answers.
  + One-to-many between Questions → Images and Answers → Images.
* **Constraints:**
  + Foreign keys for relational integrity.
  + Check constraints for image association (either QuestionId or AnswerId must be non-null).
* **Indexes:**
  + On frequently searched columns like Question Title, CreatedAt for performance.
* **Pagination & Filtering:**
  + Implemented at the query level using EF Core Skip() and Take().

### ****4. Static File Handling****

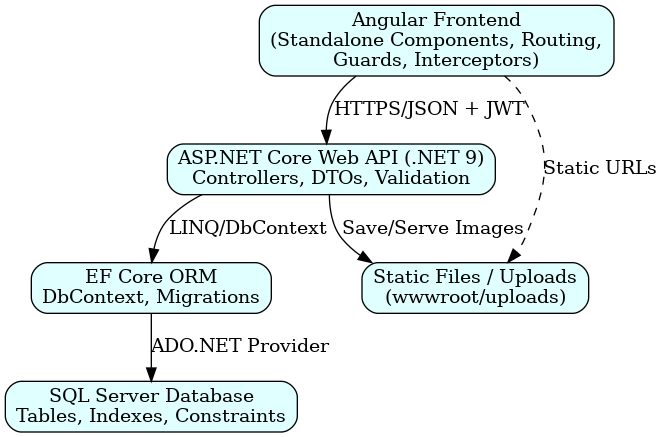
* Images uploaded by users for Questions/Answers are stored in **wwwroot/uploads**.
* The backend serves them via **ASP.NET Core Static Files Middleware**.
* File names are stored in the database for reference.

### ****5. Communication & Security****

* **API Communication:**
  + Frontend communicates with backend using **HTTPS** for all API calls.
* **Authentication:**
  + JWT token is issued upon login and validated on every request.
* **Authorization:**
  + Admin endpoints require role Admin.
  + Regular users can only modify their own questions/answers.
* **Data Validation:**
  + All inputs are validated at the backend using **Data Annotations** and Fluent Validation.

### ****6.**** ****Technology Stack Summary****

* **Frontend:** Angular, Bootstrap, HTML5, CSS3.
* **Backend:** ASP.NET Core Web API, C#.
* **Database:** Microsoft SQL Server.
* **ORM:** Entity Framework Core.
* **Authentication:** JWT tokens.
* **Testing:** xUnit and Moq for unit/integration tests.



*System Design Diagram*

# Component Breakdown & API Design

## 1. Frontend Components

* **Core Components:**
* **Login Component:** Handles user authentication and token storage.
* **Register Component:** Allows new users to sign up.
* **Dashboard Component:** Displays all approved questions and their answers.
* **Question Component:** Enables posting of new questions.
* **Answer Component:** Allows users to add answers to questions.
* **Admin Panel:** For approving/rejecting questions and answers.
* **Services:**
* **Auth Service:** Handles JWT token storage and API calls for login/register.
* **Question Service:** Manages CRUD operations for questions.
* **Answer Service:** Manages CRUD operations for answers.
* **Admin Service:** Provides moderation-related APIs.
* **Routing & Guards:**
* **Route Guards:** Prevent unauthorized users from accessing protected pages.
* **Interceptors:** Add **Authorization: Bearer token** header to all API calls.

### ****2. Backend Components (ASP.NET Core)****

* **Controllers:**
* **AuthController:** Handles authentication (Login/Register).
* **UserController:** Manages user-related actions.
* **QuestionController:** Handles questions (create, list, search).
* **AnswerController:** Handles answers.
* **AdminController:** Provides moderation actions.
* **Services Layer:**
* Contains business logic for authentication, content approval, and image management.
* **Repository Layer:**
* Handles database interactions using **Entity Framework Core**.

### ****3. API Design****

The **API endpoints** follow RESTful conventions with secure role-based access.

#### ****Authentication & User APIs****

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Description** | **Auth Required** |
| POST | /api/auth/login | User login & JWT generation | No |
| POST | /api/auth/register | Register new user | No |

#### ****Question APIs****

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Description** | **Auth Required** |
| GET | /api/questions | Get all approved questions | Yes |
| GET | /api/questions/{id} | Get question details | Yes |
| POST | /api/questions | Post a new question (Pending) | Yes |

#### ****Answer APIs****

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Description** | **Auth Required** |
| POST | /api/answers | Add answer to question | Yes |

#### ****Admin APIs****

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Description** | **Role** |
| PUT | /api/admin/questions/{id}/approve | Approve a question | Admin |
| PUT | /api/admin/questions/{id}/reject | Reject a question | Admin |
| PUT | /api/admin/answers/{id}/approve | Approve an answer | Admin |

### ****API Security & Standards****

* **Authentication:** JWT-based authentication.
* **Authorization:** Role-based (Admin/User).
* **Validation:** Data annotations and FluentValidation.
* **Response Format:** JSON with proper HTTP status codes.

# Database Design & Storage Optimization

## ****Database Design & Storage Optimization****

### ****Database Schema Overview****

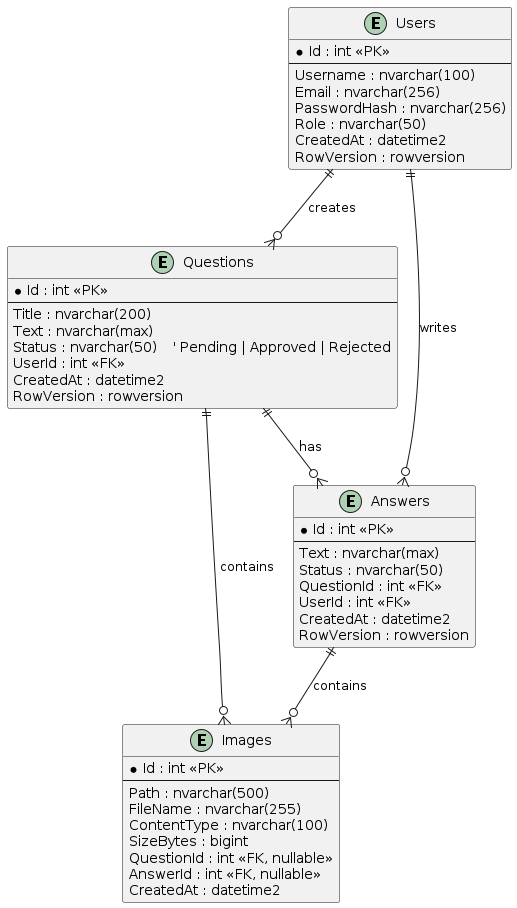
The DoConnect system uses **Microsoft SQL Server** as its primary relational database, managed through **Entity Framework Core (EF Core)**. The schema is **normalized to Third Normal Form (3NF)** to reduce redundancy and ensure data consistency. The main entities are:

* **Users:** Stores user credentials, roles, and metadata.
* **Questions:** Stores user-submitted questions with title, description, and status.
* **Answers:** Stores answers associated with questions.
* **Images:** Stores file references for questions or answers.

### ****Entity-Relationship (ER) Model****

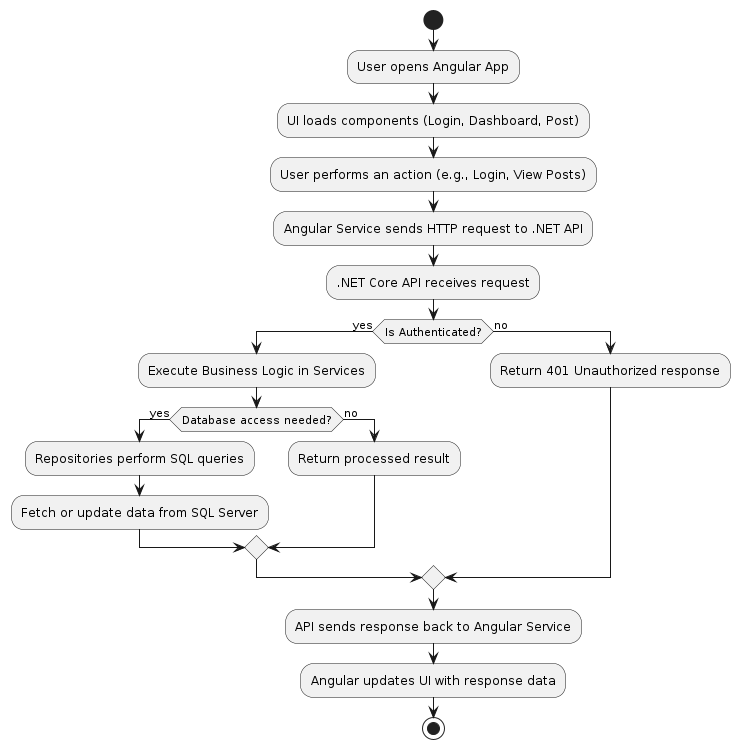
The relationships are as follows:

* **Users → Questions:** One-to-many (a user can post multiple questions).
* **Users → Answers:** One-to-many (a user can post multiple answers).
* **Questions → Answers:** One-to-many (a question can have multiple answers).
* **Questions → Images:** One-to-many (a question can have multiple images).
* **Answers → Images:** One-to-many (an answer can have multiple images).



*Entity RelationshipDiagram*

* **Process Flow Diagram**

****

***Program Flow Diagram***

### ****3. Physical Schema****

### ****Users Table****

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| Id | INT IDENTITY(1,1) | PRIMARY KEY, NOT NULL | Unique identifier for each user. |
| Username | NVARCHAR(100) | NOT NULL | Display name of the user. |
| Email | NVARCHAR(256) | UNIQUE, NOT NULL | User’s email for login and communication. |
| PasswordHash | NVARCHAR(256) | NOT NULL | Hashed password for security. |
| Role | NVARCHAR(50) | DEFAULT 'User', NOT NULL | User role: Admin or User. |
| CreatedAt | DATETIME2 | DEFAULT SYSUTCDATETIME(), NOT NULL | Timestamp of account creation. |
| RowVersion | ROWVERSION | NOT NULL | For optimistic concurrency control. |

### ****Questions Table****

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| Id | INT IDENTITY(1,1) | PRIMARY KEY, NOT NULL | Unique identifier for each question. |
| Title | NVARCHAR(200) | NOT NULL | Title of the question. |
| Text | NVARCHAR(MAX) | NOT NULL | Detailed content of the question. |
| Status | NVARCHAR(50) | DEFAULT 'Pending', NOT NULL | Status: Pending, Approved, or Rejected. |
| UserId | INT | FOREIGN KEY REFERENCES Users(Id) | The user who created the question. |
| CreatedAt | DATETIME2 | DEFAULT SYSUTCDATETIME(), NOT NULL | Timestamp of question creation. |
| RowVersion | ROWVERSION | NOT NULL | For optimistic concurrency control. |

### ****Answers Table****

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| Id | INT IDENTITY(1,1) | PRIMARY KEY, NOT NULL | Unique identifier for each answer. |
| Text | NVARCHAR(MAX) | NOT NULL | Content of the answer. |
| Status | NVARCHAR(50) | DEFAULT 'Pending', NOT NULL | Status: Pending, Approved, or Rejected. |
| QuestionId | INT | FOREIGN KEY REFERENCES Questions(Id) | The question to which this answer belongs. |
| UserId | INT | FOREIGN KEY REFERENCES Users(Id) | The user who posted the answer. |
| CreatedAt | DATETIME2 | DEFAULT SYSUTCDATETIME(), NOT NULL | Timestamp of answer creation. |
| RowVersion | ROWVERSION | NOT NULL | For optimistic concurrency control. |

### ****Images Table****

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| Id | INT IDENTITY(1,1) | PRIMARY KEY, NOT NULL | Unique identifier for each image. |
| Path | NVARCHAR(500) | NOT NULL | File path of the stored image. |
| FileName | NVARCHAR(255) | NULL | Original image file name. |
| ContentType | NVARCHAR(100) | NULL | MIME type of the image (e.g., image/png). |
| SizeBytes | BIGINT | NULL | Size of the image in bytes. |
| QuestionId | INT | FOREIGN KEY REFERENCES Questions(Id), NULL | The question associated with the image. |
| AnswerId | INT | FOREIGN KEY REFERENCES Answers(Id), NULL | The answer associated with the image. |
| CreatedAt | DATETIME2 | DEFAULT SYSUTCDATETIME(), NOT NULL | Timestamp of image upload. |
| **CHECK** | -- | (QuestionId IS NOT NULL OR AnswerId IS NOT NULL) | Ensures the image belongs to either question or answer. |

### Optimization techniques for efficient queries

### ****Indexing Strategies****

* Create **clustered indexes** on primary keys for all tables.
* Add **non-clustered indexes** on frequently queried columns:
  + Users(Email) for login lookups.
  + Questions(UserId, CreatedAt) for fetching user-specific questions.
  + Answers(QuestionId, CreatedAt) for retrieving answers for a question.
* Use **full-text indexing** for keyword searches on Title and Text columns in the Questions table.

### ****Pagination for Large Datasets****

* Use **keyset pagination** instead of OFFSET-FETCH for large datasets:
* SELECT TOP 10 \*
* FROM Questions
* WHERE CreatedAt < @LastSeenDate AND Status = 'Approved'
* ORDER BY CreatedAt DESC;

### ****Prevent N+1 Query Problems****

* In **Entity Framework Core**, use:
  + **Eager loading** (Include) for related entities only when needed.
  + **Explicit loading** for large relationships.
  + **Projections** to select only required fields.

### ****Caching Frequently Accessed Data****

* Cache popular and recent questions or user profiles in:
  + **In-memory cache** for single-server deployments.
  + **Distributed cache (Redis)** for multi-node systems.
* Set **short TTL for dynamic data** and **long TTL for static metadata**.

**Detailed Demonstration**

### ****Environment Setup****

Before running the system, ensure the environment is configured properly.

**Backend Setup:**

* Open the DoConnect backend project in Visual Studio.
* Restore NuGet packages using:
  + dotnet restore
* Applying migrations to create the database:
  + dotnet ef database update
* Start the API server:
  + dotnet run

The default URL is https://localhost:5108.

**Frontend Setup:**

* Navigate to the Angular frontend folder.
* Install dependencies using:
  + npm install
* Start the Angular development server:
  + ng serve --open

The application opens at http://localhost:4200.

### ****User Registration and Login****

**Process:**

* A new user registers by providing a username, email, and password.
* Registration API (/api/auth/register) validates input and stores user details in the database.
* The user then logs in using email and password via API (/api/auth/login), which returns a JWT token.
* The token is stored in the browser’s local storage for authentication in future requests.

### ****Posting a Question****

**Steps:**

* The logged-in user navigates to the **Ask Question** page.
* Enters the question title and description and optionally uploads an image.
* On submission:
  + The API (/api/questions) saves the question with a status of **Pending**.
  + The image file is stored in wwwroot/uploads and referenced in the Images table.
* The user receives a confirmation message stating that the question is under review.

### ****Admin Moderation of Questions****

**Process:**

* The admin logs in and accesses the **Admin Dashboard**.
* Admin sees all questions with the **Pending** status.
* Admin can approve or reject questions using:
  + /api/admin/questions/{id}/approve
  + /api/admin/questions/{id}/reject
* Once approved, the question becomes visible to all users.

### ****Posting an Answer****

**Steps:**

* A user selects an **Approved Question** and writes an answer.
* The answer is submitted via /api/answers and stored with status **Pending**.
* Admin reviews and approves or rejects answers using:
  + /api/admin/answers/{id}/approve
  + /api/admin/answers/{id}/reject
* Approved answers become visible to all users under the respective question.

### ****Image Handling****

* When uploading an image:
  + The backend saves the file in wwwroot/uploads.
  + Metadata (file path, name, content type) is stored in the Images table.
* Angular loads the image through the API using the stored path.

**Frontend Code**



**Backend Code**

****

**Database Schema**

****

**Deployment**

* Push entire project to GitHub
* Keep repository private
* Include backend & frontend files

Github link:

<https://github.com/Pilliveda/PilliSuma_DOCONNECT_CapstoneProject>

# Testing Strategy

### ****Objectives of Testing****

* Verify the correctness of all features.
* Ensure data integrity and proper interaction between components.
* Detect and fix defects early in the development cycle.
* Validate security measures such as authentication and authorization.

### ****Unit Testing****

**Tools and Frameworks:**

* Backend:
  + xUnit for .NET Core unit tests.
  + Moq for mocking dependencies in service and repository layers.
* Frontend:
  + Jasmine & Karma for Angular component and service testing.

**Scope of Unit Tests:**

* **Backend:**
  + Test service layer methods for business logic.
  + Validate input data and authentication processes.
  + Verify JWT token generation and validation.
* **Frontend:**
  + Test Angular components for proper rendering.
  + Validate form submissions and input handling.
  + Test HTTP services using mock API responses.

### ****Integration Testing****

**Tools:**

* **Postman** for API testing.
* **Swagger UI** for interactive API checks.
* **EF Core InMemory Database** for backend integration tests without impacting the real database.

**Scope of Integration Tests:**

* **Backend:**
  + Test API endpoints for expected responses and workflows such as:
    - User registration and login.
    - Question submission and admin approval.
  + Validate data persistence and relationships.
* **Frontend:**
  + Test Angular services that call backend APIs.
  + Check that UI updates correctly after API responses.

### ****Continuous Testing****

* Automated tests are run through a CI/CD pipeline.
* Builds fail if any critical test case does not pass.

# Environment

### ****Environments****

* **Development:**  
  This environment is used by developers for coding and testing the application. It runs on local machines with tools like **Visual Studio**, **Angular CLI**, and **SQL Server Express**.
* **Staging:**  
  This environment is used for **user acceptance testing (UAT)**. It mimics the production environment to test the system before going live.
* **Production:**  
  This is the **live environment** where the application runs for end-users. It is optimized for **speed**, **security**, and **stability**. Sensitive data like database credentials and API keys are stored in **environment variables** for safety.

# Conclusion

The DoConnect project is a simple and useful platform for asking and answering questions in a safe and organized way. It allows users to post questions, give answers, and upload images. Admins can check and approve content to maintain quality.

The system uses:

* **Angular** for the frontend (user interface),
* **ASP.NET Core API** for the backend (logic),
* **SQL Server** for the database.

It includes important features like:

* **Secure login with JWT**,
* **Search and pagination**,
* **Role-based access**,
* **Testing for reliability**.

The application can run on both **local servers** and it supports **scalability, security, and easy maintenance**.

In short, DoConnect is a reliable and flexible solution for knowledge sharing, and it can be improved in the future by adding features like tags, notifications, and smart recommendations.