Project Name: DoConnect (Frontend in React, Backend in ASP.NET Core MVC)

Updated Problem Statement:

DoConnect is a Q&A platform where users can ask and answer questions related to various technical topics. The application has two types of users:

- **User**
- **Admin**

Tech Stack:

- **Frontend:** React
- **Backend:** ASP.NET Core MVC with Web API
- **Database:** SQL Server (using Entity Framework Core as ORM)
- **File Storage:** File upload functionality on the server
- **API Documentation:** Swagger for Web API
- **Notifications:** SignalR (optional for real-time notifications)

User Stories:

User Stories (User)

1. **User Authentication:**
   As a user, I should be able to **log in**, **log out**, and **register** into the application.
2. **Ask Questions:**
   As a user, I should be able to ask a question under any topic.
3. **Search Questions:**
   As a user, I should be able to search for questions based on a search query.
4. **Answer Questions:**
   As a user, I should be able to answer any question posted.
5. **Multiple Answers:**
   As a user, I should be able to provide multiple answers to the same question.
6. **Image Upload:**
   As a user, I should be able to upload images along with my question or answer.

Admin Stories

1. **Admin Authentication:**
   As an admin, I should be able to **log in**, **log out**, and **register** into the application.
2. **Receive Notifications:**
   As an admin, I should receive notifications when a new question is posted or an answer is given.

3. **Approve Questions & Answers:**
   As an admin, I should be able to approve or reject questions and answers, making them visible on the platform only after approval.
4. **Moderate Content:**
   As an admin, I should be able to delete inappropriate questions or answers.

System Requirements:

- **Frontend:** React-based UI for user interaction.
- **Backend:** ASP.NET Core MVC with Web API for business logic and data access.
- **ORM:** Entity Framework Core (EF Core) for database interactions.
- **Database:** SQL Server for storing user, question, and answer data.
- **Image Uploads:** Store uploaded images in a server-side folder.

Architecture Overview:

1. **Frontend (React):**
   - User Interface for Users and Admin
   - Consumes Web API for login, registration, questions, answers, and approvals
   - Routing for User and Admin pages (React Router)
   - Manage state using Context API or Redux
2. **Backend (ASP.NET Core MVC):**
   - ASP.NET Core MVC with Web API endpoints for data exchange
   - Entity Framework Core to handle database CRUD operations
   - JWT Token Authentication for secure login/logout
   - Web API endpoints to support user authentication, question, and answer management
   - SignalR or Notification System for admin notifications
3. **Database (SQL Server):**
   - Tables for `Users`, `Questions`, `Answers`, `Images`
   - Relationships: One-to-Many between Users and Questions, Questions and Answers
4. **Image Upload:**
   - Users can upload images, and these images are stored in a server directory
   - File path references are stored in the database
5. **API Documentation:**
   - Swagger UI for testing and documenting the Web API

Sprint Plan:

Sprint I:

Objectives:

1. **Use Case Document:**
   Prepare use case documentation for all functionalities (login, question management, etc.).
2. **Database Schema Design:**
   Define tables and relationships:
   - Users (UserId, Username, Password, Role)
   - Questions (QuestionId, UserId, QuestionTitle, QuestionText, Status)
   - Answers (AnswerId, QuestionId, UserId, AnswerText, Status)
   - Images (ImageId, ImagePath, QuestionId/AnswerId)
3. **Select ORM Tool:**
   Use **Entity Framework Core (EF Core)** for database interactions.
4. **Backend Setup:**
   - Identify necessary controllers (UserController, QuestionController, AnswerController).
   - Create **ASP.NET Core MVC Views** (HTML templates) as placeholders for the frontend.

Deliverables:

- Use case document
- Database schema (ERD and .sql script)
- Static views in MVC for Questions, Answers, and Authentication

---

Sprint II:

Objectives:

1. **Frontend Setup (React):**
   - Initialize a React app for the user and admin interfaces.
   - Implement routing for login, register, ask question, answer question, and admin approval pages.
2. **Backend Implementation (ASP.NET Core MVC):**
   - Implement **CRUD operations** for users (register, login, logout).
   - Implement user authentication with **JWT Tokens**.
   - Create the **DBContext** object using EF Core.
3. **API for Questions & Answers:**
   - Create Web API for CRUD operations on `Questions` and `Answers`.
   - Implement image upload functionality in the backend.
4. **Admin and User Pages:**
   - Design and implement the layout for user pages (React) and admin pages.
   - Connect the React frontend to the Web API (Axios or Fetch).

Deliverables:

- React components for user and admin UI
- Web API endpoints for `Questions`, `Answers`, and `Images`
- Working authentication for users and admin

Sprint III:

Objectives:

1. **Search Functionality (API + Frontend):**
    - Develop the **search API** for questions based on query strings.
    - Integrate the search functionality into the React frontend.
2. **Admin Notifications:**
    - Implement notification system (optional: **SignalR**) for admins when a question or answer is added.
    - Display notifications on the admin dashboard.
3. **Admin Approval Workflow:**
    - Implement the approval workflow for questions and answers.
    - Only approved content should be visible to users.
4. **Swagger API Testing:**
    - Add **Swagger UI** for API documentation and testing.
5. **Final Integration:**
    - Ensure that all frontend components are correctly integrated with the backend API.
    - Test the entire system for any bugs or performance issues.

Deliverables:

- Search functionality in the API and frontend
- Admin approval module with notifications
- Swagger UI documentation for all Web API endpoints
- Fully functional React frontend consuming ASP.NET Core MVC backend API

Key Points for Implementation:

1. **Frontend:**
    - Use **React Router** for navigation between different views (login, register, ask question, etc.).
    - Manage API requests and responses using **Axios**.
    - State management using **Context API** or **Redux**.

2. **Backend:**
    - Implement JWT-based authentication for secure user sessions.
    - Use **Entity Framework Core** for database operations (CRUD).
    - Create Web API endpoints for users, questions, and answers, and consume them in React.
3. **Image Uploads:**
    - Handle image uploads in the backend using **IFormFile** in ASP.NET Core.
    - Store file references in the database and serve them through the API.
4. **Admin Notifications (optional):**
    - Implement **SignalR** for real-time notifications to admins.