ProfileBook (Frontend: React, Backend: ASP.NET Core MVC)

ProfileBook is an online social media platform that connects people virtually. Users can post content, message other users, and report users. Admins manage users, approve posts, and handle reported users.

Tech Stack:

- **Frontend:** React
- **Backend:** ASP.NET Core MVC with Web API
- **Database:** SQL Server (using Entity Framework Core as ORM)
- **File Storage:** Upload profile pictures and posts to a server folder.
- **API Documentation:** Swagger for Web API
- **Real-time Communication:** Use SignalR for messaging between users (optional).

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. **View Posts:**
   As a user, I should be able to see posts that are shared by other users.
3. **Create Post:**
   As a user, I should be able to submit a post for approval from the admin.
4. **Receive Post Approval Notification:**
   As a user, I should get a confirmation once my post is approved by the admin.
5. **Like/Comment Posts:**
   As a user, I should be able to like or comment on posts shared by other users.
6. **Message Other Users:**
   As a user, I should be able to send messages to other users.
7. **Report Users:**
   As a user, I should be able to report inappropriate behavior of other users.
8. **Search for Users:**
   As a user, I should be able to search for other users based on their profile.

Admin Stories

1. **Admin Authentication:**
   As an admin, I should be able to log in, log out, and register into the application.
2. **CRUD on Users:**
   As an admin, I should be able to create, read, update, and delete users.

3. **Approve Posts:**
   As an admin, I should be able to receive notifications for post approval requests and approve posts.
4. **Create User Groups:**
   As an admin, I should be able to create groups for users.
5. **View Reported Users:**
   As an admin, I should be able to view users reported by others.

System Requirements:

- **Frontend:** React-based UI for user interaction with Web API consumption.
- **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 profiles, posts, messages, and reports.
- **Image Uploads:** Profile pictures and post images stored in server directories.
- **Notifications:** Real-time notifications using SignalR for messaging (optional).

Architecture Overview:

1. **Frontend (React):**
   - User interface for user actions (posting, messaging, liking, reporting).
   - Admin interface for managing posts, users, and reports.
   - Consumes Web API for login, registration, posting, messaging, and user management.
   - Routing for User and Admin pages using React Router.
   - State management using Context API or Redux.
2. **Backend (ASP.NET Core MVC):**
   - ASP.NET Core MVC with Web API endpoints for data exchange.
   - JWT Token Authentication for secure login/logout.
   - Entity Framework Core (EF Core) to handle CRUD operations on database entities.
   - Web API endpoints to support user authentication, post approval, reporting, and messaging.
3. **Database (SQL Server):**
   - Tables for `Users`, `Posts`, `Messages`, `Reports`, and `Groups`.
   - Relationships: One-to-Many between Users and Posts, Users and Messages.
4. **Image Upload:**
   - Users can upload profile images and post-related images.
   - Images are stored on the server, and file 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 detailed use case documentation for all functionalities (login, registration, posting, messaging, etc.).
2. **Database Schema Design:**
   Define tables and relationships:
   - `Users` (UserId, Username, Password, Role, ProfileImage)
   - `Posts` (PostId, UserId, Content, PostImage, Status)
   - `Messages` (MessageId, SenderId, ReceiverId, MessageContent, TimeStamp)
   - `Reports` (ReportId, ReportedUserId, ReportingUserId, Reason, TimeStamp)
   - `Groups` (GroupId, GroupName, GroupMembers)
3. **Select ORM Tool:**
   Use **Entity Framework Core (EF Core)** to manage database operations.
4. **Backend Setup:**
   - Identify necessary controllers (UserController, PostController, MessageController, ReportController).
   - Create **ASP.NET Core MVC Views** as placeholders for the frontend.
5. **Static Frontend in React:**
   - Build static page templates for login, register, post viewing, and messaging in React.

Deliverables:

- Use case document
- Database schema (ERD and .sql script)
- Static views in MVC for post creation, messaging, and user authentication

---

Sprint II:

Objectives:

1. **Frontend Setup (React):**
   - Initialize a React app with routing for login, register, viewing posts, and messaging.
   - Build React components for user and admin interfaces (Profile, Posts, Messaging).
2. **Backend Implementation (ASP.NET Core MVC):**
   - Implement **CRUD operations** for users (register, login, logout).
   - Implement user authentication with **JWT Tokens**.

    - Create the **DbContext** using Entity Framework Core (EF Core).
    - Develop the **Web API** for user registration, login, posting, and messaging.
3. **Admin and User Page Layouts:**
    - Design and implement layouts for admin and user pages (React).
    - Connect React frontend to Web API using **Axios** or **Fetch**.
4. **Messaging API:**
    - Create a **separate Web API** for messaging between users.
5. **Post Management:**
    - Implement the user post approval functionality, where posts are submitted for admin review.

Deliverables:

- React components for user and admin UI
- Web API endpoints for user management, posting, and messaging
- JWT-based authentication
- Separate Web API for messaging functionality

Sprint III:

Objectives:

1. **Search and Filter Functionality (API + Frontend):**
    - Implement search and filter functionality for users and posts.
    - Integrate this feature into the React frontend.
2. **Post Approval (Admin-side):**
    - Allow admins to approve posts submitted by users via the Web API.
    - Notify users when their posts are approved or rejected.
3. **Reported Users (Admin-side):**
    - Implement functionality for admins to view and take action on reported users.
4. **Groups Management (Admin-side):**
    - Allow admins to create and manage user groups.
5. **Swagger API Testing:**
    - Add **Swagger UI** for documenting and testing the Web API.
6. **Final Integration:**
    - Ensure that all frontend components are correctly integrated with the backend API.
    - Test the system for bugs, scalability, and performance issues.

Deliverables:

- Search and filter functionality for users and posts
- Post approval and rejection system for admin

- Reported users management for admin
- Swagger documentation for Web API
- Fully functional React frontend with backend API integration

---

Key Points for Implementation:

1. **Frontend:**
    - Use **React Router** for navigation between user and admin pages.
    - State management via **Context API** or **Redux** for handling login sessions and post approvals.
    - Handle API requests and responses using **Axios** or **Fetch**.
2. **Backend:**
    - Implement JWT-based authentication to secure login/logout functionality.
    - Use **Entity Framework Core (EF Core)** for database operations.
    - Create a separate Web API for messaging and integrate it with the main backend system.
3. **Image Uploads:**
    - Manage image uploads for posts and profile pictures in the backend using **IFormFile** in ASP.NET Core.
4. **Notifications (Optional):**
    - Implement **SignalR** for real-time messaging between users.