Blood Link

1.Project Planning & Management

Project Proposal – Overview, Objectives, and Scope

Overview:

This project aims to develop a centralized digital platform that connects all blood banks and hospitals across Egypt. The system enables citizens to search for the nearest blood bank or hospital that has their required blood type available, while hospitals can officially request blood units through verified channels. The platform will be managed by government administrators, ensuring the registration and verification of all hospitals and blood banks to maintain credibility, transparency, and accuracy.

Objectives:

- Facilitate quick access to national blood availability data.
- Improve communication and coordination between hospitals and blood banks.
- Allow hospitals to submit, track, and manage blood requests efficiently.
- Provide the government with real-time visibility and control over national blood resources.
- Reduce emergency response time by connecting donors and hospitals to nearby sources.

Scope:

1. Admin Panel – Manage hospitals, blood banks, and verify data authenticity.
2. Hospital Module – Submit blood requests, update stock, and track request status.
3. User Search Module – Enable the public to search for available blood types by location and availability.

Resources Required:

Development Tools: Visual Studio Code, SQL Server, Figma (UI design)
Technologies: HTML, CSS, JavaScript, C# (.NET), SQL

Project Plan – Timeline, Milestones, Deliverables, and Resources

Date	Milestone	Deliverable
(15.9.2025 – 22.9.2025)	Requirements Gathering & ERD Design	Finalized requirements and ERD diagram
(23.9.2025 – 30.9.2025)	Database Design & Setup	Normalized database and schema
(5.10.2025 – 10.10.2025)	Front-End Design	UI for search, hospital, and admin modules
(11.10.2025 – 5.11.2025)	Back-End Development	APIs for search, requests, and admin actions
(6.11.2025 – 8.11.2025)	Integration & Testing	Fully connected and tested system
(9.11.2025 – 15.11.2025)	Bug Fixing & Optimization	Stable beta version
(16.11.2025 –20.11.2025)	Documentation	Technical report and user guide
(25.11.2025 – 5.12.2025)	Final Review & Deployment	System ready for public launch

Task Assignment & Roles

Role	Responsibilities	Name
Project Leader	Oversees the project timeline, manages risks, and ensures all deliverables are met	Marwa Gameil
Backend Developer	Designs APIs, database logic, and ensures data security and scalability	Marwa Gameil Hadeer Sabry
Frontend Developer	Builds UI components, search pages, and hospital forms	Walid Abdelkader
Database Designer	Designs ERD, creates schema, and maintains relational integrity	All team
Tester / QA Engineer	Conducts functionality, usability, and performance testing	All team
Documentation Lead	Prepares reports, user manuals, and technical documentation	Hadeer Sabry

Risk Assessment & Mitigation Plan

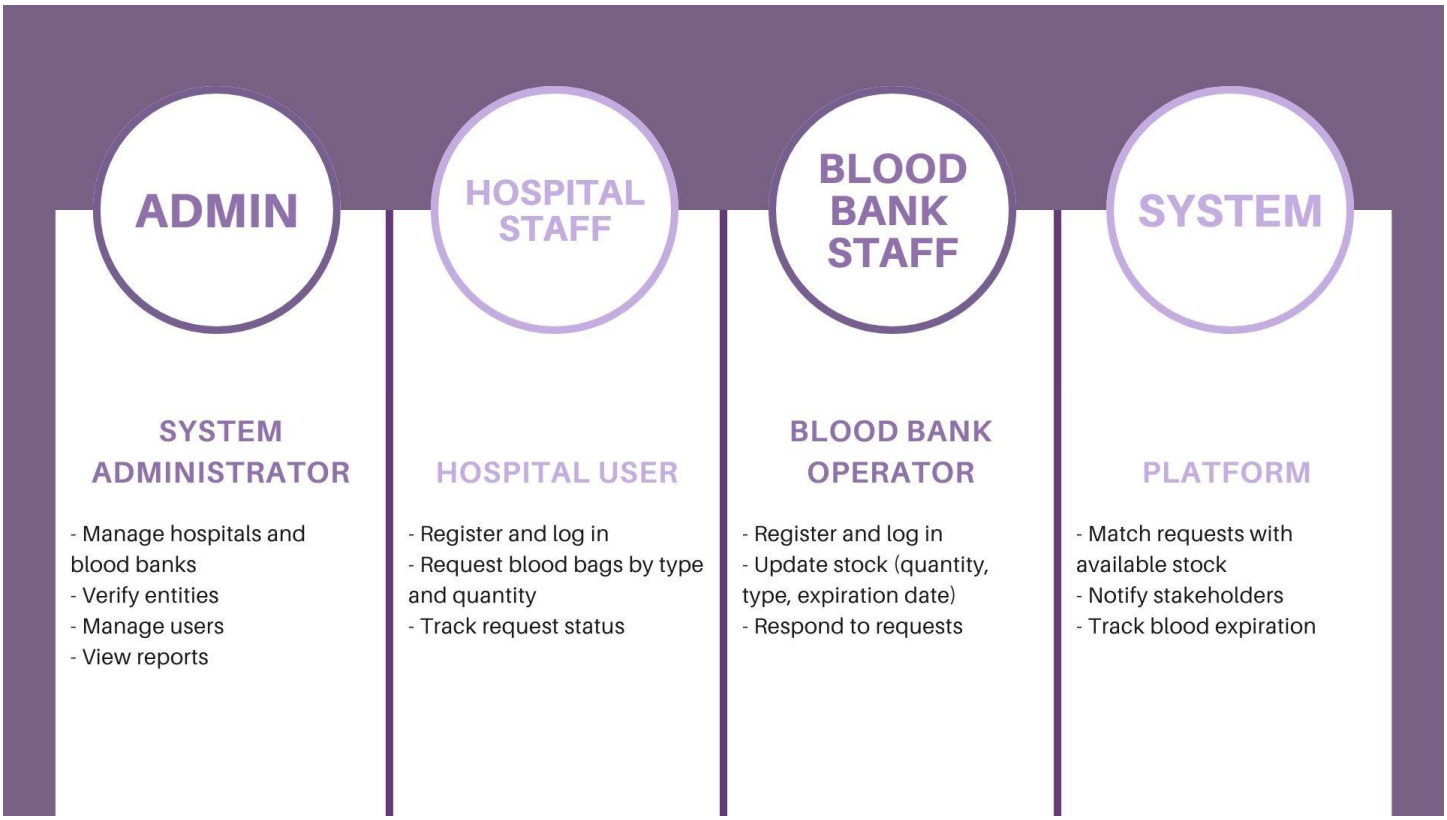
Risk	Impact	Mitigation Strategy
Data inconsistency from unverified sources	High	Restrict registration to admin-approved hospitals and banks
System downtime / slow performance	Medium	Use reliable hosting, load balancing, and regular backups
Security risks (data leaks, unauthorized access)	High	Encrypt passwords, validate inputs, and apply role-based access control
Insufficient or outdated stock data	Medium	Send automated reminders for periodic stock updates
Delay in development schedule	Medium	Follow agile methodology with weekly progress reviews

Key Performance Indicators (KPIs)

	Target / Goal	Description
System Uptime	≥ 99%	Ensure continuous system availability
Response Time	≤ 2 seconds per query	Guarantee fast and efficient search results
Data Accuracy	≥ 95%	Maintain verified and reliable hospital data
Request Fulfillment Rate	≥ 90%	Measure percentage of fulfilled hospital blood requests
User Adoption Rate	Increasing monthly usage	Track growth of users and hospitals on the platform
Admin Verification Time	≤ 24 hours	Average time to verify and activate new accounts

3. Requirements Gathering

Stakeholder Analysis



User Stories & Use Cases

User Stories

- As a hospital, I want to request a specific number of blood bags so that I can treat patients.
- As a blood bank, I want to update my stock so that hospitals know the available quantity.
- As an admin, I want to add and verify hospitals and blood banks so that the system remains trusted.
- As a blood bank, I want to see pending requests so I can fulfill them.
- As a user, I want to search for a blood type so that I can find the nearest blood bank that has it.

Use Cases

- **Use Case 1: Hospital Request Blood Bags**

Actor: Hospital

- **Preconditions:**
  - Hospital must be a verified user.
  - Hospital must be logged in.
- **Main Flow:**
  1. Hospital logs in.
  2. Hospital submits a new blood request (blood type + quantity + patient info).
  3. System checks stock availability in nearby blood banks.
  4. If a matching blood bank has sufficient stock →
     ✓ System marks the request as **Fulfilled**
     ✓ System notifies both hospital and blood bank
  5. If stock is not available →
     ✓ Request is marked as **Pending**
     ✓ System waits for stock updates
- **Postconditions:**
  - Request stored in the database.
  - Status = Pending or Fulfilled.

Use Case 2: Blood Bank Updates Stock

- **Actor: Blood Bank**

- **Preconditions:**
  - Blood bank must be verified.
  - Blood bank must be logged in.
- **Main Flow:**
  1. Blood bank logs in.
  2. Blood bank updates stock (blood type + quantity + expiration date).
  3. System saves updated stock.
  4. System checks if any pending requests can now be fulfilled.
  5. System notifies hospital(s) with pending requests.
- **Postconditions:**
  - Updated stock is stored.
  - Pending requests may be resolved automatically.

Use Case 3: Admin Management (Add & Verify Entities)

- **Actor**: Admin

- **Preconditions:**
  - Admin must be logged in.
  - Admin must have role Admin.
- **Main Flow:**
  1. Admin logs in.
  2. Admin creates a new hospital / blood bank user.
  3. Admin sets verified = true after validation.
  4. System stores who created the entry (CreatedById).
- **Postconditions**:
  - New verified entity is added.
  - Can now use system functionalities.

4. **Use Case 4: Search and Locate Blood Type**

Actor: User (Public / Not Logged In)

- **Preconditions:**
  - None (public access)

- **Main Flow:**

1. User opens the website.
2. User selects the required blood type.
3. System retrieves blood banks that have this blood type in stock.
4. System sorts blood banks by distance (using governorate).
5. User sees only:
   - Blood bank name
   - Phone number
   - Location
   - Working hours
   - Available quantity
6. **User cannot send a request**
   → Only **view information** (no login, no request action).

- **Postconditions:**
- User gets a list of available blood banks for the selected blood type.
- No request is created at any stage.

Functional Requirements

1. User Management

- The system supports user registration and login.
- Role-based access: **Admin**, **Hospital**, **Blood Bank**.
- Admin can verify hospitals and blood banks before activating their accounts.
- Only verified hospitals and blood banks can use system features.

2. Hospital Management (Admin Only)

- Admin can add, update, and delete hospital accounts.
- Admin can verify hospitals (verified = true).
- The system stores who created each hospital.

3. Blood Bank Management (Admin Only)

- Admin can add, update, and delete blood bank accounts.
- Admin can verify blood banks (verified = true).
- The system stores who created each blood bank.

4. Stock Management (Blood Bank Only)

- Blood banks can add stock (blood type, quantity, expiration date).
- Blood banks can update stock and quantities.
- The system tracks creation and update timestamps.
- Stock visibility is available for hospitals and public users.

5. Blood Request Management

Hospital:

- Hospitals can create blood requests (blood type, quantity, patient details).
- Hospitals can view request status (Pending, Fulfilled).

Blood Bank:

- Blood banks can view pending requests assigned to them.
- Blood banks can fulfill or reject requests.

System:

- Automatically updates request status based on stock availability.
- Public users **cannot** create requests.

7. Public Blood Type Search (Public User)

- Public users can search for blood types without logging in.
- System shows:
    - Blood bank name
    - Location
    - Contact info
    - Working hours
    - Available quantity
- Public users **can only view**, not request blood.

Non-functional Requirements

Performance

- System should respond within 2 seconds.
- Support multiple hospitals and blood banks simultaneously.

Security

- Passwords must be encrypted.
- Role-based access control (RBAC).
- Protection against SQL Injection and XSS.

Usability

- Simple user interface for hospitals and admins.
- Multilingual support (English, Arabic).

Reliability

- System available 24/7.
- Regular data backups.
- Effective notification system.

4.System Analysis & Design

Problem Statement & Objectives

Problem Statement:

Blood shortages and delayed communication between hospitals and blood banks often result in preventable emergencies. There is no centralized, real-time system that allows hospitals to efficiently request blood or enables the public to find the nearest blood bank with their required blood type available.

Objectives:

- **To create a centralized digital platform that connects users, hospitals, and blood banks.**
- **To allow public users to easily find the nearest blood bank that has their required blood type in stock.**
- **To enable hospitals to submit and manage blood requests efficiently.**
- **To allow blood banks to update and monitor blood stock levels in real-time.**
- **To give admins control over verifying institutions and ensuring data accuracy.**

Use Case Diagram & Descriptions

Main Actors:

1. **Admin – Verifies hospitals and blood banks, manages system data.**
2. **Hospital – Sends blood requests to nearby blood banks.**
3. **Blood Bank Staff – Updates and manages available blood stock.**
4. **Public User – Searches for available blood based on type and location.**

Use Cases:

1. **Login**
   Admin / Hospital / Blood Bank logs into the system.
2. **Search Available Blood (Public User)**
   User searches by blood type → system shows nearest blood banks with available stock (view only).
3. **Manage Stock (Blood Bank)**
   Add, update, or delete blood stock (type, quantity, expiration).

4. **Create Blood Request (Hospital)**
   Hospital submits a blood request (type + quantity + patient info).
5. **Handle Requests (Blood Bank)**
   Blood bank views pending requests and fulfills them if stock is available.
6. **Verify Accounts (Admin)**
   Admin verifies hospitals and blood banks before activating them.
7. **User Management (Admin)**
   Admin adds, updates, or deletes hospitals/blood banks.

Use Case Descriptions:

Use Case	Actor	Description
Search for Blood	**Public User**	**User search for nearest blood bank with a specific blood type available.**
Send Request	**Hospital**	**Hospital submits a request for a specific blood type and quantity.**
Update Stock	**Blood Bank Staff**	**Updates stock levels, types, and expiration dates.**
Manage Verification	**Admin**	**Reviews and verify new hospitals or blood banks before activation.**
Track Request Status	**Hospital**	**Check whether the request is pending, fulfilled, or expired.**

Functional & Non-Functional Requirements

Functional Requirements:

1. **Authentication & Authorization**
   - **Role-based accounts: donor (user), hospital, bloodbank, admin.**
   - **Secure login, password reset, session management.**
2. **Organization Management**
   - **Admin approves/verifies hospitals and blood banks.**
   - **Hospitals and blood banks can update profile and working hours.**

3. **Geolocation & Search**
    - **Store coordsX (latitude) and coordsY (longitude) for hospitals and blood banks.**
    - **Accept user location and compute nearest organizations using Haversine distance.**
    - **Filter results by bloodType, availability, city.**
4. **Stock Management**
    - **Blood bank adds/updates stock (type, quantity, availability, expirationDate, lastUpdated).**
    - **Automatic low-stock thresholds and expiry tracking.**
5. **Request Management**
    - **Hospitals create requests (bloodType, quantity, status, createdAt, endAt).**
    - **Blood banks and donors can view and respond to requests.**
    - **Request lifecycle: open → pending → Approved / cancelled.**
6. **Donation Scheduling & Tracking**
    - **Donors schedule donation; hospital/blood bank confirms; donation recorded.**
7. **Admin Dashboards & Reporting**
    - **KPIs, request fulfillment rates, verification queue, system metrics.**

Non-Functional Requirements:

- **Performance: Response time ≤ 3s for search queries under normal load.**
- **Availability: System uptime ≥ 99%.**
- **Scalability: Able to scale to thousands of concurrent searches/updates.**
- **Security: Password hashing, HTTPS/TLS, role-based access control, input validation, DB access controls.**
- **Data Integrity: ACID compliance for inventory and request transactions.**
- **Usability: Mobile-first responsive UI, accessible forms following WCAG basics.**
- **Compliance: Personal data handling per local privacy laws.**

Software Architecture

MVC + API (.NET) – mix of MVC and API for clear separation of concerns and maintainability.

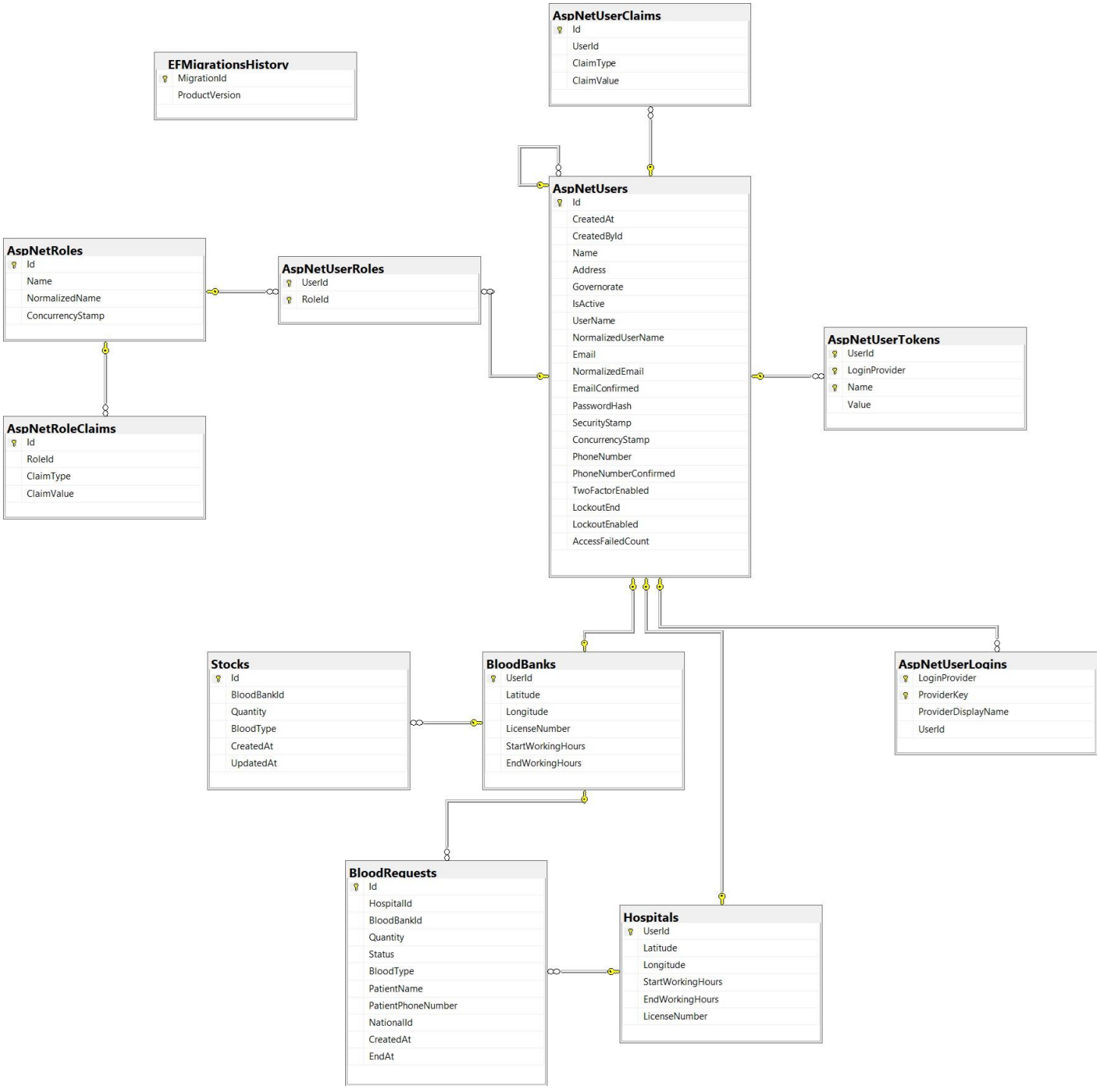
High-level components:

- **Frontend (Client): HTML, CSS, JavaScript.**
- **Backend API: .NET Core (C#) with MVC + RESTful endpoints.**
- **Database: SQL Server for structured data (Users, Requests, Stock).**
- **Authentication: Built-in .NET Auth with JWT.**
- **Admin Console: integrated into backend.**
- **Testing & Documentation: Swagger for API testing.**

Interactions:
Frontend ↔ Backend API ↔ SQL Server
Background tasks for notifications and low-stock alerts.

Database Design & Data Modeling

ER Diagram



Logical Schema (tables & keys)

Admin

PK: Id
Custom fields: Name, Address, Governorate, Latitude, Longitude, IsActive, CreatedAt, CreatedById
Relations:

- 1→Many: UserClaims, UserTokens, UserLogins
- Many→Many: Roles (via UserRoles)
- 1→1: BloodBanks, Hospitals

AspNetRoles PK: **Id**

AspNetUserRoles Composite PK: **(UserId, RoleId)**

AspNetRoleClaims PK: **Id** FK: **RoleId**

AspNetUserClaims PK: **Id** FK: **UserId**

AspNetUserLogins Composite PK: **(LoginProvider, ProviderKey)**

AspNetUserTokens Composite PK: **(UserId, LoginProvider, Name)**

Blood Banks

PK / FK: UserId → AspNetUsers.Id
Fields: Latitude, Longitude, LicenseNumber, StartWorkingHours, EndWorkingHours
Relations:

- 1→Many: Stocks
- 1→Many: BloodRequests

Hospitals

PK / FK: UserId → AspNetUsers.Id
Fields: Latitude, Longitude, LicenseNumber, StartWorkingHours, EndWorkingHours
Relations:

- 1→Many: BloodRequests

Stocks **PK:** Id

FK: BloodBankId → BloodBanks.UserId

Fields: BloodType, Quantity, CreatedAt, UpdatedAt

BloodRequests PK: Id
FKs:

- HospitalId → Hospitals.UserId
- BloodBankId → BloodBanks.UserId

Fields: Quantity, Status, BloodType, PatientName, PatientPhoneNumber, NationalId, CreatedAt, EndAt

Normalization & Constraints

Normalization (1NF → 3NF)

1NF (Atomic fields, no repeating groups)

- All columns are atomic (no multi-valued fields).
- No repeating groups.

2NF (Full dependency on primary key)

All non-key attributes depend entirely on primary keys:

Examples:

- In BloodRequests, all attributes depend on Id.
- In Stocks, all depend on Id.

No partial dependencies (no composite PK except in Identity mapping tables, which is expected).

3NF (No transitive dependencies)

No attribute depends on a non-key attribute:

- Hospitals store only their data (no derived info).
- BloodBanks store only their operational info.
- BloodRequests stores request-specific info.
- No attribute like “HospitalName” is stored redundantly (it is referenced via AspNetUsers).
-

Constraints: Business, Referential & Data Integrity

1. Primary Key Constraints

Table	Primary Key
AspNetUsers	Id
AspNetRoles	Id
AspNetUserClaims	Id
AspNetRoleClaims	Id
AspNetUserTokens	(UserId, LoginProvider, Name)
AspNetUserLogins	(LoginProvider, ProviderKey)
Stocks	Id
BloodBanks	UserId
Hospitals	UserId
BloodRequests	Id

2. Foreign Key Constraints

Child Table	**FK**	**Parent Table**
BloodBanks	UserId	AspNetUsers
Hospitals	UserId	AspNetUsers
Stocks	BloodBankId	BloodBanks
BloodRequests	HospitalId	Hospitals
BloodRequests	BloodBankId	BloodBanks
AspNetUserClaims	UserId	AspNetUsers
AspNetUserRoles	UserId	AspNetUsers
AspNetUserRoles	RoleId	AspNetRoles
AspNetRoleClaims	RoleId	AspNetRoles
AspNetUserTokens	UserId	AspNetUsers
AspNetUserLogins	UserId	AspNetUsers

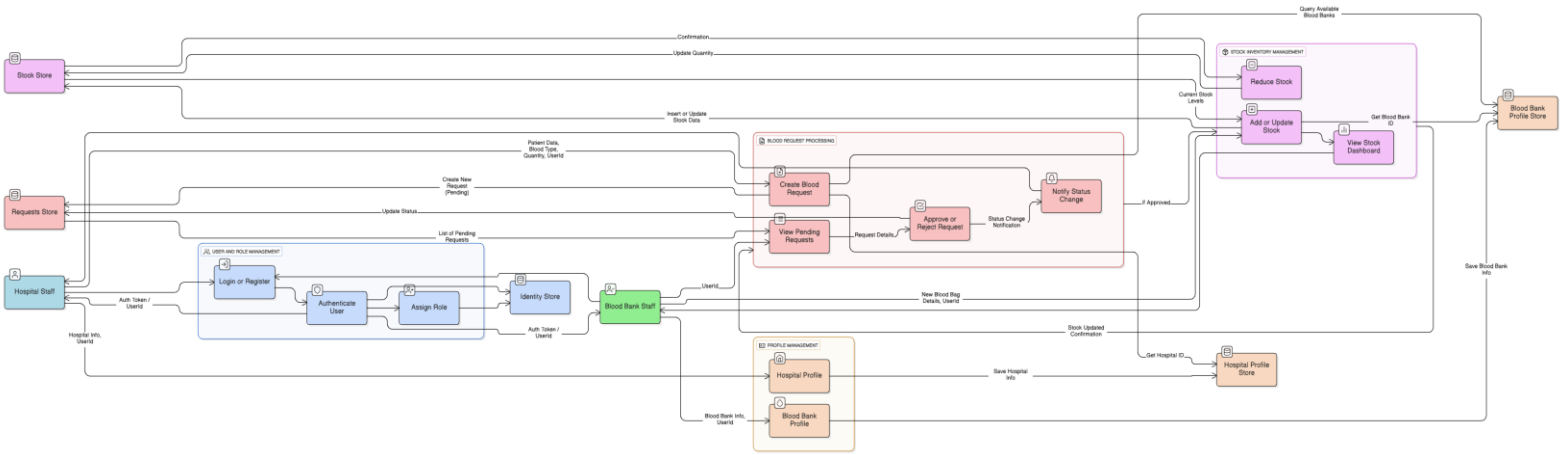
Data Flow & System Behavior

Data Flow Diagram (DFD)

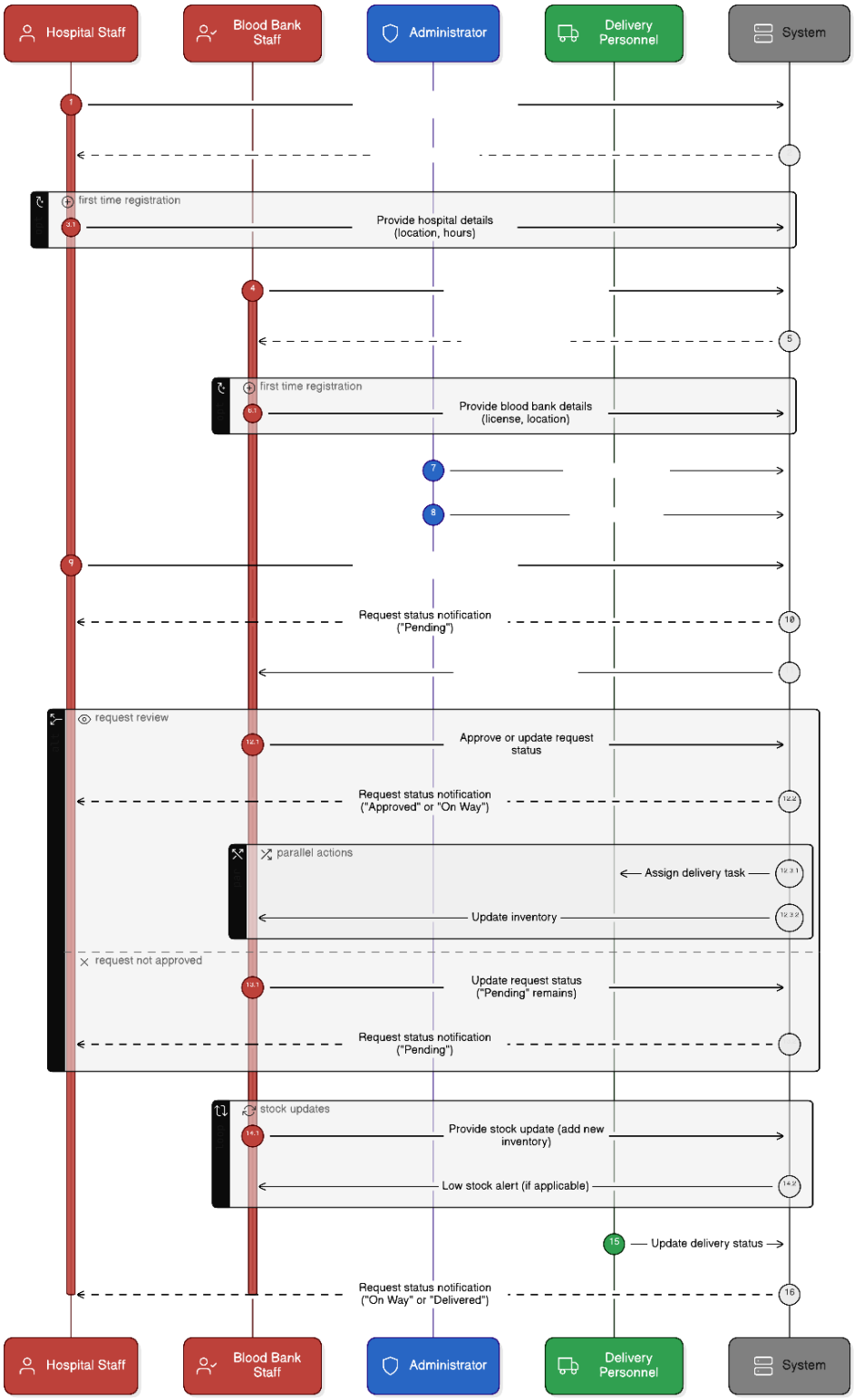
Context-level (Level 0):

- **External entities: User(Admin), Hospital, Blood Bank.**
- **Processes: Search Service, Request Service, Stock Management, Admin Verification.**
- **Data stores: Database (users, hospitals, bloodbanks, stock, requests).**
- **Data flows: location & query → Search Service → results; request creation → Request Service → DB and Notification Service.**

Level 1:



Sequence Diagrams (key flows)



Activity Diagram (example)

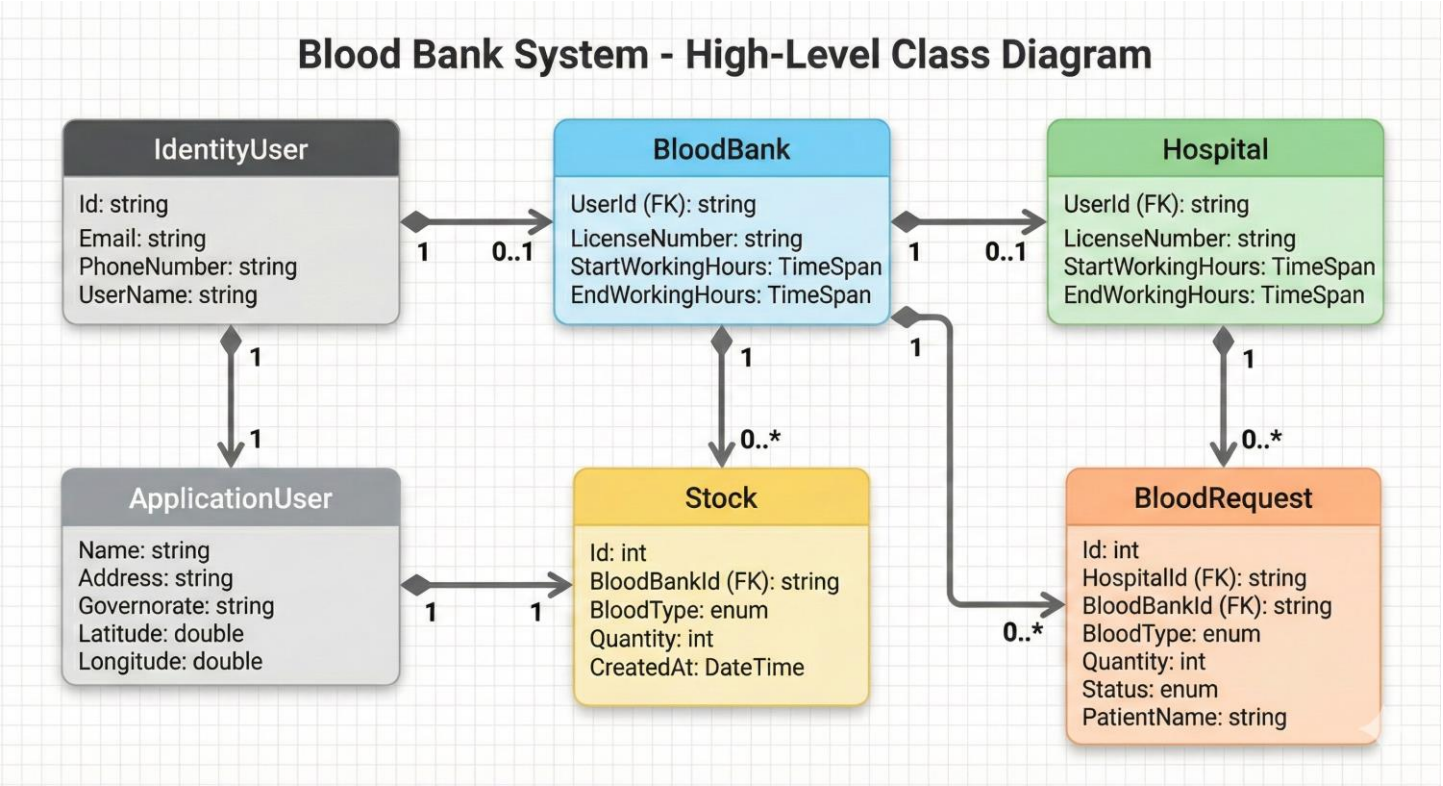
Activity: Request Fulfillment

- **Start → Hospital creates request → System validates → Notify banks/donors → Wait for responses → If response accepted → Arrange transfer/donation → Update stock & request as fulfilled → End.**

State Diagram (example for requests)

- **States: open → pending (accepted by donor/bank) → in process (transfer scheduled) → fulfilled / cancelled / expired.**
- **Transitions triggered by actions: accept, cancel, timeout, mark_fulfilled.**

Class Diagram (high-level classes)



UI/UX Design & Prototyping

Screens / Wireframes

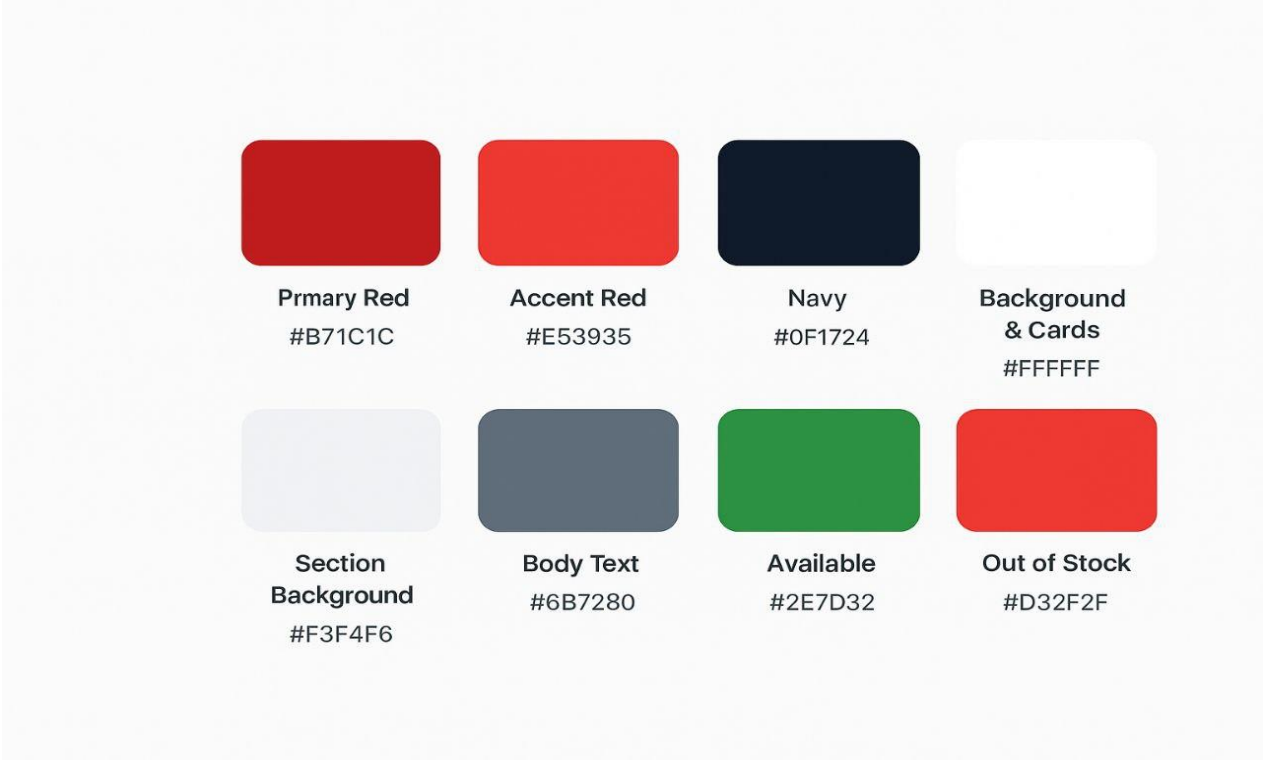
1. **Landing / Home Page — search bar for blood type + location, call-to-action (Donate / Request).**
2. **Search Results / Map View — list + map with nearest hospitals & blood banks, availability badges.**
3. **Hospital Dashboard — create request, status list, request history.**
4. **Blood Bank Dashboard — manage stock, respond to requests, expiration alerts.**
5. **Admin Panel — verification queue, metrics, user & org management.**
6. **Profile & Settings — user details, blood type, contact preferences.**

UI/UX Guidelines

- **Color scheme: Red accent (for urgency) + neutral background (white/gray) to improve readability.**
- **Icons / Visuals: Use clear icons for blood types, availability, urgency.**
- **Typography: Clear sans-serif fonts (e.g., Inter, Roboto) with readable sizes (≥14px body).**
- **Accessibility: Keyboard navigation, ARIA labels for form elements, contrast ratio ≥ 4.5:1 for text.**
- **Mobile-first: Ensure search and map are usable on small screens; quick actions for urgent requests.**
- **Feedback: Show confirmations for actions (request created, donation scheduled), and error messages where needed.**

LOGO **Brand**

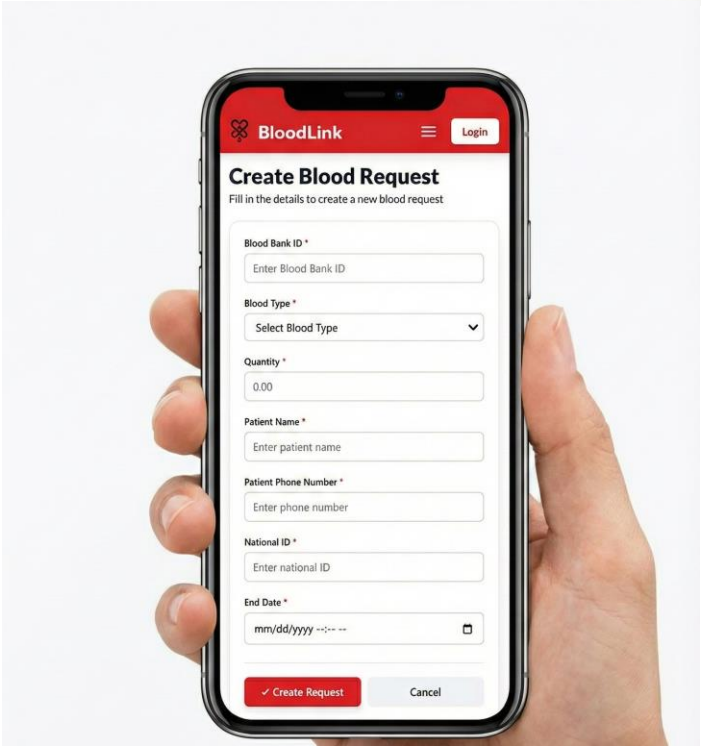
color palette



Some pages mockup:

Home Page

About Page



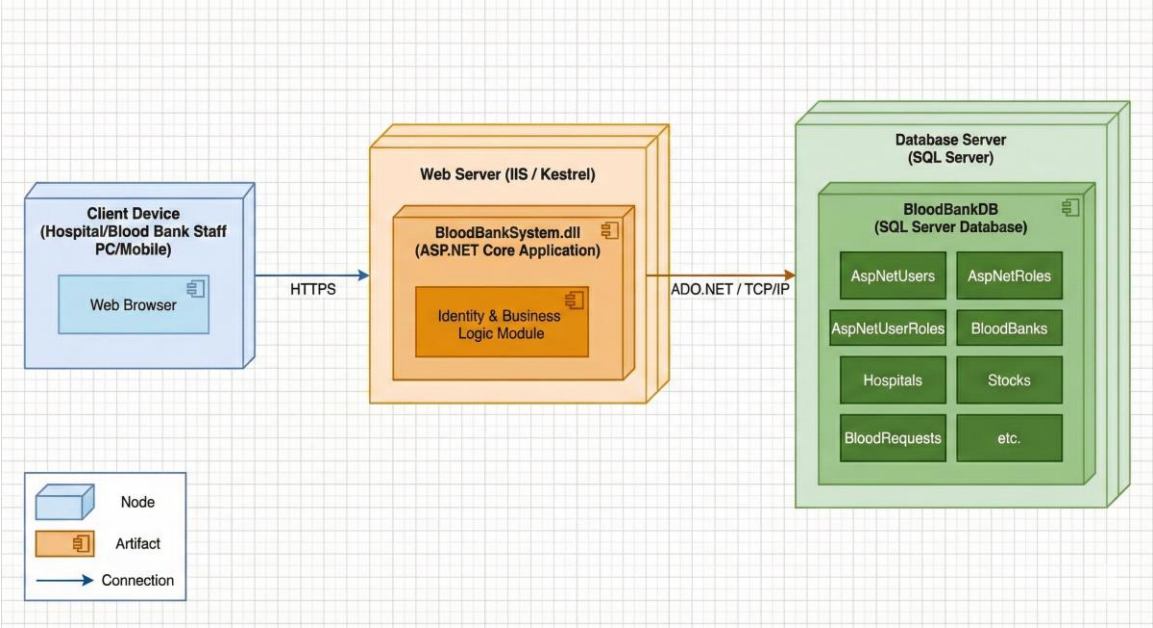
Blood Request Page

System Deployment & Integration

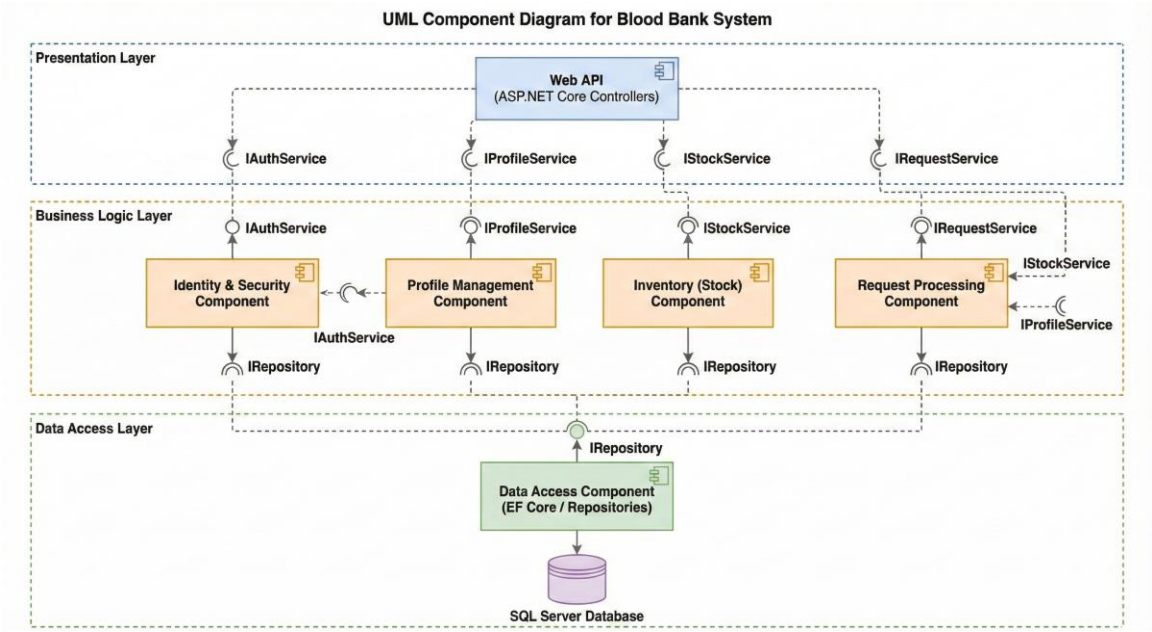
Technology Stack

- **Backend:** C#, ASP.NET Core
- **Frontend:** HTML, CSS, JavaScript
- **Database:** SQL Server / Entity Framework Core
- **Authentication:** JWT (JSON Web Tokens)
- **API Documentation:** Swagger
- **Version Control:** Git / GitHub
- **Hosting:** Local environment using IIS Express / Kestrel with MVC server-side rendered views.

Deployment Diagram



Component Diagram



Additional Deliverables

API Documentation (outline)

- **Example endpoints:**
    - **GET /api/users – Retrieve all users**
    - **POST /api/users – Create a new user**
    - **GET /api/bloodrequests – Retrieve all blood requests**
    - **POST /api/bloodrequests – Create a new blood request**

For full API documentation, see [API Documentation] .

Testing & Validation Plan

- **Unit Testing**: Each component of the system was tested individually to ensure correct behavior and to catch any errors early.
- **Integration Testing**: All modules were tested together to verify seamless interaction and correct data flow across the system.
- **User Acceptance Testing (UAT)**: Conducted to ensure that the system meets the requirements and expectations of end-users.

- **API Testing & Documentation**: We used Swagger to document all endpoints and validate API functionality interactively.

Deployment Strategy

- **Hosting Environment**: The system will be hosted locally, providing a stable and controlled platform for initial deployment.
- **Version Control & Updates**: All updates and changes will be managed via GitHub, ensuring proper versioning, collaboration, and rollback capability.
- **Scaling Considerations**: The system architecture allows for future scalability. Strategies for scaling will be applied based on increased user load or additional feature requirements.

---

5. Implementation (Source Code & Execution)

1. Source Code

- Structured & Well-Commented Code: The code is organized, clean, and easy to maintain. Each module and function includes meaningful comments explaining its purpose and logic.
- Coding Standards & Naming Conventions: All variables, functions, and classes follow consistent naming conventions and coding standards to enhance readability and maintainability.
- Modular Code & Reusability: The system is structured into reusable components, functions, and classes, allowing easy updates and extensions in the future.
- Security & Error Handling: We implemented secure coding practices, input validation, and proper exception handling to prevent errors and ensure system stability.

```
15  namespace App.Application.Services;
16
17  public class CookieAuthService : ICookieAuthService
18  {
19      private readonly UserManager<User> _userManager;
20      private readonly SignInManager<User> _signInManager;
21
22      public CookieAuthService(UserManager<User> manager, SignInManager<User> signInManager)
23      {
24          _userManager = manager;
25          _signInManager = signInManager;
26      }
27
28      public async Task<Result> LoginAsync(LoginDTO loginDTO)
29      {
30          User? user = await _userManager.FindByEmailAsync(loginDTO.Email);
31          if (user is null)
32              return Result.Fail(AppResponses.UnAuthorizedResponse);
33
34
35          if (!user.IsActive)
36              return Result.Fail(AppResponses.UnAuthorizedResponse);
37
38          var result = await _signInManager.PasswordSignInAsync(user, loginDTO.Password, false, false);
39          if (result.Succeeded)
40              return Result.Success();
41
42          return Result.Fail(AppResponses.UnAuthorizedResponse);
43      }
44
45      public async Task<Result> LogoutAsync()
46      {
47          await _signInManager.SignOutAsync();
48          return Result.Success();
49      }
50  }
```

Figure 1 part of CookieAuthService

2. Version Control & Collaboration

Repo Link : https://github.com/marwa-gameil/BloodLink.git

- Version Control Repository: The project is managed using GitHub for version control. A link to the repository is provided for reference and collaboration.
- Branching Strategy: We follow a clear workflow using Feature Branching, which ensures safe development of new features and proper integration into the main branch.
- Commit History & Documentation: All commits are meaningful, reflecting the changes made, and pull requests include detailed descriptions to facilitate collaboration and code review.

3. Deployment & Execution

README File Link

README.md

---

6. Testing & Quality Assurance

Test Cases & Test Plan:

The test plan for BloodLink includes a variety of test scenarios that cover the core features and workflows of the system. Each test case details the scenario, test steps, expected results, and requirements.

Sample Test Scenarios:

1. **User Authentication**
    - **Scenario:** User logs in with valid credentials.
    - **Expected Outcome:** Login succeeds and session is created.
    - **Scenario:** User logs in with wrong password.
    - **Expected Outcome:** Login fails with an error message.
2. **Creating a Blood Request (Hospital User)**
    - **Scenario:** Hospital submits a valid blood request.
    - **Expected Outcome:** The request is registered in the system and appears in pending requests.
3. **Blood Bank Approves/Rejects a Request**
    - **Scenario:** Blood bank user approves a pending blood request.
    - **Expected Outcome:** Request status changes to Approved and requester is notified.
4. **Stock Management**
    - **Scenario:** Blood bank increases stock for a blood type.
    - **Expected Outcome:** Stock quantity is updated in the system.
5. **Admin Deactivates a User**
    - **Scenario:** Admin deactivates a user account.
    - **Expected Outcome:** User cannot log in and is marked as inactive.
6. **Search for Blood Type Availability**
    - **Scenario:** Search for available blood banks with A+ blood type.
    - **Expected Outcome:** Correct blood banks are listed

Bug Reports:
During development, we encountered a few bugs that were fixed along the way, but they were not formally documented at the time.
Here's an example of one of the issues we faced and resolved:

Example of a bug we encountered:
There was an issue where blood requests stayed in the **"Pending"** status even after the blood bank clicked the **Approve** button. The button worked on the UI, but the request status was not updating in the database, and no confirmation message appeared.

Cause:
The approval API endpoint was not correctly updating the request status.

Fix:
We updated the approval logic in the API so that the status is properly saved and a clear response is returned to the UI. After the fix, the approval process worked as expected.

In the future, any bugs that appear will be fully documented and tracked properly.

END