

MedLink

A Secure Longitudinal HealthCare Management System

1. The Team Division :

Group A: The Architects (Backend & Database)

The "Brains" of MedLink

1. Database Integrity (SQL): Your job is to ensure that if a Patient is deleted, their Visits and Trials don't stay "floating" in the database.

- **Task:** Implement ON DELETE CASCADE and FOREIGN KEY constraints.
- **Deliverable:** A script that sets up the 3NF tables we discussed earlier.

2. The Encryption Engine (AES-256): You will write a "Wrapper" function. Whenever data is saved to the Visits table, it must pass through this function first.

- **The Logic:** Plaintext Diagnosis + Secret Key = Ciphertext.
- **Key Challenge:** Keeping the "Secret Key" safe (don't hardcode it in the script; use a .env file).

3. The Hospital API: You create the "doorways" for Group B to get data.

- **Endpoints:** * GET /api/hospitals/patients (Fetch patients only for the logged-in hospital).
 - POST /api/visits/add (Accept data from the UI and encrypt it).
-

Group B: The Builders (Frontend & UI/UX)

The "Face" of MedLink

1. Authentication & Session Management: You handle the login screen. Once Group A gives you a JWT (Token), you must store it securely in the browser.

- **Task:** Attach that Token to the "Header" of every request so the backend knows which hospital is logged in.

2. Longitudinal Timeline Visualization: This is your most important visual feature. Doctors shouldn't just see a list; they should see a story.

- **Task:** Use a library like React-Vertical-Timeline to map out the VisitDate from the database.
- **Visuals:** Use icons to distinguish between a "Doctor Visit," a "Surgery," and a "Clinical Trial Enrollment."

3. Role-Based Access (UI Side):

- **Task:** Ensure that if a user is not logged in as a "Hospital Admin," they cannot even see the "Clinical Trial Management" tab.
-

2. The Step-by-Step Roadmap :

Step 1: The Blueprint (Database & Normalization)

The Goal: Build a perfectly organized filing system.

- **Normalization (3NF):** Every piece of info has exactly one home. You don't store the Hospital's address in the Patient's file; you store it in the Hospitals table and link them. This prevents errors when data changes.
 - **Junction Tables:** In a database, a Patient can join many Trials, and a Trial has many Patients. You can't fit this into one row. You create a "Bridge" table (Trial Enrollment) that lists: Patient ID | Trial ID. It's like a registration log.
-

Step 2: The Vault (Security & Encryption)

The Goal: Make the data unreadable to hackers.

- **Asymmetric Encryption:** This is the "Postbox" method. Anyone can drop a letter into the postbox (Public Key / Hospital Level), but only the person with the key (Private Key / Authorized Doctor) can open the box to read the letters. Even if a hacker steals the database, all they see is scrambled code.
 - **Hashing (bcrypt):** We don't "encrypt" passwords; we "hash" them. Hashing is a one-way street. You can turn "Password123" into "xh77!@#," but you can't turn "xh77!@#" back into "Password123." When you log in, the system hashes your input and compares it to the saved hash.
 - **RBAC (Middleware):** This is the "Security Guard" at the door. Every time a request comes in, the middleware checks the user's ID badge. If the badge doesn't say "Hospital," the guard blocks them before they even reach the data.
-

Step 3: The Messenger (Backend API)

The Goal: Create the bridge between the Database and the Website.

- **API Routes:** These are specific URLs that perform actions.
 - **Register Patient:** The backend takes clear text from the website, "locks" it (encrypts it), and then stores the locked version in SQL.
 - **Patient History:** The backend pulls the locked data, "unlocks" it (decrypts it) using the private key, and sends the readable version to the doctor's screen.
 - **Trial Eligibility:** The backend runs a smart search: "Show me all patients who have 'Diabetes' in their history and are over age 40."
-

Step 4: The Interface (Frontend & Timeline)

The Goal: Turn rows of data into a visual story.

- **Longitudinal View:** Instead of a boring table, you build a Vertical Timeline.
- **How it works:** You sort the Patient's medical events by date.
 - The top of the line is the most recent (2023: New Medication).
 - The bottom is the past (2021: First Diagnosis).
- **React.js:** This allows the website to be "Reactive." When a doctor clicks a patient's name, only the timeline updates, not the whole page. It feels fast and professional.

Summary of Member Roles for this Roadmap

Phase	Group A (Architects)	Group B (Builders)
Week 1	Write the SQL CREATE scripts.	Sketch the Dashboard layout.
Week 2	Code the Encryption functions.	Build the Login/Sign-up screens.
Week 3	Build the API endpoints (Node/Python).	Connect the screens to the API.
Week 4	Final testing & Audit Logs.	Code the Timeline & CSS styling.

MedLink
Secure. Longitudinal. Connected.

3. The Technology Stack :

1. Frontend: React.js + Tailwind CSS

- **What it is:** The "Face" of the website.
- **Easy Explanation:** * React.js is like Lego blocks. You build a "Patient Card" or a "Navigation Bar" once and reuse it everywhere. It makes the website feel fast because it only updates the part of the screen you are looking at, rather than reloading the whole page.
 - Tailwind CSS is like a massive box of pre-painted stickers. Instead of writing long CSS files to make a button blue, you just give it a "sticker" (a class name), and it looks professional instantly.
- **Why for MedLink?**

It allows you to build that complex Longitudinal Timeline smoothly.

2. Backend: Node.js (Express)

- **What it is:** The "Manager" or "Brain."
- **Easy Explanation:** When a doctor clicks "Search Patient," the Frontend asks the Manager (Node.js). Node.js then talks to the Database, checks security, and sends the answer back.
- **Why for MedLink?**

It is "Asynchronous." This means it can handle 100 doctors searching for 100 different patients at the exact same time without the website freezing.

3. Database: PostgreSQL

- **What it is:** The "Vault" or "Filing Cabinet."
- **Easy Explanation:** While simple databases store things like an Excel sheet, PostgreSQL is much smarter. it is built to handle complex relationships (e.g., linking a Patient to a Visit, then to a Doctor, then to a Hospital) without losing data.
- **Why for MedLink?**

Healthcare data is complex and relational. PostgreSQL is famous for "Data Integrity"—it ensures that your medical records never get mixed up or corrupted.

4. Security: Crypto.js / OpenSSL

- **What it is:** The "Scrambler."
- **Easy Explanation:** This tool takes a readable diagnosis like "*Patient has a Fever*" and turns it into a secret code like "*8f2gH9kL0...*" before it ever reaches the database.
- **Why for MedLink?**

Even if a hacker steals the database, the patient's privacy is protected because the hacker doesn't have the "Key" to unscramble the code

5. Auth: JWT (JSON Web Tokens)

- **What it is:** The "Digital ID Badge."
- **Easy Explanation:** After a doctor logs in, the system gives them a JWT. It's like a stamp on your hand at a club. For the rest of the day, whenever the doctor wants to see a patient's record, they just show the "stamp" (the token). The system sees the stamp and says, *"Okay, you are a doctor from City Hospital, you are allowed to see this."*
- **Why for MedLink?**

It's a secure way to remember who is logged in without making them enter their password on every single page.

Quick Summary for Members

Tool	Real-World Role
React	The beautiful dashboard the doctor sees.
Node.js	The engine that moves data back and forth.
PostgreSQL	The secure organized filing cabinet.
Crypto.js	The lock on the medical files.
JWT	The security badge that grants access.

MedLink
Secure. Longitudinal. Connected.

4. The MedLink ER Diagram Blueprint :

1. Entities & Attributes

- **Hospital:** HospitalID (PK), Name, Location, LicenseNumber.
- **Doctor:** DoctorID (PK), HospitalID (FK), Name, Specialization, Email.
- **Patient:** PatientID (PK), Name, DOB, Gender, Contact, MedicalHistory_Hash.
- **Visit (The Longitudinal Anchor):** VisitID (PK), PatientID (FK), DoctorID (FK), VisitDate, Diagnosis (Encrypted), Symptoms (Encrypted).
- **Treatment:** TreatmentID (PK), VisitID (FK), Medication, Dosage, Duration.
- **ClinicalTrial:** TrialID (PK), TrialName, Phase, Sponsor, StartDate.
- **TrialEnrollment (Junction Table):** EnrollmentID (PK), PatientID (FK), TrialID (FK), Status, EnrollmentDate.
- **TrialOutcome:** OutcomeID (PK), EnrollmentID (FK), ObservationDate, Result (Encrypted).

2. Logical Relationships (The "Rules")

To ensure **Referential Integrity** and your "Hospital-Only" requirement, follow these connections:

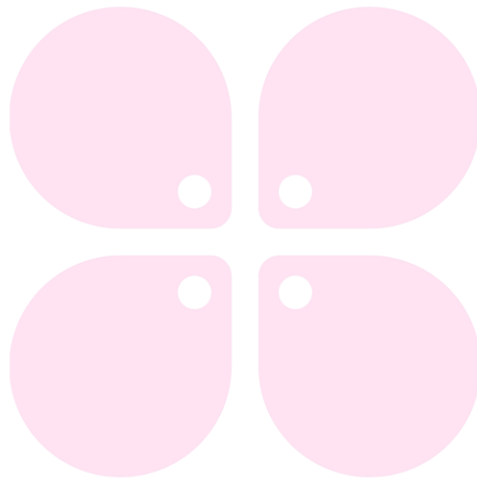
1. **Hospitals to Doctors (1:N):** One hospital employs many doctors. A doctor belongs to exactly one hospital.
2. **Doctors to Visits (1:N):** One doctor can conduct many visits.
3. **Patients to Visits (1:N):** One patient has many visits over time (this creates the **Longitudinal** timeline).
4. **Visits to Treatments (1:N):** One visit can result in multiple prescribed medications/treatments.
5. **Patients to Clinical Trials (M:N):** A patient can participate in many trials, and a trial has many patients. We resolve this using the **TrialEnrollment** table.

3. Implementation Roadmap (Task Division)

Step	Group A (Backend & DB - 2 Members)	Group B (Frontend & UI - 2 Members)
Phase 1	SQL Schema: Write CREATE TABLE scripts with PK/FK constraints in PostgreSQL.	Wireframing: Design the Hospital Login and the Patient Timeline Dashboard.
Phase 2	Security Layer: Implement AES-256 logic. Create a function: Encrypt(Diagnosis) -> Ciphertext.	Auth System: Build the login forms and protected routes (No login = No data).
Phase 3	API Development: Create endpoints (e.g., GET /patient/:id/timeline) that join Visit and Treatment tables.	Data Display: Fetch the API data and render it using a vertical timeline library.
Phase 4	Referential Integrity Check: Ensure deleting a patient also handles their visit history (Cascading).	Final Integration: Add search filters for doctors to find patients by ID or Chronic Condition.

4. Recommended Tech Stack

- **Modeling:** draw.io or Lucidchart (for the ER Diagram).
 - **Database:** PostgreSQL (It handles relational data and complex joins better than MySQL).
 - **Encryption:** CryptoJS (for Node.js) or PyCryptodome (for Python).
 - **Frontend:** React with Tailwind CSS (for a clean "Pro" hospital look).
-



MedLink

Secure. Longitudinal. Connected.

5. Security Workflow (Hospital Only Access) :

Step 1: Authentication (The Badge Check)

- **Who does this:** Frontend (Group B) collects credentials; Backend (Group A) verifies them.
 - **Process:** A doctor enters their username and password. The system checks the Doctors table.
 - **Key Detail:** Passwords are never stored as plain text. They are "Hashed" (transformed into a random string of characters) so no one, not even the admin, knows the actual password.
-

Step 2: Authorization (The Golden Ticket - JWT)

- **Who does this:** Backend (Group A).
 - **Process:** Once the password is confirmed, the server gives the browser a **JWT (JSON Web Token)**.
 - **Analogy:** It's like a "Day Pass" at a theme park. The doctor doesn't have to log in every time they click a button; they just show their digital "Day Pass" (the token).
 - **Security:** The token contains the HospitalID, ensuring the doctor can only stay within their authorized area.
-

Step 3: Data Request (The Permission Gate)

- **Who does this:** Both Groups.
 - **Process:** The doctor clicks "View Patient History."
 - **The Guard:** Before the database runs the query, the Backend checks the token. It says: *"Is this doctor from Hospital A? Yes. Is he trying to access a record? Okay, proceed."* * **Restriction:** If a doctor from Hospital B tries to use their token to see Hospital A's data, the system blocks the request immediately (**403 Forbidden**).
-

Step 4: Decryption (Opening the Vault)

- **Who does this:** Backend (Group A).
 - **Process:** The data coming out of the database looks like gibberish (Ciphertext).
 - **The Action:** The server uses a secret **Private Key** (stored securely on the server, never in the DB) to turn that gibberish back into readable text (e.g., "Patient has Hypertension").
 - **Benefit:** This is **End-to-End Encryption**. The data is only "readable" for a split second in the server's memory before being sent to the doctor's screen.
-

Step 5: Audit Log (The Black Box Recorder)

- **Who does this:** Backend (Group A).
 - **Process:** Every time Step 4 happens, the system automatically writes a secret entry into an AuditLogs table.
 - **Data Stored:** [Timestamp] | [Doctor_ID] | [Patient_ID] | [Action: Viewed Record].
 - **Why?** If a data leak is ever suspected, the hospital can look at the "Black Box" to see exactly who looked at what and when.
-

Workflow Visualization

Member	Task
Member 1 (Backend)	Write the Encryption/Decryption functions using a library like crypto.
Member 2 (Backend)	Setup the JWT Logic and the AuditLog table in the database.
Member 3 (Frontend)	Create the Login Form and store the JWT in "Local Storage" or "Cookies."
Member 4 (Frontend)	Build the Dashboard that sends the JWT in the header of every API request.

