Project Goal: People Connection & Event Manager (PCEM)

1. Project Objective

To deliver a multi-user, relational application backend (PCEM) that allows authenticated users to securely store, manage, and receive proactive, time-based notifications for special events related to their contacts. This system is designed to be a reliable source of truth for personal relationship data, moving beyond simple address books by focusing on temporal intelligence, data versatility, and enterprise-grade reliability.

Key Success Metrics

1. **Versatility:** Successful implementation of the `metadata` JSONB column in PostgreSQL for flexible data storage. Achieving versatility means the user can track arbitrary, non-standard information (e.g., "Favorite Book Title," "Last Discussed Topic," "Children's School Name") for any given contact **without requiring a database schema migration.** This design future-proofs the application against evolving user data needs.
2. **Integrity:** Robust data integrity via foreign keys, cascade deletion, and **database transactions**. This ensures that complex write operations (like updating a person and logging the change) are atomic and that the system state is never corrupted.
3. **Reliability & Scalability:** Successful operation of the Asynchronous Notification Runner (M4) using **Bull/Redis job queues**. The system must guarantee that scheduled notifications are processed even during peak API traffic or system restarts, requiring robust background task management with persistent queuing systems.
4. **Security:** All data access is strictly scoped by `userId`. This mandate applies equally to CRUD operations in M2 and M3, and query operations within the M4 Notification Runner.

2. Technical Stack and Constraints

The project utilizes a powerful relational stack optimized for scalability, transactional integrity, and maintainability.

Component	Technology	Rationale/Constraint
Backend Framework	NestJS (TypeScript)	Existing project environment. Provides modularity, dependency injection, and clear structure for complex applications, accelerating development speed and simplifying maintenance.
Database	PostgreSQL	Existing project environment. Chosen for its proven reliability, strong transactional integrity (ACID compliance), and advanced features, particularly native `JSONB` storage which bridges the gap between relational structure and NoSQL flexibility.
Data ORM	TypeORM or Prisma	To interact with PostgreSQL entities. Will be used to define schemas, manage database migrations, and handle complex **transactions**.

Async Tasks	**Bull/Redis**	To manage the recurring Notification Runner (M4). **Bull/Redis is mandatory** for guaranteed execution, queue visibility, and robust retry mechanisms, offering the scalability needed for production.
External Delivery	TBD (e.g., SendGrid, Twilio)	Placeholder for external communication APIs. The integration module must be abstract to allow easy switching between providers without rewriting core notification logic.

3. Project Goals: Sequential Implementation Roadmap

The project is structured into three developer goals. Each goal must be completed and thoroughly tested before starting the next, ensuring a stable foundation and building complexity layer-by-layer.

GOAL 1: People/Contacts Module (M2)

Objective: Establish the core data structure, secure access, and advanced indexing for versatile contact management.

Task ID	Task Description	Dependencies	Output / Artifacts
G1-T1	Implement `Person` Entity & CRUD	None	`Person` Entity/Model (with `userId`, `firstName`, `lastName`, `primaryContact`), corresponding `PeopleController`, `PeopleService`, and API endpoints. This establishes the primary table schema and basic routing.
G1-T2	Implement **Security Scoping**	G1-T1	Implement a global NestJS Passport Guard and a custom Decorator to extract the `userId`. Service layer functions must *always* pass the `userId` to the repository layer for rigorous access control checks.
G1-T3	Implement **Flexible Schema (** `JSONB` **) Indexing**	G1-T1	Update `Person` entity to include a `metadata` column (`JSONB`). **Crucially**, implement a **GIN or GIST index** on this column to enable efficient searching (e.g., finding all people where `metadata.linkedinUrl` is not null, or searching text within the JSON).
G1-T4	Implement `Relationship` Entity & Linking	G1-T1	Design and implement the `Relationship` Entity (`sourceId`, `targetId`, `type`). Create API/service methods to manage defined, directed connections between contacts.
G1-T5	Implement **Soft Deletion** (Best Practice)	G1-T1	Update the `Person` entity to include a `deletedAt` timestamp. All `GET` queries must automatically exclude records where `deletedAt` is set, and the `DELETE` operation simply sets this timestamp instead of physically removing the row. This provides an audit trail and easy recovery.

GOAL 2: Event Module & Integrity (M3)

Objective: Implement the `Event` entity, enforce transactional and data integrity rules, and prepare data for asynchronous processing.

Task ID	Task Description	Dependencies	Output / Artifacts
G2-T1	Implement `Event` Entity & CRUD	G1-T1	`Event` Entity/Model (with required `personId` foreign key, `eventDate` , `title` , and `isSpecialEvent` boolean), corresponding controllers and services.
G2-T2	Implement **Transactional Integrity**	G2-T1	Refactor service methods that perform multi-step writes (e.g., updating a person's profile and simultaneously creating an associated audit log event). Use **TypeORM/Prisma transactions** to guarantee that either *all* writes succeed or *all* writes fail, maintaining an atomic state.
G2-T3	Implement **Deletion Cascade**	G2-T1	Update the PostgreSQL schema (via migration) to explicitly set the `ON DELETE CASCADE` constraint for the `Event.personId` foreign key. This database-level enforcement ensures integrity with maximum efficiency.
G2-T4	Implement **Timezone Management & Query**	G2-T1	Ensure all dates are stored as UTC in the database. Service logic must handle converting the stored UTC `eventDate` to the user's preferred local timezone for accurate display. The upcoming query (G2-T5) must still operate strictly on UTC date ranges.
G2-T5	Implement **Optimized Upcoming Events Query**	G2-T1	Develop a performant service method (`/api/v1/events/upcoming`) to execute the filtered date range query. The query requires **composite indexing** on `(userId, eventDate)` for optimal performance, as this query will be executed frequently by both the API and the Notification Runner.

GOAL 3: Asynchronous Notification System (M4)

Objective: Implement the resilient, decoupled notification architecture using a Message Queue (Bull/Redis) for scheduling, querying, and external delivery, focusing on failure handling and recurrence logic.

Task ID	Task Description	Dependencies	Output / Artifacts
G3-T1	Implement **Notification Preference Entity**	G1-T1	Create a `NotificationPreference` Entity (linked to `userId`) to store granular settings: `deliveryChannel` (Email/SMS), `leadTimeDays` (e.g., 3, 7), and `dailyDeliveryTime` (the UTC hour for notification checks).
G3-T2	Setup **Bull/Redis Queue Infrastructure**	G2-T5, G3-T1	Implement the **Bull Queue module** in NestJS. Configure a dedicated `notification-schedule` queue and a `delivery` queue. This task sets up the central nervous system for asynchronous processing.

G3-T3	Implement **Notification Query Runner (Producer)**	G2-T5, G3-T2	Implement a **NestJS Cron Job** that runs hourly. Its only job is to: 1. Query the database for users whose `dailyDeliveryTime` aligns with the current hour. 2. Execute the `Upcoming Events Query` (G2-T5) for that user. 3. **Add a job** to the `delivery` queue for each required notification.
G3-T4	Implement **Delivery Worker (Consumer)** & **Failure Handling**	G3-T3	Implement a **NestJS Worker Module** that consumes jobs from the `delivery` queue. This service handles the external API integration (SendGrid/Twilio). **Crucially**, configure Bull to automatically **retry** failed jobs (e.g., 3 times with exponential backoff) before moving them to a "Failed Jobs" set for manual review.