TrackED Manual

Goal

The goal of the TrackED backend was to track student behavior, location presence, clinical assignments, and any incidents in a structured and scalable way. To accomplish that, the project uses a relational database implemented through Entity Framework Core, with a Table-Per-Hierarchy (TPH) inheritance pattern. The core idea was eliminating duplicated data while controlling how each user type behaves in the system.

Basic Design

At the center of the design is the AppUser table. Both Students and Professors inherit from AppUser. This avoids redundant fields such as email, name, phone, and credentials being stored in multiple tables. AppUser holds generalized attributes relevant to all users, while Student and Professor add role-specific fields. This allows the database to scale without rewriting core identity logic. If new user classes are needed in the future (such as clinical supervisors or administrators), they can inherit from AppUser without changes to existing tables.

Assignments were treated as a direct relationship between a Student and a Professor, occurring at a specific time and in a designated Location. An Assignment links three critical pieces: StudentEnumber, ProfessorEnumber, and LocationID. This establishes accountability while maintaining clear relational integrity. The Assignment entity avoids generic calendar-style design where a placement could exist without a responsible party attached. Every assignment has a designated authority and participant.

Location is kept as its own table instead of embedded into entities. This allows locations to evolve, be reused, or be replaced without modifying Assignment or Incident records. Latitude, longitude, and radius are stored as dedicated fields so that any future geofencing logic can be computed without restructuring. These values are intentionally simple numeric primitives rather than GIS-specific types because the project does not require spatial indexing or full geospatial engines.

Incidents were given their own domain model instead of being absorbed into Assignment. This choice keeps student clinical performance records isolated and auditable. Incident stores the parties involved, the date of occurrence, the assignment context, and the reason code. Storing IncidentReason as a separate lookup table ensures consistent reporting. This prevents ad-hoc text descriptions from becoming data quality problems. The professor can review or escalate incidents, but the underlying structure remains consistent regardless of how many times a student is flagged.

The system enforces clear relational constraints to preserve data quality. AppUser is always the parent type in the hierarchy, so a Student or Professor cannot exist without the base identity. Assignments cannot be created unless the referenced Student, Professor, and Location exist. This prevents "floating assignments" or orphaned records. Incident entries are also constrained to an Assignment context. A student cannot receive an incident report without an active or historical assignment linked to it. These rules force accountability into the schema and stop the UI layer or API from injecting bad data. Every action in the system ties back to a real user and a real event.

Throughout the database, primary keys are explicit and never depend on composite keys or surrogate hybrid models. Enumber functions as a stable identifier for all users, aligning with the university context. Auto-incrementing integers were used for internal entities such as Location, Incident, and Assignment, which makes indexing straightforward and avoids collisions.

Guidance for Future Developers Working on This Database

Anyone extending this backend should approach development through the domain model first. Begin in the Entities folder and understand how inheritance is implemented. AppUser is the root class. Student and Professor do not exist independently in the database; their tables merge into one through EF's TPH mapping. All new user-type entities should follow this pattern. Modifying Student or Professor without understanding their AppUser dependency will cause schema conflicts.

Before creating new controllers or endpoints, determine whether the data you want belongs to a relationship or an entity. The architecture is built to avoid "god tables" that accumulate unrelated fields. If new features revolve around events, tracking, or performance metrics, they should be represented as separate domain models. This keeps the database normalized and supports clear data flow from the API.

When implementing new features, use DTOs to control exposure. Entities should never be returned directly. The existing codebase separates read and write DTOs, which prevents accidental exposure of hashed credentials, internal IDs, or fields that should remain immutable. Follow that same pattern: create a CreateDto for requests, UpdateDto for mutable fields, and ReadDto to shape output.

Future migrations should be deliberate. This system is not designed for frequent schema churn. If a developer makes an entity change, they should review any ripple effects on controllers, seeding, and DTOs before writing migrations. This prevents breaking the endpoints or corrupting seeded baseline data.

Seeding is kept intentionally minimal. The goal was not to produce large fake datasets, but to ensure every core relationship can be tested immediately. A baseline Professor and Student are inserted, a centralized Location is created, and a small number of Assignments are added to

validate controller endpoints without manual data entry. These seeds roughly simulate the lifecycle the system is built around: a student belongs to the user pool, receives an assignment from a professor, is placed at a location, and may trigger an incident. Developers who add additional test data should mirror this pattern, never seed data that violates the same relational constraints enforced by the schema.

Conclusion

In summary, the database design is intentionally tight: one source of truth for users, isolated context tables, and consistent relationships. Maintaining these principles will preserve performance, prevent duplication, and keep the project extendable.