Team Members:

- Gurleen Bassali (101260100)
- Karan Modi (101193808)

Application Functionality:

- **Functions implemented (TOTAL 10): (Constraints in RED)**

1. *Member*:
    a) ***User Registration*:** Creates a new member with unique email, password and profile info. Validates email uniqueness before insertion.
    (email must be unique; password required; email format validated)

    b) Profile Management: Updates member details. And allows adding/modifying fitness goals.
    (member must exist; email uniqueness checked on update)

    c) Health history: Records timestamp heals metrics without overwriting previous entries. Supports tracking.
    (**member must exist; metricType and value required; append-only)**

    d) Dashboard: Displays latest health metrics, active fitness goals, upcoming PT sessions, and registered classes in a single view.
    (member must exist; only show future classes/sessions)

    e) PT Session Scheduling: Books personal training sessions after validating trainer availability, room availability, and member schedule conflicts.
    (**trainer must be available; room must be free; member cannot have overlapping bookings; session time must be in future)**

    f) Group Class Registration: Registers members for fitness classes after checking class capacity and schedule conflicts. Prevents duplicate registrations.
    (class capacity not exceeded; no duplicate registrations; no schedule conflicts with existing session/classes)

2. *Trainer:*
    a) ***Set Availability:*** Defines available time slots as either one-time intervals or recurring weekly schedules. Prevents duplicate registrations.
    (no overlapping availability slots; start time must be before end time)

b) ***Schedule View:*** Displays combined view of assigned PT sessions and fitness classes, sorted by date/time.
(**trainer must exist; only shows future appointments)**

3. *Admin:*
   a) ***Room Booking:*** Manages room assignments for sessions and classes. Validates against double-booking conflicts.
   (room cannot be double-booked; room must exist; time slot validated)

   b) ***Class Management:*** Creates, updates and cancels fitness classes. Assign trainers and rooms with correct validation
   (trainer must exist and be available; room must exist and be free; capacity must be positive; start time before end time)

- Tech stack used:
  a) ***Frontend***: React + Typescript
  b) ***Backend:*** Node.js + Express.
  c) ***Database:*** PostgreSQL
  d) ***ORM:*** Prisma

Schema Quality and Normalization:
We evaluated our final relational schema (derived from our Prisma models) for normalization. All tables satisfy the First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF).

1. First Normal Form (1NF)

All tables in our schema satisfy 1NF because:

- Every attribute stores atomic values (no multi-valued fields, comma-separated lists, or repeating groups).
- Each table has a defined primary key.
- There are no repeating column groups (e.g., no metric1, metric2, etc.).
- Historical data such as health metrics and goals are stored as separate rows, not overwritten.

2. Second Normal Form (2NF)

A schema is in 2NF if:

- It is already in 1NF.
- It has no partial dependencies. This means it has no non-key attribute depending on part of a composite key.

In our schema, all tables use a single-column primary key (id), not composite keys.
Therefore, partial dependencies are impossible.

3. Third Normal Form (3NF)

A schema is in 3NF if:

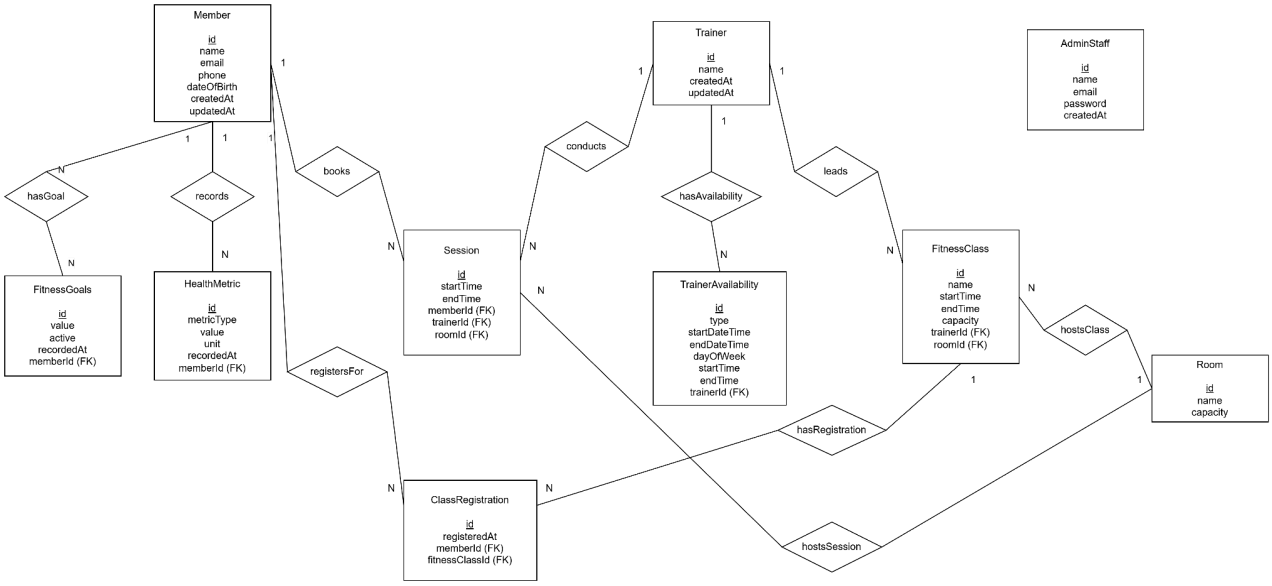
- It is in 2NF.
- It has no transitive dependencies. This means non-key attributes must not depend on other non-key attributes.

In our schema, we reviewed each entity to ensure that no non-key determines another non-key attribute.
Entities:

- Member
    - All attributes depend solely on id.
    - No attribute (e.g., email) determines another non-key attribute.
    - We explicitly removed a previously redundant attribute (pastClassCount), which was a derived value and would have violated 3NF. Its value is now computed dynamically in the dashboard service.

- FitnessClass
    - This entity has foreign keys: trainerId and roomId.
    - These are foreign keys only. No non-key attributes depend on them.
    - All trainer/room attributes are stored in their respective tables, not duplicated.
- TrainerAvailability
    - The attribute nullability of startDateTime, endDateTime, dayOfWeek, startTime, and endTime depends on the availability type. But this is a design rule, not a functional dependency.
    - No non-key attribute determines another.
- Session, FitnessGoals, Room, Trainer, AdminStaff, HealthMetric
    - All follow the same pattern.
    - Every non-key attribute depends directly on the primary key.
    - There is no derived, duplicated, or indirectly-determined data stored within the table.

Conceptual Database Design (ER Model):



Notes:

- The TrainerAvailability entity supports two different kinds of availability:
    - One-time availability for a specific date/time range (e.g., “Nov 30, 3–5pm”)
    - Recurring weekly availability (e.g., “Every Monday, 3–5pm”)
    - To model both in a single table, we include:
        - type: enum with values ONE_TIME or WEEKLY
        - For ONE_TIME: we use startDateTime and endDateTime
        - For WEEKLY: we use dayOfWeek, startTime, and endTime
    - Only the fields relevant to the chosen type are used. This design lets us express both one-off and repeating availability without creating separate tables, while still keeping the model normalized.
- ClassRegistration exists to implement the many-to-many relationship between Member and FitnessClass. A member can register for many classes, and a class can have many members, so we use ClassRegistration(memberId, fitnessClassId) instead of putting a member list inside FitnessClass.
- Both HealthMetric and FitnessGoals include recordedAt timestamps. This lets us keep a history of metrics and goals instead of overwriting values, which is required for “health history” and for showing the “latest” goal or metrics on the dashboard.

ER to Relational Schema Mapping:

ER Construct	Assumptions	Mapping Rule	Resulting Relational Schema
Member (Strong Entity)	Email must be unique.	Strong entity maps directly to a table with all attributes.	**Member** (id INT PK AUTO_INCREMENT, **name** VARCHAR NOT NULL, **email** VARCHAR UNIQUE NOT NULL, **phone** VARCHAR NULL, **password** VARCHAR NOT NULL, **dateOfBirth** DATETIME NULL, **createdAt** DATETIME, **updatedAt** DATETIME)
Trainer (Strong Entity with attributes: name)	Trainers don't have login credentials. (managed by admin)	Strong entity maps directly to a table with surrogate primary key.	**Trainer**(id **INT PK** AUTO_INCREMENT, **name** VARCHAR NOT NULL, **createdAt** DATETIME, **updatedAt** DATETIME)
AdminStaff (Strong Entity with attributes: name, email, password)	Admins have a separate login flow compared to members.	Strong entity maps directly to a table. Email is a candidate key.	**AdminStaff** (id INT PK AUTO_INCREMENT, **name** VARCHAR NOT NULL, **email** VARCHAR UNIQUE NOT NULL, **password** VARCHAR NOT NULL, **createdAt** DATETIME)
Room (Strong Entity with attributes: name, capacity)	Room capacity is fixed, no equipment tracking.	Strong entity maps directly to a table with surrogate primary key.	**Room** (id INT PK AUTO_INCREMENT, **name** VARCHAR NOT NULL, **capacity** INT NOT NULL)
HealthMetric (Weak Entity of Member with attributes: recordedAt, metricType, value, unit)	Metrics are append-only, never overwritten for history tracking.	Weak entity maps to a table with partial key + FK to identifying owner. ON DELETE CASCADE maintains referential integrity.	**HealthMetric** (id INT PK, **memberId** INT FK→Memberd(id) ON DELETE CASCADE, **recordedAt** DATETIME, **metricType** VARCHAR NOT NULL, **value** FLOAT NOT NULL, **unit** VARCHAR NULL)
FitnessGoals (Weak Entity of Member with attributes: recordedAt, active, value)	Multiple goals allowed; acti ve flag indicates current goal	Weak entity maps to a table with partial key + FK to identifying owner.	**FitnessGoals**(id INT PK, **memberId** INT FK→Member(id) ON DELETE CASCADE, **recordedAt** DATETIME, **active** BOOLEAN NOT NULL, **value** VARCHAR NOT NULL)
TrainerAvailability (We ak Entity of Trainer with attributes: type, startDateTime, endDateTime, dayOfWeek, startTime, endTime)	Can be one-time or weekly	Weak entity maps to a table with partial key + FK to identifying owner. Type is an ENUM for availability variants.	**TrainerAvailability**(id INT PK, **trainerI** d INT FK→Trainer(id) ON DELETE CASCADE, **type** ENUM('ONE_TIME','WEEKLY') NO T NULL, **startDateTime** DATETIME NULL, **endDateTime** DATETIME NULL, **dayOfWeek** INT NULL, **startTime** TIME NULL, **endTime** TIME NULL)
"**Attends**" Relationship (Member a ttends Session with Trainer in Room) — Ternary **1**:N:N relationsh ip	-	Ternary relationship with attributes (startTi me, endTime) maps to a separate relationship table with FKs to all participating entit ies.	**Session**(id INT PK, **memberId** INT FK→Member(id) ON DELETE CASCADE, **trainerId** INT FK→Trainer(id) ON DELETE CASCADE, **roomId** INT FK→Room(id) NULL, **startTime** DATETIME NOT NULL, **endTime** DATETIME NOT NULL)

FitnessClass (Entity participating in "Teaches" with Trainer and "Held In" with Room) — Two **1**:N relationships merged	A class must have a trainer assigned.	Entity with two N:**1** relationship. FKs placed on the "many" sides (FitnessClass). Has own attributes (name, capacity, times).	**FitnessClass**(id INT PK, **name** VARCHAR NOT NULL, **trainerId** INT FK→Trainer(id) ON DELETE CASCADE, **roomId** INT FK→Room(id) NOT NULL, **startTime** DATETIME NOT NULL, **endTime** DATETIME NOT NULL, **capacity** INT NOT NULL)
"Registers For" Relationship (Member ↔ FitnessClass) — M:N relationship with attribute registeredAt	No duplicate registrations and members can cancel registrations.	M:N relationship maps to a junction/associative table with FKs to both entities. Composite unique constraint prevents duplicate registrations.	**ClassRegistration**(id INT PK, **memberId** INT FK→Member(id) ON DELETE CASCADE, **fitnessClassId** INT FK→FitnessClass(id) ON DELETE CASCADE, **registeredAt** DATETIME, UNIQUE(memberId, fitnessClassId))

ORM Integration:

Our project uses Prisma ORM as the primary abstraction layer for all database interactions within our Health & Fitness Club Management System. Instead of manually writing SQL statements or constructing queries by hand, Prisma allows us to define our entire relational schema in a single declarative schema.prisma file. This schema includes entity definitions, relationships, constraints, and indexes.
From this schema, Prisma automatically generates database access code in the /generated directory (commonly called the Prisma Client). This generated client contains TypeScript classes, enums, model typings, and CRUD helpers for every entity in our schema, enabling fully type-safe queries such as prisma.member.findMany() or prisma.session.create(). The generated client is rebuilt whenever the schema changes, ensuring that our TypeScript code and our database schema never get out of sync.
Prisma also manages the creation and evolution of the database through its migration system. Every modification to the schema (like adding a model, updating a relationship, or creating an index) is captured in a migration file inside the prisma/migrations/ folder. Each migration contains the SQL needed to reflect that exact schema change in PostgreSQL. When running the command "npx prisma migrate dev", Prisma applies the migration, updates the database structure, and regenerates the TypeScript Prisma Client. This ensures that the ORM, the database schema, and the application's backend logic are always aligned.
So throughout the application, all CRUD operations like registering members, scheduling sessions, checking trainer availability, and managing classes, are performed using the Prisma Client, preventing common SQL errors. This ORM-based workflow allowed us to write cleaner service-layer logic while maintaining referential integrity and adhering to 3NF normalization.

Below is an example entity from our Prisma schema, showing how tables and relationships are mapped. Prisma handles the underlying foreign keys, cascades, and indexing automatically:

```
model FitnessGoals {
  id         Int      @id @default(autoincrement())
  memberId   Int
  recordedAt DateTime @default(now())
  active     Boolean  //indicates
  value      String   //thats the actual goal

  member     Member   @relation(fields: [memberId], references: [id], onDelete: Cascade)

  @@index([memberId, recordedAt])
}
```

Here is an example function that uses the Prisma Client to query entities without any raw SQL. This function retrieves all goals for a member:

```
export async function getFitnessGoalsForMember(memberId: number): Promise<FitnessGoals[]> {
  return prisma.fitnessGoals.findMany({
    where: { memberId },
    orderBy: { recordedAt: "desc" },
  });
}
```

Mapped Entities:
All tables in our database are represented as ORM-mapped models, including:

1. Member
2. Trainer
3. AdminStaff
4. FitnessClass
5. Session
6. TrainerAvailability
7. Room
8. ClassRegistration
9. FitnessGoals
10. HealthMetric

Each entity includes the appropriate relations, primary keys, unique constraints, and indexing for efficient lookup.

Views, Triggers, and Indexes:

- Index: We explicitly define indexes in schema.prisma (for example @@index([memberId, recordedAt]) in FitnessGoals and HealthMetric, @@index([trainerId, startTime, endTime]) in TrainerAvailability, and @@unique([memberId, fitnessClassId]) plus @@index([fitnessClassId]) in ClassRegistration). Prisma automatically creates these as real PostgreSQL indexes/constraints. This is how our app implements indexes handled via the ORM.
- View (implemented via ORM): Instead of a physical SQL view, we implement view-like aggregations of data from multiple tables. An example of this is in dashboardService.ts. Functions such as getActiveGoalsForMember, getPastClassCountForMember, and getUpcomingClassSessionsForMember join and filter data from multiple tables using Prisma queries and return a virtual “dashboard” for a member. This is conceptually equivalent to defining a named SQL view (ex. MemberDashboard) but kept entirely in the ORM layer.
- Trigger (implemented via ORM): Constraints that would typically be enforced with database triggers such as preventing overlapping trainer bookings or avoiding double registrations, are implemented as reusable checks in our service layer before calling prisma create or update. For example, our session scheduling logic calls a helper like trainerHasConflict(trainerId, startTime, endTime, options) to detect conflicts with

existing sessions and classes; if a conflict exists, it throws an error instead of inserting the row. Functionally, this acts as a BEFORE INSERT/UPDATE trigger, but is implemented in TypeScript utilizing Prisma rather than in raw SQL.