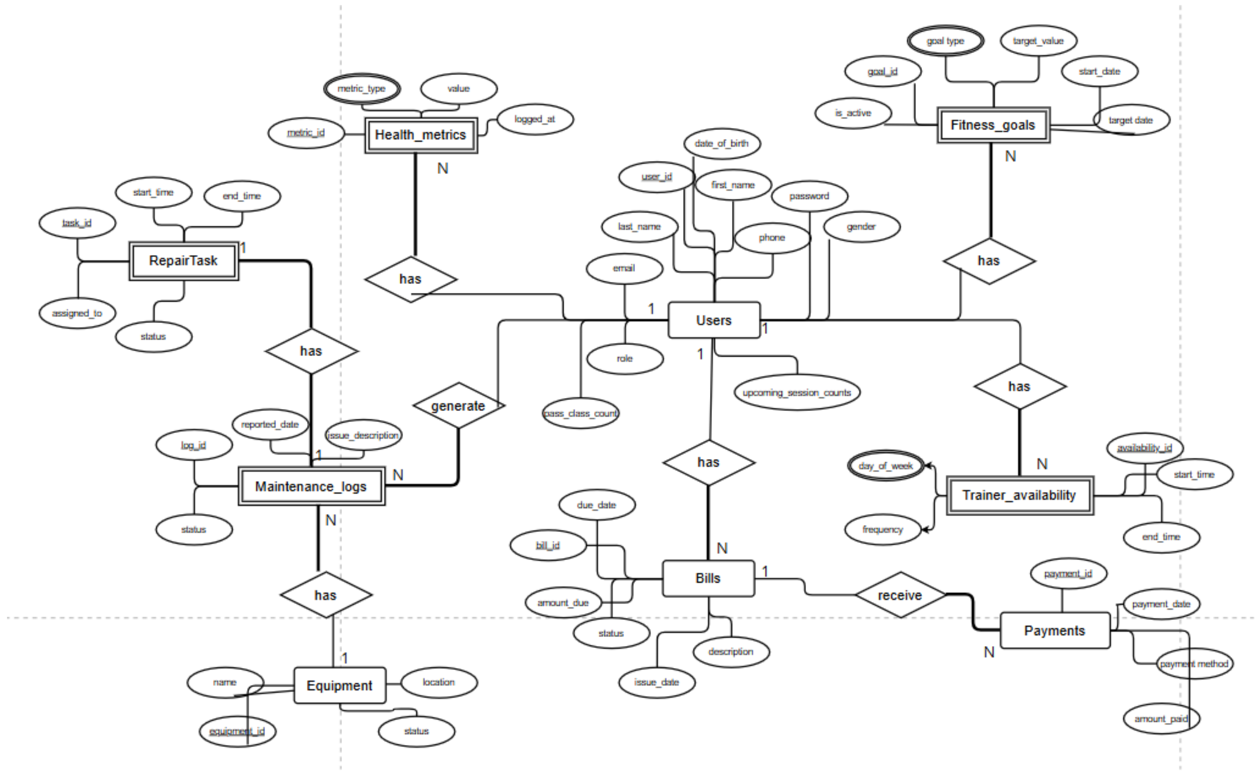
## 1. ER Model



## 2. Assumption

- **Single User Table Approach:** All system users (members, trainers, admins) are stored in a single users table with role-based differentiation, simplifying authentication and maintaining data consistency

- **VIEW + TRIGGER + INDEX:** Those operations can be done in raw SQL (in PostgreSQL) even though I used ORM to handle my database

- **Flexible Billing:** Support for partial payments and multiple payment methods with automated status updates

- **Maintenance Workflow:** Equipment issues follow a sequential workflow from reporting → repair assignment → completion with status tracking

## 3. ER Mapping (ORM)

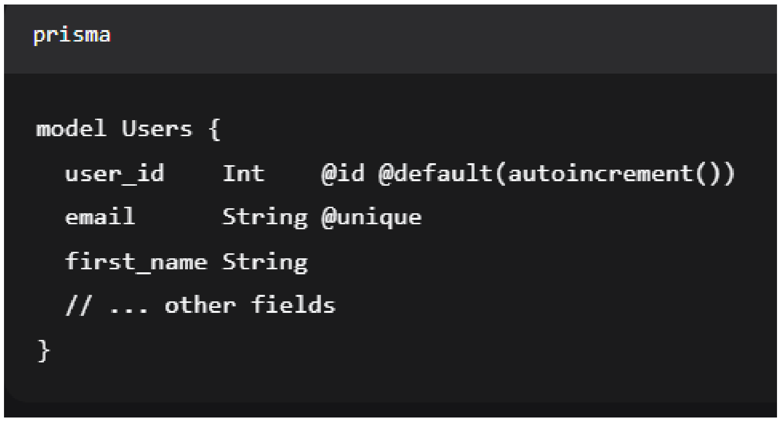
The transition from the Entity-Relationship (ER) model to a working relational database is handled by the Prisma ORM. Prisma uses its own schema definition language (Prisma Schema) to define the application's data model, which it then uses to generate both the SQL migration files for the database and a type-safe client for executing queries.

### Mapping Entities and Relationships

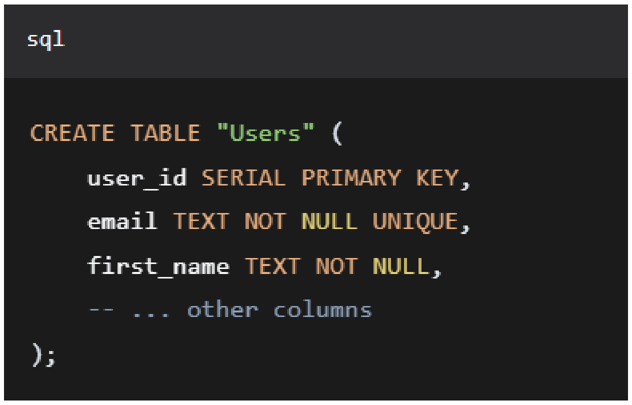
In Prisma, the model blocks in the schema.prisma file are the direct equivalent of entity classes in traditional ORMs like Hibernate. Each model maps to a table in the PostgreSQL database.

### 1. Entity Mapping (Models as Tables):

Each model (e.g., Users, FitnessGoal, Bill) is translated by Prisma into a corresponding SQL CREATE TABLE statement.



```prisma
model Users {
  user_id    Int    @id @default(autoincrement())
  email      String @unique
  first_name String
  // ... other fields
}
```

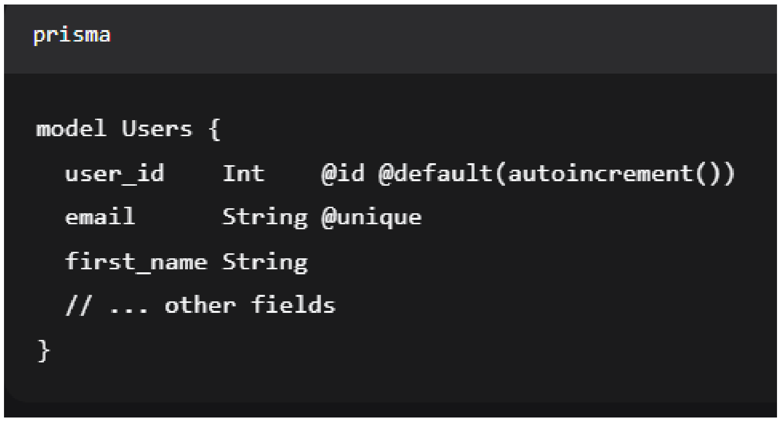


```sql
CREATE TABLE "Users" (
    user_id SERIAL PRIMARY KEY,
    email TEXT NOT NULL UNIQUE,
    first_name TEXT NOT NULL,
    -- ... other columns
);
```

## 2. Attribute Mapping (Fields as Columns):

The fields within each Prisma model define the columns' names, data types, and constraints.

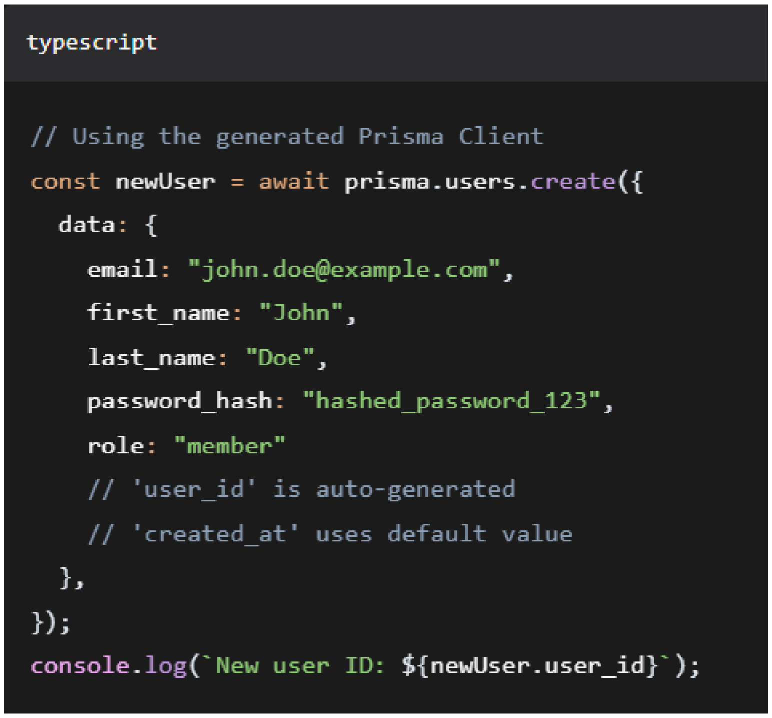
- Int, String, DateTime, Boolean, Float map to their respective SQL types.
- Constraints are added via attributes:
- @id and @default(autoincrement()) create a SERIAL PRIMARY KEY.
- @unique creates a UNIQUE constraint.



prisma

```prisma
model Users {
  user_id    Int    @id @default(autoincrement())
  email      String @unique
  first_name String
  // ... other fields
}
```

Insert instance in Users Table



typescript

```typescript
// Using the generated Prisma Client
const newUser = await prisma.users.create({
  data: {
    email: "john.doe@example.com",
    first_name: "John",
    last_name: "Doe",
    password_hash: "hashed_password_123",
    role: "member"
    // 'user_id' is auto-generated
    // 'created_at' uses default value
  },
});
console.log(`New user ID: ${newUser.user_id}`);
```

## 3. Relationship Mapping:

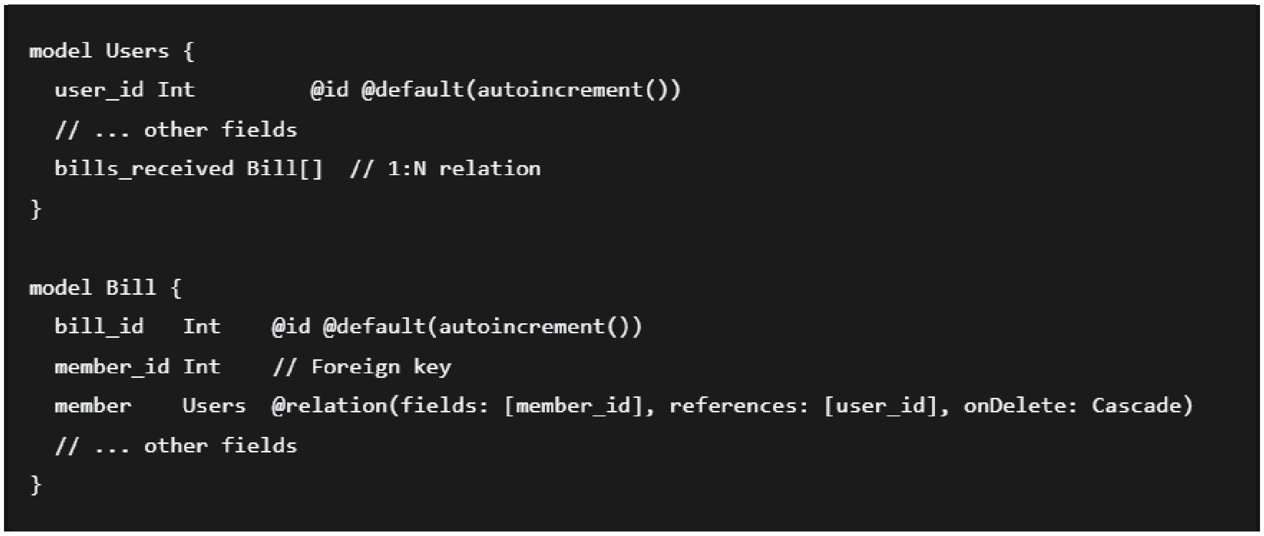
Prisma handles relationships using fields and the @relation attribute.

- **One-to-Many (1:N): This is the most common relationship in my schema.**

Example: Users has many Bills.

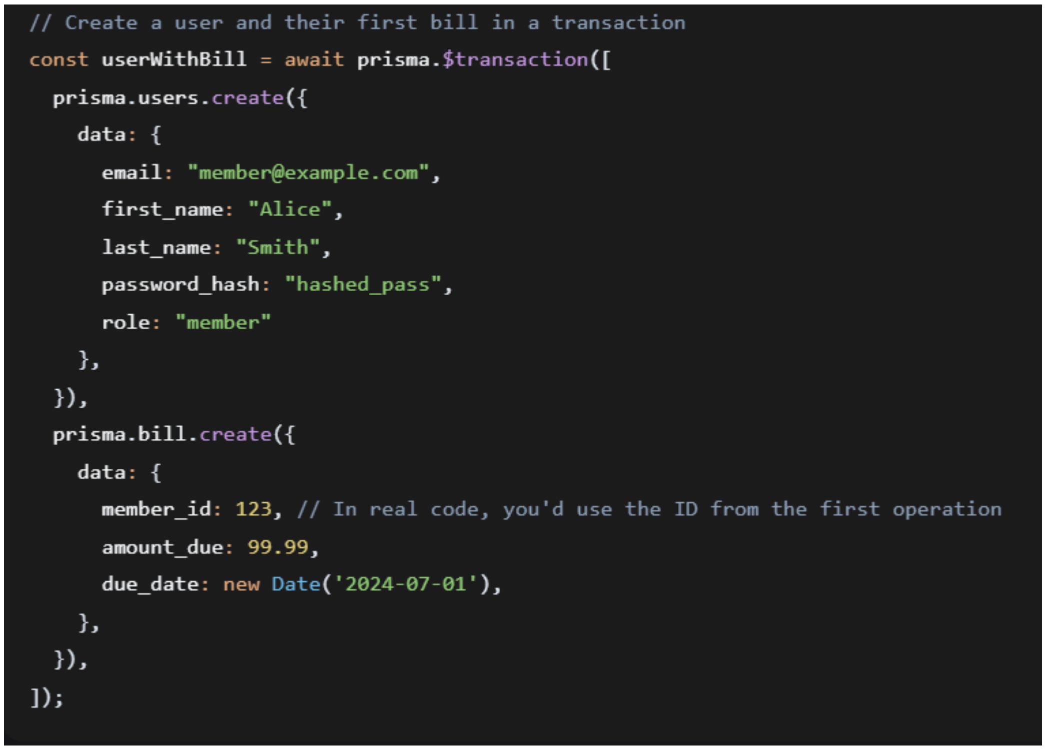
**Prisma Implementation:**

- On the "one" side (Users), we have a list: bills_received Bill[].
- On the "many" side (Bill), we have a scalar foreign key: member_id Int and a relation field: member Users @relation(fields: [member_id], references: [user_id], ...).



```
model Users {
  user_id Int         @id @default(autoincrement())
  // ... other fields
  bills_received Bill[]  // 1:N relation
}

model Bill {
  bill_id   Int    @id @default(autoincrement())
  member_id Int    // Foreign key
  member    Users  @relation(fields: [member_id], references: [user_id], onDelete: Cascade)
  // ... other fields
}
```



```
// Create a user and their first bill in a transaction
const userWithBill = await prisma.$transaction([
  prisma.users.create({
    data: {
      email: "member@example.com",
      first_name: "Alice",
      last_name: "Smith",
      password_hash: "hashed_pass",
      role: "member"
    },
  }),
  prisma.bill.create({
    data: {
      member_id: 123, // In real code, you'd use the ID from the first operation
      amount_due: 99.99,
      due_date: new Date('2024-07-01'),
    },
  }),
]);
```

- **One-to-One (1:1): Used for exclusive relationships.**

Example: A MaintenanceLog can have one RepairTask.

**Prisma Implementation:**

- The parent (MaintenanceLog) has an optional field: repair_task RepairTask?.
- The child (RepairTask) has a unique scalar foreign key: log_id Int @unique and the relation field. The @unique constraint enforces the one-to-one cardinality at the database level.

```
model MaintenanceLog {
  log_id      Int          @id @default(autoincrement())
  // ... other fields
  repair_task RepairTask?  // 1:1 relation
}

model RepairTask {
  task_id Int    @id @default(autoincrement())
  log_id  Int    @unique  // Enforces 1:1
  // ... other fields
  maintenance_log MaintenanceLog @relation(fields: [log_id], references: [log_id])
}
```

```
// Create a maintenance log with its repair task
const logWithTask = await prisma.maintenanceLog.create({
  data: {
    equipment_id: 1,
    reported_by: 456,
    issue_description: "Treadmill belt slipping",
    repair_task: {
      create: {
        assigned_to: "Tech Team A",
        status: "pending"
      }
    }
  },
  include: {
    repair_task: true  // Eager load the related task
  }
});
console.log(`Repair task ID: ${logWithTask.repair_task?.task_id}`);
```

## 4. Cascading Actions:

The onDelete: Cascade ensures referential integrity by automatically deleting related records.

```
// When we delete a user, all their fitness goals are automatically deleted
await prisma.users.delete({
  where: { user_id: 123 },
});

// Verify the goals are gone
const remainingGoals = await prisma.fitnessGoal.findMany({
  where: { user_id: 123 }
});
console.log(remainingGoals); // Output: []
```