ORM Implementation

This NODEJS Fitness Gym DB System incorporates the ORM-Layer in the standard conventional norm. PRISMA was selected due to it's wide spread use among the NODEJS ecosystem. There are no application-level object classes, but a schema file located in the PRISMA folder which acts as Wrapper over native SQL. PRISMA introduces higher-level abstractions like Index, View,. As a pleasant add-on, it comes bootstrapped with a native localhost GUI for streamlined database debugging.

As I am working on the project in the real-time, let me share my own frustrations with the code. First, PRISMA does not support native VIEW. A view is a shortcut that aggregates multiple tables together, to return a streamlined result to the application code. Because PRISMA lacked this feature I had to write raw SQL boiler-plate code to support this functionality. The PRISMA still runs the SQL code (It's essentially a SQL transpiler), but I found the experience frustrating nevertheless.

Here is the code

```
-- This is an empty mi-- Drop if re-running in dev
DROP VIEW IF EXISTS member_dashboard_view;

CREATE VIEW member_dashboard_view AS
SELECT
   row_number() OVER ()::int            AS id,
   m.id                                 AS "memberId",
   m."fullName"                         AS "memberName",
   hm."metricType"                      AS "metricType",
   hm.value                             AS "metricValue",
   hm."recordedAt"                      AS "metricRecordedAt",
   fg."goalText"                        AS "goalText",
   fg."createdAt"                       AS "goalCreatedAt"
FROM members m
LEFT JOIN "HealthMetric" hm ON hm."memberId" = m.id
LEFT JOIN "FitnessGoal" fg ON fg."memberId" = m.id;
```

Inline-schema INDEX works normally, thank goodness. This just means that some tables have a hash map that allows for blazing fast query retrieval, rather than linearly searching through the table for a hit. It's obviously going to be expensive memory wise, think of production level millions of records, hence why it's just an option. I believe I used it on the member table and spammed it like crazy in my application code.

As for lazy/eager loading, It only supports eager-loading by default. I didn't quite understand why that was the design choice, but I researched and discovered lazy-loading to be a bad-practice for it's reputation to leak performance if not careful. The syntax for lazy loaded ORM looks indistinguishable from regular code, Juniors Devs are notorious for forgetting this fact, and often invoke a database query in a loop. This causes the DB to stall and performance to bottle neck. So I decided to not go against the well and remained with the default settings of

Eager loading.

As for the Prisma.Schema, all entities are mapped to it. Mind you, I didn’t create application-entity level classes so you may dock some marks there.