MENU

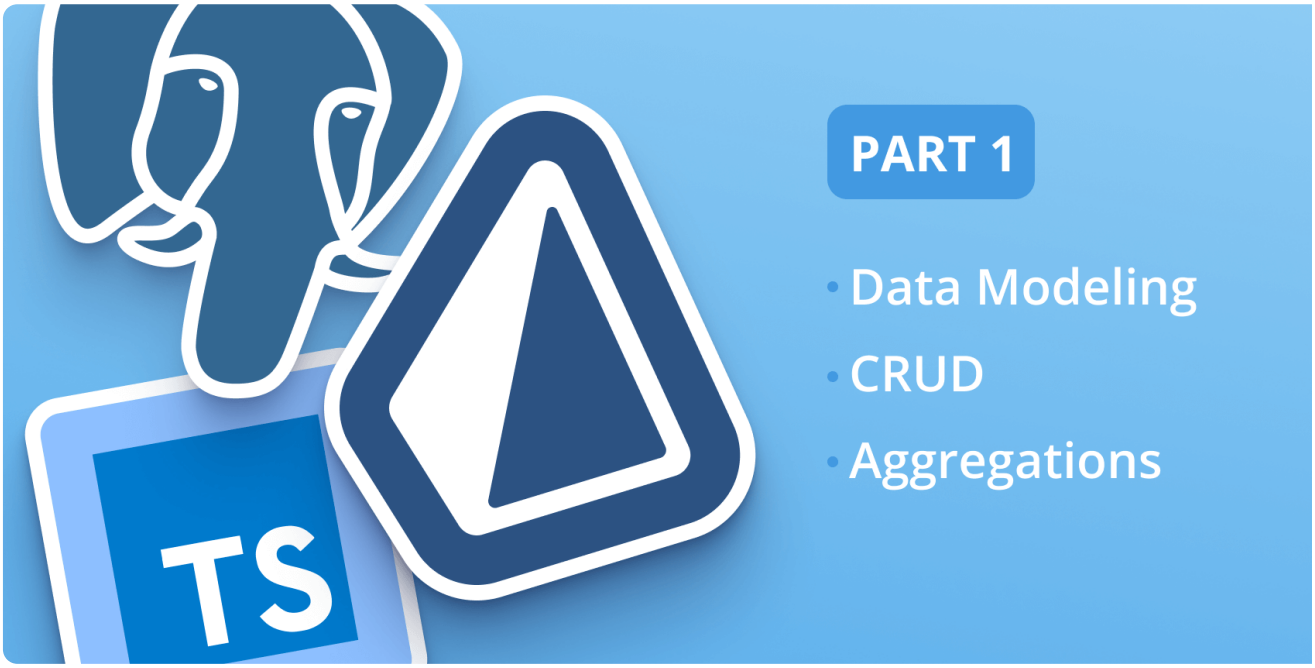
July 30, 2020

Modern Backend with TypeScript, PostgreSQL, and Prisma: Data Modeling, CRUD, and Aggregates

SERIES

Daniel Norman
@daniel2color

This article is part of a series of live streams and articles on building a backend with TypeScript, PostgreSQL, and Prisma. In this article, which summarizes the first live stream, we'll look at how to design the data model, perform CRUD operations, and query aggregates using Prisma.



PART 1 (Currently reading)

Modern Backend with TypeScript, PostgreSQL, and Prisma: Data Modeling, CRUD, and Aggregates

PART 2

Modern Backend with TypeScript, PostgreSQL, and Prisma: REST API, Validation, and Testing

PART 3

Modern Backend with TypeScript, PostgreSQL, and Prisma: Passwordless Authentication and Authorization

PART 4

Modern Backend with TypeScript, PostgreSQL, and Prisma: Continuous Integration and Continuous Deployment

Introduction

The goal of the series is to explore and demonstrate different patterns, problems, and architectures for a modern backend by solving a concrete problem: **a grading system for online courses.** This is a good example because it features diverse relations types and is complex enough to represent a real-world use-case.

The recording of the live stream is available above and covers the same ground as this article.

What the series will cover

The series will focus on the role of the database in every aspect of backend development covering:

- Data modeling
- CRUD
- Aggregations
- API layer
- Validation
- Testing

Authentication

Authorization

Integration with external APIs

Deployment

What you will learn today

This first article of the series will begin by laying out the problem domain and developing the following aspects of the backend:

Data modeling: Mapping the problem domain to a database schema

CRUD: Implement Create, Read, Update, and Delete queries with Prisma Clientagainst the database

Aggregation: Implement aggregate queries with Prisma to calculate averages, etc.

By the end of this article you will have a Prisma schema, a corresponding database schema created by Prisma Migrate, and a seed script which uses Prisma Client to perform CRUD and aggregation queries.

The next parts of this series will cover the other aspects from the list in detail.

> ***Note:*** *Throughout the guide you'll find various* ***checkpoints*** *that enable you to validate whether you performed the steps correctly.*

Prerequisites

Assumed knowledge

This series assumes basic knowledge of TypeScript, Node.js, and relational databases. If you're experienced with JavaScript but haven't had the chance to try TypeScript, you should still be able to follow along. The series will use PostgreSQL, however, most of the concepts apply to other relational databases such as MySQL. Beyond that, no prior knowledge of Prisma is required as that will be covered in the series.

Development environment

You should have the following installed:

Node.js

Docker (will be used to run a development PostgreSQL database)

If you're using Visual Studio Code, the Prisma extension is recommended for syntax highlighting, formatting, and other helpers.

> ***Note**: If you don't want to use Docker, you can set up a local PostgreSQL database or a hosted PostgreSQL database on Heroku.*

Clone the repository

The source code for the series can be found on GitHub.

To get started, clone the repository and install the dependencies:

```
git clone -b part-1 git@github.com:2color/real-world-grading-app.git
cd real-world-grading-app
npm install
```

> ***Note:** By checking out the `part-1` branch you'll be able to follow the article from the same starting point.*

Start PostgreSQL

To start PostgreSQL, run the following command from the `real-world-grading-app` folder:

```
docker-compose up -d
```

> ***Note:** Docker will use the `docker-compose.yml` file to start the PostgreSQL container.*

Data model for a grading system for online courses

Defining the problem domain and entities

When building a backend, one of the foremost concerns is a proper understanding of the *problem domain*. The problem domain (or problem space) is a term referring to all information that defines the problem and constrains the solution (the constraints being part of the problem). By understanding the problem domain, the shape and structure of the data model should become clear.

The online grading system will have the following entities:

User: A person with an account. A user can be either a teacher or a student through their relation to a course. In other words, the same user who's a teacher of one course can be a student in another course.

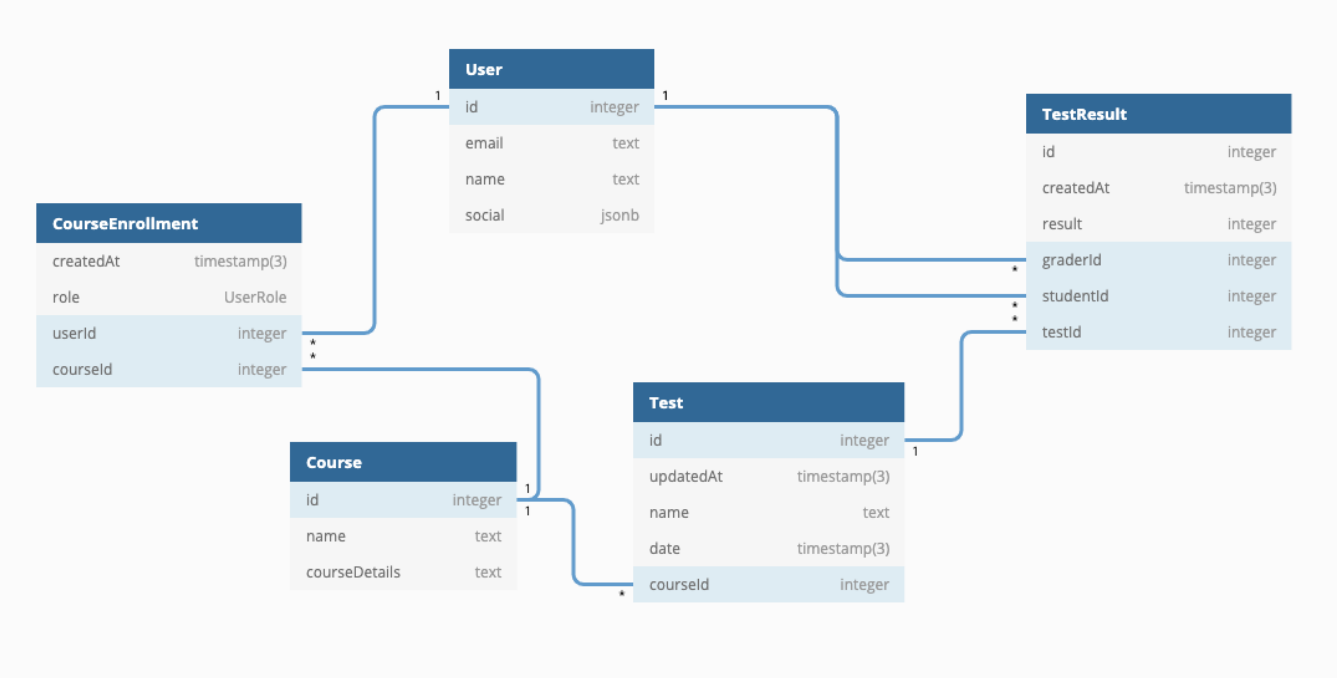
Course: A learning course with one or more teachers and students as well as one or more tests. For example: an "Introduction to TypeScript" course can have two teachers and ten students.

Test: A course can have many tests to evaluate the students' comprehension. Tests have a date and are related to a course.

Test result: Each test can have multiple test result records per student. Additionally, a TestResult is also related to the teacher who graded the test.

> ***Note**: An entity represents either a physical object or an intangible concept. For example, a **user** represents a person, whereas a **course** is an intangible concept.*

The entities can be visualized to demonstrate how they would be represented in a relational database (in this case PostgreSQL). The diagram below adds the columns relevant for each entity and foreign keys to describe the relationships between the entities.



The first thing to note about the diagram is that every entity maps to a database table.

The diagram has the following relations:

one-to-many (also known as `1-n`):

```
Test <> TestResult

Course <> Test
```

`User <> TestResult (via graderId )`

`User <> TestResult (via student )`

many-to-many (also known as `m-n`):

`User <> Course` (via the `CourseEnrollment` relation table with two *foreign keys*: `userId` and `courseId`). Many-to-many relations typically require an additional table. This is necessary so that the grading system can have the following properties:

A single course can have many associated users (as students or teachers)

A single user can be associated with many courses.

> ***Note**: A relation table (also known as a JOIN table) connects two or more other tables to create a relation between them. Creating relation tables is a common data modeling practice in SQL to represent relationships between different entities. In essence, it means that "one m-n relation is modeled as two 1-n relations in the database".*

Understanding the Prisma schema

To create the tables in your database, you first need to define your Prisma schema. The Prisma schema is a declarative configuration for your database tables which will be used by Prisma Migrate to create the tables in your database. Similar to the entity diagram above, it defines the columns and relations between the database tables.

The Prisma schema is used as the source of truth for the generated Prisma Client and Prisma Migrate to create the database schema.

The Prisma schema for the project can be found in `prisma/schema.prisma` . In the schema you will find stub models which you will define in this step and a `datasource` block. The `datasource` block defines the kind of database that you'll connect to and the connection string. With `env("DATABASE_URL")` , Prisma will load the database connection URL from an environment variable.

> ***Note:** It's considered best practice to keep secrets out of your codebase. For this reason the `env("DATABASE_URL")` is defined in the datasource block. By setting an environment variable you keep secrets out of the codebase.*

Define models

The fundamental building block of the Prisma schema is `model` . Every model maps to a database table.

Here is an example showing the basic signature of a model:

```
model User {
  id        Int    @default(autoincrement()) @id
  email     String @unique
  firstName String
  lastName  String
  social    Json?
}
```

Here you define a `User` model with several fields. Each field has a name followed by a type and optional field attributes. For example, the `id` field could be broken down as follows:

NAME	TYPE	SCALAR VS RELATION	TYPE MODIFIER	ATTRIBUTES
`id`	`Int`	Scalar	-	`@id` (denote the primary key) and `@default(autoincrement())` (set a default auto-increment value)
`email`	`String`	Scalar	-	`@unique`
`firstName`	`String`	Scalar	-	-
`lastName`	`String`	Scalar	-	-
`social`	`Json`	Scalar	`?` (optional)	-

Prisma defines a set of data types that map to the native database types depending on the database used.

The `Json` data type allows storing free form JSON. This is useful for information that can be inconsistent across `User` records and can change without affecting the core functionality of the backend. In the `User` model above it'd be used to store social links, e.g. Twitter, LinkedIn, etc. Adding new social profile links to the `social` requires no database migration.

With a good understanding of your problem domain and modeling data with Prisma, you can now add the following models to your `prisma/schema.prisma` file:

```
model User {
  id        Int    @default(autoincrement()) @id
  email     String @unique
```

```
  firstName String
  lastName  String
  social    Json?
}

model Course {
  id            Int     @default(autoincrement()) @id
  name          String
  courseDetails String?
}

model Test {
  id        Int      @default(autoincrement()) @id
  updatedAt DateTime @updatedAt
  name      String // Name of the test
  date      DateTime // Date of the test
}

model TestResult {
  id        Int      @default(autoincrement()) @id
  createdAt DateTime @default(now())
  result    Int // Percentage precise to one decimal point represented as `result * 10^-1`
}
```

Each model has all the relevant fields while ignoring relations (which will be defined in the next step).

Define relations

One-to-many

In this step you will define a *one-to-many* relation between `Test` and `TestResult`.

First, consider the `Test` and `TestResult` models defined in the previous step:

```
model Test {
  id        Int      @default(autoincrement()) @id
  updatedAt DateTime @updatedAt
  name      String
  date      DateTime
}

model TestResult {
  id        Int      @default(autoincrement()) @id
  createdAt DateTime @default(now())
```

```
  result    Int // Percentage precise to one decimal point represented result * 10^-1
}
```

To define a one-to-many relation between the two models, add the following three fields:

- `testId` field of type `Int` (*relation scalar*) on the "many" side of the relation: `TestResult`. This field represents the *foreign key* in the underlying database table.
- `test` field of type `Test` (*relation field*) with a `@relation` attribute mapping the relation scalar `testId` to the `id` primary key of the `Test` model.
- `testResults` field of type `TestResult[]` (*relation field*)

```
model Test {
  id        Int      @default(autoincrement()) @id
  updatedAt DateTime @updatedAt
  name      String
  date      DateTime

+ testResults TestResult[] // relation field
}

model TestResult {
  id        Int      @default(autoincrement()) @id
  createdAt DateTime @default(now())
  result    Int // Percentage precise to one decimal point represented result * 10^-1
+ testId    Int // relation scalar field
+ test      Test @relation(fields: [testId], references: [id]) // relation field
}
```

Relation fields like `test` and `testResults` can be identified by their value type pointing to another model, e.g. `Test` and `TestResult`. Their name will affect the way that relations are accessed programmatically with Prisma Client, however, they don't represent a real database column.

Many-to-many relations

In this step, you will define a *many-to-many* relation between the `User` and `Course` models.

Many-to-many relations can be *implicit* or *explicit* in the Prisma schema. In this part, you will learn the difference between the two and when to choose implicit or explicit.

First, consider the `User` and `Course` models defined in the previous step:

```
model User {
  id        Int    @default(autoincrement()) @id
  email     String @unique
  firstName String
  lastName  String
  social    Json?
}

model Course {
  id            Int     @default(autoincrement()) @id
  name          String
  courseDetails String?
}
```

To create an implicit many-to-many relation, define relation fields as lists on both sides of the relations:

```
model User {
  id        Int    @default(autoincrement()) @id
  email     String @unique
  firstName String
  lastName  String
  social    Json?
+ courses     Course[]
}

model Course {
  id            Int     @default(autoincrement()) @id
  name          String
  courseDetails String?
+ members User[]
}
```

With this, Prisma will create the relation table so the grading system can maintain the properties defined above:

> A single course can have many associated users.
>
> A single user can be associated with many courses.

However, one of the requirements of the grading system is to allow relating users to a course with a role as either a *teacher* or a *student*. This means we need a way to store "meta-information" about the relation in the database.

This can be achieved using an explicit many-to-many relation. The relation table connecting `User` and `Course` requires an extra field to indicate whether the user is a teacher or a student of a course. With explicit many-to-many relations, you can define extra fields on the relation table.

To do so, define a new model for the relation table named `CourseEnrollment` and update the `courses` field in the `User` model and the `members` field in the `Course` model to type `CourseEnrollment[]` as follows:

```
model User {
  id        Int    @default(autoincrement()) @id
  email     String @unique
  firstName String
  lastName  String
  social    Json?
+ courses     CourseEnrollment[]
}

model Course {
  id            Int     @default(autoincrement()) @id
  name          String
  courseDetails String?
+ members CourseEnrollment[]
}

+model CourseEnrollment {
+  createdAt DateTime @default(now())
+  role      UserRole
+
+  // Relation Fields
+  userId   Int
+  user     User   @relation(fields: [userId], references: [id])
+  courseId Int
+  course   Course @relation(fields: [courseId], references: [id])
+  @@id([userId, courseId])
+  @@index([userId, role])
+}
+
+enum UserRole {
+  STUDENT
+  TEACHER
+}
```

Things to note about the `CourseEnrollment` model:

It uses the `UserRole` enum to denote whether a user is a student or a teacher of a course.

The `@@id[userId, courseId]` defines a multi-field primary key of the two fields. This will ensure that every `User` can only be associated to a `Course` once, either as a student or as a teacher but not both.

To learn more about relations, check out the relation docs.

The full schema

Now that you've seen how relations are defined, update the Prisma schema with the following:

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        Int    @default(autoincrement()) @id
  email     String @unique
  firstName String
  lastName  String
  social    Json?

  // Relation fields
  courses     CourseEnrollment[]
  testResults TestResult[]       @relation(name: "results")
  testsGraded TestResult[]       @relation(name: "graded")
}

model Course {
  id            Int     @default(autoincrement()) @id
  name          String
  courseDetails String?

  // Relation fields
  members CourseEnrollment[]
  tests   Test[]
}

model CourseEnrollment {
  createdAt DateTime @default(now())
  role      UserRole

  // Relation Fields
```

```
  userId   Int
  courseId Int
  user     User   @relation(fields: [userId], references: [id])
  course   Course @relation(fields: [courseId], references: [id])

  @@id([userId, courseId])
  @@index([userId, role])
}

model Test {
  id        Int      @default(autoincrement()) @id
  updatedAt DateTime @updatedAt
  name      String
  date      DateTime

  // Relation Fields
  courseId   Int
  course     Course       @relation(fields: [courseId], references: [id])
  testResults TestResult[]
}

model TestResult {
  id        Int      @default(autoincrement()) @id
  createdAt DateTime @default(now())
  result    Int // Percentage precise to one decimal point represented as `result * 10^-1`

  // Relation Fields
  studentId Int
  student   User @relation(name: "results", fields: [studentId], references: [id])
  graderId  Int
  gradedBy  User @relation(name: "graded", fields: [graderId], references: [id])
  testId    Int
  test      Test @relation(fields: [testId], references: [id])
}

enum UserRole {
  STUDENT
  TEACHER
}
```

Note that `TestResult` has two relations to the `User` model: `student` and `gradedBy` to represent both the teacher who graded the test and the student who took the test. The `name` argument on

the `@relation` attribute is necessary to disambiguate the relationwhen a single model has more than one relation to the same model.

Migrating the database

With the Prisma schema defined, you will now use Prisma Migrate to create the actual tables in the database.

First, set the `DATABASE_URL` environment variable locally so that Prisma can connect to your database.

```
export DATABASE_URL="postgresql://prisma:prisma@127.0.0.1:5432/grading-app"
```

> ***Note**: The username and password for the local database are both defined as `prisma` in `docker-compose.yml` .*

To run a migration with Prisma Migrate, two steps are required:

Saving the migration: In this step, Prisma Migrate will take a snapshot of your schema and figure out the steps necessary to carry out the migration. The migration files will be saved to `prisma/migrations`

Running the migration: In this step, Prisma Migrate will use the migration files to run the migration and alter (or create) the database schema

> ***Note:** Prisma Migrate is currently in an experimental state. This means that it is not recommended to use Prisma Migrate in production. Instead, you can perform schema migrations using plain SQL or another migration tool of your choice and then bring the changes into your Prisma schema using introspection.*

Run the following commands in your terminal:

```
# Save the migration
npx prisma migrate save --experimental --name "init-db" --create-db

# Run the migration
npx prisma migrate up --experimental
```

Checkpoint: You should see the following in the output: 🚀 `Done with 1 migration in 263ms.`

Congratulations, you have successfully designed the data model and created the database schema. In the next step, you will use Prisma Client to perform CRUD and aggregation queries against your database.

Generating Prisma Client

Prisma Client is an auto-generated database client that's tailored to your database schema. It works by parsing the Prisma schema and generating a TypeScript client that you can import in your code.

Generating Prisma Client, typically requires three steps:

Add the following `generator` definition to your Prisma schema:

```
generator client {
  provider        = "prisma-client-js"
  previewFeatures = ["aggregateApi"]
}
```

Install the `@prisma/client` npm package

```
npm install --save @prisma/client
```

Generate Prisma Client with the following command:

```
npx prisma generate
```

Checkpoint: You should see the following in the output: `✓ Generated Prisma Client to ./node_modules/@prisma/client in 57ms`

Seeding the database

In this step, you will use Prisma Client to write a seed script to fill the database with some sample data.

A *seed script* in this context is a bunch of CRUD operations (create, read, update, and delete) with Prisma Client. You will also use nested writes to create database rows for related entities in a single operation.

Open the skeleton `src/seed.ts` file, where you will find the Prisma Client imported and two Prisma Client function calls: one to instantiate Prisma Client and the other to disconnect when the script finishes running.

Creating a user

Begin by creating a user as follows in `main` function:

```
const grace = await prisma.user.create({
  data: {
    email: 'grace@hey.com',
    firstName: 'Grace',
    lastName: 'Bell',
    social: {
      facebook: 'gracebell',
      twitter: 'therealgracebell',
    },
  },
})
```

The operation will create a row in the *User* table and return the created user (including the created `id`). It's worth noting that `user` will infer the type `User` which is defined in `@prisma/client`:

```
export type User = {
  id: number
  email: string
  firstName: string
  lastName: string
  social: JsonValue | null
}
```

To execute the seed script and create the `User` record, you can use the `seed` script in the `package.json` as follows:

```
npm run seed
```

As you follow the next steps, you will run the seed script more than once. To avoid hitting unique constraint errors, you can delete the contents of the database in the beginning of the `main` functions as follows:

```
await prisma.testResult.deleteMany({})
await prisma.courseEnrollment.deleteMany({})
await prisma.test.deleteMany({})
await prisma.user.deleteMany({})
await prisma.course.deleteMany({})
```

> ***Note:*** *These commands delete all rows in each database table. Use carefully and avoid this in production!*

Creating a course and related tests and users

In this step, you will create a *course* and use a nested write to create related *tests*.

Add the following to the `main` function:

```
const weekFromNow = add(new Date(), { days: 7 })
const twoWeekFromNow = add(new Date(), { days: 14 })
const monthFromNow = add(new Date(), { days: 28 })

const course = await prisma.course.create({
  data: {
    name: 'CRUD with Prisma',
    tests: {
      create: [
        {
          date: weekFromNow,
          name: 'First test',
        },
        {
          date: twoWeekFromNow,
          name: 'Second test',
        },
        {
          date: monthFromNow,
          name: 'Final exam',
        },
      ],
    },
  },
})
```

This will create a row in the `Course` table and three related rows in the `Tests` table (`Course` and `Tests` have a one-to-many relationship which allows this).

What if you wanted to create a relation between the user created in the previous step to this course as a teacher?

`User` and `Course` have an explicit many-to-many relationship. That means that we have to create rows in the `CourseEnrollment` table and assign a role to link a `User` to a `Course`.

This can be done as follows (adding to the query from the previous step):

```
const weekFromNow = add(new Date(), { days: 7 })
const twoWeekFromNow = add(new Date(), { days: 14 })
const monthFromNow = add(new Date(), { days: 28 })

const course = await prisma.course.create({
  data: {
    name: 'CRUD with Prisma',
    tests: {
      create: [
        {
          date: weekFromNow,
          name: 'First test',
        },
        {
          date: twoWeekFromNow,
          name: 'Second test',
        },
        {
          date: monthFromNow,
          name: 'Final exam',
        },
      ],
    },
    members: {
      create: {
        role: 'TEACHER',
        user: {
          connect: {
            email: grace.email,
          },
        },
      },
    },
  },
  include: {
    tests: true,
  },
})
```

> ***Note:*** *the* `include` *argument allows you to fetch relations in the result. This will be useful in a later step to relate test results with tests*

When using nested writes (as with `members` and `tests`) there are two options:

`connect` : Create a relation with an existing row

`create` : Create a new row and relation

In the case of `tests` , you passed an array of objects which are linked to the created course.

In the case of `members` , both `create` and `connect` were used. This is necessary because even though the `user` already exists, a *new* row in the relation table (`CourseEnrollment` referenced by `members`) needs to be created which uses `connect` to form a relation with the previously-created user.

Creating users and relating to a course

In the previous step, you created a course, related tests, and assigned a teacher to the course. In this step you will create more users and relate them to the course as *students*.

Add the following statements:

```
const shakuntala = await prisma.user.create({
  data: {
    email: 'devi@prisma.io',
    firstName: 'Shakuntala',
    lastName: 'Devi',
    courses: {
      create: {
        role: 'STUDENT',
        course: {
          connect: { id: course.id },
        },
      },
    },
  },
})

const david = await prisma.user.create({
  data: {
    email: 'david@prisma.io',
    firstName: 'David',
    lastName: 'Deutsch',
    courses: {
      create: {
        role: 'STUDENT',
        course: {
          connect: { id: course.id },
        },
      },
```

```
    },
  },
})
```

Adding test results for the students

Looking at the `TestResult` model, it has three relations: `student`, `gradedBy`, and `test`. To add test results for Shakuntala and David, you will use nested writes similarly to the previous steps.

Here is the `TestResult` model again for reference:

```
model TestResult {
  id        Int      @default(autoincrement()) @id
  createdAt DateTime @default(now())
  result    Int // Percentage precise to one decimal point represented as `result * 10^-1`

  // Relation Fields
  studentId Int
  student   User @relation(name: "results", fields: [studentId], references: [id])
  graderId  Int
  gradedBy  User @relation(name: "graded", fields: [graderId], references: [id])
  testId    Int
  test      Test @relation(fields: [testId], references: [id])
}
```

Adding a single test result would look as follows:

```
await prisma.testResult.create({
  data: {
    gradedBy: {
      connect: { email: grace.email },
    },
    student: {
      connect: { email: shakuntala.email },
    },
    test: {
      connect: { id: test.id },
    },
    result: 950,
  },
})
```

To add a test result for both David and Shakuntala for each of the three tests, you can create a loop:

```
const testResultsDavid = [650, 900, 950]
const testResultsShakuntala = [800, 950, 910]

let counter = 0
for (const test of course.tests) {
  await prisma.testResult.create({
    data: {
      gradedBy: {
        connect: { email: grace.email },
      },
      student: {
        connect: { email: shakuntala.email },
      },
      test: {
        connect: { id: test.id },
      },
      result: testResultsShakuntala[counter],
    },
  })

  await prisma.testResult.create({
    data: {
      gradedBy: {
        connect: { email: grace.email },
      },
      student: {
        connect: { email: david.email },
      },
      test: {
        connect: { id: test.id },
      },
      result: testResultsDavid[counter],
    },
  })

  counter++
}
```

Congratulations, if you have reached this point, you successfully created sample data for users, courses, tests, and test results in your database.

To explore the data in the database, you can run Prisma Studio. Prisma Studio is a visual editor for your database. To run Prisma Studio, run the following command in your terminal:

```
npx prisma studio --experimental
```

> ***Note:*** *Prisma Studio is currently in an experimental state.*

Aggregating the test results with Prisma Client

Prisma Client allows you to perform aggregate operations on number fields (such as `Int` and `Float`) of a model. Aggregate operations compute a single result from a set of input values, i.e. multiple rows in a table. For example, calculating the *minimum*, *maximum*, and *average* value of the `result` column over a set of `TestResult` rows.

In this step, you will run two kinds of aggregate operations:

For each **test** in the course across all **students**, resulting in aggregates representing how difficult the test was or the class' comprehension of the test's topic:

```
for (const test of course.tests) {
  const results = await prisma.testResult.aggregate({
    where: {
      testId: test.id,
    },
    avg: { result: true },
    max: { result: true },
    min: { result: true },
    count: true,
  })
  console.log(`test: ${test.name} (id: ${test.id})`, results)
}
```

This results in the following:

```
test: First test (id: 1) {
 avg: { result: 725 },
 max: { result: 800 },
 min: { result: 650 },
 count: 2
}
test: Second test (id: 2) {
 avg: { result: 925 },
 max: { result: 950 },
```

```
  min: { result: 900 },
  count: 2
 }
 test: Final exam (id: 3) {
  avg: { result: 930 },
  max: { result: 950 },
  min: { result: 910 },
  count: 2
 }
```

For each **student** across all **tests**, resulting in aggregates representing the student's performance in the course:

```
// Get aggregates for David
const davidAggregates = await prisma.testResult.aggregate({
  where: {
    student: { email: david.email },
  },
  avg: { result: true },
  max: { result: true },
  min: { result: true },
  count: true,
})
console.log(`David's results (email: ${david.email})`, davidAggregates)

// Get aggregates for Shakuntala
const shakuntalaAggregates = await prisma.testResult.aggregate({
  where: {
    student: { email: shakuntala.email },
  },
  avg: { result: true },
  max: { result: true },
  min: { result: true },
  count: true,
})
console.log(`Shakuntala's results (email: ${shakuntala.email})`, shakuntalaAggregates)
```

This results in the following terminal output:

```
 David's results (email: david@prisma.io) {
  avg: { result: 833.3333333333334 },
  max: { result: 950 },
```

```
 min: { result: 650 },
 count: 3
}
Shakuntala's results (email: devi@prisma.io) {
 avg: { result: 886.6666666666666 },
 max: { result: 950 },
 min: { result: 800 },
 count: 3
}
```

Summary and next steps

This article covered a lot of ground starting with the problem domain and then delving into data modeling, the Prisma Schema, database migrations with Prisma Migrate, CRUD with Prisma Client, and aggregations.

Mapping out the problem domain is generally good advice before jumping into the code because it informs the design of the data model which impacts every aspect of the backend.

While Prisma aims to make working with relational databases easy it can be helpful to have a deeper understanding of the underlying database.

Check out the Prisma's Data Guide to learn more about how databases work, how to choose the right one, and how to use databases with your applications to their full potential.

In the next parts of the series, you'll learn more about:

- API layer
- Validation
- Testing
- Authentication
- Authorization
- Integration with external APIs
- Deployment

Join for the the next live stream which will be streamed live on YouTube at 6:00 PM CEST August 12th.

Join the discussion

Follow @prisma on Twitter

Twitter