Purwadhika
Digital Technology School

Full Stack AI Software Development

Database Integration with ORM Tools

Job Connector Program

Outline

Introduction to ORM
Learn how ORM simplifies database interaction by mapping data between objects and relational tables.

Prisma ORM
Discover how to use Prisma to define models, manage relations, and perform efficient database queries.

Model and Migration
Understand how to create, update, and migrate database schemas using Prisma migrations.

Querying and Implementation
Learn how to query, filter, and manipulate data with Prisma Client and implement it in real APIs.

Database ORM

ORM stands for object-relational mapping, it might seem complex, but its purpose is to make your life as a programmer easier. To get data out of a database, you need to write a query. Does that mean you have to learn SQL? Well, no. **Object relational mapping makes it possible for you to write queries in the language of your choice**.

There are many types of ORM: Knex.js, Sequelize, Mongoose, TypeORM, Prisma, etc

Database ORM

If you're building a small project, installing an ORM library isn't required. Using SQL statements to drive your application should be sufficient. **An ORM is quite beneficial for medium- to large-scale projects that source data from hundreds of database tables**. In such a situation, you need a framework that allows you to **operate and maintain your application's data layer in a consistent and predictable way**.

ORM Libraries

ORM is commonly undertaken with help of a library. The term ORM most commonly refers to an actual ORM library — an object relational mapper — that carries out the work of object relational mapping for you.

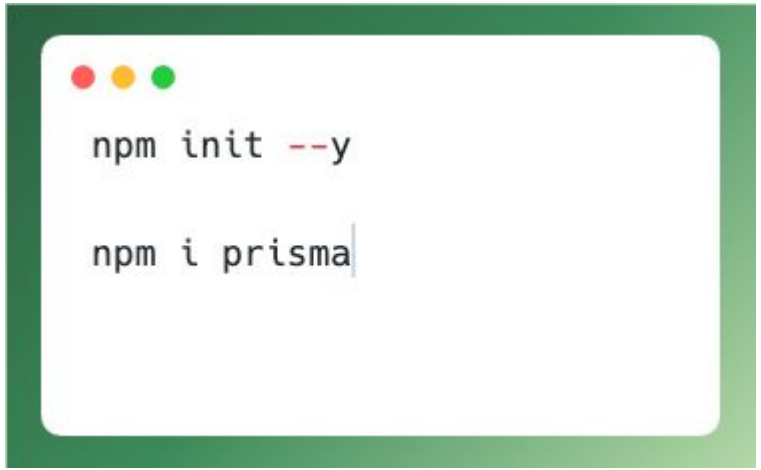
Hence, using an ORM library to build your data layer helps ensure that the database will always remain in a consistent state. ORM libraries often contain many more essential features, such as:

- Query builders
- Migration scripts
- CLI tool for generating boilerplate code
- Seeding feature for pre-populating tables with test data

Introduction to Prisma

Prisma is a modern Javascript/TypeScript and Node.js Focusing on easy to access data model declaration making project is well-documented and easy to understand.

Find out more: Prisma official website (prisma.io)



Prisma - Getting Started

Prisma is a very mature and popular Node.js ORM library with excellent documentation containing well explained concept and guides. There are several database that could handle by Prisma such as:

- Postgres
- Mysql
- Mariadb
- Sqlite
- MongoDB (NoSQL)

Prisma - Getting Started

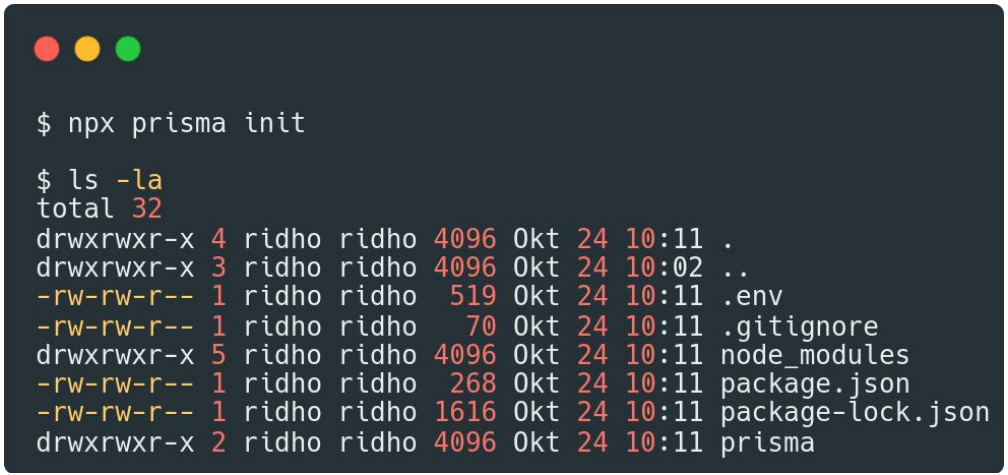
You can use prisma with new database/existing database. In this session we will use new database for our example.

To start using prisma, we need to setup our project directory using prisma CLI tools:

Prisma CLI will automatically generating files for you, such as:

- .env
- prisma/prisma.schema

Next, we will discuss usage of this file.



Prisma - Getting Started

.env stand for **environment**, this file used to describe configuration used in our project, such as database connection, secret key and many more. Usually, values stored in this file is quite sensitive and private. **Prisma** will use this file to find necessary configuration to connect to our database instance.

```
$ cat .env
DATABASE_URL="postgresql://johndoe:randompassword@localhost:5432/mydb?schema=public"
```

```
$ cat .env
DATABASE_URL="mysql://root:root@localhost:3306/learn-prisma"
```

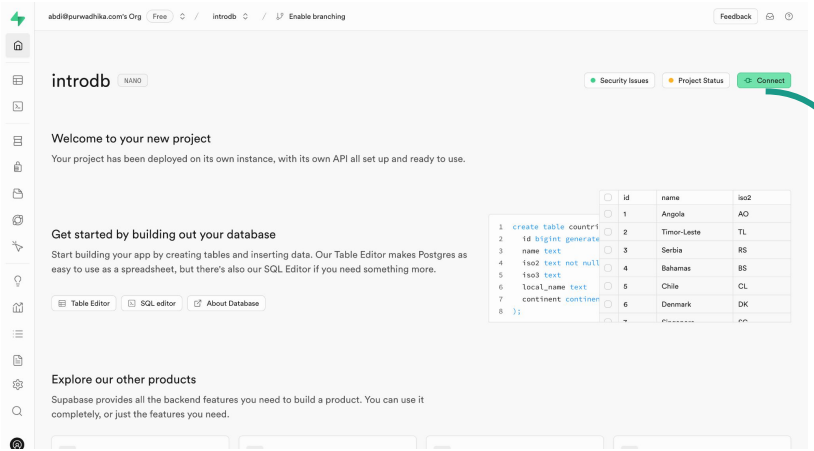
Since we use **MySQL** as our database, we need to configure it properly, with this format:

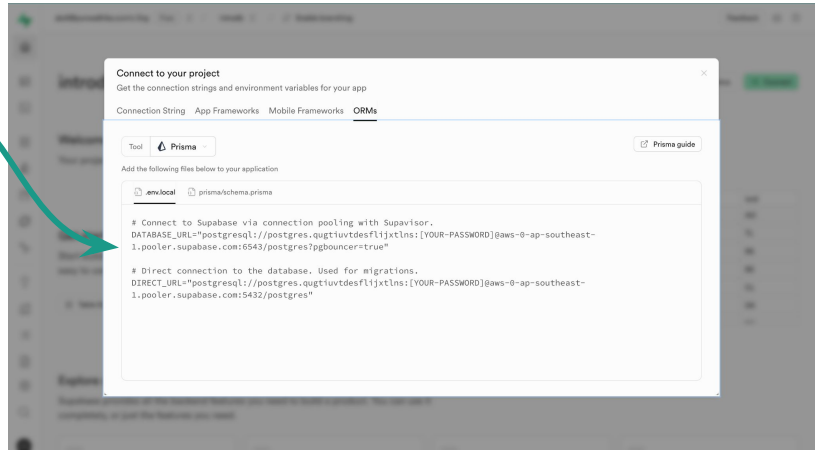
mysql://<user>:<password>@<host>:<port>/<db_name>

You can find more details in prisma documentation: prisma.io/docs/concepts/database-connectors/mysql

Prisma - Connect Supabase PostgreSQL

Open your project in Supabase dashboard and click button connection.
https://supabase.com/partners/integrations/prisma

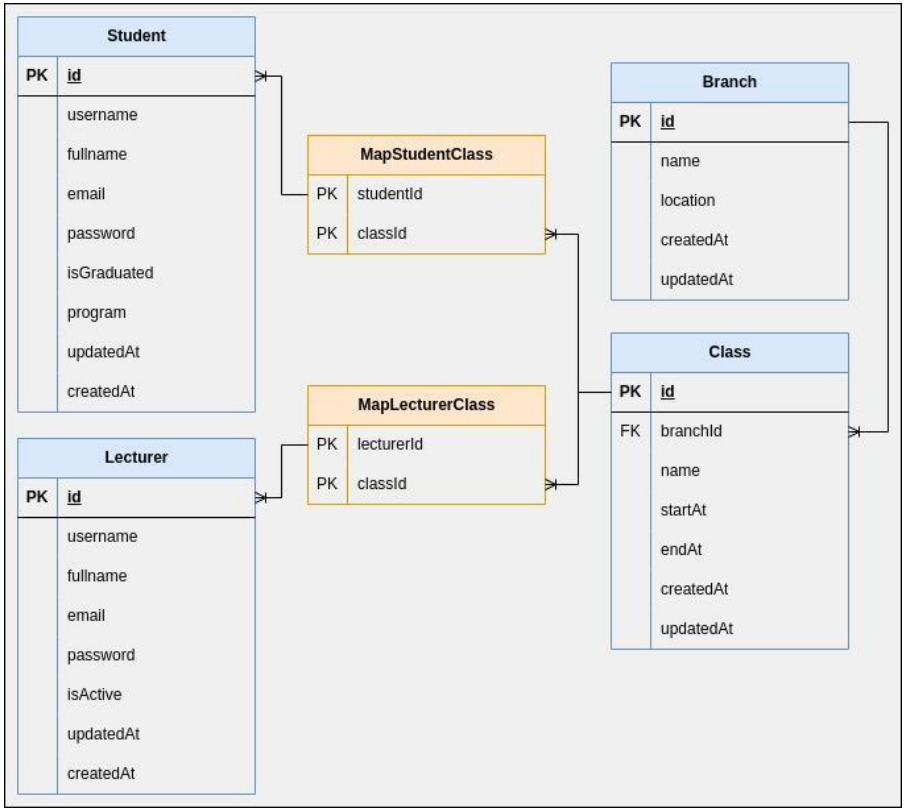




Prisma - Getting Started

To understand clearly how to use prisma, let's imagine we are creating **class usage management** application, where the goal is to track each class used in all programs. We can follow this database diagram as our example.

Now, let's get started.



Prisma - Creating our first model

Prisma use **.prisma** file as data definition source of truth. In this file you can define how your data model structured.

Now, let's add new **Branch** model as following.

More details about model definition: prisma.io/docs/concepts/components/prisma-schema

prisma > schema.prisma > ...

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

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

model Branch {
  id        Int      @id @default(autoincrement())
  name      String
  location  String
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}
```

Prisma - Model Naming Conventions

- Model names must adhere to the following regular expression: [A-Za-z][A-Za-z0-9_]*
- Model names must start with a letter and are typically spelled in **PascalCase**
- Model names should use the singular form (for example, **User** instead of **user**, **users** or **Users**)
- Prisma has a number of reserved words that are being used by Prisma internally and therefore cannot be used as a model name.

Prisma - Fields Naming Conventions

- Must start with a letter
- Typically spelled in **camelCase**
- Must adhere to the following regular expression: [A-Za-z][A-Za-z0-9_]

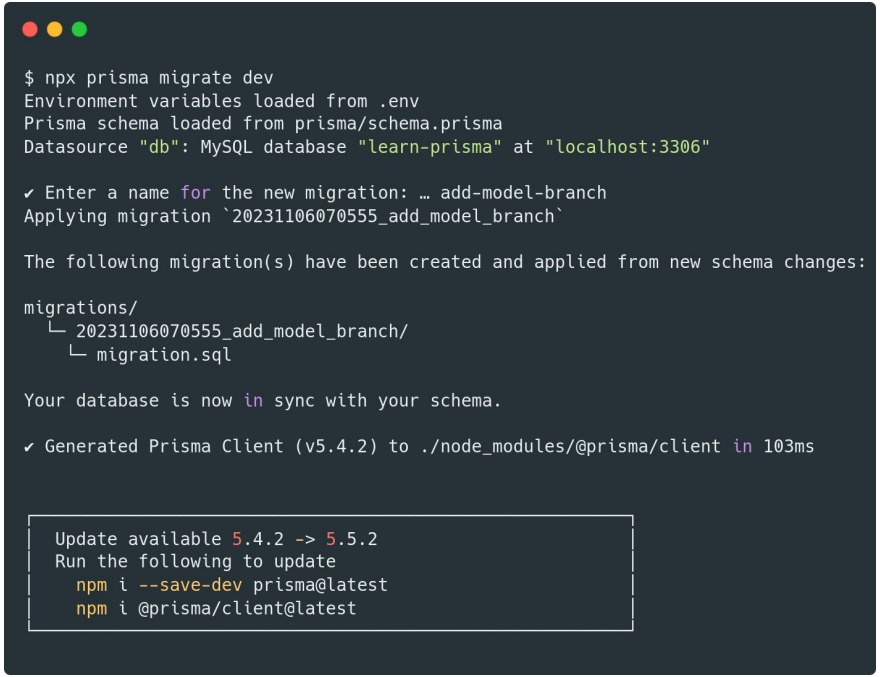
https://www.prisma.io/docs/orm/reference/prisma-schema-reference#naming-conventions

Prisma - Migrations

Since you already create a **Model**, now we need to synchronize it to our database, in order to do that, we need to use **Prisma CLI** command :

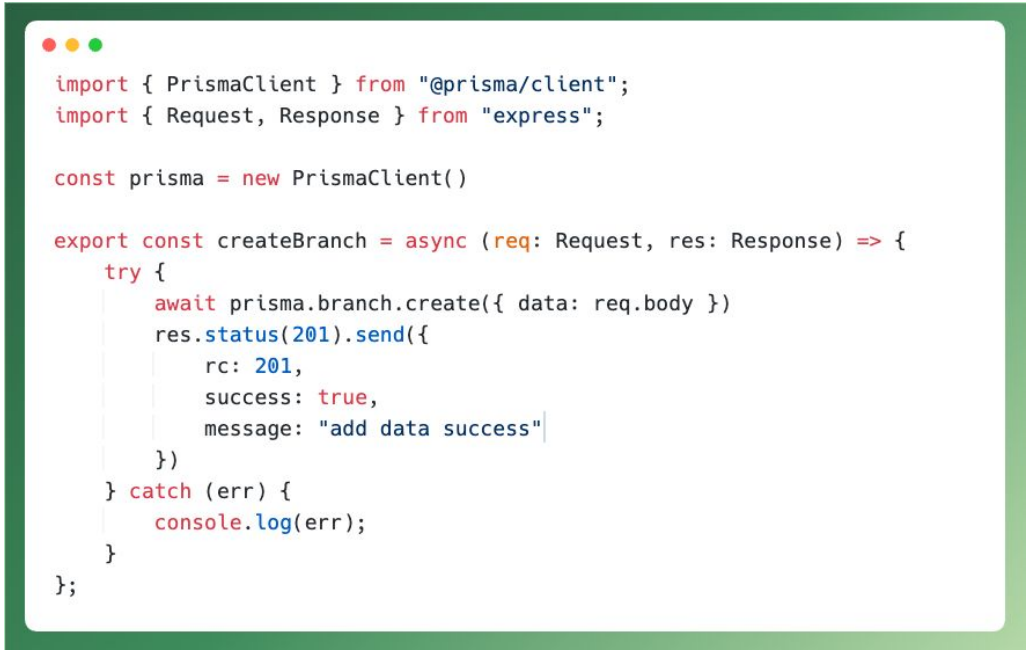
npx prisma migrate dev

This command will prompt you to fill a **migration name**. To add one, simply type it in your command line and press **Enter**.



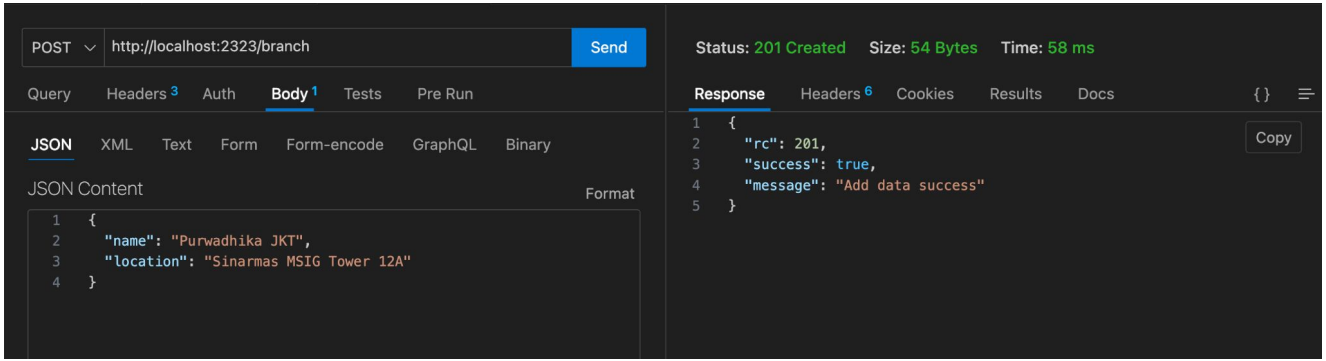
Model Query Basics - Create

In order to use prisma in our REST API apps, we need to construct **PrismaClient** class and use the value to interact with our data.
Create controller file *branch.ts* and setup our service middleware.



```
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient()

export const createBranch = async (req: Request, res: Response) => {
    try {
        await prisma.branch.create({ data: req.body })
        res.status(201).send({
            rc: 201,
            success: true,
            message: "add data success"
        })
    } catch (err) {
        console.log(err);
    }
};
```

Model Query Basics - Create - Testing



If you want to try multiple insert, check this documentation
https://www.prisma.io/docs/orm/prisma-client/queries/crud#create-multiple-records

Model Query Basics - Read Multiple

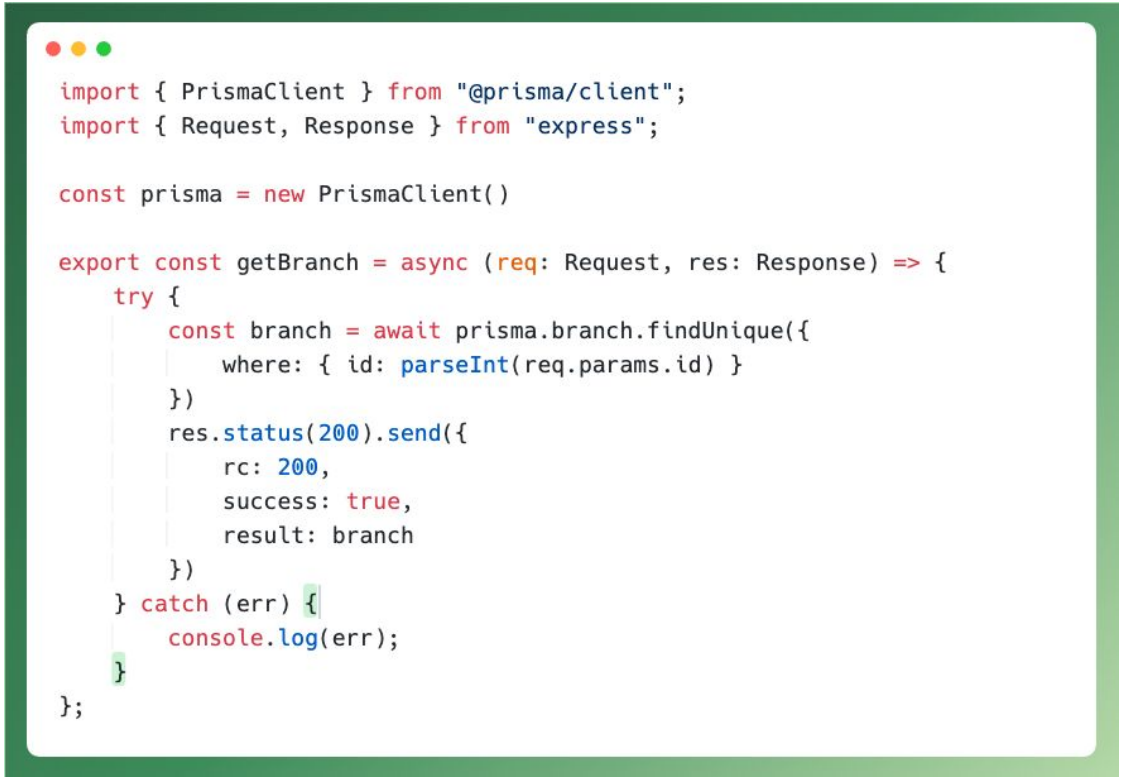
To get all data we can use **findMany** function

```
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient()

export const getBranches = async (req: Request, res: Response) => {
    try {
        const branches = await prisma.branch.findMany()
        res.status(200).send({
            rc: 201,
            success: true,
            result: branches
        })
    } catch (err) {
        console.log(err);
    }
};
```

Model Query Basics - Read

To get single unique data we can use **findUnique** function

Note: only unique/primary key field can be used in where statement if you are using **findUnique** function.



```
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient()

export const getBranch = async (req: Request, res: Response) => {
    try {
        const branch = await prisma.branch.findUnique({
            where: { id: parseInt(req.params.id) }
        })
        res.status(200).send({
            rc: 200,
            success: true,
            result: branch
        })
    } catch (err) {
        console.log(err);
    }
};
```

Model Query Basics - Dynamic filter with req.query

Just like SQL statement, in prisma we can add condition using **where** options, this method will apply **WHERE** to the generated SQL query used by prisma.

This example is how we can implement dynamic filter with parameter from req.query.

```
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient()

export const getBranches = async (req: Request, res: Response) => {
    try {
        const { name } = req.query

        const filterData: any = {}

        if(name){
            filterData.name = name as string
        }

        const branches = await prisma.branch.findMany({
            where: filterData
        })
        res.status(200).send({
            rc: 200,
            success: true,
            result: branches
        })
    } catch (err) {
        console.log(err);
    }
};
```

Model Query Basics - Condition “Contain”

Contains same with “LIKE” with “%params%” in SQL Queries.

This example with output batam & jakarta branch since both contains **ta**.

```
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient();

export const getBranches = async (req: Request, res: Response) => {
  try {
    const { name } = req.query;

    const filterData: any = {};

    if (name) {
      filterData.name = {
        contains: name as string,
        mode: "insensitive",
      };
    }

    const branches = await prisma.branch.findMany({
      where: filterData,
    });

    res.status(200).send({
      rc: 200,
      success: true,
      result: branches,
    });
  } catch (err) {
    console.log(err);
  }
};
```

Model Query Basics - Update

Update queries also accept the where option.

Prisma protect table property by defined data type. Don’t forget to give value by defined data type.

Req.params.id is a string, but in column configuration id is a number.

```
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient();

export const updateBranch = async (req: Request, res: Response) => {
  try {
    const { id } = req.params;

    await prisma.branch.update({
      where: { id: parseInt(id) },
      data: req.body,
    });

    res.status(200).send({
      rc: 200,
      success: true,
      message: "Update data success",
    });
  } catch (err) {
    console.log(err);
  }
};
```

Model Query Basics - Delete

Delete queries also accept the where option.

```
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient();

export const deleteBranch = async (req: Request, res: Response) => {
  try {
    const { id } = req.params;

    await prisma.branch.delete({
      where: { id: parseInt(id) },
    });

    res.status(200).send({
      rc: 200,
      success: true,
      message: "Delete data success",
    });
  } catch (err) {
    console.log(err);
  }
};
```

Model Query Basics - Pagination

Prisma Client supports both offset pagination and cursor-based pagination.

Offset pagination uses skip and take to skip a certain number of results and select a limited range. The following query skips the first 3 Post records and returns records 4 - 7:

https://www.prisma.io/docs/orm/prisma-client/queries/pagination

Model Query Basics - Pagination Example

```ts
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient();

export const getBrances = async (req: Request, res: Response) => {
  try {
    const branches = await prisma.branch.findMany({
      skip: 1,
      take: 2,
    });

    res.status(200).send({
      rc: 200,
      success: true,
      result: branches,
    });
  } catch (err) {
    console.log(err);
  }
};
```

Model Query Basics - Agregation

Prisma Client support aggregation like SQL Query.
We can find average, min, max, count etc.

https://www.prisma.io/docs/orm/prisma-client/queries/aggregation-grouping-summarizing

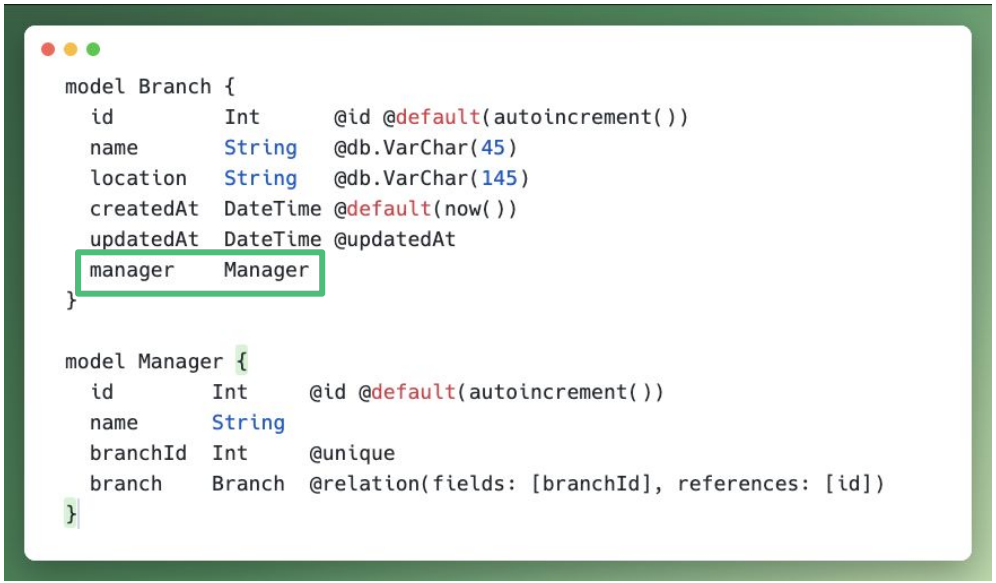
```
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient();

export const getBranchesAggregation = async (req: Request, res: Response)
    => {
  try {
    const aggregation = await prisma.branch.aggregate({
      _count: true, // count all branches
      _avg: {
        employeeCount: true, // numeric field example
      },
      _sum: {
        employeeCount: true,
      },
      _min: {
        createdAt: true,
      },
      _max: {
        createdAt: true,
      },
    });

    res.status(200).send({
      rc: 200,
      success: true,
      result: aggregation,
    });
  } catch (err) {
    console.log(err);
  }
};
```

Model Query Basics - Relation queries

Prisma, to implement relationships between tables, we simply use the options include and select.

- Use include to include related records, such as a user's posts or profile, in the query response.
- Use a nested select to include specific fields from a related record. You can also nest select inside an include.

Model Query Basics - Model Relation One to One

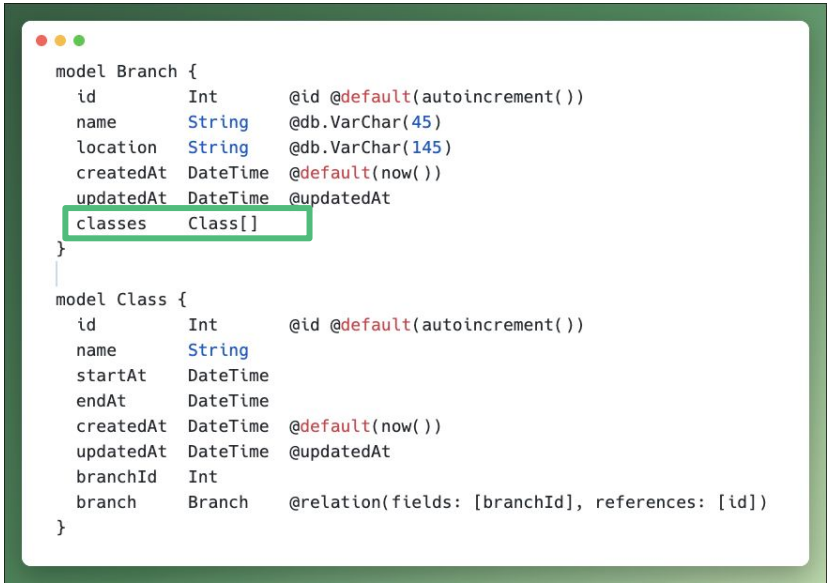


```
model Branch {
  id          Int      @id @default(autoincrement())
  name        String   @db.VarChar(45)
  location    String   @db.VarChar(145)
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
  manager     Manager
}

model Manager {
  id        Int     @id @default(autoincrement())
  name      String
  branchId  Int     @unique
  branch    Branch  @relation(fields: [branchId], references: [id])
}
```

Now, let's configure the table relationship model first.

In this model, we need to add field manager with initialize to Manager model. This indicates between Branch and Manager only has relationship with one data.

Model Query Basics - Model Relation One to Many



```
model Branch {
  id          Int       @id @default(autoincrement())
  name        String    @db.VarChar(45)
  location    String    @db.VarChar(145)
  createdAt   DateTime  @default(now())
  updatedAt   DateTime  @updatedAt
  classes     Class[]
}

model Class {
  id          Int       @id @default(autoincrement())
  name        String
  startAt     DateTime
  endAt       DateTime
  createdAt   DateTime  @default(now())
  updatedAt   DateTime  @updatedAt
  branchId    Int
  branch      Branch    @relation(fields: [branchId], references: [id])
}
```

Now, let's configure the table relationship model first.

In this model, we need to add field **classes** with initialize **Class[ ]**. This indicates that the relationship between Branch and Class is in a one-to-many form, as denoted by the presence of "[ ]".

Model Query Basics - Relation queries

In this program, we are attempting to retrieve class data for a specific branch based on **“req.params.id”**. It is assumed that each branch has a relationship with several class data, where the connecting columns are **“branchId”** in the **“Class”** table and **“id”** in the **“Branch”** table.

https://www.prisma.io/docs/orm/prisma-client/queries/relation-queries

```
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient();

export const getBranchesWithClasses = async (req: Request, res: Response)
    => {
  try {
    const branches = await prisma.branch.findMany({
      include: {
        classes: true, // join with related classes
      },
    });

    res.status(200).send({
      rc: 200,
      success: true,
      result: branches,
    });
  } catch (err) {
    console.log(err);
  }
};
```

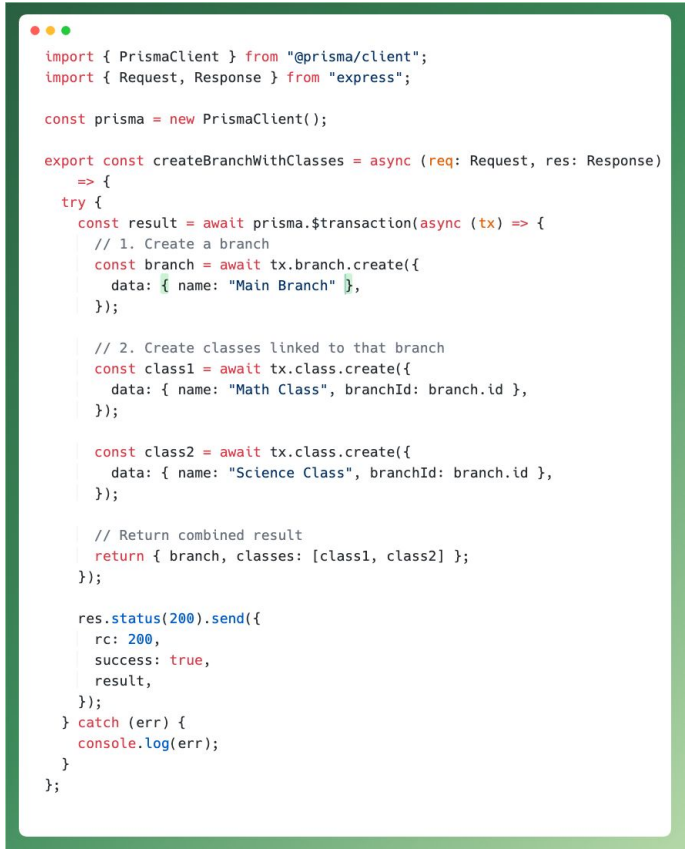
Transactions

Prisma use **$transaction** api for using transactions in to way :

- Interactive transactions: Pass a function that can contain user code including Prisma Client queries, non-Prisma code and other control flow to be executed in a transaction.
- Sequential operations: Pass an array of Prisma Client queries to be executed sequentially inside of a transaction.

https://www.prisma.io/docs/orm/prisma-client/queries/transactions#the-transaction-api

Transactions - Interactive Transaction



```
import { PrismaClient } from "@prisma/client";
import { Request, Response } from "express";

const prisma = new PrismaClient();

export const createBranchWithClasses = async (req: Request, res: Response)
    => {
  try {
    const result = await prisma.$transaction(async (tx) => {
      // 1. Create a branch
      const branch = await tx.branch.create({
        data: { name: "Main Branch" },
      });

      // 2. Create classes linked to that branch
      const class1 = await tx.class.create({
        data: { name: "Math Class", branchId: branch.id },
      });

      const class2 = await tx.class.create({
        data: { name: "Science Class", branchId: branch.id },
      });

      // Return combined result
      return { branch, classes: [class1, class2] };
    });

    res.status(200).send({
      rc: 200,
      success: true,
      result,
    });
  } catch (err) {
    console.log(err);
  }
};
```

In this example:

We use **"prisma.$transaction"** to wrap operations within a transaction.

Step 1 involves adding new data to the **"branch"** table using **"prisma.branch.create"**.

If both operations succeed, we return a response, and the transaction is committed.

If there's an error in any of the operations, we catch it, print an error message, and the transaction is rolled back.

Exercise - Simple Social Media API using Express, Prisma, and TypeScript

Users:

1. **Account Registration:**
   - Users can create an account by providing information such as name, email, and password.
2. **Authentication:**
   - Users can log in with their created accounts.

Posts:

1. **Create Post:**
   - Users can create new posts with text and optionally an image.
2. **View Posts:**
   - Users can see a list of posts, both their own and those from other users.
3. **Edit Posts:**
   - Users can like posts and add comments.

API Endpoints:

- **Authentication:**
  - `POST /api/auth/register`: Register a new account.
  - `POST /api/auth/login`: Log into an account.
- **Users:**
  - `GET /api/users`: Get a list of users.
  - `GET /api/users/:id`: Get user information by ID.
  - `PUT /api/users/:id`: Update user information.
  - `GET /api/users/:id/posts`: Get posts from a specific user.
- **Posts:**
  - `GET /api/posts`: Get a list of posts.
  - `GET /api/posts/:id`: Get post information by ID.
  - `POST /api/posts`: Create a new post.
  - `PUT /api/posts/:id`: Update a post.

Thank you