

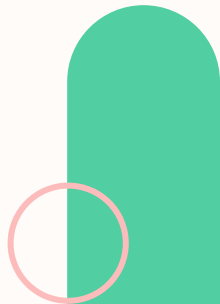
Full Stack Web Development

Database ORM

Database ORM

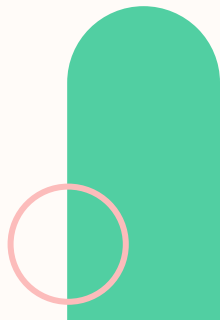
ORM stands for object-relational mapping, it might seem complex, but its purpose is to make your life as a programmer easier. To get data out of a database, you need to write a query. Does that mean you have to learn SQL? Well, no. **Object relational mapping makes it possible for you to write queries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

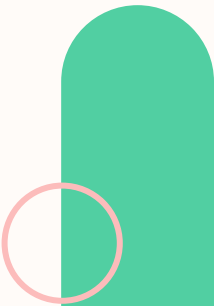


ORM Libraries

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

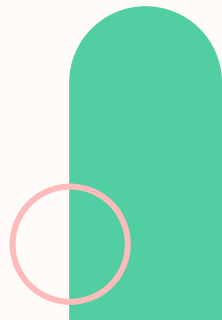
Prisma is a modern Javascript/TypeScript and Node.js Focusing on easy to access data model declaration making project is well-documented and easy to understand.

Find out more: [Prisma official website](https://prisma.io) (prisma.io)



```
$ npm init --y
```

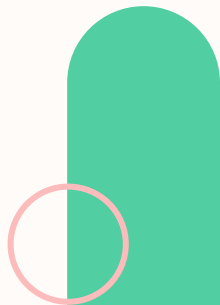
```
$ npm install --save prisma
```



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.

```
$ npx prisma init

$ ls -la
total 32
drwxrwxr-x 4 ridho ridho 4096 0kt 24 10:11 .
drwxrwxr-x 3 ridho ridho 4096 0kt 24 10:02 ..
-rw-rw-r-- 1 ridho ridho 519 0kt 24 10:11 .env
-rw-rw-r-- 1 ridho ridho 70 0kt 24 10:11 .gitignore
drwxrwxr-x 5 ridho ridho 4096 0kt 24 10:11 node_modules
-rw-rw-r-- 1 ridho ridho 268 0kt 24 10:11 package.json
-rw-rw-r-- 1 ridho ridho 1616 0kt 24 10:11 package-lock.json
drwxrwxr-x 2 ridho ridho 4096 0kt 24 10:11 prisma
```

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:

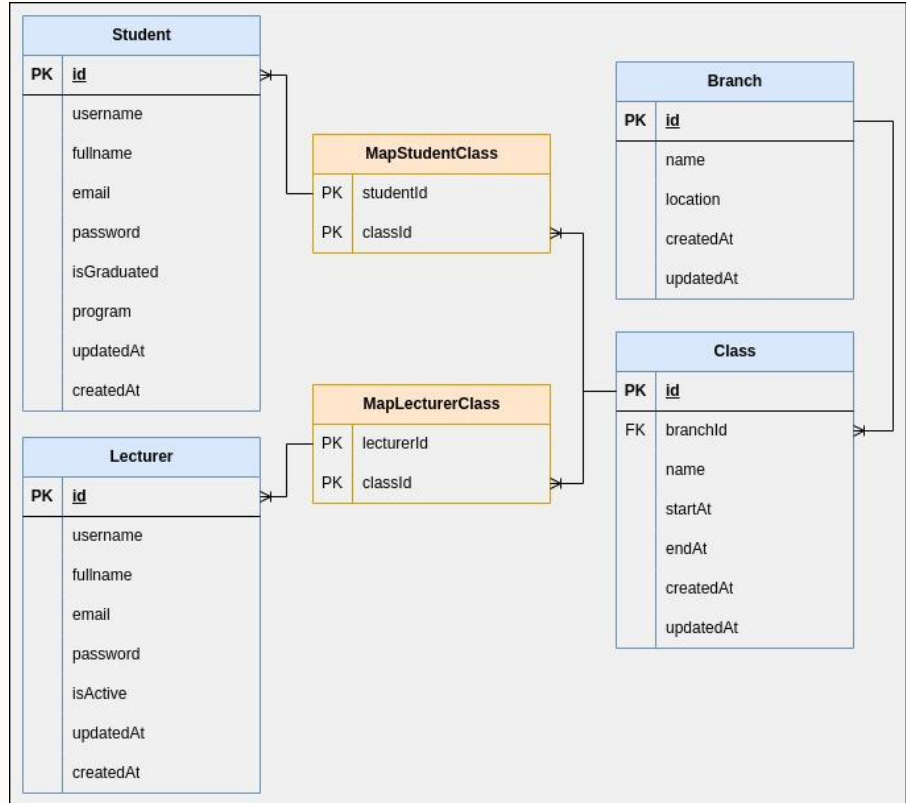
prisma.io/docs/concepts/database-connectors/mysql



Prisma - 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.

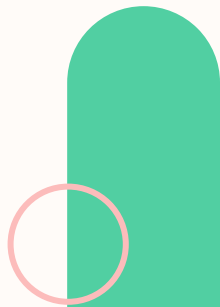
More details about model definition:

prisma.io/docs/concepts/components/prisma-schema
[@](#)

```
prisma > schema.prisma > ...
1 // This is your Prisma schema file,
2 // learn more about it in the docs: https://pris.ly/d/prisma-schema
3
4 generator client {
5   provider = "prisma-client-js"
6 }
7
8 datasource db {
9   provider = "mysql"
10  url       = env("DATABASE_URL")
11 }
12
13 model Branch {
14   id          Int      @id @default(autoincrement())
15   name        String
16   location    String
17   createdAt   DateTime @default(now())
18   updatedAt   DateTime @updatedAt
19 }
```

Prisma - Model Naming Conventions

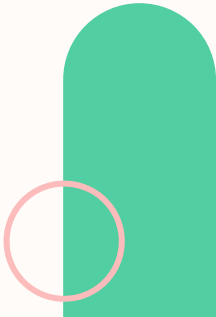
- Model names must adhere to the following regular expression: `[A-Za-z][A-Za-z0-9_]*`
- Model names must start with a letter and are typically spelled in **PascalCase**
- Model names should use the singular form (for example, **User** instead of **user**, **users** or **Users**)
- Prisma has a number 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**.

```
$ npx prisma migrate dev
Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": MySQL database "learn-prisma" at "localhost:3306"

✓ Enter a name for the new migration: ... add-model-branch
Applying migration `20231106070555_add_model_branch`

The following migration(s) have been created and applied from new schema changes:

migrations/
├─ 20231106070555_add_model_branch/
│   └─ migration.sql

Your database is now in sync with your schema.

✓ Generated Prisma Client (v5.4.2) to ./node_modules/@prisma/client in 103ms
```

```
Update available 5.4.2 -> 5.5.2
Run the following to update
npm i --save-dev prisma@latest
npm i @prisma/client@latest
```

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.

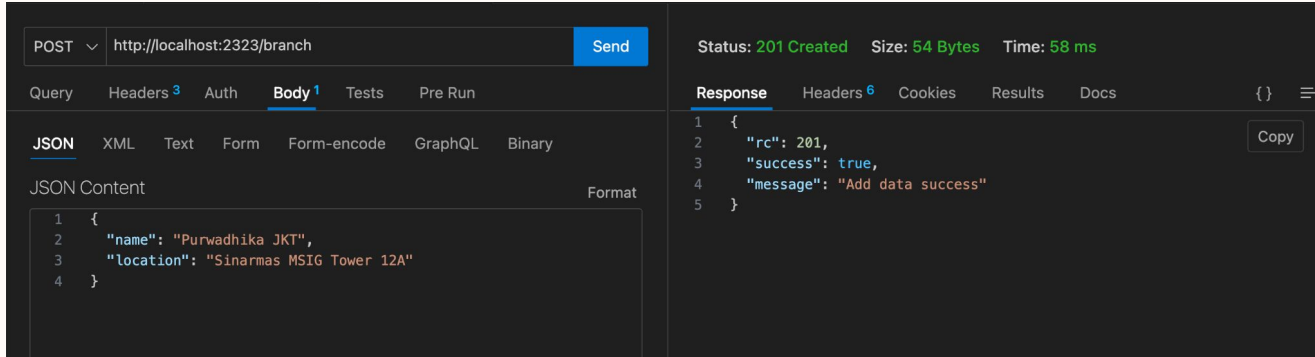
PrismaClient Setup

We can use **create** function to add new data to our database.

```
1  import { PrismaClient } from "@prisma/client";
2  import { Request, Response } from "express";
3
4  const prisma = new PrismaClient();
5
6  export const createBranch = async (req: Request, res: Response) => {
7    try {
8      const branch = await prisma.branch.create({
9        data: req.body,
10      });
11      res.status(201).send({
12        rc: 201,
13        success: true,
14        message: "Add data success",
15      });
16    } catch (error) {
17      console.log(error);
18    }
19  };

```

Model Query Basics - Create - Testing

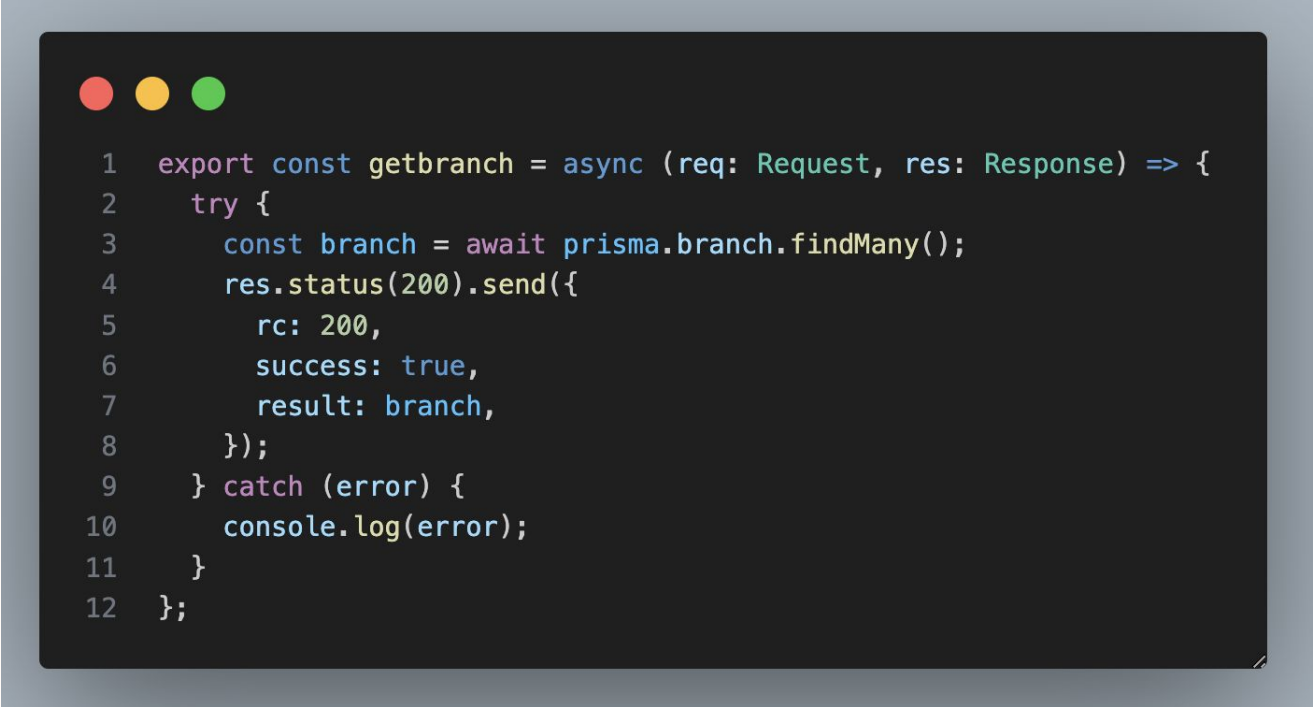


If you want to try multiple insert, check this documentation

<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



```
1 export const getbranch = async (req: Request, res: Response) => {
2   try {
3     const branch = await prisma.branch.findMany();
4     res.status(200).send({
5       rc: 200,
6       success: true,
7       result: branch,
8     });
9   } catch (error) {
10    console.log(error);
11  }
12 };
```


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.

```
1 export const getbranchUnique = async (req: Request, res: Response) => {
2   try {
3     const branchUnique = await prisma.branch.findUnique({
4       where: {
5         id: parseInt(req.params.id),
6       },
7     });
8     res.status(200).send({
9       rc: 200,
10      success: true,
11      result: branchUnique,
12    });
13   } catch (error) {
14     console.log(error);
15   }
16 };
```

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.

```
1  export const getbranch = async (req: Request, res: Response) => {
2    try {
3      interface FilterQuery {
4        id?: number;
5        name?: string;
6      }
7      const { id, name } = req.query;
8      const filterData: FilterQuery = {};
9      if (id) {
10       filterData.id = parseInt(id as string);
11     }
12     if (name) {
13       filterData.name = name as string;
14     }
15     const branch = await prisma.branch.findMany({
16       where: filterData,
17     });
18
19     res.status(200).send({
20       rc: 200,
21       success: true,
22       result: branch,
23     });
24   } catch (error) {
25     console.log(error);
26   }
27 }
```

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**.

```
async function main() {
  const result = await prisma.branch.findMany({
    where: {
      name: {
        contains: "ta",
      },
    },
  });
  console.log(result);
}
```

Model Query Basics - Other Condition

Prisma support all kind of filter conditioning, here are some available parameter you can use to apply your condition.

More details about prisma filtering:

prisma.io/docs/concepts/components/prisma-client/filtering-and-sorting#filtering

```
async function main() {
  const result = await prisma.branch.findMany({
    where: {
      // AND, OR, NOT = for multiple condition
      AND: [
        {
          location: {
            // not = for single condition
            not: {
              // contains, startsWith, endsWith
              contains: "ta",
            },
          },
          name: {
            // in, notIn
            notIn: ["Batam", "Jakarta"],
          },
        },
      ],
    },
    createdAt: {
      // gt, lt, gte, lte
      gte: new Date("2023-01-01"),
    },
  });
  console.log(result);
}
```

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.

```
1 export const updateBranch = async (req: Request, res: Response) => {
2   try {
3     const { id } = req.params;
4     const branchUpdate = await prisma.branch.update({
5       where: { id: parseInt(id) },
6       data: req.body,
7     });
8     return res.status(200).send({
9       rc: 200,
10      success: true,
11      message: "Update data success",
12      result: branchUpdate,
13    });
14   } catch (error) {
15     console.log(error);
16   }
17 };
```

Model Query Basics - Delete

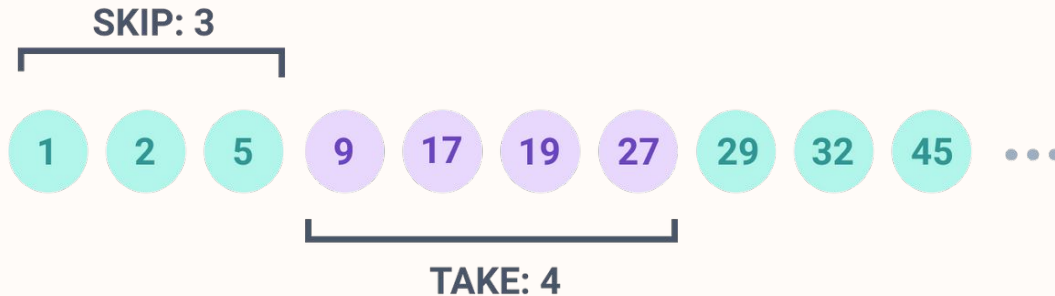
Delete queries also accept the where option.

```
1 export const deleteBranch = async (req: Request, res: Response) => {
2   try {
3     const { id } = req.params;
4     const branchDelete = await prisma.branch.delete({
5       where: { id: parseInt(id) },
6     });
7     return res.status(200).send({
8       rc: 200,
9       success: true,
10      message: "Delete data success",
11      result: branchDelete,
12    });
13   } catch (error) {
14     console.log(error);
15   }
16 };
```

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



```
1 export const getbranch = async (req: Request, res: Response) => {
2   try {
3     const branch = await prisma.branch.findMany({
4       skip: 1,
5       take: 2,
6     });
7
8     return res.status(200).send({
9       rc: 200,
10      success: true,
11      result: branch,
12    });
13   } catch (error) {
14     console.log(error);
15   }
16 };
```


Model Query Basics - Agregation

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

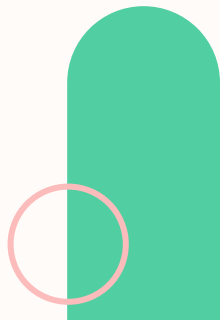
<https://www.prisma.io/docs/orm/prisma-client/queries/aggregation-grouping-summarizing>

```
1 export const getBranchStats = async (req: Request, res: Response) => {
2   try {
3     const branchStats = await prisma.branch.aggregate({
4       _count: {
5         _all: true,
6       },
7       _min: {
8         createdAt: true,
9       },
10      _max: {
11        createdAt: true,
12      },
13    });
14    console.log(`Total branches created: ${branchStats._count._all}`);
15    console.log(`Earliest creation time: ${branchStats._min.createdAt}`);
16    console.log(`Latest creation time: ${branchStats._max.createdAt}`);
17
18    return res.status(200).send({
19      rc: 200,
20      success: true,
21      result: branchStats,
22    });
23  } catch (error) {
24    console.log(error);
25  }
26  };
```

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

```
1 model Branch {
2   id      Int      @id @default(autoincrement())
3   name    String   @db.VarChar(45)
4   location String   @db.VarChar(145)
5   createdAt DateTime @default(now())
6   updatedAt DateTime @updatedAt
7   manager Manager
8 }
9
10 model Manager {
11   id      Int      @id @default(autoincrement())
12   name    String
13   branchId Int      @unique
14   branch  Branch   @relation(fields: [branchId], references: [id])
15 }
```

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

```
1 model Branch {
2   id      Int      @id @default(autoincrement())
3   name     String  @db.VarChar(45)
4   location String  @db.VarChar(145)
5   createdAt DateTime @default(now())
6   updatedAt DateTime @updatedAt
7   classes Class[]
8 }
9
10 model Class {
11   id      Int      @id @default(autoincrement())
12   name     String
13   startAt  DateTime
14   endAt    DateTime
15   createdAt DateTime @default(now())
16   updatedAt DateTime @updatedAt
17   branchId Int
18   branch   Branch  @relation(fields: [branchId], references: [id])
19 }
```

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>

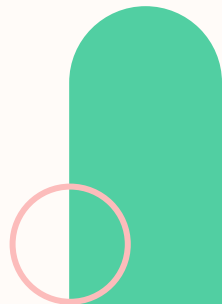
```
1 export const getBranchWithClass = async (req: Request, res: Response) => {
2   try {
3     const branchWithClasses = await prisma.branch.findUnique({
4       where: { id: parseInt(req.params.id) },
5       include: {
6         classes: true,
7       } as any,
8     });
9
10    console.log("Branch with classes:", branchWithClasses);
11
12    return res.status(200).send({
13      rc: 200,
14      success: true,
15      result: branchWithClasses,
16    });
17  } catch (error) {
18    console.log(error);
19  }
20  };
```

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

```
1 export const createBranch = async (req: Request, res: Response) => {
2   const transactionResult = await prisma.$transaction(async (prisma) => {
3     try {
4       // Langkah 1: Menambah data baru ke dalam tabel branch
5       const branch = await prisma.branch.create({
6         data: req.body,
7       });
8       // Jika semua operasi berhasil, commit transaksi
9       return res.status(201).send({
10         rc: 201,
11         success: true,
12         message: "Add data success",
13       });
14     } catch (error) {
15       // Jika ada kesalahan, transaksi akan di-rollback
16       console.error("Transaction error:", error);
17       throw error;
18     }
19   });
20 }
```

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!

