PRISMA

(https://projects.100xdevs.com/tracks/gZf9uBBNSbBR7UCqyyqT/prisma-1)

What are ORMs

Definition:

ORM stands for Object-Relational Mapping, a programming technique used in software development to convert data between incompatible type systems in object-oriented programming languages. This technique creates a **"virtual object database"** that can be used from within the programming language. ORMs are used to abstract the complexities of the underlying database into simpler, more easily managed objects within the code

Other definitions:

ORMs let you easily interact with your database without worrying too much about the underlying syntax (SQL language for eg)

Prisma is Node.js specifics

Why ORMS?

1. Simpler syntax (converts objects to SQL queries under the hood)

Non ORM

```
const query = 'SELECT * FROM users WHERE email = $1';
const result = await client.query(query, ["harkirat@gmail.com"]);
```

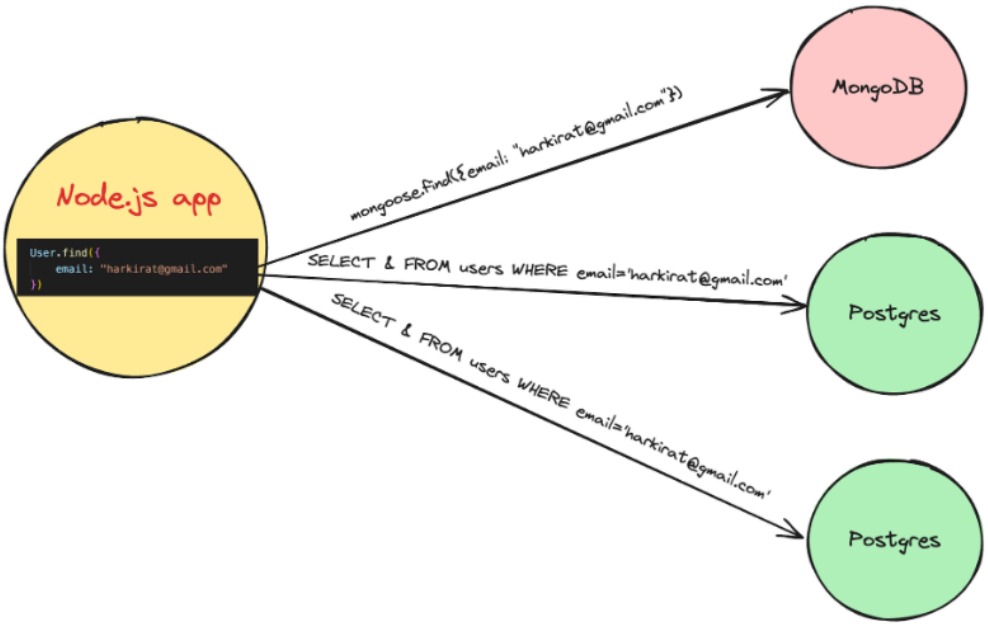
ORM

```
User.find({
    email: "harkirat@gmail.com"
})
```

We have to understand the SQL query and then pass it

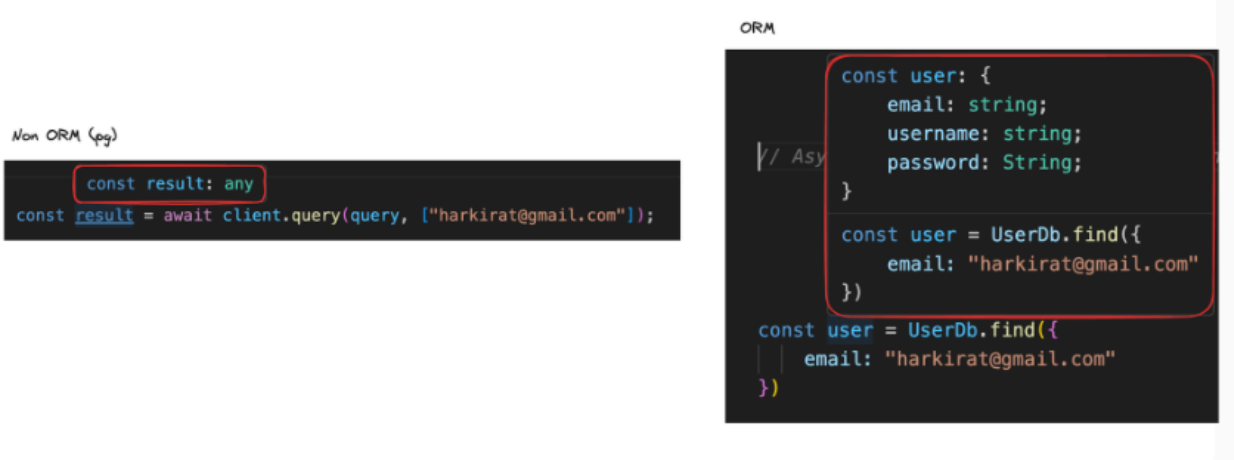
2. Abstraction that lets you flip the database you are using. Unified API irrespective of the DB

Prisma not specific to Postgress, prisma code looks same even if we use it in MongoDB and postgres.



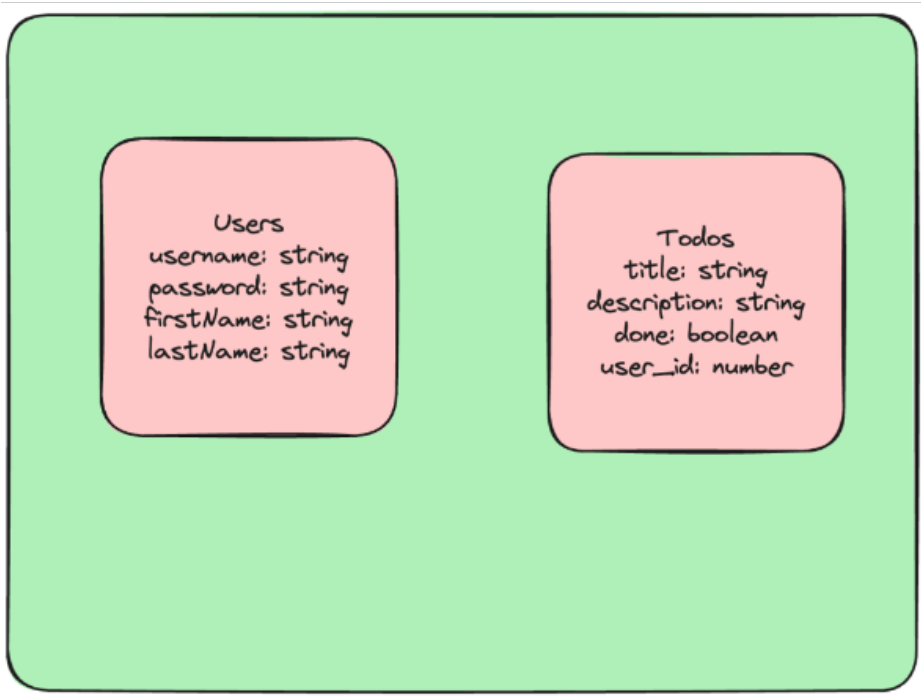
3. Type safety/ Auto completion

We want result here user want to look like this.



4. Automatic migrations

In case of simple Postgres app, it's very hard to keep track of all the components that were ran that led to the current schema of the table.




```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE NOT NULL
);

ALTER TABLE users
ADD COLUMN phone_number VARCHAR(15);
// now even supporting the phoneNo
```

As your app grows, you will have a lot of these CREATE and ALTER commands.

ORMs (or more specifically Prisma) maintains all of these for you.

For example - https://github.com/code100x/cms/tree/main/prisma/migrations

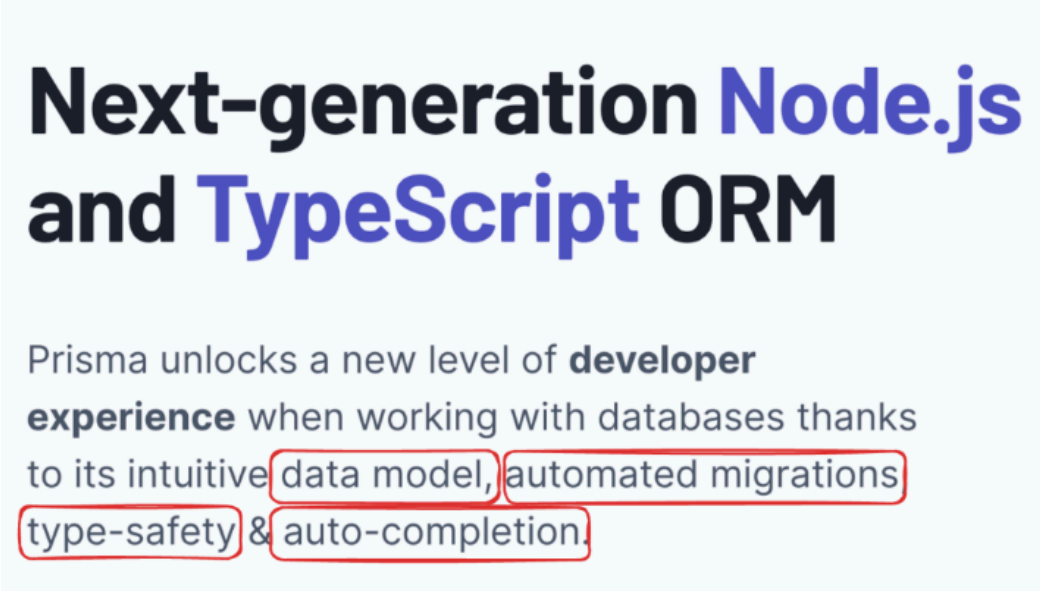
How will be know what tables we formed, migration is very long files, this specif schema was reached by running the following commands, and putting them in a centralised place.

WE have to keep telling the database that this is schema of the DB

Migration is telling DB what our User table will look like
IT shows hows overtime schema changes.

Lets say if NeonDB is down at least we have schema to shift to different database.

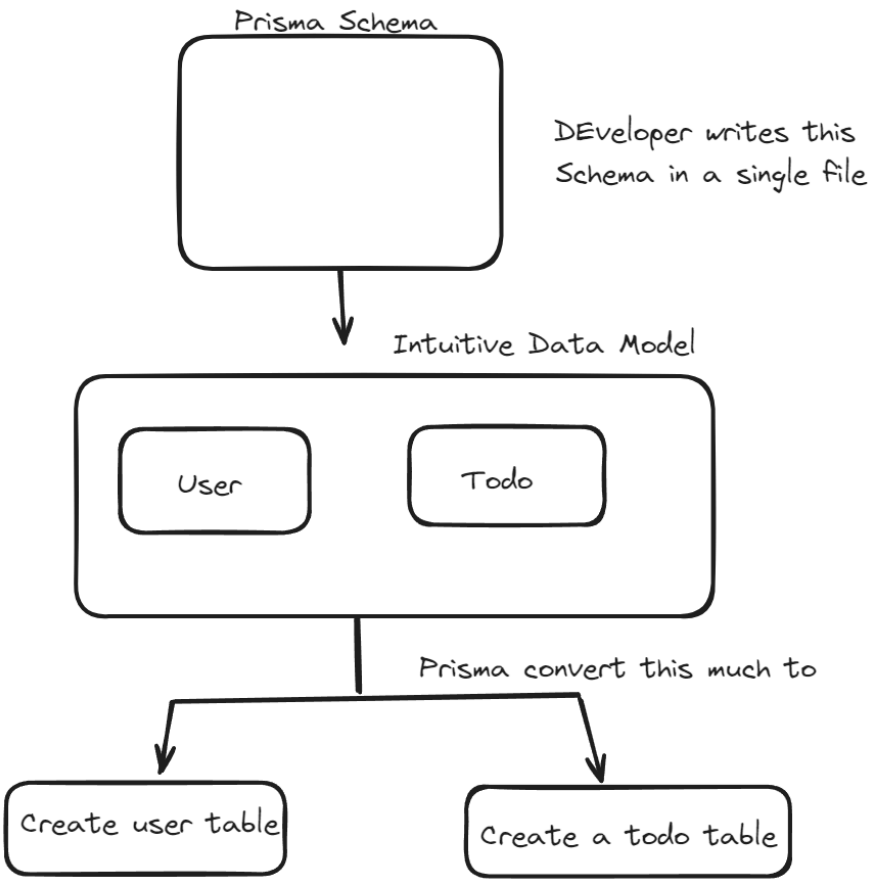
WHat is Prisma



1. Data model

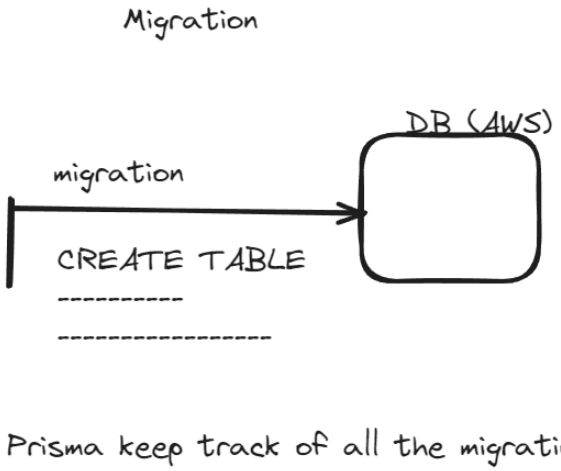
In a single file , define your schema. What it looks like what tables you haven what field each table has, how are rows related to each other.

Schema.prisma file will automatically create migration or basically generate files showing the changes we made.



2. Automated migrations

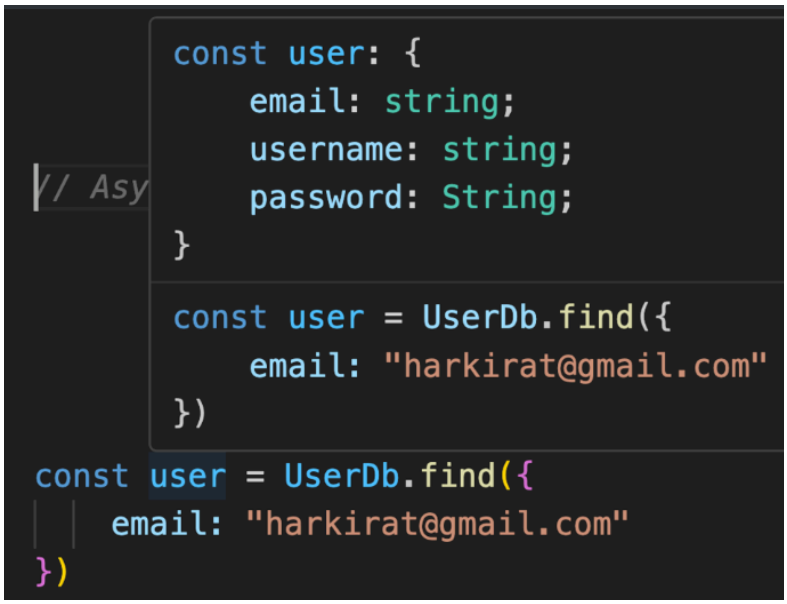
Prisma generates and runs database migrations based on changes to the Prisma schema.



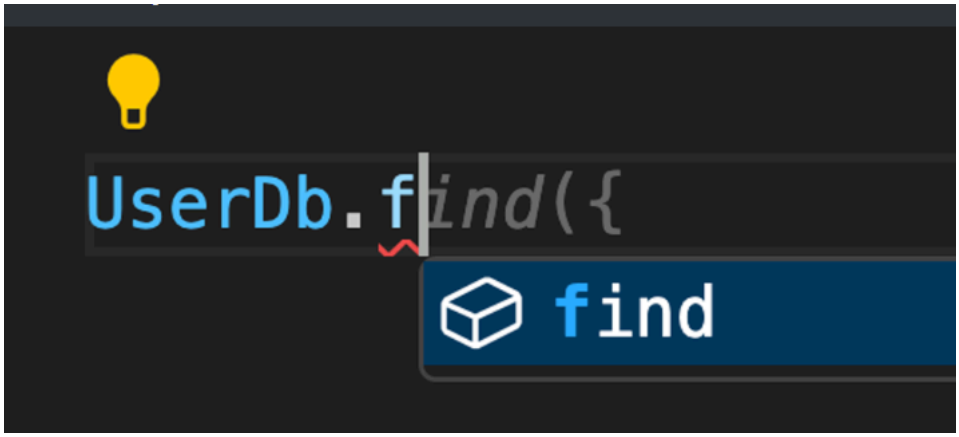
Prisma keep track of all the migrations

3. Type Safety

Prisma generates a type-safety database client based on the Prisma schema.



4. Auto-Completion



Installing prisma in a fresh app

Let's create a simple TODO app

1. npm init -y
2. npm install prisma typescript ts-node @types/node --save-dev

3. Initializing typescript

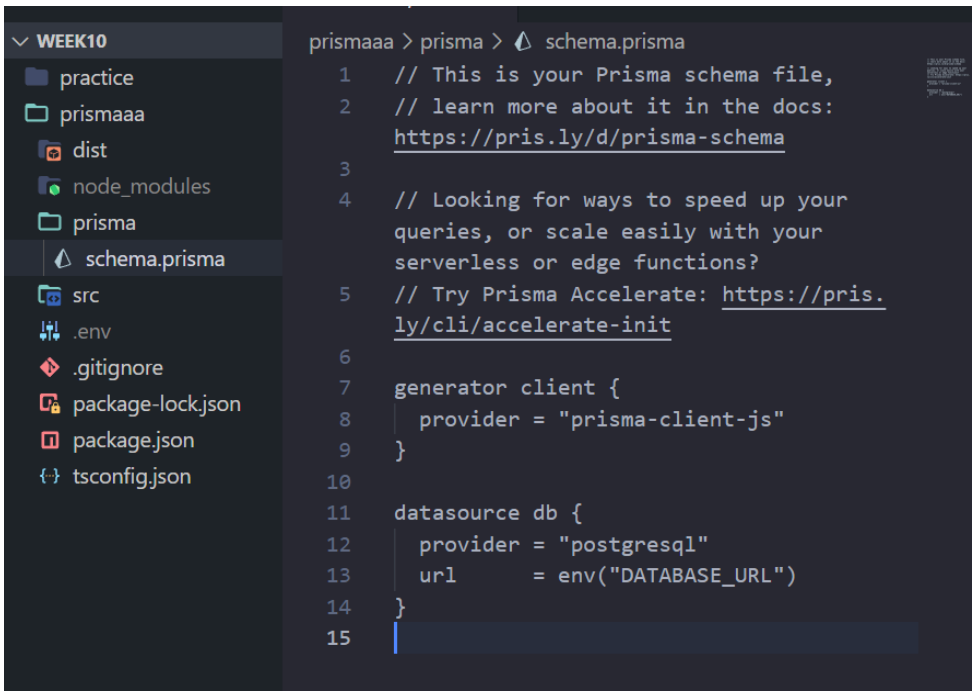
npx tsc --init

Change `rootDit` to `src`

Change `outDir` to `dist`

4. npx prisma init

```
Next steps:
1. Set the DATABASE_URL in the .env file to point to your existing data
base. If your database has no tables yet, read https://pris.ly/d/gettin
g-started
2. Set the provider of the datasource block in schema.prisma to match y
 mysql, sqlite, sqlserver, mongodb or cockroachdb.
3. Run prisma db pull to turn your database schema into a Prisma schema
.
4. Run prisma generate to generate the Prisma Client. You can then star
t querying your database.
More information in our documentation:
https://pris.ly/d/getting-started
```



Schema.prisma

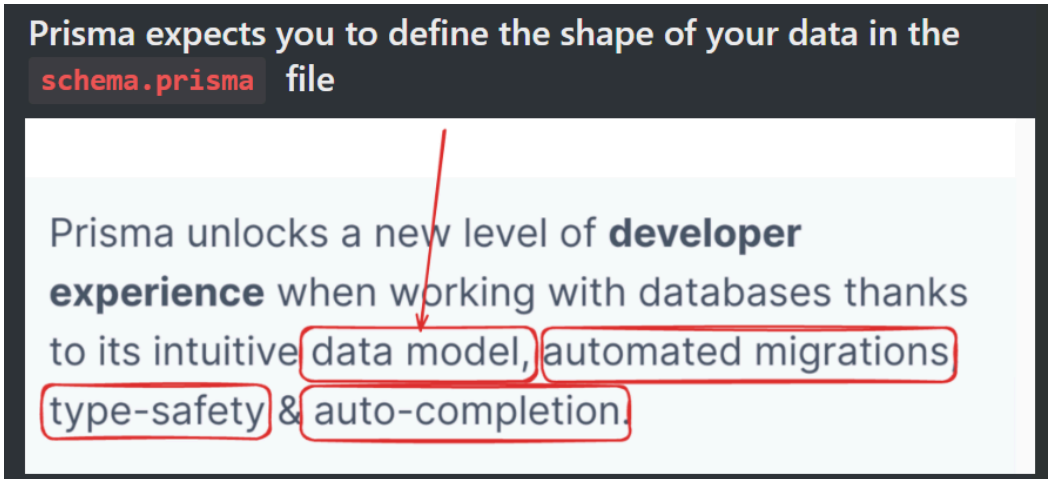
```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url
="postgresql://neondb_owner:tWcwiYRK8A3U@ep-fancy-pine-a5vggaa1.us-east-2.
aws.neon.tech/neondb?sslmode=require"
}
```

Selecting your database

Prisma lets you chose between a few databases (MySQL, Postgres, Mongo)

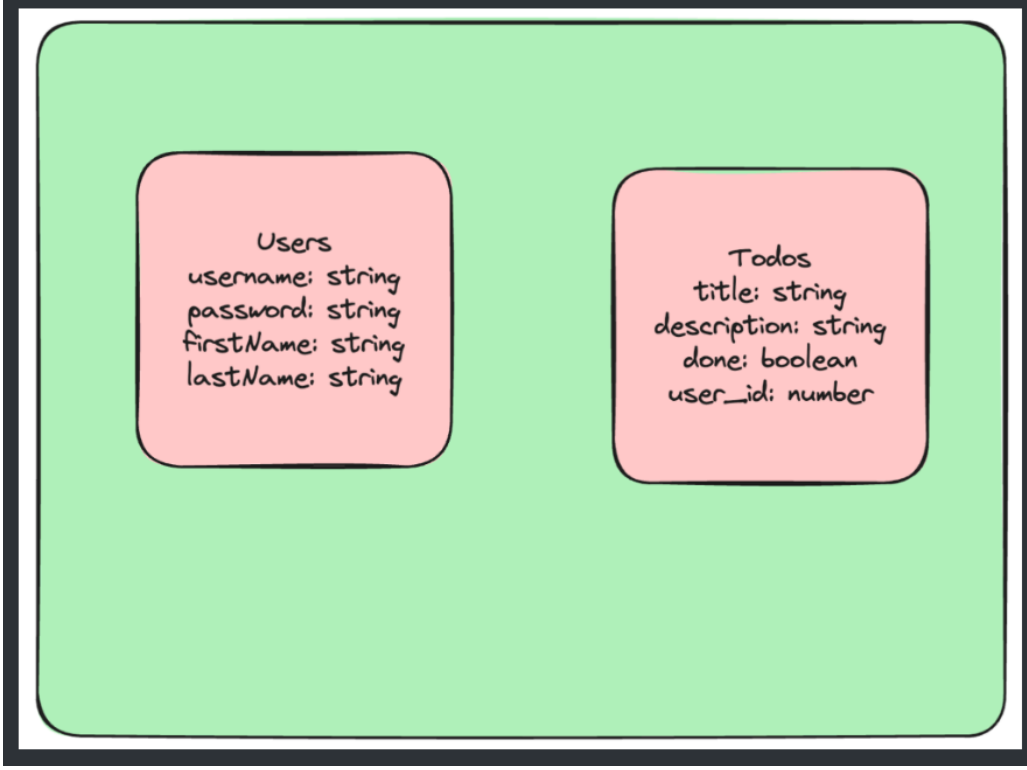
You can update prisma/schema.prisma to setup what database you want to use.

Defining you data model



Assignment

Add a Users and a Todo table in your application. Don't worry about `foreign keys` / `relationships` just yet



schema.prisma

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

datasource db {
  provider = "postgresql"
  url
="postgresql://neondb_owner:tWcwiYRK8A3U@ep-fancy-pine-a5vggaa1.us-east-2.
aws.neon.tech/neondb?sslmode=require"
}

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

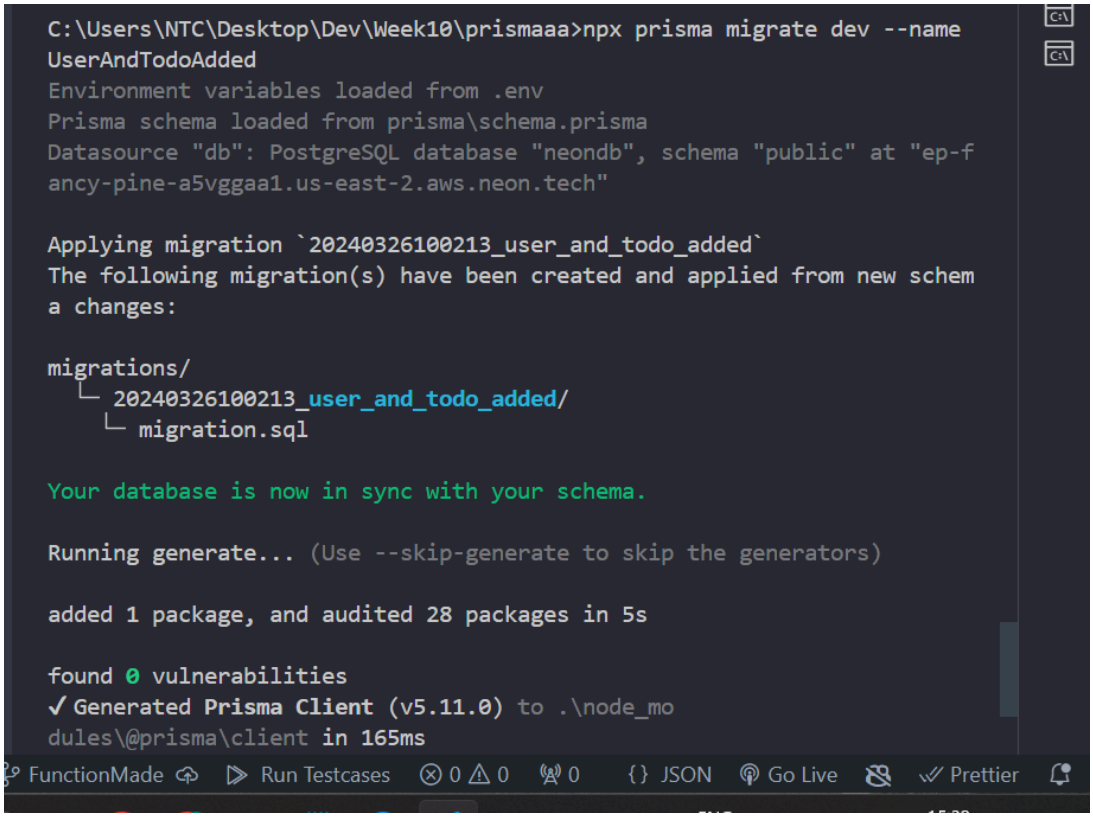
```
}
// THis is how we define model , in future we may have todo model which is
defined in the same way
model Todo{
  id Int @id @default(autoincrement())
  title String
  description String
  done Boolean @default(false)
  userId Int
}
```

This is most important bit , where we will spend our major time

Generate Migrations

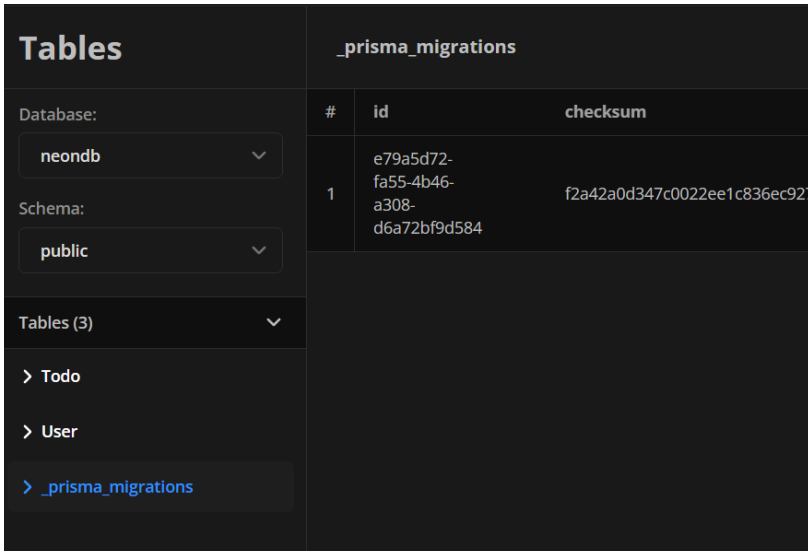
You have created a single schema file. You haven't yet run the **CREATE TABLE** commands. To run those and create **migration files** run

npx prisma migrate dev --name Initialize the schema



Exploring database

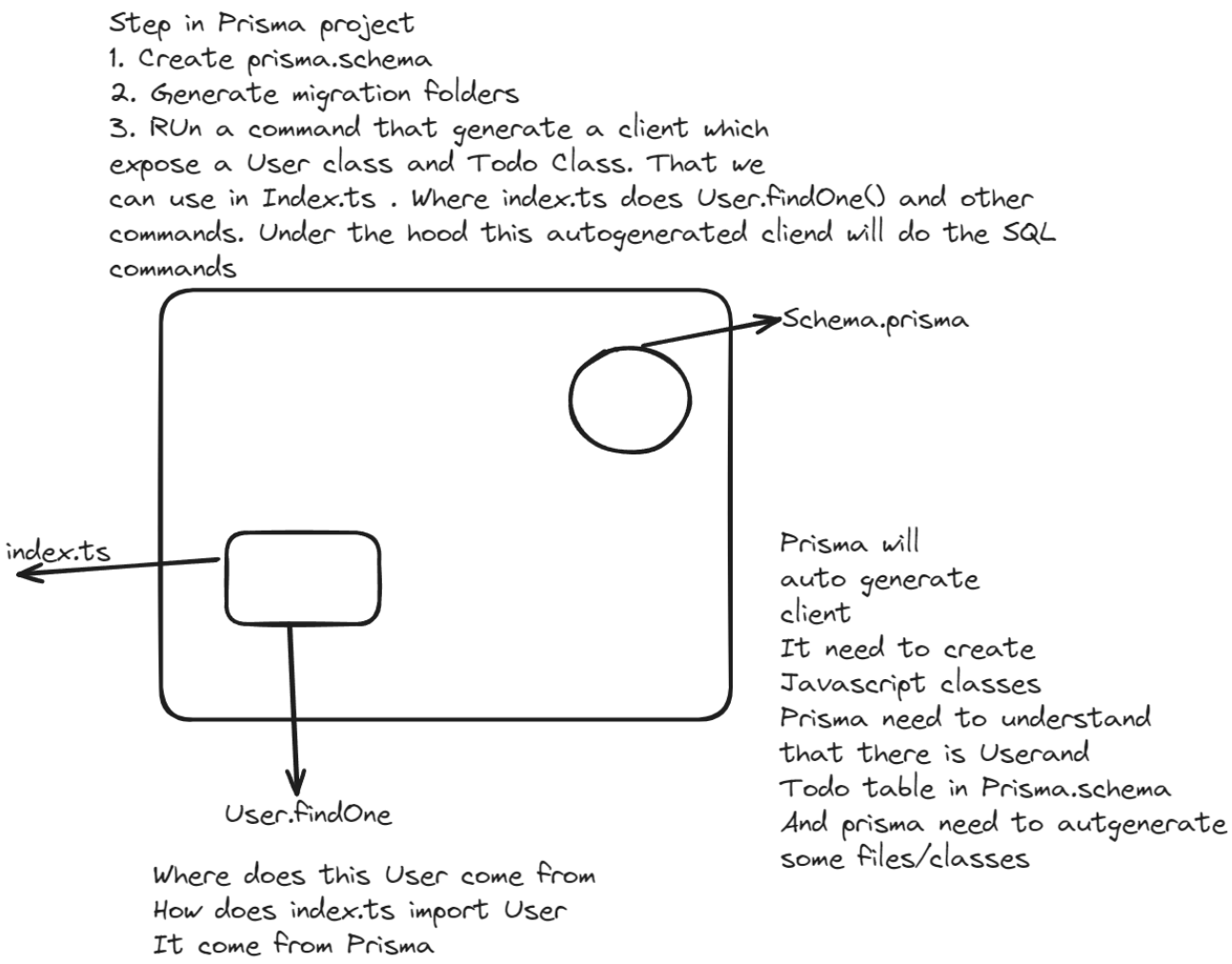
- Drift detected can be shown while we use prisma generate command it means that we have used the same database for the other project and our migration history is not matching up.
- docker exec -it my-postgres1 psql -U postgres **if we are running in docker**



Generating the prisma client

What is client??

autogenerated client??



Client represent all the function that convert

User.create({email: "harkirat@gmail.com"})

Wher does this user come from.

Answer is the autogenerated client

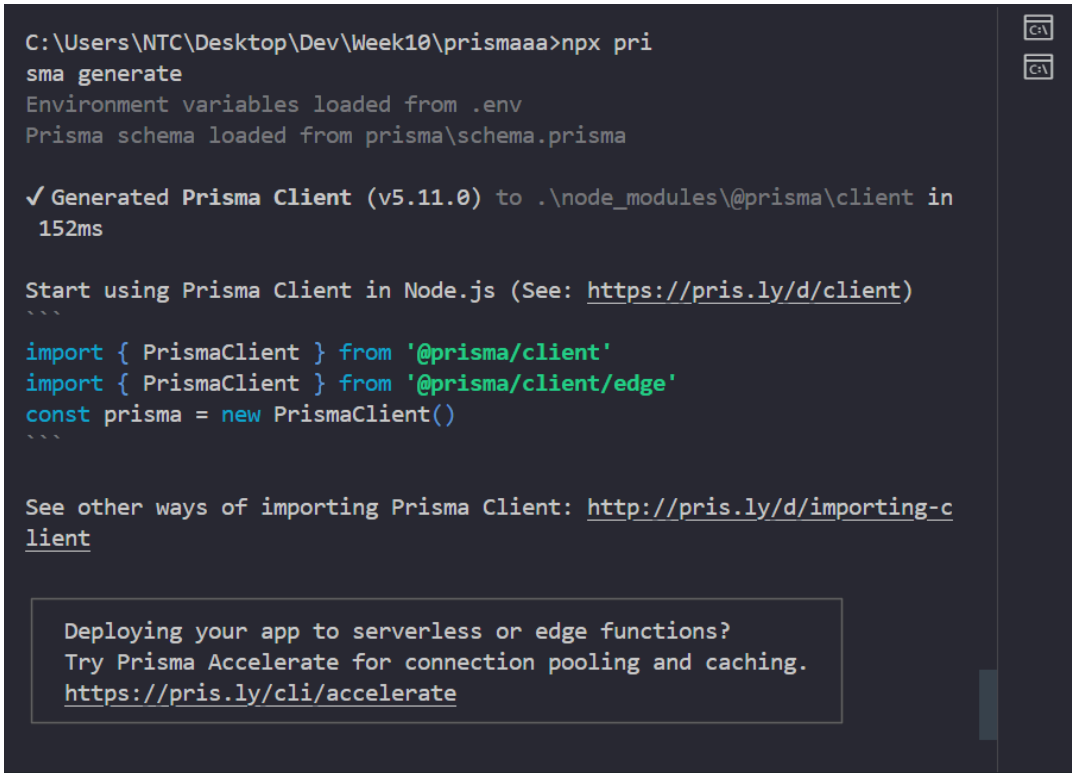
We want prisma to make a User class

Once you ve created the **prisma/schema.prisma** you can generate these **clients** that you can use in Node.js app.

How to generate the client??

npx prisma generate

This generate a new client



Anytime we change the schema.prisma file we need to do two steps

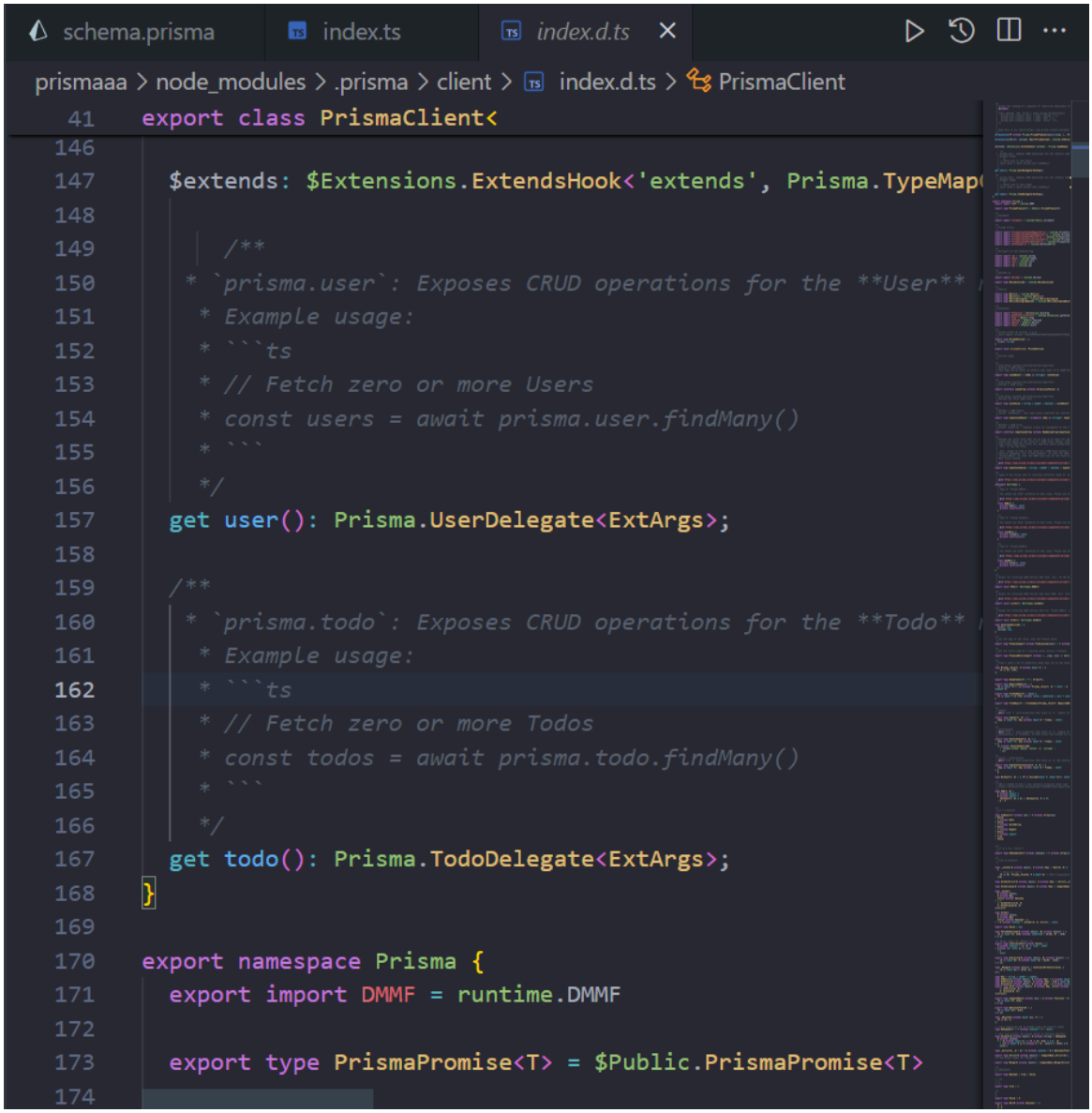
1. Create the migrations folder
2. Generate the Prisma client

src => index.ts

```
import { PrismaClient } from "@prisma/client";
const prisma = new PrismaClient();
// similar to import mongoose from mongoose and connecting it


prisma.user
// How did it know that our database had User??
// When we do npx prisma generate , some extra code got putten in
@prisma/client based on schema.prisma
```

```
// When we hover on it and click it it will take us to a file which
consist of all the info of user
```

Hovering on the User




```
// process of converting schema.prisma to this client is called the
autogenerated client
```

Creating your first app

Insert

Write a function that let's you insert data in the Users table.

Typescript will help you out, here's a starter code –

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function insertUser(username: string, password: string, firstName:
string, lastName: string) {

}
```

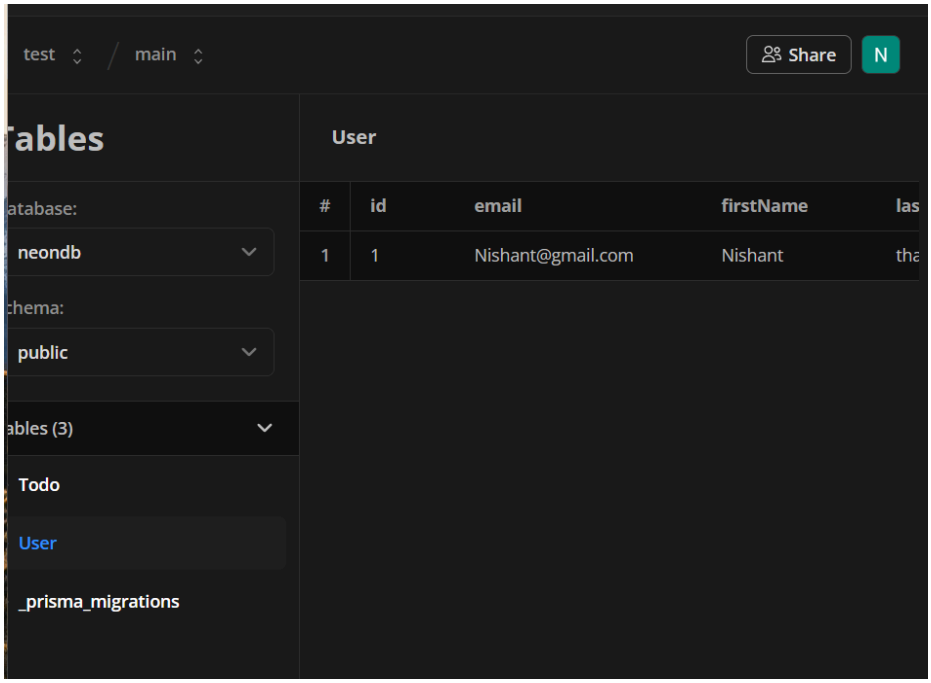
```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function insertUser(username: string, password: string, firstName:
string, lastName: string) {
//   ORM provide us a simple autogenerate client which can be used to
write the functions
    const res = await prisma.user.create({
    data:{
        email: username,
        password,
        firstName,
        lastName
    },
    // what should res contain , id yes password yes
    select:{
        id:true,
        password:true
    }
  })
  console.log(res);
}
insertUser("Nishant@gmail.com","password12","Nishant","thapu")
```

tsc -b

Then run the index.js file generated

```
C:\Users\NTC\Desktop\Dev\Week10\prismaaa>tsc -b

C:\Users\NTC\Desktop\Dev\Week10\prismaaa>node dist/index.js
{ id: 1, password: 'password12' }

C:\Users\NTC\Desktop\Dev\Week10\prismaaa>
```

Let's check our database



If we again run with same it will show error that user exist with the same email

Update

Write a function that let's you update data in the Users table.

Starter code –

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();
```

```
interface UpdateParams {
    firstName: string;
    lastName: string;
}

async function updateUser(username: string, {
    firstName,
    lastName
}: UpdateParams) {

}
```

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

// Whoever want to update should give the above information
interface UpdateParams {
    firstName: string;
    lastName: string;
}

async function updateUser(username: string, {
    firstName,
    lastName
}: UpdateParams) {
 const res = await prisma.user.update({
    where: {email: username},
    data: {
        firstName,
        lastName
    }
  })
  console.log(res);
}
updateUser('Nishant@gmail.com',{
    firstName:'Nishant111',
    lastName:'Thapa'
}).then(()=>{
    console.log("User updated")
```

```
}).catch((e) => {
    console.error(e)
}).finally(async()=>{
    await prisma.$disconnect()
    // disconnect from database
})
```

```
C:\Users\NTC\Desktop\Dev\Week10\prismaaa>node dist/index.js
{
  id: 1,
  email: 'Nishant@gmail.com',
  firstName: 'Nishant111',
  lastName: 'Thapa',
  password: 'password12'
}
User updated

C:\Users\NTC\Desktop\Dev\Week10\prismaaa>
```

Get a user's details

Write a function that let's you fetch the details of a user given their email

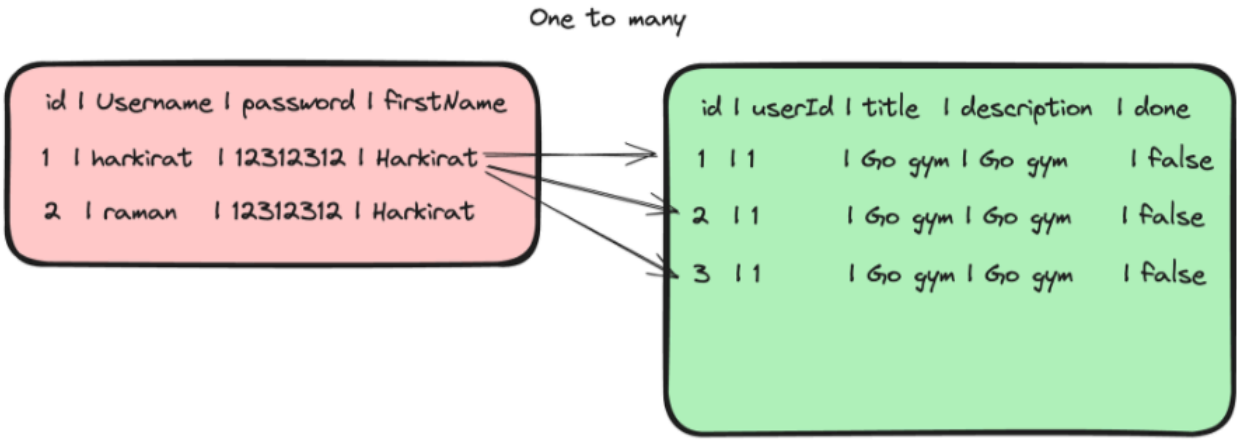
```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function getUser(username: string) {
  const user = await prisma.user.findFirst({
    where: {
        username: username
    }
  })
  console.log(user);
}

getUser("admin1");
```

Relationship

Types of Relationship:

1. One to One
2. One to Many
3. Many to Many
4. Many to one

Example For TODO app there is one to many Relationship



To store many to many relationship it is usual that we have third table which connect both tables.

```
model User {
  id          Int      @id @default(autoincrement())
  username    String   @unique
  password    String
  firstName   String
  lastName    String
  todos       Todo[]
}

model Todo {
  id          Int     @id @default(autoincrement())
  title       String
  description String
  done        Boolean @default(false)
  userId      Int
  user        User    @relation(fields: [userId], references: [id])
}
```

Todo functions:

1. createTodo

Starter code:

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function createTodo(userId: number, title: string, description:
string) {

}

createTodo(1, "go to gym", "go to gym and do 10 pushups");
```

Solution:

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();
```

```
async function createTodo(userId: number, title: string, description:
string) {
  const todo = await prisma.todo.create({
    data: {
      title,
      description,
      userId
    },
  });
  console.log(todo);

}

getUser(1, "go to gym", "go to gym and do 10 pushups");
```

2. getTodos

Starter code:

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function getTodos(userId: number, ) {

}

getTodos(1);
```

Solution:

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function getTodos(userId: number, ) {
    const todos = await prisma.todo.findMany({
        where: {
```

```
        userId: userId,
        },
    });
    console.log(todos);
}

getTodos(1);
```

3. getTodosAndUserDetails

Starter code:

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function getTodosAndUserDetails(userId: number, ) {

}

getTodosAndUserDetails(1);
```

Bad solution:

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function getTodosAndUserDetails(userId: number) {
    const user = await prisma.user.findUnique({
        where: {
            id: userId
        }
    });
    const todos = await prisma.todo.findMany({
        where: {
            userId: userId,
        }
    });
    console.log(todos);
```

```
    console.log(user)
}

getTodosAndUserDetails(1);
```

Good solution

```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function getTodosAndUserDetails(userId: number, ) {
    const todos = await prisma.todo.findMany({
        where: {
            userId: userId,
        },
        select: {
            user: true,
            title: true,
            description: true
        }
    });
    console.log(todos);
}

getTodosAndUserDetails(1);
```

Q/NA

1.