

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"]);
```

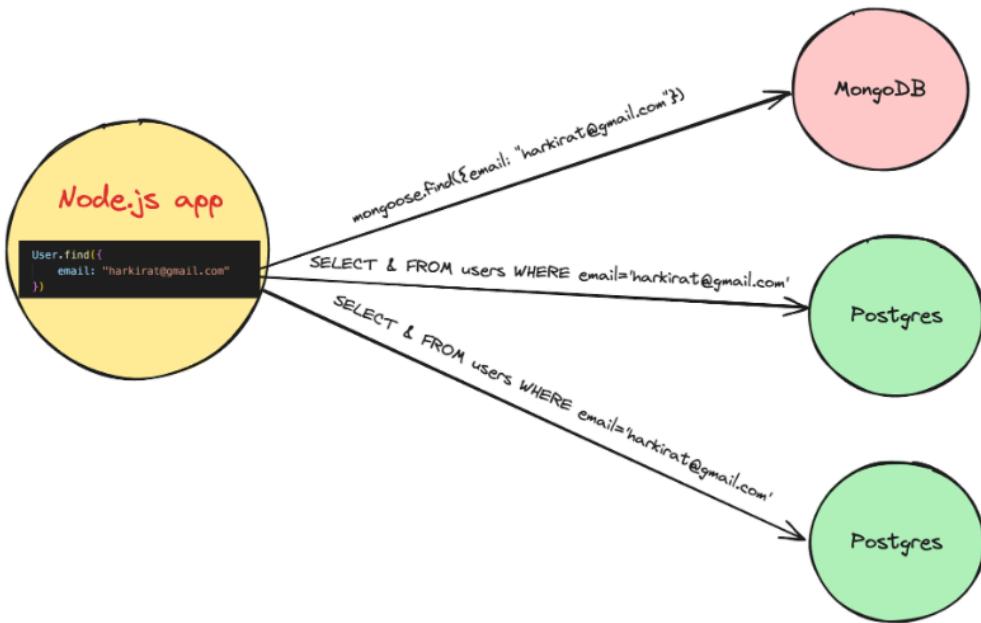
ORM

```
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 Postgres, 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.

Non ORM (pg)

```
const result: any
const result = await client.query(query, ["harkirat@gmail.com"]);
```

ORM

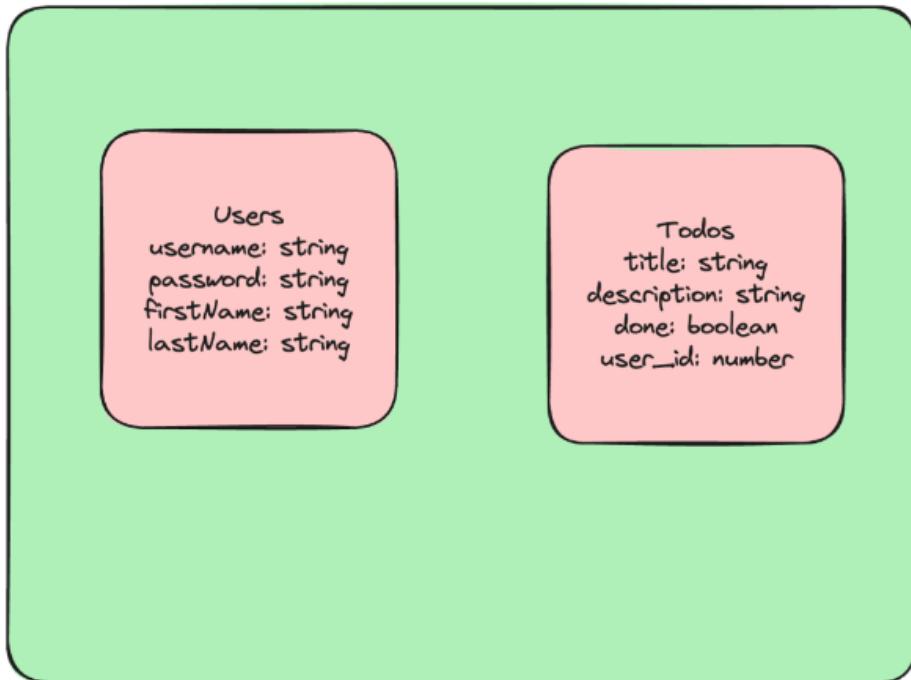
```
// Asy
const user: {
  email: string;
  username: string;
  password: String;
}

const user = UserDb.find({
  email: "harkirat@gmail.com"
})

const user = UserDb.find({
  email: "harkirat@gmail.com"
})
```

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 we know what tables we formed, migration is very long files, this specific 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

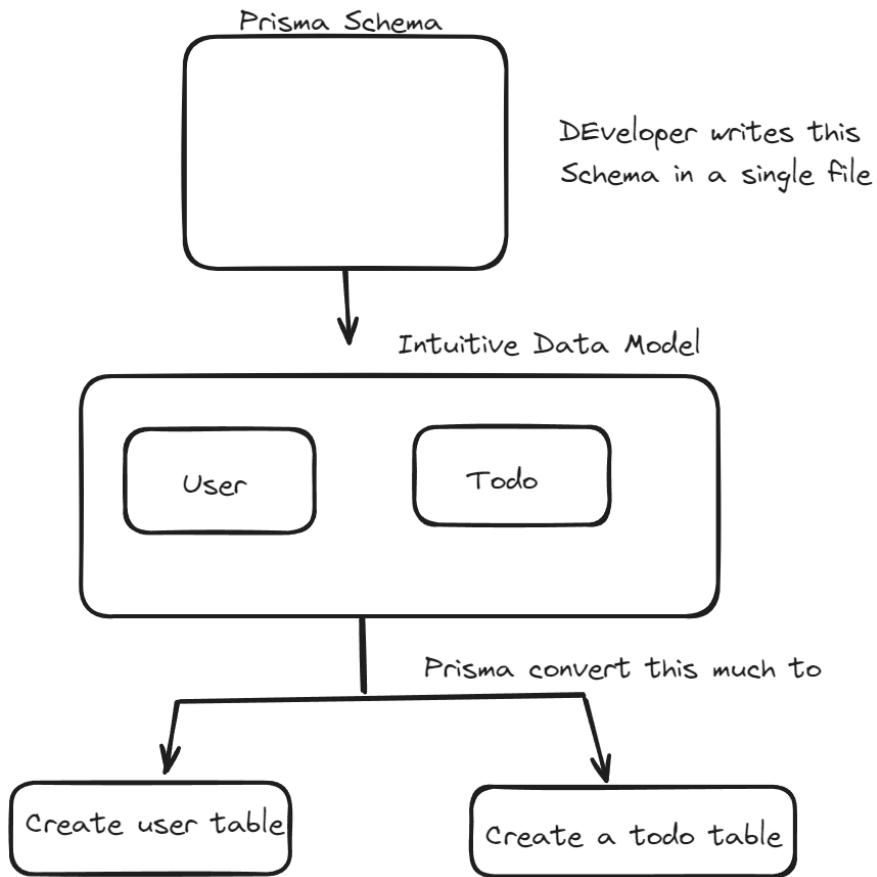
Next-generation Node.js and TypeScript ORM

Prisma unlocks a new level of **developer experience** when working with databases thanks to its intuitive **data model**, **automated migrations**, **type-safety** & **auto-completion**.

1. Data model

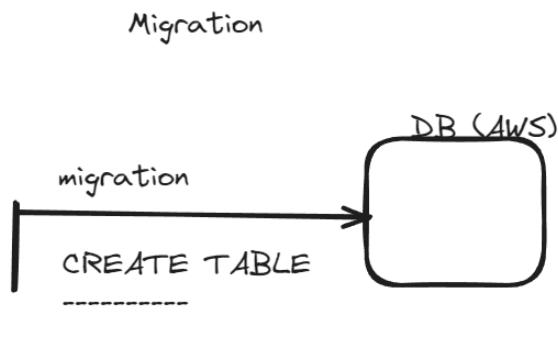
In a single file , define your schema. What it looks like what tables you haven what field each table has, how are rows related to each other.

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.

```
// Asy
const user: {
  email: string;
  username: string;
  password: String;
}

const user = UserDb.find({
  email: "harkirat@gmail.com"
})

const user = UserDb.find({
  email: "harkirat@gmail.com"
})
```

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 `rootDir` 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/getting-started>
2. Set the `provider` of the `datasource` block in `schema.prisma` to match your `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 start querying your database.

More information in our documentation:

<https://pris.ly/d/getting-started>

```
prismaaa > prisma > schema.prisma
  1 // This is your Prisma schema file,
  2 // learn more about it in the docs:
  3 // https://pris.ly/d/prisma-schema
  4 // Looking for ways to speed up your
  5 // queries, or scale easily with your
  6 // serverless or edge functions?
  7 // Try Prisma Accelerate: https://pris.
  8 // ly/cli/accelerate-init
  9
 10 generator client {
 11   provider = "prisma-client-js"
 12 }
 13
 14 datasource db {
 15   provider = "postgresql"
 16   url      = env("DATABASE_URL")
 17 }
```

Schema.prisma

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

datasource db {
  provider = "postgresql"
  url
  ="postgresql://neondb_owner:tWcwiYRK8A3U@ep-fancy-pine-a5vggaal.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.

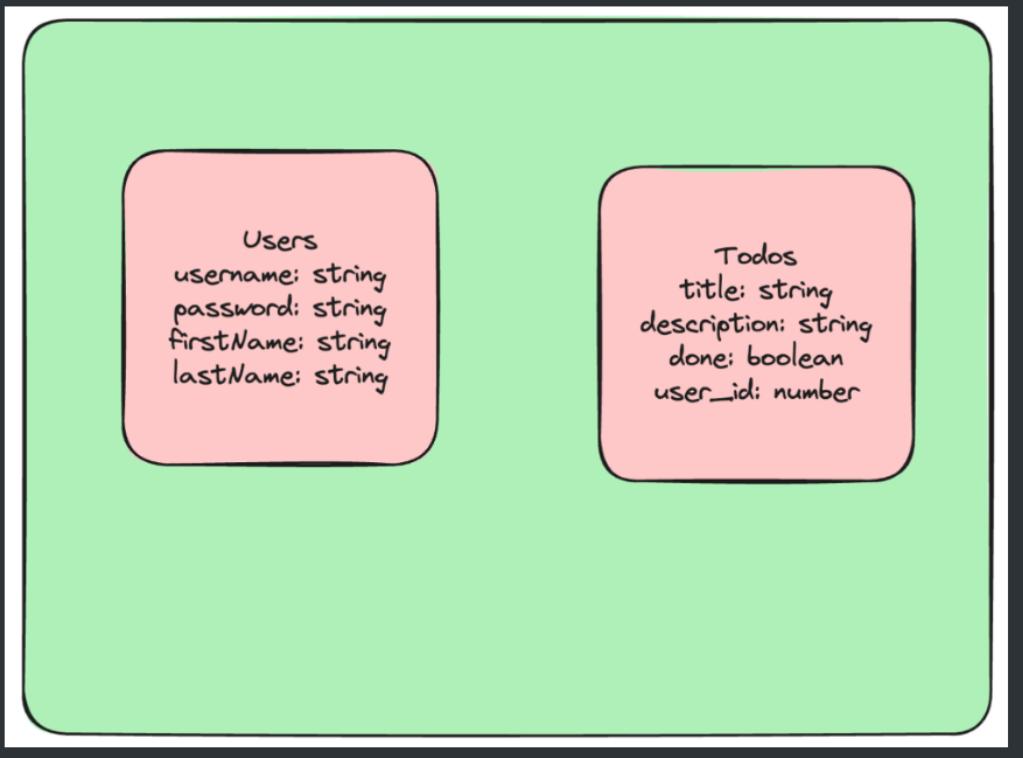
Defining your data model

Prisma expects you to define the shape of your data in the `schema.prisma` file

Prisma unlocks a new level of **developer experience** when working with databases thanks to its intuitive **data model**, **automated migrations**, **type-safety** & **auto-completion**.

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-a5vggaal.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

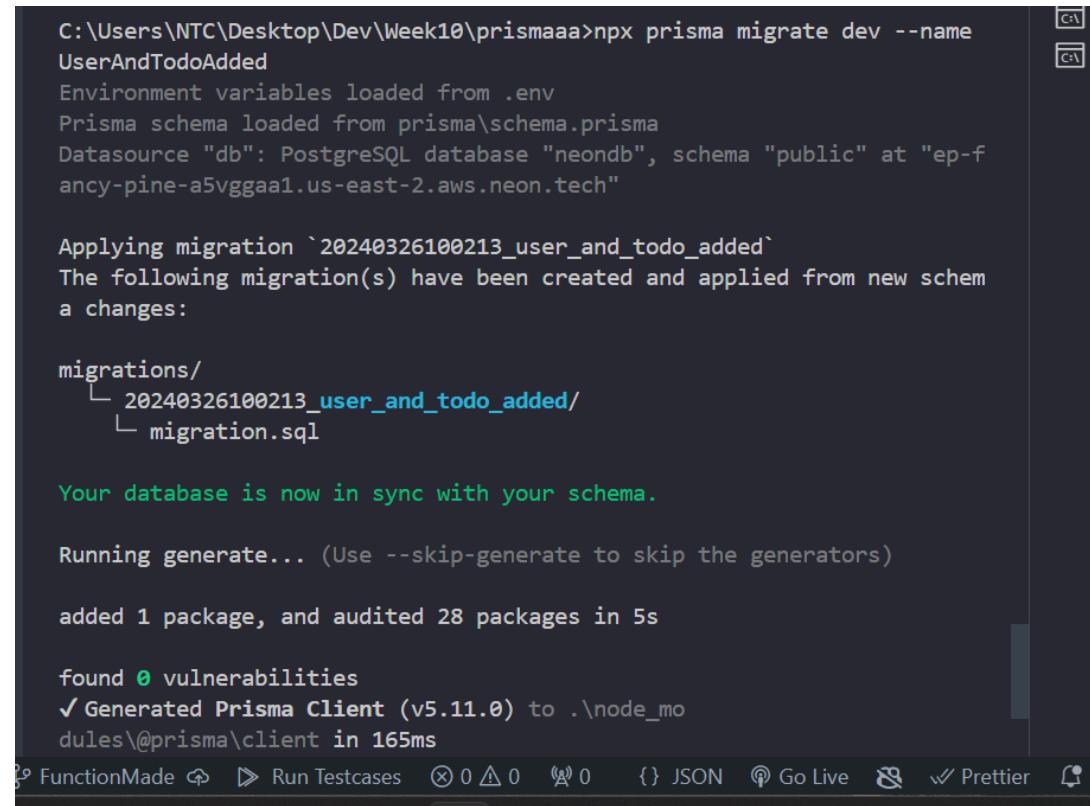
model Todo{
    id Int @id @default(autoincrement())
    title String
    description String
    done Boolean @default(false)
    userId Int
}

```

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



```

C:\Users\NTC\Desktop\Dev\Week10\prismaaa>npx prisma migrate dev --name
UserAndTodoAdded
Environment variables loaded from .env
Prisma schema loaded from prisma\schema.prisma
Datasource "db": PostgreSQL database "neondb", schema "public" at "ep-f
ancy-pine-a5vggaa1.us-east-2.aws.neon.tech"

Applying migration `20240326100213_user_and_todo_added`
The following migration(s) have been created and applied from new schem
a changes:

migrations/
└── 20240326100213_user_and_todo_added/
    └── migration.sql

Your database is now in sync with your schema.

Running generate... (Use --skip-generate to skip the generators)

added 1 package, and audited 28 packages in 5s

found 0 vulnerabilities
✓ Generated Prisma Client (v5.11.0) to .\node_mo
dules\@prisma\client in 165ms

```

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**

Tables		_prisma_migrations		
Database:	neondb	#	id	checksum
Schema:	public	1	e79a5d72-fa55-4b46-a308-d6a72bf9d584	f2a42a0d347c0022ee1c836ec92
Tables (3)				
> Todo				
> User				
> _prisma_migrations				

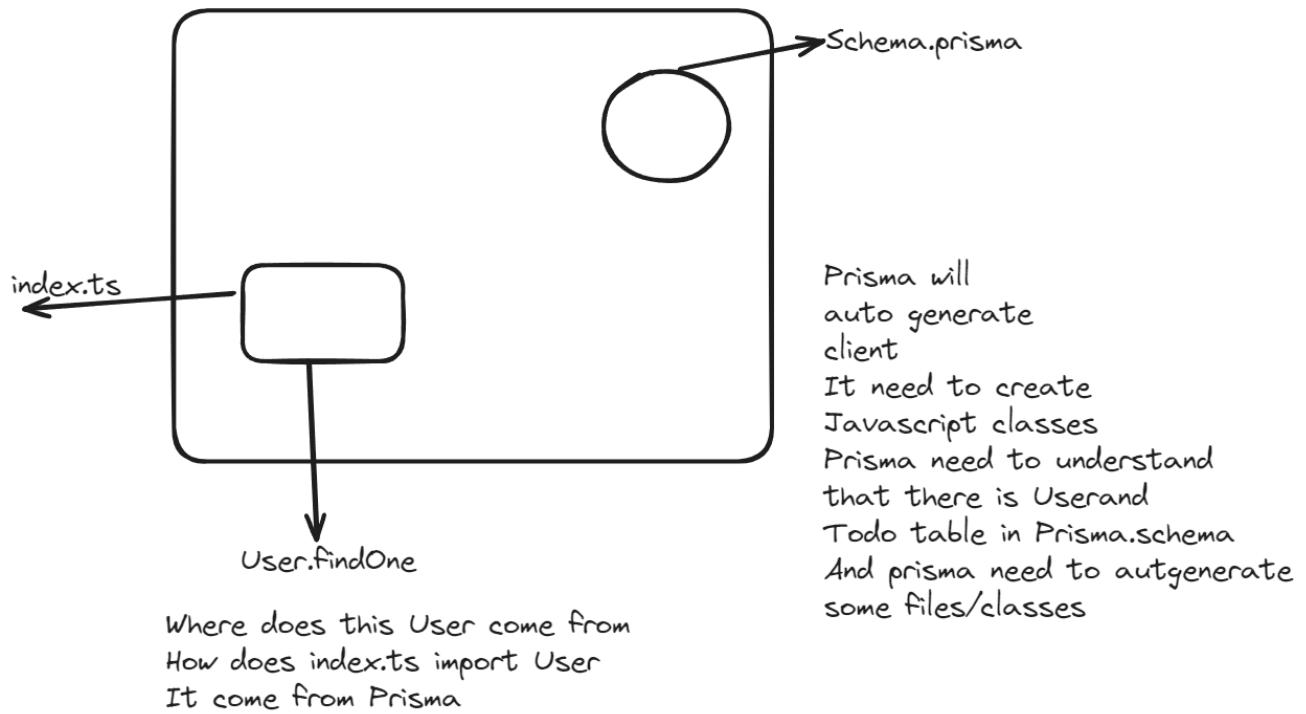
Generating the prisma client

What is client??

autogenerated client??

Step in Prisma project

1. Create prisma.schema
2. Generate migration folders
3. Run a command that generate a client which expose a User class and Todo Class. That we can use in Index.ts . Where index.ts does User.findOne() and other commands. Under the hood this autogenerated client will do the SQL commands



Client represent all the function that convert

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

Wher does this user come from.

Answer is the autogenerated client

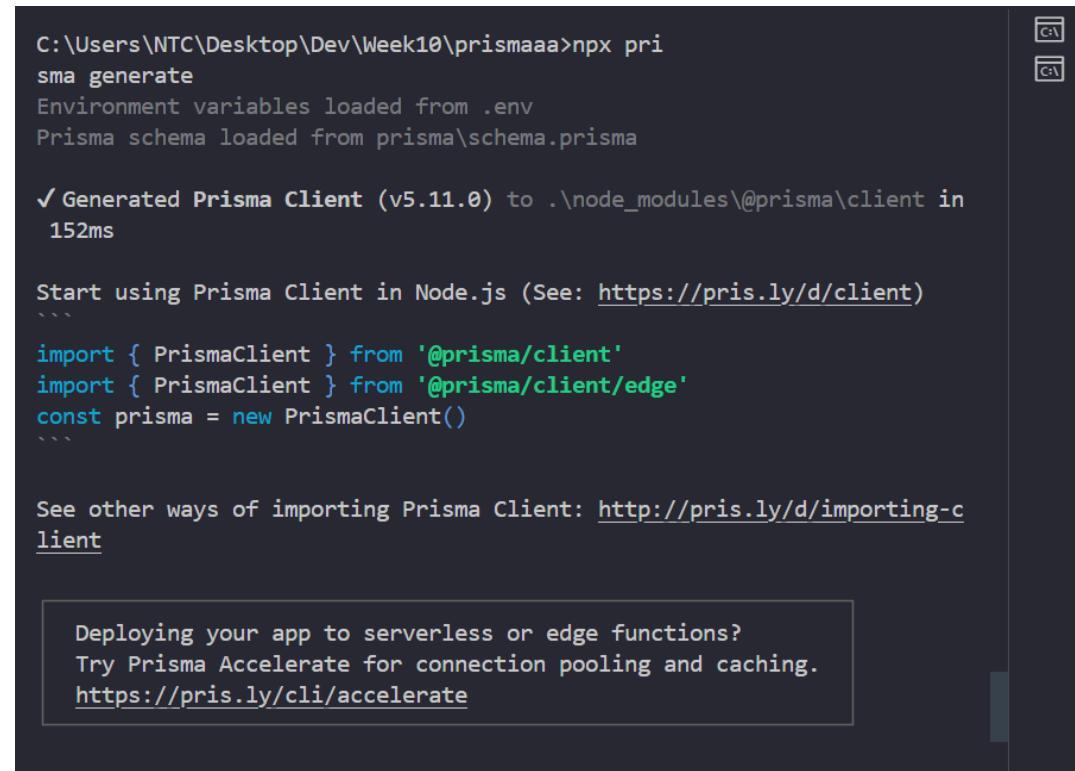
We want prisma to make a User class

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



```
C:\Users\NTC\Desktop\Dev\Week10\prismaaa>npx prisma generate
Environment variables loaded from .env
Prisma schema loaded from prisma\schema.prisma

✓ Generated Prisma Client (v5.11.0) to ./node_modules/@prisma/client in
152ms

Start using Prisma Client in Node.js (See: https://pris.ly/d/client)
```
import { PrismaClient } from '@prisma/client'
import { PrismaClient } from '@prisma/client/edge'
const prisma = new PrismaClient()
```

See other ways of importing Prisma Client: http://pris.ly/d/importing-client

Deploying your app to serverless or edge functions?
Try Prisma Accelerate for connection pooling and caching.
https://pris.ly/cli/accelerate
```

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

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

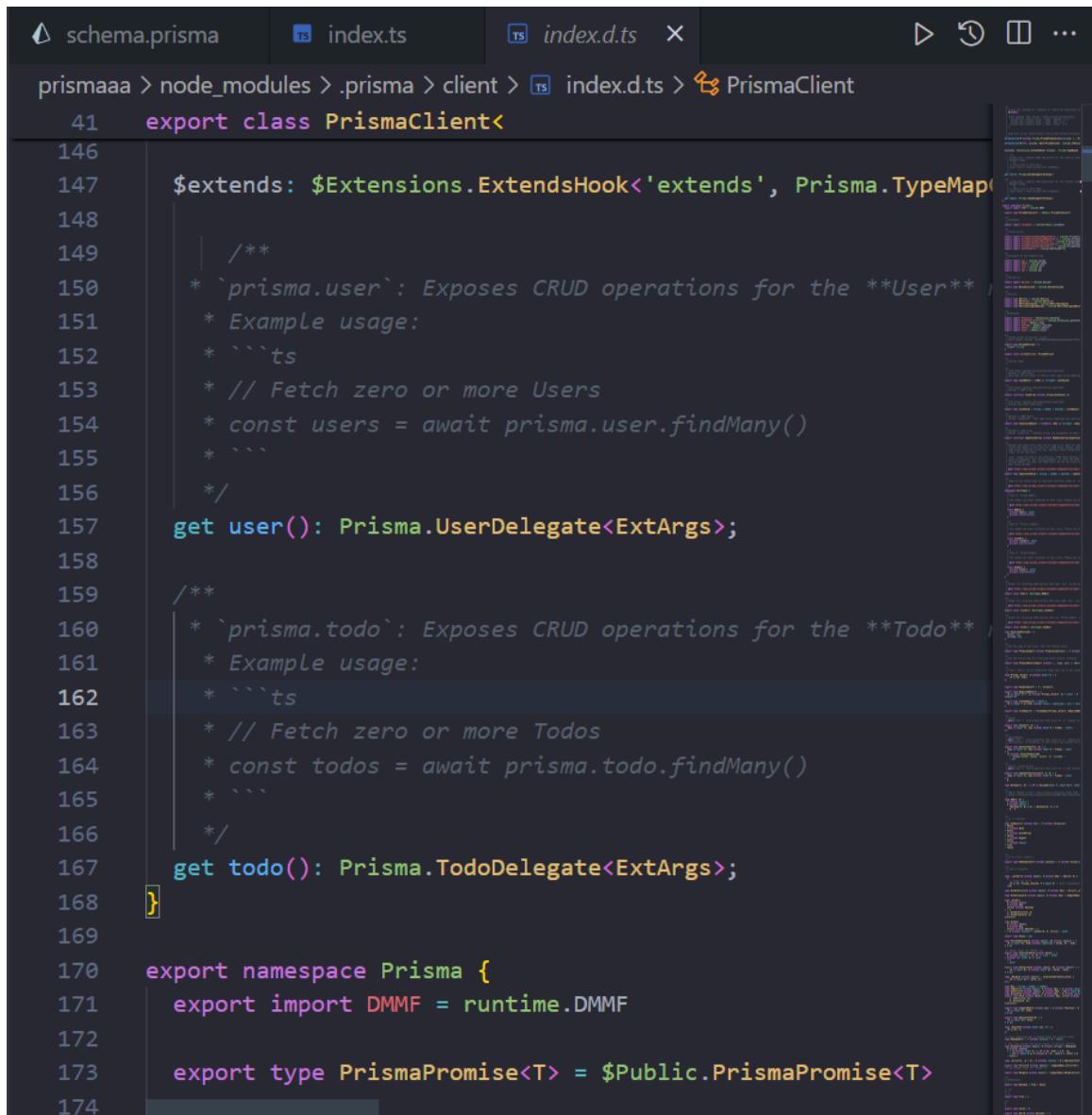
src => index.ts

```
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



```
schema.prisma      ts index.ts      ts index.d.ts X  
prismaaaa > node_modules > .prisma > client > ts index.d.ts > PrismaClient  
41  export class PrismaClient<  
146  
147    $extends: $Extensions.ExtendsHook<'extends', Prisma.TypeMap  
148  
149    /**  
150     * `prisma.user`: Exposes CRUD operations for the **User** type.  
151     * Example usage:  
152     * ``ts  
153     * // Fetch zero or more Users  
154     * const users = await prisma.user.findMany()  
155     * ``  
156     */  
157   get user(): Prisma.UserDelegate<ExtArgs>;  
158  
159   /**  
160    * `prisma.todo`: Exposes CRUD operations for the **Todo** type.  
161    * Example usage:  
162    * ``ts  
163    * // Fetch zero or more Todos  
164    * const todos = await prisma.todo.findMany()  
165    * ``  
166    */  
167   get todo(): Prisma.TodoDelegate<ExtArgs>;  
168 }  
169  
170 export namespace Prisma {  
171   export import DMMF = runtime.DMMF  
172  
173   export type PrismaPromise<T> = $Public.PrismaPromise<T>  
174 }
```

```
// 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

The screenshot shows the Prisma Studio interface. On the left, there's a sidebar titled 'Tables' with dropdown menus for 'database' (set to 'neondb'), 'schema' (set to 'public'), and 'tables (3)' (listing 'Todo', 'User', and '_prisma_migrations'). The main area is titled 'User' and shows a table with one row:

#	id	email	firstName	lastName
1	1	Nishant@gmail.com	Nishant	Patel

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);
}

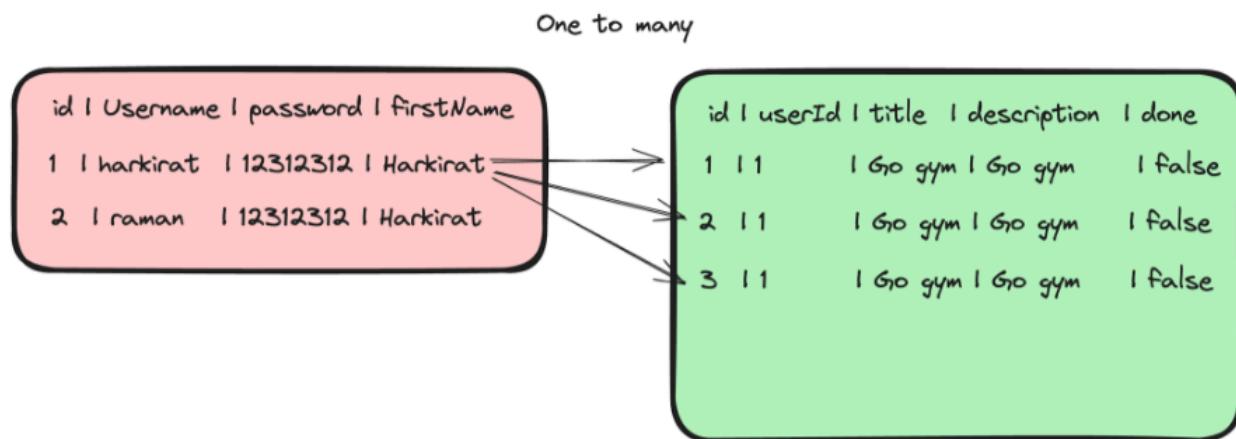
getUser("admin1");
```

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.