Cloudfare for servers or Amazon dynamodb they do everything themselves scales well and expensive also

Firebase also good

ORMS (Object Relational Mapping)

1. Boring official defination

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

2. Easier to digest defination

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

Step 2 - Why ORMs?

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

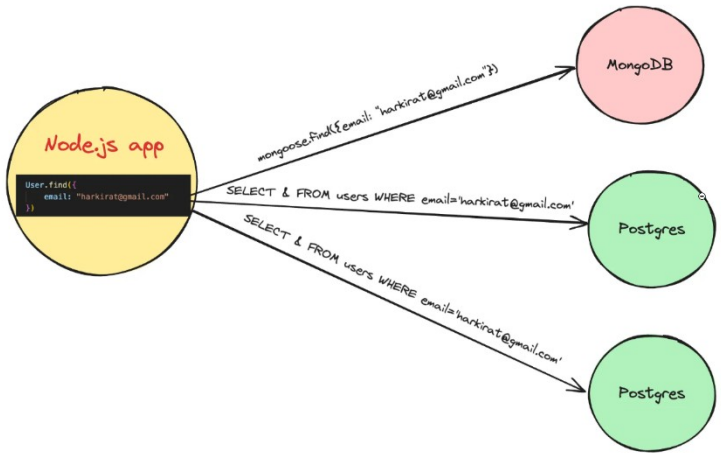
Its like syntax is easier

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"
})
```



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

Helps to migrate from the db easier RARELY HAPPENS

3. Type safety/Auto completion

Non ORM (pg)

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

ORM

```
const user: {
    email: string;
    username: string;
    password: String;
}
const user = UserDb.find({
    email: "harkirat@gmail.com"
})
// Asy
const user = UserDb.find({
    email: "harkirat@gmail.com"
})
```

In the orm there is a auto complete as the the type is defined and in non orm the type was not defined so its better

4. Automatic migrations

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

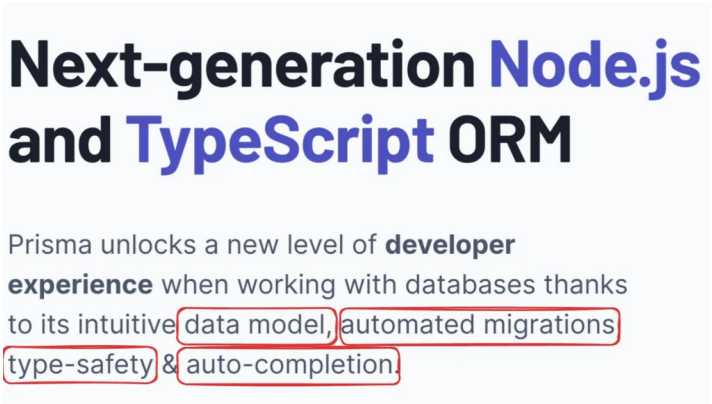
As your app grows, you will have a lot of these CREATE and ALTER commands.

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

IN THIS THERE IS A ISSUE IF LIKE YOU ARE RUNNING ANY ALTER COMMANDS AND ALL EARLIER THERE WE NEVER USED TO STORE THE LOGS LIKE WE ALTERED THIS OR THAT, SO THAT CAN BE FIXED USING THE PRISMA AS IT MAINTAIN THE LOGS WHAT ALTER COMMANDS WE DID ON THE DATABASE AND TO RUN THE PROJECT WE HAVE TO RUN THOSE MIGRATIONS AND THEN ONLY THE PROJECT WILL WORK

There are many more orms such as Drizzle, Prisma etc

What is Prisma????



1. Data model

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

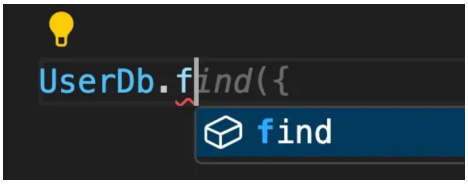
2. Automated migrations

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

3. Type Safety

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

4. Auto-Completion



https://www.prisma.io/docs/prisma-orm/quickstart/postgresql#6-create-and-apply-your-first-migrationuse this

npm install prisma

npx prisma init

schema.prisma contains many things such as models etc but some more things

prisma can be used with mongodb, mysql, postgress etc

it has the provider which we are using is postgress and url we will take from neon.tech which gives us a free sql db

also move the prisma.config.ts to the src folder

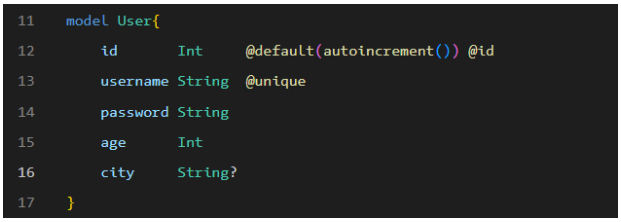
and then add the database url in the env file

npx prisma migrate dev to create the migration file and give the name as initialize

and if the tsconfig make problems then use

add this at the end of tsconfig.json

This is like we have to write the data in this



? at the end means it can be null also

GOOD QUESTION IS THINK IF SOMEONE WANT TO ADD A NEW FIELD LATER BUT FOR THAT TABLE ALREADY 10 ROWS EXIST THEN WHAT TO DO WITH THE OTHER ROWS THEN?

You can like define is like this

time DateTime? Or time DateTime @default(now())

Generating the Prisma Client (Autogenerate Clients)

npx prisma generate

alt click on the username while entering data to know the structure of the table

```
import { prisma } from "./lib/prisma";



async function createUser(){
    await prisma.user.create({
        data:{
            username:"pallab1",
            password: "123123121",
            age:123,
            city:"namaste1"
        }
    })

}
```

Setup the whole things using the docs and then work

the side code shows how to insert the data into the db

```
async function createUser(){
    await prisma.user.delete({
        where:{
            id:1
        }
    })

}
```

```
async function createUser(){
    await prisma.user.update({
        where:{
            id:2
        },
        data:{
            username:"harkirat"
        }
    })

}
```

to find a user you can use findOne , findFirst, findMany

Main thing for prisma is the auto suggestions it gives based on the db we made

```
async function createUser(){
    const user = await prisma.user.findFirst({
        where:{
            id:2
        },
        select:{
            username:true
        }
    })
    console.log(user?.password)
}
```

To select and print specific thing we use select

And to print it we can do like user.password it gives auto suggestions also

Can also return null if the user is not present in the db

Pion, mediasoup, livekit for like live streaming shit bcrypt , argon2

Relationships (to relate tables with each other)

npx tsx script.ts

npx prisma migrate dev

1. One to Many: Its like adding a foreign key like there is a user table and a Todo table and one user can have many todos but one todo cannot exist without a user

```
async function getUser(){
    const user = await prisma.user.findFirst({
        where:{
            id:2
        },
        include:{
            todos:true
        }
    })
    console.log(user)
}

getUser();
```

also you can return the data from other table also with same id

```
prisma > TS seed.ts > createDummyUsers > user > data > todos > create > done
import { PrismaClient } from "@prisma/client"

const client = new PrismaClient();

async function createDummyUsers() {
    let user = await client.user.create({
        data: {
            username: "harkirat",
            age: 21,
            password: "123123",
            city: "Delhi",
            todos: {
                create: {
                    description: "Go to gym",
                    title: "Gym",
                    done: false
                }
            }
        }
    })
}

createDummyUsers()
```

YOU CAN ALSO CREATE A SEED FILE WHICH WILL CONTAIN A DUMMY DATA SUCH AS DUMMY COURSES, DUMMY USERS ETC IN THE SCHEMA FOLDER ONLY

SOMETHING LIKE THE SIDE THING

TO USE THEY HAVE TO RUN

npx prisma db seed and need to add some dependencies and code also

this is called **SEEDING** for the user

In **Prisma ORM**, the terms **client** and **adapter** are related but serve different roles in how Prisma talks to your database. I'll explain them in a simple way.

1️⃣ Prisma Client

Prisma Client is the **auto-generated database query library** you use in your code.

When you run:

```
npx prisma generate
```

Prisma reads your **schema.prisma** and generates **Prisma Client**.

You then use it like this:

```
import { PrismaClient } from '@prisma/client'

const prisma = new PrismaClient()

const users = await prisma.user.findMany()
```

What Prisma Client does

It provides **typed functions to query your database**.

Example queries:

```
await prisma.user.create()

await prisma.user.findMany()

await prisma.post.update()

await prisma.user.delete()
```

So think of it like:

Prisma Client = API for your database

Your Code

↓

Prisma Client

↓

Database
✔ Type-safe queries
✔ Auto-complete
✔ No raw SQL needed (optional)

2️⃣ **Prisma Adapter**

An **adapter** is the **connector that lets Prisma Client talk to a specific database driver or environment**.

It defines **how Prisma connects to the database internally**.

Adapters are used especially when:

- Using **edge runtimes**
- Using **serverless databases**
- Using **custom drivers**

Example adapters:

- **Prisma Neon Adapter** → for **Neon**
- **Prisma PlanetScale Adapter** → for **PlanetScale**

Example usage:

```
import { PrismaClient } from '@prisma/client'
import { PrismaNeon } from '@prisma/adapter-neon'
```

```
const adapter = new PrismaNeon({ connectionString })
const prisma = new PrismaClient({
  adapter
})
```

What the adapter does

```
Your Code
   ↓
Prisma Client
   ↓
Adapter
   ↓
Database Driver
   ↓
Database
```

It basically says:

> "Use this driver/connection method to talk to the database."