



Step 1 - What are ORMs





Step 2 – Why ORMs?

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

Non ORM

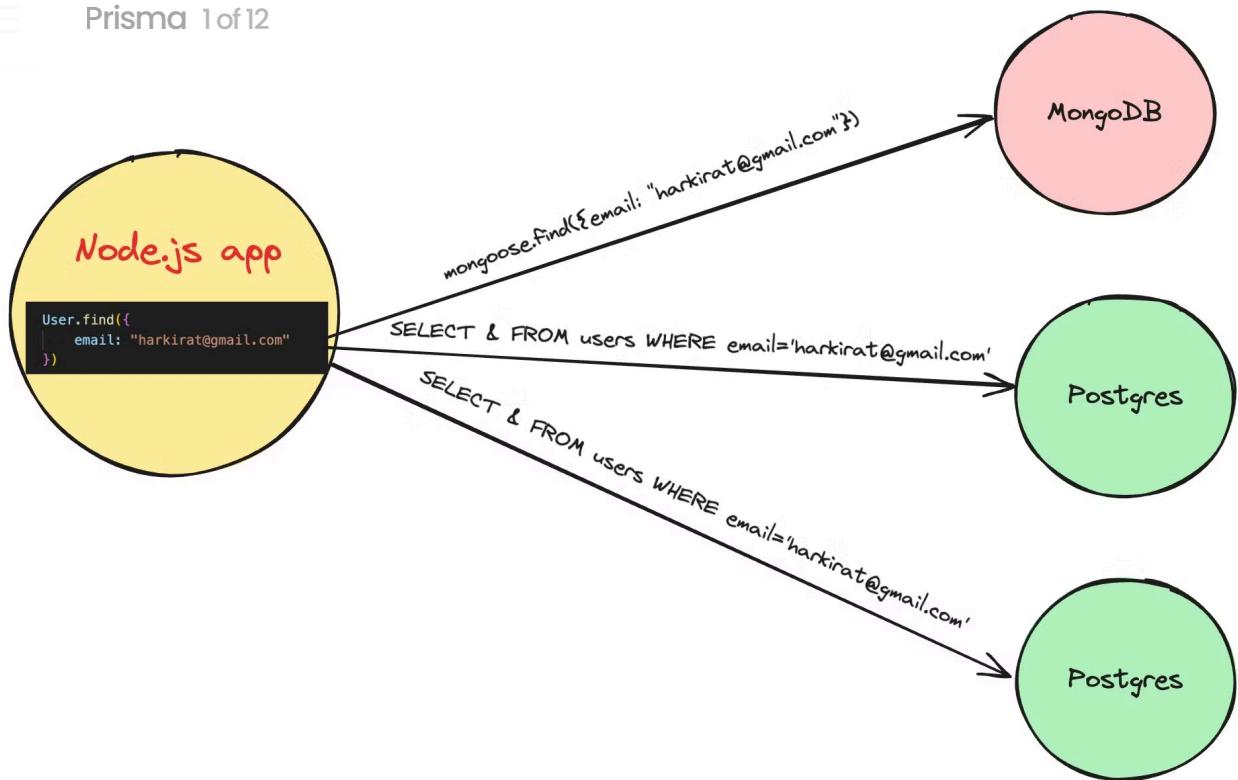
```
const query = 'SELECT * FROM users WHERE email = $1';
const result = await client.query(query, ['harkirat@gmail.com']);
```

ORM

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

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

Prisma 1 of 12



3. Type safety/Auto completion

Non ORM (pg)

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

ORM

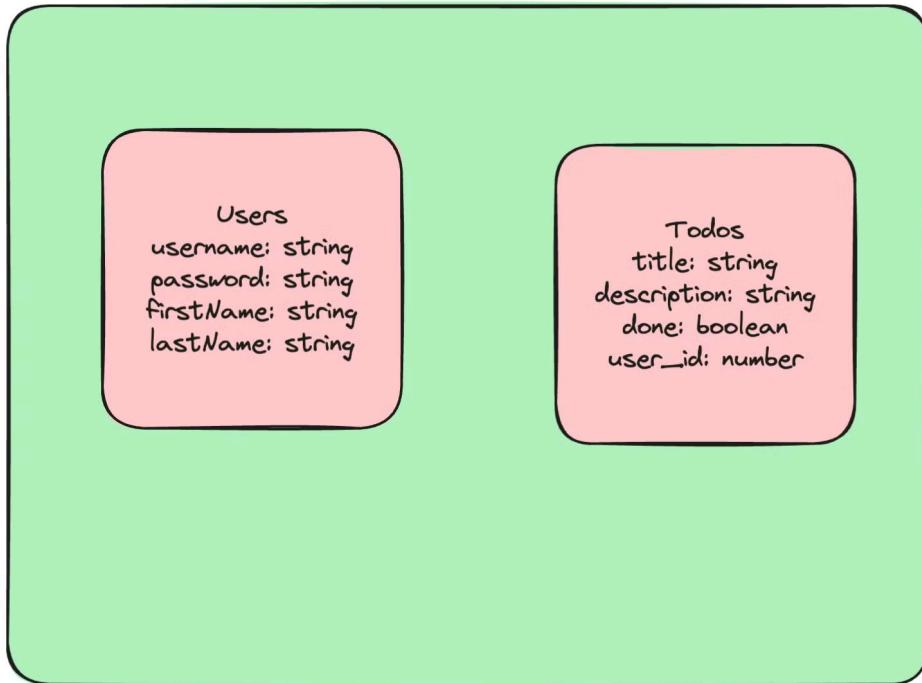
```
// Asy
const user: {
  email: string;
  username: string;
  password: String;
}

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

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

4 Automatic migrations

In case of a simple Postgres app, it's very hard to keep track of all the changes 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
```

Step 3 – What is Prisma

Prisma - 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 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.

The screenshot shows a code editor interface. On the left, there's a sidebar with the text "Prisma 1 of 12". The main area contains the following code:

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

Step 4 – Installing prisma in a fresh app

Let's create a simple TODO app

1. Initialize an empty Node.js project

```
npm init -y
```

1. Add dependencies

```
npm install prisma typescript ts-node --save-dev
```

```
npx tscc --init  
Prisma 1 of 12 `outDir` to `src`  
Change `outDir` to `dist`
```

1. Initialize a fresh prisma project

```
npx prisma init
```

```
✓ Your Prisma schema was created at prisma/schema.prisma  
You can now open it in your favorite editor.  
  
Next steps:  
1. Set the DATABASE_URL in the .env file to point to your existing database. 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 database: postgresql, 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
```

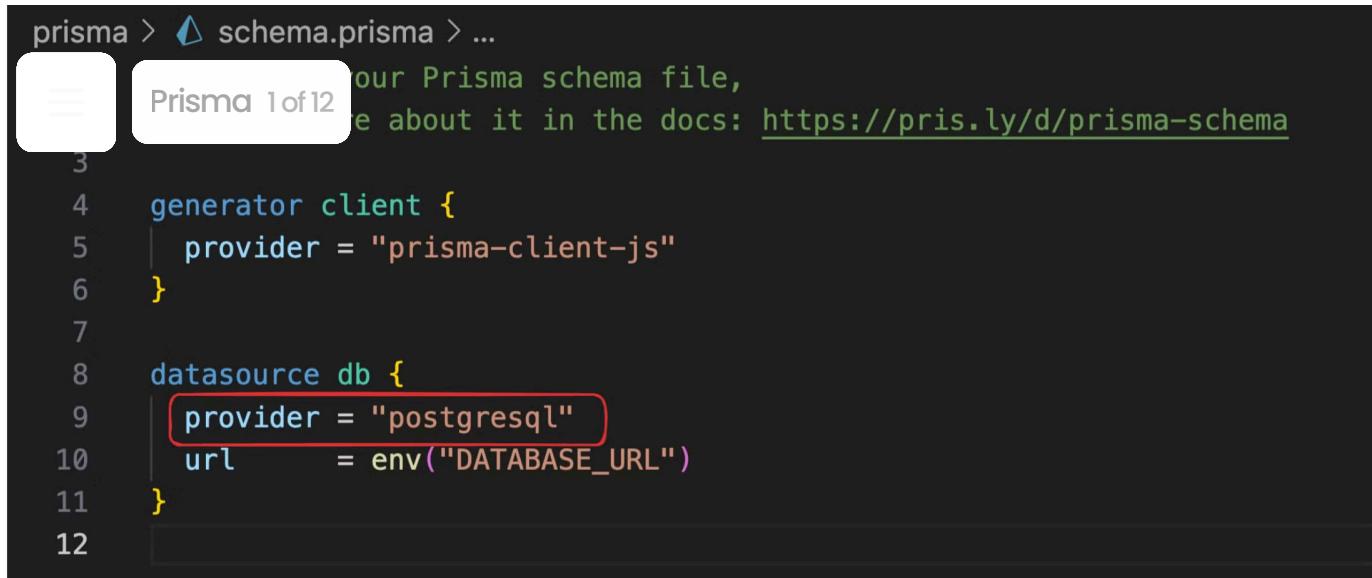
Step 5 – Selecting your database

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.



Also replace the database url with your test url for now

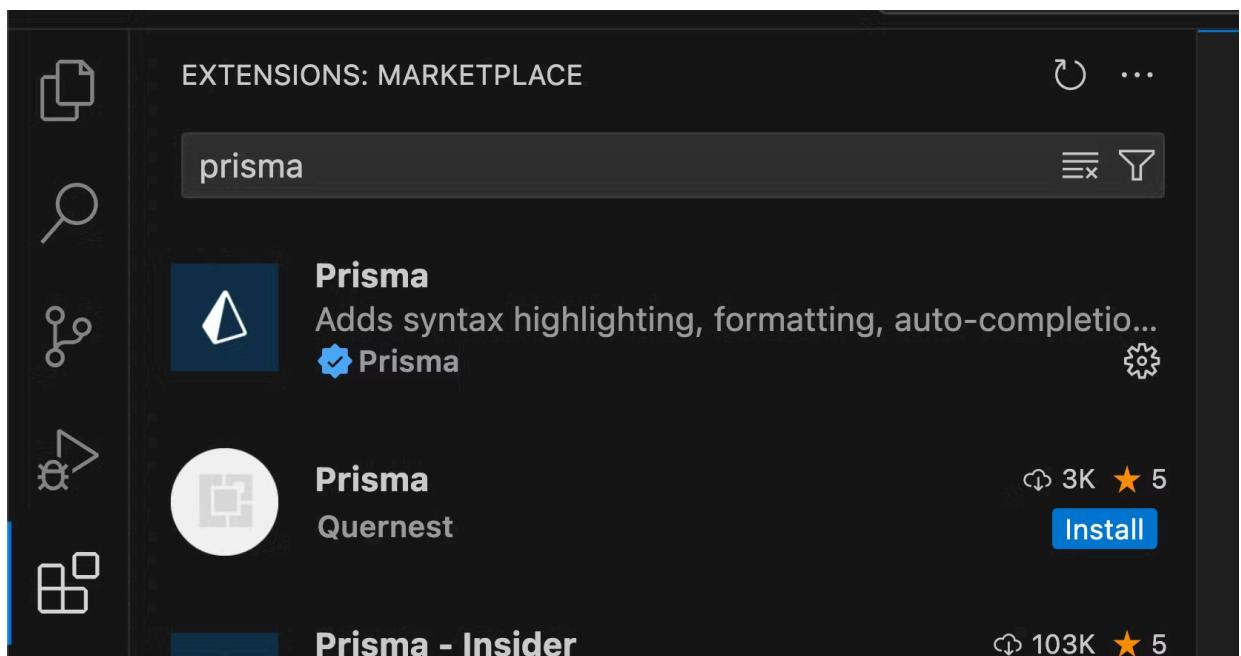


prisma > **schema.prisma** > ...

Your Prisma schema file, 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 = "postgresql"
10  url      = env("DATABASE_URL")
11 }
12 
```

💡 Good to have the VSCode extension that lets you visualise prisma better



Step 6 – 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**.

If your final app will have a Users table, it would look like this in the `schema.prisma` file

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

Assignment

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

Prisma 1 of 12

Users

```
username: string  
password: string  
firstName: string  
lastName: string
```

Todos

```
title: string  
description: string  
done: boolean  
user_id: number
```

▼ Answer

```
// 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 = "postgresql"  
  url    = env("DATABASE_URL")  
}
```

```
model User {  
  id    Int    @id @default(autoincrement())  
  username String @unique  
  password String  
  
  }  
}
```

```
| Prisma 1 of 12 @id @default(autoincrement())  
title String  
description String  
done Boolean @default(false)  
userId Int  
}
```

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
```

Your DB should now have the updated schema.

Step 7 – Exploring your database

If you have `psql`, try to explore the tables that `prisma` created for you.

```
psql -h localhost -d postgres -U postgres
```

```
postgres=# \dt;
          Prisma 1of12   List of relations
 Schema |      Name       | Type  | Owner
-----+-----+-----+-----+
 public | Todo          | table | postgres
 public | User          | table | postgres
 public | _prisma_migrations | table | postgres
(3 rows)
```

Step 8 – Generating the prisma client

What is a client?

Foundation	Frontend
Prisma 1 of 12	Reconcilers and Frontend frameworks React beginner to advance Internals of state, Context API State management using recoil CSS you need to know of, Flexbox, basic styling Frontend UI frameworks, Deep dive into Tailwind Containerization, Docker Next.js Custom hooks In house auth using next auth
Backend	Basic Devops
Backend communication protocols Express basic to advance ORMs Middlewares, routes, status codes, global catches Zod MonoRepos, turborepo Serverless Backends OpenAPI Spec Autogenerated clients Authentication using external libraries Scaling Node.js, performance benchmarks Deploying npm packages	Docker end to end Deploying to AWS servers Newer clouds like fly/Remix Nginx and reverse proxies
Projects	
	GSoC Project setting up and issue solving Building Paytm/Wallet End to End

Client represents all the functions that convert

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

into

INSERT INTO users VALUES ...

Once you've created the `prisma/schema.prisma`, you can generate these `clients` that you can use in your Node.js app

How to generate the client?

`npx prisma generate`

This generates a new `client` for you

Step 9 – Creating your





Prisma 1 of 12

Write a function that lets 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, firstN
}


```



▼ Solution

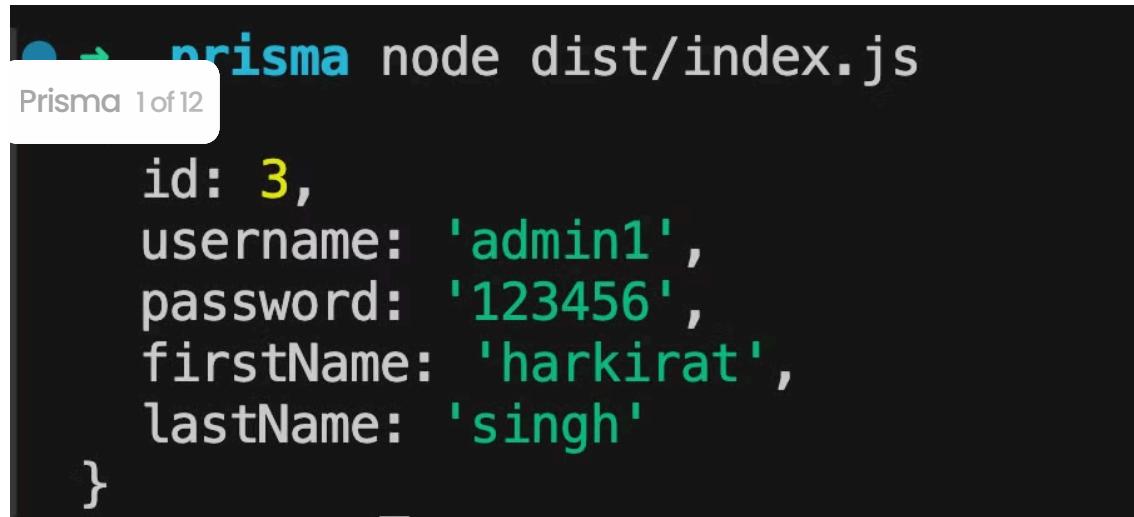
```
import { PrismaClient } from "@prisma/client";

const prisma = new PrismaClient();

async function insertUser(username: string, password: string, fir
const res = await prisma.user.create({
  data: {
    username,
    password,
    firstName,
    lastName
  }
})
console.log(res);
}

insertUser("admin1", "123456", "harkirat", "singh")
```





```
prisma node dist/index.js
Prisma 1 of 12
{
  id: 3,
  username: 'admin1',
  password: '123456',
  firstName: 'harkirat',
  lastName: 'singh'
}
```

Update

Write a function that lets 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) {

}
```

▼ Solution

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

const prisma = new PrismaClient();

interface UpdateParams {
  firstName: string;
```

```
Prisma 1 of 12
  inction updateUser(username: string, {
    name,
    lastName
  ): UpdateParams) {
    const res = await prisma.user.update({
      where: { username },
      data: {
        firstName,
        lastName
      }
    });
    console.log(res);
  }

  updateUser("admin1", {
    firstName: "new name",
    lastName: "singh"
  })
}
```

```
● → prisma node dist/index.js
{
  id: 3,
  username: 'admin1',
  password: '123456',
  firstName: 'new name',
  lastName: 'singh'
}
```

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";
Prisma 1 of 12
const prisma = new PrismaClient();

async function getUser(username: string) {
}
```

▼ Solution

```
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");
```

Step 10 – Relationships.

Prisma let's you define **relationships** to relate tables with each other.

1 Types of relationships

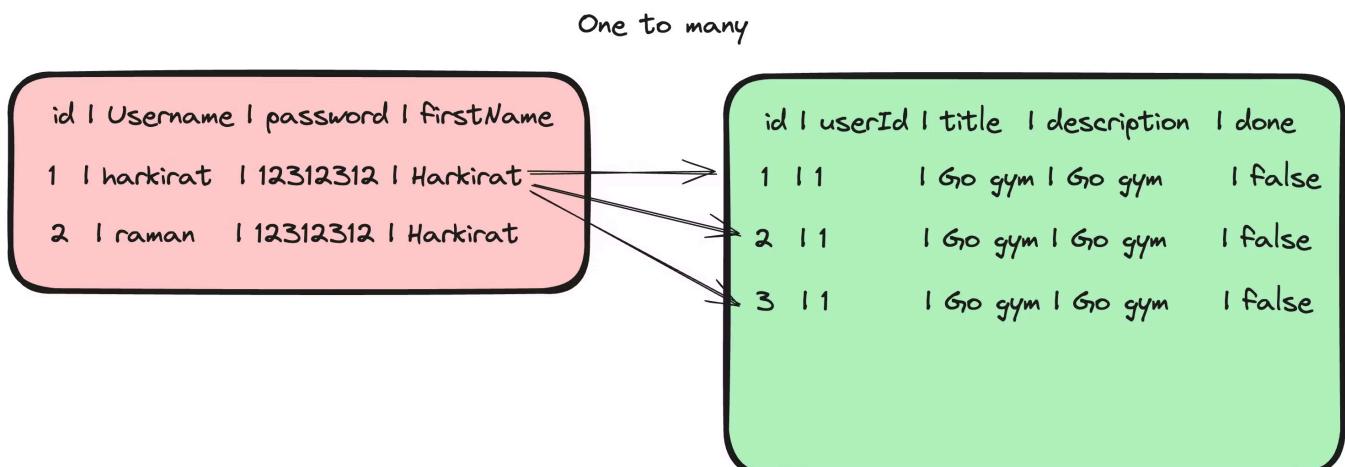
Prisma 1 of 12

2. One to Many

3. Many to One

4. Many to Many

2. For the TODO app, there is a **one to many** relationship



3. Updating the prisma schema

```

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])
}

```

▼ Updated schema

```

// 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 = "postgresql"
    url    = "postgresql://postgres:mysecretpassword@localhost:5432/postgres"
}

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

```

```
title  String
      tring
Prisma 1 of 12  lean @default(false)
userId  Int
user    User  @relation(fields: [userId], references: [id])
}
```

4. Update the database and the prisma client

```
npx prisma migrate dev --name relationship
npx prisma generate
```



Try exploring the `prisma/migrations` folder now. Do you see more

Step 11 – Todo functions

1. createTodo

Write a function that let's you put a `todo` in the database.

Starter code –

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

const prisma = new PrismaClient();

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

}

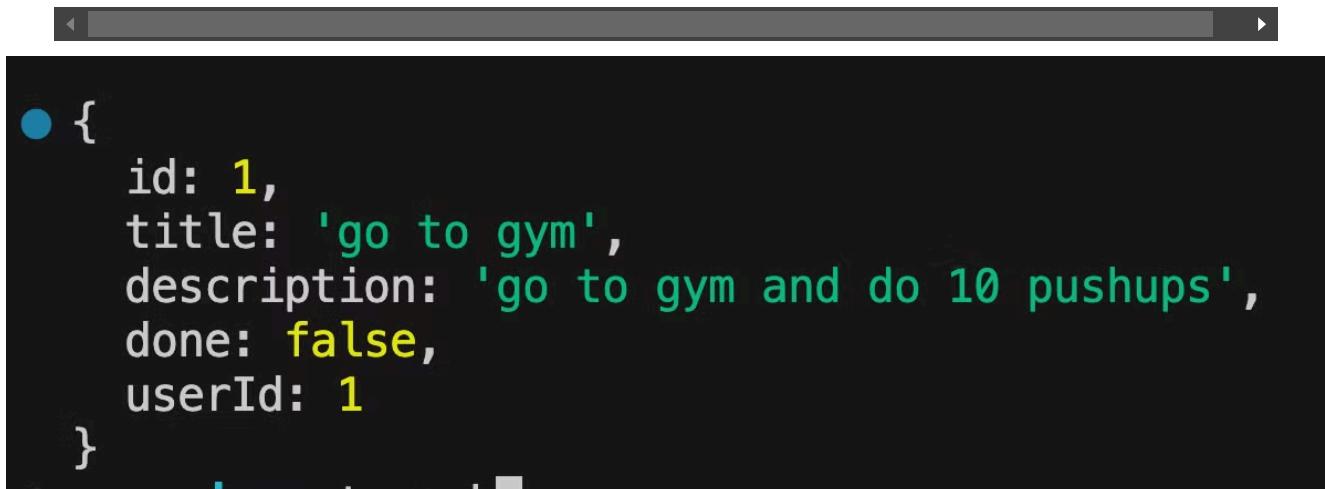
createTodo(1, "go to gym", "go to gym and do 10 pushups");
```

```
import { PrismaClient } from "@prisma/client";
Prisma 1 of 12
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");
```



```
● {
  id: 1,
  title: 'go to gym',
  description: 'go to gym and do 10 pushups',
  done: false,
  userId: 1
}
```

2. getTodos

Write a function to get all the todos for a user.

Starter code

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

const prisma = new PrismaClient();

async function getTodos(userId: number, ) {
```

et
Prisma 1 of 12

▼ 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);
```

```
[{"id": 1, "title": "go to gym", "description": "go to gym and do 10 pushups", "done": false, "userId": 1}]
```

3. getTodosAndUserDetails (Does/should it use joins?)

Write a function that gives you the todo details of a user along with the user details

Starter Code

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

```
async function getTodosAndUserDetails(userId: number, ) {
```

☰ Prisma 1 of 12

```
getTodosAndUserDetails(1);
```

▼ Bad solution (2 queries)

```
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 (using joins)

```
import { PrismaClient } from "@prisma/client";  
  
const prisma = new PrismaClient();  
  
async function getTodosAndUserDetails(userId: number, ) {  
  const todos = await prisma.todo.findMany({  
    where: {  
      userId: userId,  
    },  
    select: {  
      description: true  
    }  
  });  
  const user = await prisma.user.findUnique({  
    where: {  
      id: userId  
    }  
  });  
  return {  
    todos,  
    user  
  };  
}  
  
getTodosAndUserDetails(1);
```

```
    }  
  }  
  
  Prisma 1 of 12 (todos);  
}  
  
getTodosAndUserDetails(1);
```



See <https://github.com/prisma/prisma/issues/5026> to log the actual SQL queries

Page 12 – Expressify it

Assignment for this week

Try creating a todo application that lets a user signup, put todos and fetch todos.

Use

1. Typescript as the language
2. Prisma as the ORM
3. Postgres as the database
4. Zod as validation library

