emanuelefavero / prisma

Code Issues Pull requests Actions Projects Security Insights

main

prisma / README.md

emanuelefavero update README 6 months ago

473 lines (366 loc) · 9.9 KB

Prisma

This is a cheat sheet repo for the Prisma database ORM. Prisma is a database toolkit that makes it easy to query, migrate and model your database

> Prisma can use any database, but this cheat sheet is focused on PostgreSQL (*Note: Very little would change with a different database, that's the magic of Prisma*)

Table of Contents

Installation

- setup a new project with `npm init -y`
- install Prisma and needed dev dependencies with `npm i -D prisma @prisma/client`

> Note: For a Typescript project, you'll need to install `typescript` and `ts-node` as well as well as any other dev dependencies you need for your project (such as `@types/node` for a Node project)
>
> It is also recommended to install `nodemon` for development

- Full Command for a Node Typescript Project

```
npm i -D prisma typescript ts-node @types/node nodemon
```

- create a `tsconfig.json` file:

```
{
  "compilerOptions": {
    "sourceMap": true,
    "outDir": "dist",
    "strict": true,
    "lib": ["esnext"],
    "esModuleInterop": true
  }
}
```

Initialize Prisma

- this will create a `prisma` folder with a `schema.prisma` file

```
npx prisma init --datasource-provider postgresql
```

> *--datasource-provider is optional and will default to* `postgresql`

Setup a Database

- Setup any database you want to use with Prisma and get the connection string

> Note: I've created a new database with Supabase which is a firebase-like database service that uses PostgreSQL
>
> - Create a new database with Supabase
> - Go to `Project Settings / Database / Connection string / URI` and copy the `URI` string

- Add your database connection URI string to `.env`

```
DATABASE_URL="postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=SCHEMA
```

IF you already have data in your database

- run `npx db pull` if you already have data in your database and you want to generate the Prisma schema
- add your schema in `schema.prisma`

*Prisma VS Code Extension

Install the prisma vs-code extension for syntax highlighting and more

Add the following to your `settings.json` file to enable this extension for `.prisma` files:

```
"[prisma]": {
  "editor.defaultFormatter": "Prisma.prisma"
}
```

Define your Database Schema

- Define your database models

```
model User {
id String @id @default(uuid())
```

```
  name String
}
```

> Note: uuid is of type String, autoincrement is of type Int

Initialize your database

> **Remember to run this command after any changes to your schema**

```
npx prisma migrate dev
```

> if prisma complains, run this command: `npx prisma migrate dev --name init`

Install Prisma Client

```
npm i @prisma/client
```

> When you install Prisma Client, it automatically generates a client for your defined models, if you need to regenerate the client, run `npx prisma generate`

Use Prisma Client

- create a `prisma.ts` or any file you want to use Prisma in
- import the client

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

- create a new instance of the client

```
const prisma = new PrismaClient()
```

> Note: Tell prisma to log all database queries **Useful WHEN debugging**

```
const prisma = new PrismaClient({
  log: ['query', 'info', 'warn'],
})
```

- use the client to query your database

```
async function main() {
  const allUsers = await prisma.user.findMany()
  console.log(allUsers)
  // ... WRITE HERE ALL YOUR QUERIES
}
main()
  .catch((e) => {
    throw e
  })
  .finally(async () => {
    await prisma.$disconnect()
  })
```

> Note: Check the example project in this repo for prisma client and schema models examples

Schema Models

- `schema.prisma` file

```
model User {
id String @id @default(uuid()) // @id sets the primary key
// id Int @id @default(autoincrement())
email String @unique // @unique sets the field as unique
name String? // ? optional
createdAt DateTime @default(now()) // * default value (now)
updatedAt DateTime @updatedAt // * auto update this field on update

posts Post[] // * one user to many posts relation

// ? BLOCK LEVEL ATTRIBUTE
@@unique([age, name]) // now we cannot have two users with the same age a
@@index([email]) // index this field for faster queries when filtering an
}

model Post {
id String @id @default(uuid())
title String
content String?
published Boolean @default(false)
createdAt DateTime @default(now())
updatedAt DateTime @updatedAt

// * one user to many posts relation
author User @relation(fields: [authorId], references: [id])
authorId String
}
```

> Note: uuid is of type String, autoincrement is of type Int

Enums

- define a custom enum type in your schema

```
enum Role {
  USER
  ADMIN
}

model User {
  id String @id @default(uuid())
  role Role @default(USER)
}
```

> Note: Enums are useful for determining the role of a user, or the status of a post (draft, published, etc...)

CRUD Operations

CREATE

```
// * CREATE
const createUser = await prisma.user.create({
  data: {
    name: 'Pam',
    email: 'pam@paper.com',
    age: 26,

    // * Create a userPreference object at the same time. (relation)
    userPreference: {
      create: {
        emailUpdates: true,
      },
    },
  },

  // * Include the userPreference object in the response
```

```
    // include: {
    //   userPreference: true,
    // },

    // * Only show the name and the id of userPreference in the response
    select: {
      name: true,
      userPreference: { select: { id: true } },
    },
  })

  const createUsers = await prisma.user.createMany({
    data: [
      {
        name: 'Michael',
        email: 'michael@paper.com',
        age: 41,
      },
      {
        name: 'Dwight',
        email: 'dwight@paper.com',
        age: 35,
      },
    ],

    // ? You can't use include or select with createMany
  })
```

UPDATE

```
// * UPDATE
// Update One
const updateOne = await prisma.user.update({
  where: {
    email: 'michael@paper.com',
  },

  data: {
    age: {
      increment: 1, // ? increment, decrement, multiply, divide, append,
    },
  },
})

// Update Many
const updateMany = await prisma.user.updateMany({
  where: {
    age: { gt: 40 },
  },

  data: {
    email: '...@paper.com',
  },
})
```

CONNECT, DISCONNECT, SET

```
// * CONNECT, DISCONNECT, SET
const connect = await prisma.user.update({
  where: {
    email: 'pam@paper.com',
  },

  data: {
    userPreference: {
      connect: {
        id: '9c7c2634-5cab-428d-8ca8-0db26bc3c684', // ? userPreferenceId
      },
    },
  },
})

const disconnect = await prisma.user.update({
  where: {
    email: 'pam@paper.com',
  },

  data: {
    userPreference: {
      disconnect: true, // ? now pam's userPreference is null
    },
  },
})
```

DELETE

```
// * DELETE
// * delete all
const deleteAll = await prisma.user.deleteMany()
```

```
// * delete many that match a condition
const deleteAllUsersAged40Plus = await prisma.user.deleteMany({
  where: {
    age: { gt: 40 },
  },
})

// * delete one
// You need a unique identifier to delete one (you can setup a unique ide
const deleteOne = await prisma.user.delete({
  where: {
    email: 'pam@paper.com',
  },
})
```

READ

```
// * READ
// * find all users
const findUsers = await prisma.user.findMany()

// * find one user by an unique field (email)
const findUser = await prisma.user.findUnique({
  where: {
    email: 'pam@paper.com',
  },
})

// * find user by multiple unique fields that we specified
// ? @@unique([age, name])
const findUserByMultipleUniqueFields = await prisma.user.findUnique({
  where: {
    age_name: {
      age: 26,
      name: 'Pam',
    },
  },
})

// * find users, sort and limit results
const findSortAndLimitResults = await prisma.user.findMany({
  take: 2, // limit
  skip: 1, // skip
  orderBy: {
    age: 'desc', // sort
  },
})

// ? findFirst - find a user by any field that is not unique
// ? distinct - return only distinct results (only first occurence of eac
```

FILTERS

```
// * FILTERS
// * not
const notFilter = await prisma.user.findMany({
  where: {
    name: { not: 'Pam' },
  },
})

// * in, notIn
const inFilter = await prisma.user.findMany({
  where: {
    name: { in: ['Pam', 'Dwight'] },
  },
})

// * lt, lte, gt, gte
const ltFilter = await prisma.user.findMany({
  where: {
    age: { lt: 30 },
  },
})
```

main prisma / README.md Top

Preview Code Blame Raw

```
  },
})

// * AND, OR, NOT
const andFilter = await prisma.user.findMany({
  where: {
    AND: [{ name: 'Pam' }, { age: { lt: 30 } }],
  },
})

// ARRAY FILTERING
// * some, none, every
// ! hypothetical example
// const someFilter = await prisma.user.findMany({
//   where: {
//     posts: {
//       some: {
//         title: 'Hello World',
//       },
//     },
//   },
// })
```

Resources

- Prisma Docs
- Prisma Quick Start
- Prisma Playground

License

- MIT

Go To Top ⬆