Prisma

Prisma is an open-source ORM for Node.js and TypeScript. It is used as an **alternative** to writing plain SQL, or using another database access tool such as SQL query builders (like knex.js) or ORMs (like TypeORM and Sequelize). Prisma currently supports PostgreSQL, MySQL, SQL Server, SQLite, MongoDB and CockroachDB (Preview).

While Prisma can be used with plain JavaScript, it embraces TypeScript and provides a level to type-safety that goes beyond the guarantees other ORMs in the TypeScript ecosystem. You can find an in-depth comparison of the type-safety guarantees of Prisma and TypeORM here.

> info **Note** If you want to get a quick overview of how Prisma works, you can follow the Quickstart or read the Introduction in the documentation. There also are ready-to-run examples for REST and GraphQL in the `prisma-examples` repo.

Getting started

In this recipe, you'll learn how to get started with NestJS and Prisma from scratch. You are going to build a sample NestJS application with a REST API that can read and write data in a database.

For the purpose of this guide, you'll use a SQLite database to save the overhead of setting up a database server. Note that you can still follow this guide, even if you're using PostgreSQL or MySQL – you'll get extra instructions for using these databases at the right places.

> info **Note** If you already have an existing project and consider migrating to Prisma, you can follow the guide for adding Prisma to an existing project. If you are migrating from TypeORM, you can read the guide Migrating from TypeORM to Prisma.

Create your NestJS project

To get started, install the NestJS CLI and create your app skeleton with the following commands:

```
$ npm install -g @nestjs/cli
$ nest new hello-prisma
```

See the First steps page to learn more about the project files created by this command. Note also that you can now run `npm start` to start your application. The REST API running at `http://localhost:3000/` currently serves a single route that's implemented in `src/app.controller.ts`. Over the course of this guide, you'll implement additional routes to store and retrieve data about *users* and *posts*.

Set up Prisma

Start by installing the Prisma CLI as a development dependency in your project:

```
$ cd hello-prisma
$ npm install prisma --save-dev
```

In the following steps, we'll be utilizing the Prisma CLI. As a best practice, it's recommended to invoke the CLI locally by prefixing it with `npx`:

```
$ npx prisma
```

▶ Expand if you're using Yarn

If you're using Yarn, then you can install the Prisma CLI as follows:

```
$ yarn add prisma --dev
```

Once installed, you can invoke it by prefixing it with `yarn`:

```
$ yarn prisma
```

Now create your initial Prisma setup using the `init` command of the Prisma CLI:

```
$ npx prisma init
```

This command creates a new `prisma` directory with the following contents:

- `schema.prisma`: Specifies your database connection and contains the database schema
- `.env`: A dotenv file, typically used to store your database credentials in a group of environment variables

Set the database connection

Your database connection is configured in the `datasource` block in your `schema.prisma` file. By default it's set to `postgresql`, but since you're using a SQLite database in this guide you need to adjust the `provider` field of the `datasource` block to `sqlite`:

```
datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

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

Now, open up `.env` and adjust the `DATABASE_URL` environment variable to look as follows:

```
DATABASE_URL="file:./dev.db"
```

Make sure you have a ConfigModule configured, otherwise the `DATABASE_URL` variable will not be picked up from `.env`.

SQLite databases are simple files; no server is required to use a SQLite database. So instead of configuring a connection URL with a *host* and *port*, you can just point it to a local file which in this case is called `dev.db`. This file will be created in the next step.

▶ Expand if you're using PostgreSQL or MySQL

With PostgreSQL and MySQL, you need to configure the connection URL to point to the *database server*. You can learn more about the required connection URL format here.

PostgreSQL

If you're using PostgreSQL, you have to adjust the `schema.prisma` and `.env` files as follows:

`schema.prisma`

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

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

`.env`

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

Replace the placeholders spelled in all uppercase letters with your database credentials. Note that if you're unsure what to provide for the `SCHEMA` placeholder, it's most likely the default value `public`:

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

If you want to learn how to set up a PostgreSQL database, you can follow this guide on setting up a free PostgreSQL database on Heroku.

MySQL

If you're using MySQL, you have to adjust the `schema.prisma` and `.env` files as follows:

`schema.prisma`

```
datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}

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

`.env`

```
DATABASE_URL="mysql://USER:PASSWORD@HOST:PORT/DATABASE"
```

Replace the placeholders spelled in all uppercase letters with your database credentials.

Create two database tables with Prisma Migrate

In this section, you'll create two new tables in your database using Prisma Migrate. Prisma Migrate generates SQL migration files for your declarative data model definition in the Prisma schema. These migration files are fully customizable so that you can configure any additional features of the underlying database or include additional commands, e.g. for seeding.

Add the following two models to your `schema.prisma` file:

```
model User {
  id    Int     @default(autoincrement()) @id
  email String  @unique
  name  String?
  posts Post[]
}

model Post {
  id        Int      @default(autoincrement()) @id
  title     String
  content   String?
  published Boolean? @default(false)
  author    User?    @relation(fields: [authorId], references: [id])
  authorId  Int?
}
```

With your Prisma models in place, you can generate your SQL migration files and run them against the database. Run the following commands in your terminal:

```
$ npx prisma migrate dev --name init
```

This `prisma migrate dev` command generates SQL files and directly runs them against the database. In this case, the following migration files was created in the existing `prisma` directory:

```
$ tree prisma
prisma
├── dev.db
├── migrations
│   └── 20201207100915_init
│       └── migration.sql
└── schema.prisma
```

▶ Expand to view the generated SQL statements

The following tables were created in your SQLite database:

```
-- CreateTable
CREATE TABLE "User" (
    "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    "email" TEXT NOT NULL,
    "name" TEXT
);

-- CreateTable
CREATE TABLE "Post" (
    "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" TEXT NOT NULL,
    "content" TEXT,
    "published" BOOLEAN DEFAULT false,
    "authorId" INTEGER,

    FOREIGN KEY ("authorId") REFERENCES "User"("id") ON DELETE SET NULL ON
UPDATE CASCADE
);

-- CreateIndex
CREATE UNIQUE INDEX "User.email_unique" ON "User"("email");
```

Install and generate Prisma Client

Prisma Client is a type-safe database client that's *generated* from your Prisma model definition. Because of this approach, Prisma Client can expose CRUD operations that are *tailored* specifically to your models.

To install Prisma Client in your project, run the following command in your terminal:

```
$ npm install @prisma/client
```

Note that during installation, Prisma automatically invokes the `prisma generate` command for you. In the future, you need to run this command after *every* change to your Prisma models to update your generated Prisma Client.

> info **Note** The `prisma generate` command reads your Prisma schema and updates the generated Prisma Client library inside `node_modules/@prisma/client`.

Use Prisma Client in your NestJS services

You're now able to send database queries with Prisma Client. If you want to learn more about building queries with Prisma Client, check out the API documentation.

When setting up your NestJS application, you'll want to abstract away the Prisma Client API for database queries within a service. To get started, you can create a new `PrismaService` that takes care of instantiating `PrismaClient` and connecting to your database.

Inside the `src` directory, create a new file called `prisma.service.ts` and add the following code to it:

```
import { Injectable, OnModuleInit } from '@nestjs/common';
import { PrismaClient } from '@prisma/client';

@Injectable()
export class PrismaService extends PrismaClient implements OnModuleInit {
  async onModuleInit() {
    await this.$connect();
  }
}
```

> info **Note** The `onModuleInit` is optional — if you leave it out, Prisma will connect lazily on its first call to the database.

Next, you can write services that you can use to make database calls for the `User` and `Post` models from your Prisma schema.

Still inside the `src` directory, create a new file called `user.service.ts` and add the following code to it:

```
import { Injectable } from '@nestjs/common';
import { PrismaService } from './prisma.service';
import { User, Prisma } from '@prisma/client';

@Injectable()
export class UserService {
  constructor(private prisma: PrismaService) {}

  async user(
    userWhereUniqueInput: Prisma.UserWhereUniqueInput,
  ): Promise<User | null> {
    return this.prisma.user.findUnique({
      where: userWhereUniqueInput,
    });
  }

  async users(params: {
    skip?: number;
    take?: number;
    cursor?: Prisma.UserWhereUniqueInput;
    where?: Prisma.UserWhereInput;
    orderBy?: Prisma.UserOrderByWithRelationInput;
  }): Promise<User[]> {
    const { skip, take, cursor, where, orderBy } = params;
    return this.prisma.user.findMany({
      skip,
      take,
      cursor,
      where,
      orderBy,
    });
  }

  async createUser(data: Prisma.UserCreateInput): Promise<User> {
    return this.prisma.user.create({
      data,
    });
  }

  async updateUser(params: {
    where: Prisma.UserWhereUniqueInput;
    data: Prisma.UserUpdateInput;
  }): Promise<User> {
    const { where, data } = params;
    return this.prisma.user.update({
      data,
      where,
    });
  }

  async deleteUser(where: Prisma.UserWhereUniqueInput): Promise<User> {
    return this.prisma.user.delete({
      where,
    });
  }
}
```

Notice how you're using Prisma Client's generated types to ensure that the methods that are exposed by your service are properly typed. You therefore save the boilerplate of typing your models and creating additional interface or DTO files.

Now do the same for the `Post` model.

Still inside the `src` directory, create a new file called `post.service.ts` and add the following code to it:

```
import { Injectable } from '@nestjs/common';
import { PrismaService } from './prisma.service';
import { Post, Prisma } from '@prisma/client';

@Injectable()
export class PostService {
  constructor(private prisma: PrismaService) {}

  async post(
    postWhereUniqueInput: Prisma.PostWhereUniqueInput,
  ): Promise<Post | null> {
    return this.prisma.post.findUnique({
      where: postWhereUniqueInput,
    });
  }

  async posts(params: {
    skip?: number;
    take?: number;
    cursor?: Prisma.PostWhereUniqueInput;
    where?: Prisma.PostWhereInput;
    orderBy?: Prisma.PostOrderByWithRelationInput;
  }): Promise<Post[]> {
    const { skip, take, cursor, where, orderBy } = params;
    return this.prisma.post.findMany({
      skip,
      take,
      cursor,
      where,
      orderBy,
    });
  }

  async createPost(data: Prisma.PostCreateInput): Promise<Post> {
    return this.prisma.post.create({
      data,
    });
  }

  async updatePost(params: {
    where: Prisma.PostWhereUniqueInput;
    data: Prisma.PostUpdateInput;
  }): Promise<Post> {
    const { data, where } = params;
    return this.prisma.post.update({
      data,
      where,
    });
  }
  async deletePost(where: Prisma.PostWhereUniqueInput): Promise<Post> {
    return this.prisma.post.delete({
      where,
    });
  }
}
```

Your `UserService` and `PostService` currently wrap the CRUD queries that are available in Prisma Client. In a real world application, the service would also be the place to add business logic to your application. For example, you could have a method called `updatePassword` inside the `UserService` that would be responsible for updating the password of a user.

Implement your REST API routes in the main app controller

Finally, you'll use the services you created in the previous sections to implement the different routes of your app. For the purpose of this guide, you'll put all your routes into the already existing `AppController` class.

Replace the contents of the `app.controller.ts` file with the following code:

```
import {
  Controller,
  Get,
  Param,
  Post,
  Body,
  Put,
  Delete,
} from '@nestjs/common';
import { UserService } from './user.service';
import { PostService } from './post.service';
import { User as UserModel, Post as PostModel } from '@prisma/client';

@Controller()
export class AppController {
  constructor(
    private readonly userService: UserService,
    private readonly postService: PostService,
  ) {}

  @Get('post/:id')
  async getPostById(@Param('id') id: string): Promise<PostModel> {
    return this.postService.post({ id: Number(id) });
  }

  @Get('feed')
  async getPublishedPosts(): Promise<PostModel[]> {
    return this.postService.posts({
      where: { published: true },
    });
  }
  @Get('filtered-posts/:searchString')
  async getFilteredPosts(
    @Param('searchString') searchString: string,
  ): Promise<PostModel[]> {
    return this.postService.posts({
      where: {
        OR: [
          {
            title: { contains: searchString },
          },
          {
            content: { contains: searchString },
          },
        ],
      },
    });
  }

  @Post('post')
  async createDraft(
    @Body() postData: { title: string; content?: string; authorEmail:
string },
  ): Promise<PostModel> {
    const { title, content, authorEmail } = postData;
    return this.postService.createPost({
      title,
      content,
      author: {
        connect: { email: authorEmail },
      },
    });
  }

  @Post('user')
  async signupUser(
    @Body() userData: { name?: string; email: string },
  ): Promise<UserModel> {
    return this.userService.createUser(userData);
  }

  @Put('publish/:id')
  async publishPost(@Param('id') id: string): Promise<PostModel> {
    return this.postService.updatePost({
      where: { id: Number(id) },
      data: { published: true },
    });
  }

  @Delete('post/:id')
  async deletePost(@Param('id') id: string): Promise<PostModel> {
    return this.postService.deletePost({ id: Number(id) });
  }
}
```

This controller implements the following routes:

`GET`

- `/post/:id`: Fetch a single post by its `id`
- `/feed`: Fetch all *published* posts
- `/filter-posts/:searchString`: Filter posts by `title` or `content`

`POST`

- `/post`: Create a new post
  - Body:
    - `title: String` (required): The title of the post
    - `content: String` (optional): The content of the post
    - `authorEmail: String` (required): The email of the user that creates the post
- `/user`: Create a new user
  - Body:
    - `email: String` (required): The email address of the user
    - `name: String` (optional): The name of the user

`PUT`

- `/publish/:id`: Publish a post by its `id`

`DELETE`

- `/post/:id`: Delete a post by its `id`

Summary

In this recipe, you learned how to use Prisma along with NestJS to implement a REST API. The controller that implements the routes of the API is calling a `PrismaService` which in turn uses Prisma Client to send queries to a database to fulfill the data needs of incoming requests.

If you want to learn more about using NestJS with Prisma, be sure to check out the following resources:

- NestJS & Prisma
- Ready-to-run example projects for REST & GraphQL
- Production-ready starter kit
- Video: Accessing Databases using NestJS with Prisma (5min) by Marc Stammerjohann