Progressive Coder

Build a NestJS Prisma REST API – The Simplest Step-By-Step Guide

Published by **Saurabh Dashora** on February 2, 2022

In this post, we will learn how to build a **NestJS Prisma REST API** from scratch. Unlike many other posts about the topic, we will start with the **basics of Prisma and also its advantages**. Once we cover the essentials, we will put together our project in a step-by-step manner. This will help us build a comprehensive **NestJS Prisma Tutorial** that can help you get started with Prisma ORM in your own projects with ease.

1 – What is Prisma?

ORMs.

Instead of classes, Prisma uses a special **Schema Definition Language**. Basically, developers describe their schemas using this language. Prisma runs over the schemas and writes the appropriate migrations depending on the chosen database. It also **generates type-safe code** to interact with the database.

In other words, **Prisma provides an alternative to writing plain SQL queries or other ORMs** (such as TypeORM or Sequelize). It can work with various databases such as PostgreSQL, MySQL and SQLite.

Prisma consists of two main parts:

- **Prisma Migrate –** This is the **migration tool** provided by Prisma. It helps us keep our database schema in sync with the Prisma schema. For every change to our schema, the Prisma migrate generates a migration file. In this way, it also **helps maintain a history of all changes that have happened to our schema**.
- **Prisma Client –** This is the **auto-generated query builder**. The Prisma Client acts as the bridge between our application code and the database. It also provides type-safety.

2 – Is Prisma ORM Good?

While this might be a subjective question, Prisma aims to make it easy for developers to deal with database queries.

As any developer will know, it is absolutely necessary for most applications to interact with databases to manage data. This interaction can be in the form of raw queries or ORM frameworks (such as TypeORM). While raw queries or query builders provide more control, also leads to **object-impedance mismatch**.

INFO

Object Impedance Mismatch is a conceptual problem when an object oriented programming language interacts with a relational database. In a relational database, **data is normalized** and links between different entities is via foreign keys. However, **objects establish the same relation using nested structures**. Developers writing application code are used to thinking about objects and their structure. This causes a mismatch when dealing with a relational database.

Prisma attempts to solve the issues around object relational mapping by making developers more productive and at the same time, giving them more control. Some important examples of how Prisma achieves this is as follows:

- It allows developers to **think in terms of objects**.
- **Avoid complex model objects**
- **Single source of truth for both database and application** using schema files
- **Type safe queries** to catch errors during compile time

3 – Setting up a NestJS Prisma Project

With a basic understanding of Prisma, we can now start to build our **NestJS Prisma REST API**.

As the first step, we create a new NestJS Project.

```
$ nest new nestjs-prisma-demo-app
$ cd nestjs-prisma-demo-app
```

Basically, this command creates a working NestJS project. In case you are new to NestJS, you can read more about it in our post about NestJS Basics.

In the next step, we install **Prisma**.

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

Note that this is only a development dependency.

We will be using the **Prisma CLI** to work with Prisma. To invoke the CLI, we use **npx** as below.

```
$ npx prisma
```

Once Prisma is activated, we create our initial Prisma setup.


```
$ npx prisma init
```

This command creates a new directory named **prisma** and a **configuration file** within our project directory.

- **schema.prisma** – This is the most important file from Prisma perspective. It will be present within the **prisma** directory. It specifies the database connection and also contains the database schema. You could think of it as the core of our application.
- **.env** – This is like an environment configuration file. It stores the database host name and credentials. Take care **not** to commit this file to source repository if it contains database credentials.

4 – Prisma Database Connection Setup

to setup database servers.

To configure our database connection, we have to make changes in the **schema.prisma** file.

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

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

Basically, we have to change the **provider** in **datasource db** section to **sqlite**. By default, it uses **postgres**.

Second change is in the **.env** file.

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

Here, we specify the path to our database file. The file name is **test.db**. You can name it anything you want.

5 – Prisma Migrate Command

Now, we are ready to create tables in our **SQLite** database.

To do so, we will first write our schema. As discussed earlier, the schema is defined in **prisma.schema** file. See below:

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

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

model Book {
  id  Int @default(autoincrement()) @id
  title String
  author String
  publishYear Int
}
```

integer and is set to auto-increment.

This schema is the single source of truth for our application and also the database. No need to write any other classes as with NestJS TypeORM.

We will now generate the migration files using **Prisma Migrate**.

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

Basically, this command generates SQL files and also runs them on the configured database. You should be able to find the below SQL file within the **prisma/migrations** directory.

```
CREATE TABLE "Book" (
    "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    "title" TEXT NOT NULL,
    "author" TEXT NOT NULL,
    "publishYear" INTEGER NOT NULL
);
```

Also, the database file **test.db** will be created.

6 – Installing and Generating Prisma Client

Our database is now ready. However, we still don't have a way to interact with the database from our application.

This is where the **Prisma Client** comes into the picture.

But what is Prisma Client?

Prisma Client is a type-safe database client to interact with our database. It is **generated** using the model definition from our **prisma.schema** file. In other words, the

We install Prisma Client using the below command.

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

Under the hood, the installation step also executes the **prisma generate** command. In case we make changes to our schema (like adding a field or a new model), we can simply invoke **prisma generate** command to update our Prisma Client accordingly.

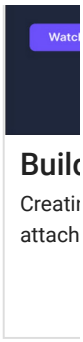
Once the installation is successful, the Prisma Client library in **node_modules/@prisma/client** is updated accordingly.

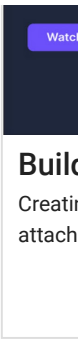
7 – Creating the Database Service

db.service.ts

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

@Injectable()
export class DBService extends PrismaClient implements OnModuleInit {

    async onModuleInit() {
        await this.$connect();
    }

    async enableShutdownHooks(app: INestApplication) {
        this.$on('beforeExit', async () => {
            await app.close();
        })
    }
}
```

Basically, we create a **DBService** that extends the **PrismaClient** that we generated in the previous step. It implements the interface **OnModuleInit**. If we don't use **OnModuleInit**, Prisma connects to the database lazily.





Prisma also has its own shut down mechanism where it destroys the database connection. Therefore, we implement the **enableShutdownHooks()** method and also call it in the **main.ts** file.

main.ts

Progressive Coder

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const dbService: DBService = app.get(DBService);
  dbService.enableShutdownHooks(app)
  await app.listen(3000);
}
bootstrap();
```

8 – Creating the Application Service

Now that our database service is setup, we can create the actual application service. This service exposes methods to read, create, update and delete a book from the database.

See below:

book.service.ts

```
import { Injectable } from "@nestjs/common";
import { Book, Prisma } from "@prisma/client";
import { DBService } from "./db.service";

@Injectable()
export class BookService {
    return this.dbService.book.findUnique({
        where: id
    })
  }

  async createBook(data: Prisma.BookCreateInput): Promise<Book> {
    return this.dbService.book.create({
        data,
    })
  }

  async updateBook(params: {
    where: Prisma.BookWhereUniqueInput;
    data: Prisma.BookUpdateInput;
  }): Promise<Book> {
    const { where, data } = params;
    return this.dbService.book.update({
        data,
        where,
```

```
    async deleteBook(where: Prisma.BookWhereUniqueInput): Promise<Book> {
        return this.dbService.book.delete({
            where,
        });
    }
}
```

This is a standard NestJS Service where we inject an instance of the **DBService**. However, important point to note is the use of **Prisma Client's generated types** such as **BookCreateInput**, **BookUpdateInput**, **Book** etc to ensure that the methods of our service are properly typed. There is no need to create any additional DTOs or interfaces to support type-safety.

9 – Creating the REST API Controller

Finally, we can create a NestJS Controller to implement the REST API endpoints.

book.controller.ts

```
import { Body, Controller, Delete, Get, Param, Post, Put } from "@nestjs/common";
import { Book } from "@prisma/client";
import { BookService } from "./book.service";

@Controller()
export class BookController {
    constructor(
        private readonly bookService: BookService
    ) {}

    @Get('books/:id')
    async getBookById(@Param('id') id: string): Promise<Book> {
        return this.bookService.getBook({id: Number(id)});
```

Progressive Coder

```
    async createBook(@Body() bookData: {title: string, author: string, publishYear: Number}):
        const { title, author } = bookData;
        const publishYear = Number(bookData.publishYear);
        return this.bookService.createBook({
            title,
            author,
            publishYear
        })
    }

    @Put('books/:id')
    async updateBook(@Param('id') id: string, @Body() bookData: {title: string, author: strin
        const { title, author } = bookData;
        const publishYear = Number(bookData.publishYear);

        return this.bookService.updateBook({
            where: {id: Number(id)},
            data: {
                title,
                author,
                publishYear
            }
        })
    }

    @Delete('books/:id')
    async deleteBook(@Param('id') id: string): Promise<Book> {
        return this.bookService.deleteBook({
            id: Number(id)
```

Here also, we use the **Book** type generated by the **Prisma Client** to implement type-safety.

As a last step, we configure the controllers and services in the App Module so that NestJS can discover them during application startup.

app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { BookController } from './book.controller';
import { BookService } from './book.service';
import { DBService } from './db.service';
```

```
10    controllers: [AppController, BookController],
11    providers: [AppService, BookService, DBService],
12  })
13  export class AppModule {}
```

Our application is ready. We can start the application using **npm run start** and test the endpoints available at **http://localhost:3000/books**.

Conclusion

With this, we have completed our goal of building **NestJS Prisma REST API**. We started from the very basics of Prisma and worked our way to writing the schema, generating a migration and client to interact with our database tables.

The code for this post is available on Github for reference.

If you have any comments or queries about this post, please feel free to mention in the comments section below.

2 Comments

dayu · April 19, 2022 at 7:07 am

i have error when npm run start

Leave a Reply

Name

Search ...

How to setup KoaJS Cache Middleware using koa-static-cache package?

A Guide to Koa JS Error Handling with Examples

How to setup Koa JS Redirect URL to handle redirection?

A Guide to KoaJS Response Object

Subscription Form

Related Posts

BLOG

How to create a KoaJS REST API?

APIs are the backbone of modern-day software development. Whether we create mobile apps or single-page applications, we need APIs to fetch the data from the backend. Usually, we tend to create REST APIs to solve Read more...

BLOG

How to setup KoaJS Cache Middleware using koa-static-cache package?

Caching is one of the most important aspects of building a high-performance application. Most web frameworks provide caching capabilities. The same is the case with KoaJS. In this post, we will learn how to setup Read more...

BLOG

A Guide to Koa JS Error Handling with Examples

Recent Posts

HOW TO CREATE A KOAJS REST API? July 14, 2022

HOW TO SETUP KOAJS CACHE MIDDLEWARE USING KOA-STATIC-CACHE PACKAGE? July 13, 2022

A GUIDE TO KOA JS ERROR HANDLING WITH EXAMPLES July 12, 2022

Links

ABOUT

SUBSCRIBE

PRIVACY POLICY

ABOUT SUBSCRIBE PRIVACY POLICY

Created my free logo at LogoMakr.com