This article will teach you how to build a backend REST API with NestJS, Prisma, PostgreSQL and Swagger.

PART 1 (Currently reading)

Building a REST API with NestJS and Prisma

PART 2

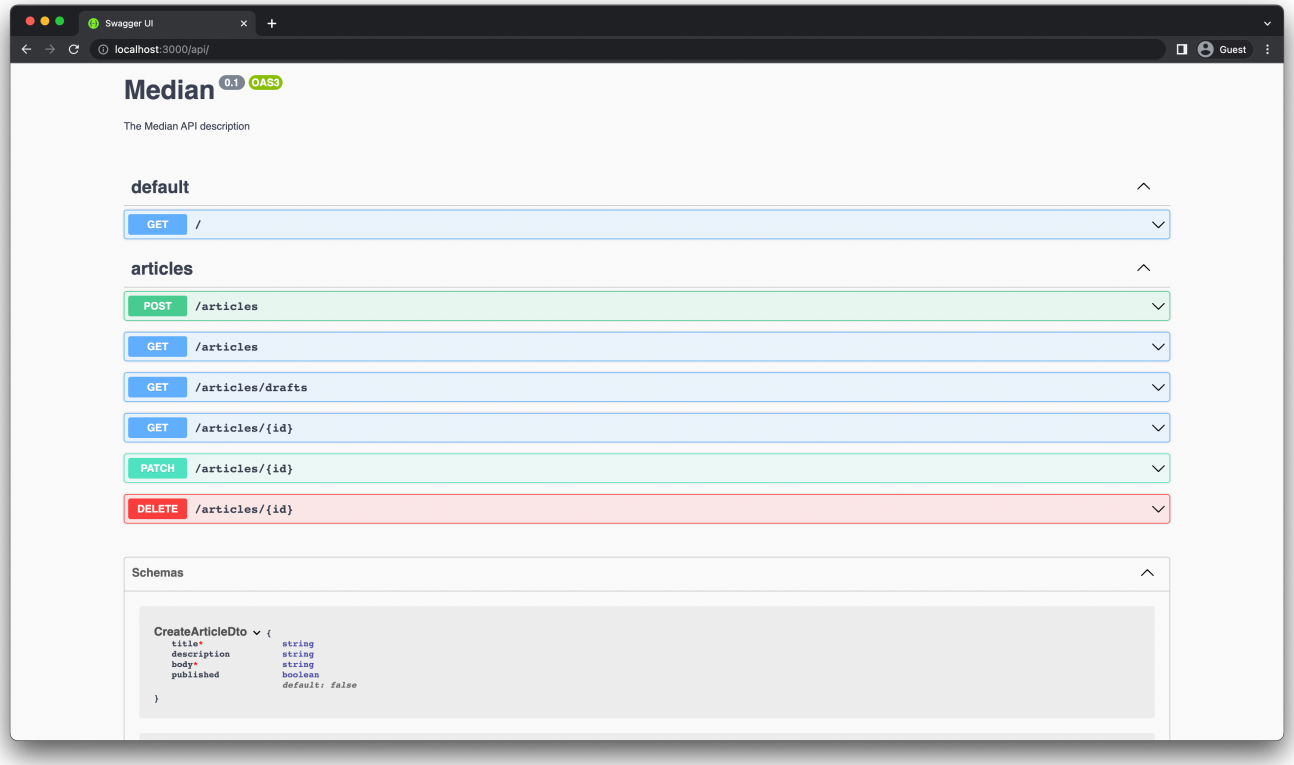
Building a REST API with NestJS and Prisma: Input Validation & Transformation

Table Of Contents

- Update Swagger response types
- Summary and final remarks

Introduction

In this tutorial, you will learn how to build the backend REST API for a blog application called "Median" (a simple Medium clone). You will get started by creating a new NestJS project. Then you will start your own PostgreSQL server and connect to it using Prisma. Finally, you will build the REST API and document it with Swagger.



The final application

Technologies you will use

- Swagger as the API documentation tool
- TypeScript as the programming language

Prerequisites

Assumed knowledge

This is a beginner friendly tutorial. However, this tutorial assumes:

- Basic knowledge of JavaScript or TypeScript (preferred)
- Basic knowledge of NestJS

> ***Note**: If you're not familiar with NestJS, you can quickly learn the basics by following the overview section in the NestJS docs.*

Development environment

To follow along with this tutorial, you will be expected to:

- ... have Node.js installed.
- ... have Docker or PostgreSQL installed.
- ... have the Prisma VSCode Extension installed. *(optional)*
- ... have access to a Unix shell (like the terminal/shell in Linux and macOS) to run the commands provided in this series. *(optional)*

> ***Note 1**: The optional Prisma VSCode extension adds some really nice IntelliSense and syntax highlighting for Prisma.*
>
> ***Note 2**: If you don't have a Unix shell (for example, you are on a Windows machine)*

The first thing you will need is to install the NestJS CLI. The NestJS CLI comes in very handy when working with a NestJS project. It comes with built-in utilities that help you initialize, develop and maintain your NestJS application.

You can use the NestJS CLI to create an empty project. To start, run the following command in the location where you want the project to reside:

```
npx @nestjs/cli new median
```

COPY

The CLI will prompt you to choose a *package manager* for your project — choose **npm**. Afterward, you should have a new NestJS project in the current directory.

Open the project in your preferred code editor (we recommend VSCode). You should see the following files:

```
median
  ├── node_modules
  ├── src
  │   ├── app.controller.spec.ts
  │   ├── app.controller.ts
  │   ├── app.module.ts
  │   ├── app.service.ts
  │   └── main.ts
  ├── test
  │   ├── app.e2e-spec.ts
  │   └── jest-e2e.json
  ├── README.md
  ├── nest-cli.json
  ├── package-lock.json
  ├── package.json
  ├── tsconfig.build.json
  └── tsconfig.json
```

- `src/app.controller.ts` : A basic controller with a single route: `/` . This route will return a simple `'Hello World!'` message.
- `src/main.ts` : The entry point of the application. It will start the NestJS application.

You can start your project by using the following command:

```
npm run start:dev
```

COPY

This command will watch your files, automatically recompiling and reloading the server whenever you make a change. To verify the server is running, go to the URL `http://localhost:3000/` . You should see an empty page with the message `'Hello World!'` .

> ***Note**: You should keep the server running in the background as you go through this tutorial.*

Create a PostgreSQL instance

You will be using PostgreSQL as the database for your NestJS application. This tutorial will show you how to install and run PostgreSQL on your machine through a Docker container.

> ***Note**: If you don't want to use Docker, you can set up a PostgreSQL instance natively or get a hosted PostgreSQL database on Heroku.*

First, create a `docker-compose.yml` file in the main folder of your project:

COPY

```
# docker-compose.yml

version: '3.8'
services:

  postgres:
    image: postgres:13.5
    restart: always
    environment:
      - POSTGRES_USER=myuser
      - POSTGRES_PASSWORD=mypassword
    volumes:
      - postgres:/var/lib/postgresql/data
    ports:
      - '5432:5432'

volumes:
  postgres:
```

A few things to understand about this configuration:

- The `image` option defines what Docker image to use. Here, you are using the `postgres` image version 13.5.
- The `environment` option specifies the environment variables passed to the container during initialization. You can define the configuration options and secrets – such as the username and password – the container will use here.
- The `volumes` option is used for persisting data in the host file system.
- The `ports` option maps ports from the host machine to the container. The format follows a `'host_port:container_port'` convention. In this case, you are mapping the port `5432` of the host machine to port `5432` of the `postgres`

```
docker-compose up
```

COPY

If everything worked correctly, the new terminal window should show logs that the database system is ready to accept connections. You should see logs similar to the following inside the terminal window:

```
...
postgres_1  | 2022-03-05 12:47:02.410 UTC [1] LOG:  listening on IPv4 address "0.0.0.(
postgres_1  | 2022-03-05 12:47:02.410 UTC [1] LOG:  listening on IPv6 address "::", pc
postgres_1  | 2022-03-05 12:47:02.411 UTC [1] LOG:  listening on Unix socket "/var/rur
postgres_1  | 2022-03-05 12:47:02.419 UTC [1] LOG:  database system is ready to accepi
```

Congratulations 🎉. You now have your own PostgreSQL database to play around with!

> ***Note**: If you close the terminal window, it will also stop the container. You can avoid this if you add a `-d` option to the end of the command, like this: `docker-compose up -d`. This will indefinitely run the container in the background.*

Set up Prisma

Now that the database is ready, it's time to set up Prisma!

Initialize Prisma

To get started, first install the Prisma CLI as a development dependency. The Prisma CLI will allow you to run various commands and interact with your project.

This will create a new `prisma` directory with a `schema.prisma` file. This is the main configuration file that contains your database schema. This command also creates a `.env` file inside your project.

Set your environment variable

Inside the `.env` file, you should see a `DATABASE_URL` environment variable with a dummy connection string. Replace this connection string with the one for your PostgreSQL instance.

```
// .env
DATABASE_URL="postgres://myuser:mypassword@localhost:5432/median-db"
```

COPY

> ***Note**: If you didn't use docker (as shown in the previous section) to create your PostgreSQL database, your connection string will be different from the one shown above. The connection string format for PostgreSQL is available in the Prisma Docs.*

Understand the Prisma schema

If you open `prisma/schema.prisma`, you should see the following default schema:

```
// prisma/schema.prisma

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

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

- **Data source**: Specifies your database connection. The above configuration means that your database *provider* is PostgreSQL and the database connection string is available in the `DATABASE_URL` environment variable.
- **Generator**: Indicates that you want to generate Prisma Client, a type-safe query builder for your database. It is used to send queries to your database.
- **Data model**: Defines your database *models*. Each model will be mapped to a table in the underlying database. Right now there are no models in your schema, you will explore this part in the next section.

> ***Note**: For more information on Prisma schema, check out the Prisma docs.*

Model the data

Now it's time to define the data models for your application. For this tutorial, you will only need an `Article` model to represent each article on the blog.

Inside the `prisma/prisma.schema` file, add a new model to your schema named `Article`:

```
// prisma/schema.prisma

model Article {
  id          Int      @id @default(autoincrement())
  title       String   @unique
  description String?
  body        String
  published   Boolean  @default(false)
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
}
```

COPY

indicates that this field should be automatically incremented and assigned to any newly created record.

The `published` field is a flag to indicate whether an article is published or in draft mode. The `@default(false)` attribute indicates that this field should be set to `false` by default.

The two `DateTime` fields, `createdAt` and `updatedAt`, will track when an article is created and when it was last updated. The `@updatedAt` attribute will automatically update the field with the current timestamp whenever an article is modified.date the field with the current timestamp any time an article is modified.

Migrate the database

With the Prisma schema defined, you will run migrations to create the actual tables in the database. To generate and execute your first migration, run the following command in the terminal:

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

COPY

This command will do three things:

1. **Save the migration**: Prisma Migrate will take a snapshot of your schema and figure out the SQL commands necessary to carry out the migration. Prisma will save the migration file containing the SQL commands to the newly created `prisma/migrations` folder.
2. **Execute the migration**: Prisma Migrate will execute the SQL in the migration file to create the underlying tables in your database.
3. **Generate Prisma Client**: Prisma will generate Prisma Client based on your latest

> ***Note**: You can learn more about Prisma Migrate in the Prisma docs.*

If completed successfully, you should see a message like this :

```
The following migration(s) have been created and applied from new schema changes:

migrations/
  └─ 20220528101323_init/
    └─ migration.sql

Your database is now in sync with your schema.
...
✓ Generated Prisma Client (3.14.0 | library) to ./node_modules/@prisma/client in 31ms
```

Check the generated migration file to get an idea about what Prisma Migrate is doing behind the scenes:

```
-- prisma/migrations/20220528101323_init/migration.sql

-- CreateTable
CREATE TABLE "Article" (
    "id" SERIAL NOT NULL,
    "title" TEXT NOT NULL,
    "description" TEXT,
    "body" TEXT NOT NULL,
    "published" BOOLEAN NOT NULL DEFAULT false,
    "createdAt" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
    "updatedAt" TIMESTAMP(3) NOT NULL,

    CONSTRAINT "Article_pkey" PRIMARY KEY ("id")
);
```

This is the SQL needed to create the `Article` table inside your PostgreSQL database. It was automatically generated and executed by Prisma based on your Prisma schema.

Seed the database

Currently, the database is empty. So you will create a *seed script* that will populate the database with some dummy data.

Firstly, create a seed file called `prisma/seed.ts`. This file will contain the dummy data and queries needed to seed your database.

```
touch prisma/seed.ts
```

COPY

Then, inside the seed file, add the following code:

```
// prisma/seed.ts

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

// initialize Prisma Client
const prisma = new PrismaClient();

async function main() {
  // create two dummy articles
  const post1 = await prisma.article.upsert({
    where: { title: 'Prisma Adds Support for MongoDB' },
    update: {},
    create: {
      title: 'Prisma Adds Support for MongoDB',
      body: 'Support for MongoDB has been one of the most requested features since the
      description:
```

COPY

```
    update: {},
    create: {
      title: "What's new in Prisma? (Q1/22)",
      body: 'Our engineers have been working hard, issuing new releases with many impr
      description:
        'Learn about everything in the Prisma ecosystem and community from January to
      published: true,
    },
  });

  console.log({ post1, post2 });
}

// execute the main function
main()
  .catch((e) => {
    console.error(e);
    process.exit(1);
  })
  .finally(async () => {
    // close Prisma Client at the end
    await prisma.$disconnect();
  });
```

Inside this script, you first initialize Prisma Client. Then you create two articles using the `prisma.upsert()` function. The `upsert` function will only create a new article if no article matches the `where` condition. You are using an `upsert` query instead of a `create` query because `upsert` removes errors related to accidentally trying to insert the same record twice.

You need to tell Prisma what script to execute when running the seeding command. You can do this by adding the `prisma.seed` key to the end of your `package.json` file:

```
  "dependencies": {
    // ...
  },
  "devDependencies": {
    // ...
  },
  "jest": {
    // ...
  },
+  "prisma": {
+    "seed": "ts-node prisma/seed.ts"
+  }
```

The `seed` command will execute the `prisma/seed.ts` script that you previously defined. This command should work automatically because `ts-node` is already installed as a dev dependency in your `package.json`.

Execute seeding with the following command:

```
npx prisma db seed
```

COPY

You should see the following output:

```
Running seed command `ts-node prisma/seed.ts` ...
{
  post1: {
    id: 1,
    title: 'Prisma Adds Support for MongoDB',
    description: "We are excited to share that today's Prisma ORM release adds stable
    body: 'Support for MongoDB has been one of the most requested features since the i
    published: false,
    createdAt: 2022-04-24T14:20:27.674Z
```

```
    published: true,
    createdAt: 2022-04-24T14:20:27.705Z,
    updatedAt: 2022-04-24T14:20:27.705Z
  }
}

🌱  The seed command has been executed.
```

> ***Note**: You can learn more about seeding in the Prisma Docs.*

Create a Prisma service

Inside your NestJS application, it is good practice to abstract away the Prisma Client API from your application. To do this, you will create a new service that will contain Prisma Client. This service, called `PrismaService`, will be responsible for instantiating a `PrismaClient` instance and connecting to your database.

The Nest CLI gives you an easy way to generate modules and services directly from the CLI. Run the following command in your terminal:

```
npx nest generate module prisma
npx nest generate service prisma
```

COPY

> ***Note 1**: If necessary, refer to the NestJS docs for an introduction to services and modules.*
>
> ***Note 2**: In some cases running the `nest generate` command with the server already running may result in NestJS throwing an exception that says: `Error: Cannot find module './app.controller'`. If you run into this error, run the following command from the terminal: `rm -rf dist` and restart the server.*

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

@Injectable()
export class PrismaService extends PrismaClient {
  async enableShutdownHooks(app: INestApplication) {
    this.$on('beforeExit', async () => {
      await app.close();
    });
  }
}
```

The `enableShutdownHooks` definition is needed to ensure your application shuts down gracefully. More information is available in the NestJS docs.

The Prisma module will be responsible for creating a singleton instance of the `PrismaService` and allow sharing of the service throughout your application. To do this, you will add the `PrismaService` to the `exports` array in the `prisma.module.ts` file:

COPY

```
// src/prisma/prisma.module.ts

import { Module } from '@nestjs/common';
import { PrismaService } from './prisma.service';

@Module({
  providers: [PrismaService],
  exports: [PrismaService],
})
export class PrismaModule {}
```

Now, any module that *imports* the `PrismaModule` will have access to `PrismaService`

Set up Swagger

Swagger is a tool to document your API using the OpenAPI specification. Nest has a dedicated module for Swagger, which you will be using shortly.

Get started by installing the required dependencies:

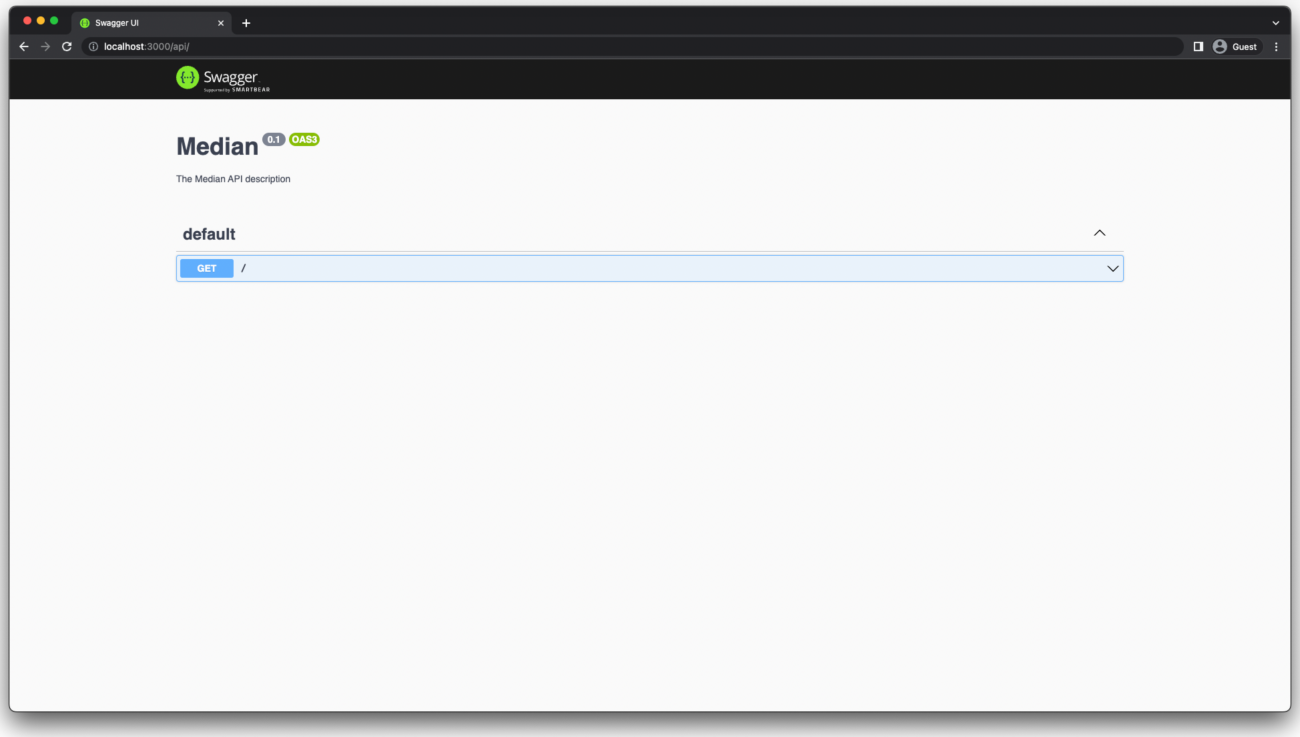
```
npm install --save @nestjs/swagger swagger-ui-express
```

COPY

Now open `main.ts` and initialize Swagger using the `SwaggerModule` class:

```
// src/main.ts

import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  const config = new DocumentBuilder()
    .setTitle('Median')
    .setDescription('The Median API description')
    .setVersion('0.1')
    .build();

  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api', app, document);

  await app.listen(3000);
}
bootstrap();
```

COPY



Swagger User Interface

Implement CRUD operations for `Article` model

In this section, you will implement the Create, Read, Update, and Delete (CRUD) operations for the `Article` model and any accompanying business logic.

Generate REST resources

Before you can implement the REST API, you will need to generate the REST resources for the `Article` model. This can be done quickly using the Nest CLI. Run the following command in your terminal:

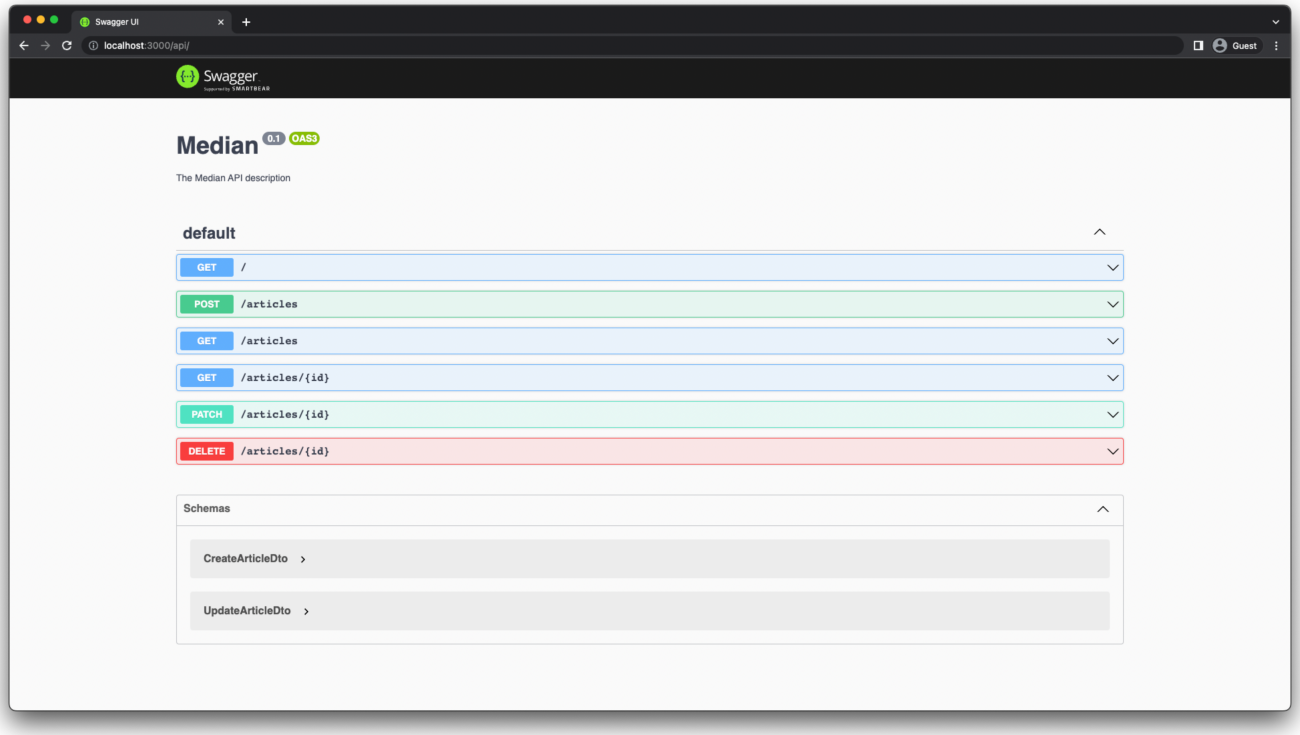
```
npx nest generate resource
```

COPY

3. `Would you like to generate CRUD entry points?` **Yes**

You should now find a new `src/articles` directory with all the boilerplate for your REST endpoints. Inside the `src/articles/articles.controller.ts` file, you will see the definition of different routes (also called route handlers). The business logic for handling each request is encapsulated in the `src/articles/articles.service.ts` file. Currently, this file contains dummy implementations.

If you open the Swagger API page again, you should see something like this:



Auto-generated "articles" endpoints

The `SwaggerModule` searches for all `@Body()`, `@Query()`, and `@Param()` decorators on the route handlers to generate this API page.

Add `PrismaClient` to the `Articles` module

```
import { ArticlesService } from './articles.service';
import { ArticlesController } from './articles.controller';
import { PrismaModule } from 'src/prisma/prisma.module';

@Module({
  controllers: [ArticlesController],
  providers: [ArticlesService],
  imports: [PrismaModule],
})
export class ArticlesModule {}
```

You can now inject the `PrismaService` inside the `ArticlesService` and use it to access the database. To do this, add a constructor to `articles.service.ts` like this:

```
// src/articles/articles.service.ts

import { Injectable } from '@nestjs/common';
import { CreateArticleDto } from './dto/create-article.dto';
import { UpdateArticleDto } from './dto/update-article.dto';
import { PrismaService } from 'src/prisma/prisma.service';

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

  // CRUD operations
}
```

Define `GET /articles` endpoint

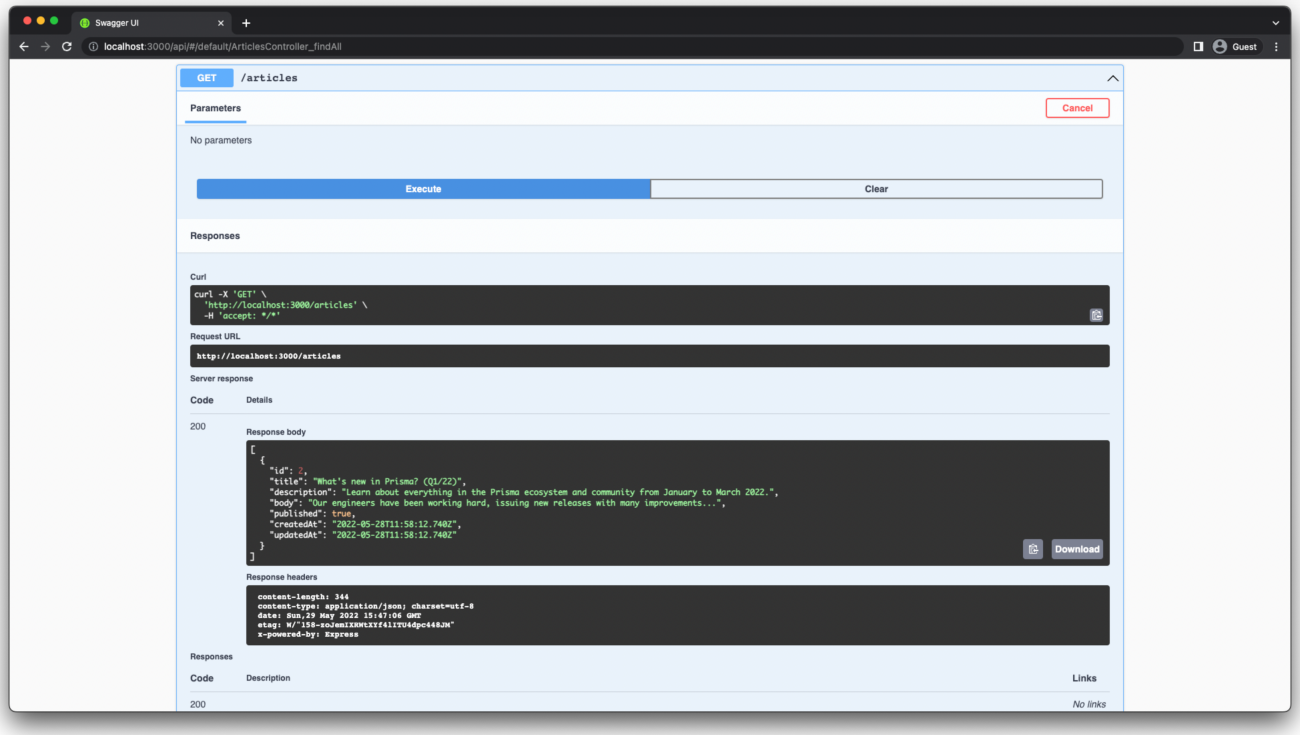
The controller for this endpoint is called `findAll`. This endpoint will return all published articles in the database. The `findAll` controller looks like this:

You need to update `ArticlesService.findAll()` to return an array of all published articles in the database:

```
// src/articles/articles.service.ts

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

  create(createArticleDto: CreateArticleDto) {
    return 'This action adds a new article';
  }

  findAll() {
-   return `This action returns all articles`;
+   return this.prisma.article.findMany({ where: { published: true } });
  }
```

The `findMany` query will return all `article` records that match the `where` condition.

You can test out the endpoint by going to `http://localhost:3000/api` and clicking on the **GET/articles** dropdown menu. Press **Try it out** and then **Execute** to see the result.



> ***Note**: You can also run all requests in the browser directly or through a REST client (like Postman). Swagger also generates the curl commands for each request in case you want to run the HTTP requests in the terminal.*

Define `GET /articles/drafts` endpoint

You will define a new route to fetch all *unpublished* articles. NestJS did not automatically generate the controller route handler for this endpoint, so you have to write it yourself.

```
// src/articles/articles.controller.ts

@Controller('articles')
export class ArticlesController {
  constructor(private readonly articlesService: ArticlesService) {}

  @Post()
```

COPY

```
+  }

   // ...

}
```

Your editor should show an error that no function called `articlesService.findDrafts()` exists. To fix this, implement the `findDrafts` method in `ArticlesService`:

COPY

```
// src/articles/articles.service.ts

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

  create(createArticleDto: CreateArticleDto) {
    return 'This action adds a new article';
  }

+  findDrafts() {
+    return this.prisma.article.findMany({ where: { published: false } });
+  }

  // ...

}
```

The `GET /articles/drafts` endpoint will now be available in the Swagger API page.

> ***Note**: I recommend testing out each endpoint through the Swagger API page once you finish implementing it.*

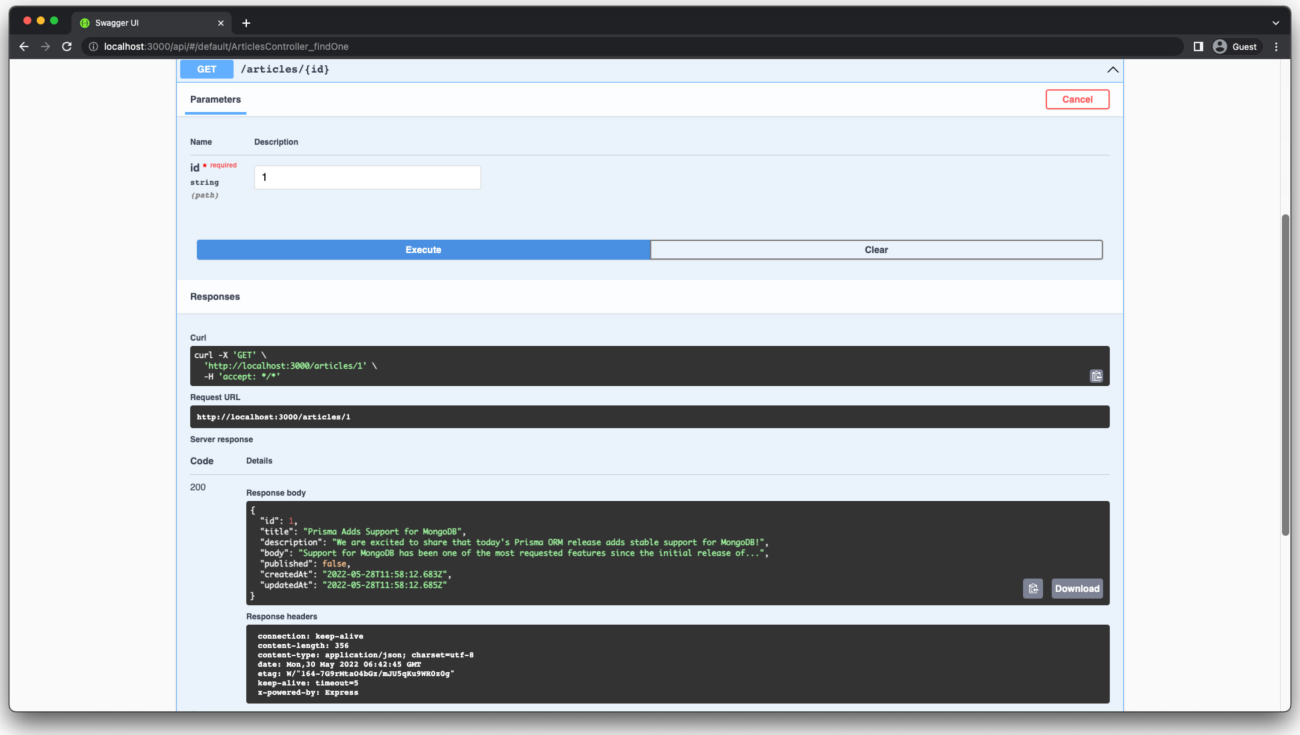
```
@Get(':id')
findOne(@Param('id') id: string) {
  return this.articlesService.findOne(+id);
}
```

The route accepts a dynamic `id` parameter, which is passed to the `findOne` controller route handler. Since the `Article` model has an integer `id` field, the `id` parameter needs to be casted to a number using the `+` operator.

Now, update the `findOne` method in the `ArticlesService` to return the article with the given id:

COPY

```
// src/articles/articles.service.ts
@Injectable()
export class ArticlesService {
  constructor(private prisma: PrismaService) {}

  create(createArticleDto: CreateArticleDto) {
    return 'This action adds a new article';
  }

  findAll() {
    return this.prisma.article.findMany({ where: { published: true } });
  }

  findOne(id: number) {
-    return `This action returns a #${id} article`;
+    return this.prisma.article.findUnique({ where: { id } });
  }
}
```



Define `POST /articles` endpoint

This is the endpoint for creating new articles. The controller route handler for this endpoint is called `create`. It looks like this:

```
// src/articles/articles.controller.ts

@Post()
create(@Body() createArticleDto: CreateArticleDto) {
  return this.articlesService.create(createArticleDto);
}
```

Notice that it expects arguments of type `CreateArticleDto` in the request body. A DTO (Data Transfer Object) is an object that defines how the data will be sent over the network. Currently, the `CreateArticleDto` is an empty class. You will add properties to it to define the shape of the request body.

```
  title: string;

  @ApiProperty({ required: false })
  description?: string;

  @ApiProperty()
  body: string;

  @ApiProperty({ required: false, default: false })
  published?: boolean = false;
}
```

The `@ApiProperty` decorators are required to make the class properties visible to the `SwaggerModule`. More information about this is available in the NestJS docs.

The `CreateArticleDto` should now be defined in the Swagger API page under **Schemas**. The shape of `UpdateArticleDto` is automatically inferred from the `CreateArticleDto` definition. So `UpdateArticleDto` is also defined inside Swagger.

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

  create(createArticleDto: CreateArticleDto) {
-   return 'This action adds a new article';
+   return this.prisma.article.create({ data: createArticleDto });
  }

  // ...
}
```

Define `PATCH /articles/:id` endpoint

This endpoint is for updating existing articles. The route handler for this endpoint is called `update`. It looks like this:

```
// src/articles/articles.controller.ts

@Patch(':id')
update(@Param('id') id: string, @Body() updateArticleDto: UpdateArticleDto) {
  return this.articlesService.update(+id, updateArticleDto);
}
```

The `updateArticleDto` definition is defined as a `PartialType` of `CreateArticleDto`. So it can have all the properties of `CreateArticleDto`.

```
// src/articles/dto/update-article.dto.ts

import { PartialType } from '@nestjs/swagger';
```

COPY

```
// src/articles/articles.service.ts

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

  // ...

  update(id: number, updateArticleDto: UpdateArticleDto) {
-    return `This action updates a #${id} article`;
+    return this.prisma.article.update({
+      where: { id },
+      data: updateArticleDto,
+    });
  }

  // ...
}
```

The `article.update` operation will try to find an `Article` record with the given `id` and update it with the data of `updateArticleDto`.

If no such `Article` record is found in the database, Prisma will return an error. In such cases, the API does not return a user-friendly error message. You will learn about error handling with NestJS in a future tutorial.

Define `DELETE /articles/:id` endpoint

This endpoint is to delete existing articles. The route handler for this endpoint is called `remove`. It looks like this:

```
// src/articles/articles.controller.ts
```

Just like before, go to `ArticlesService` and update the corresponding method:

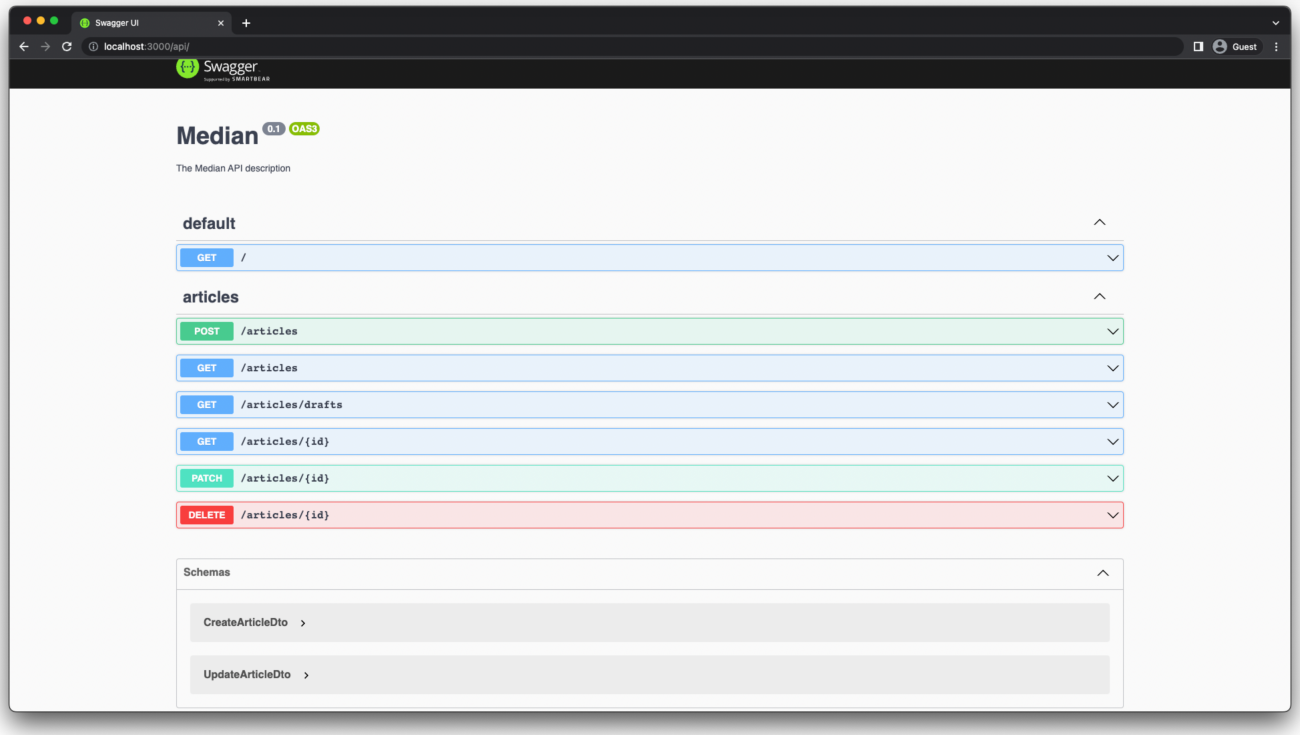
```
// src/articles/articles.service.ts

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

  // ...

  remove(id: number) {
-    return `This action removes a #${id} article`;
+    return this.prisma.article.delete({ where: { id } });
  }
}
```

That was the last operation for the `articles` endpoint. Congratulations your API is almost ready! 🎉

Group endpoints together in Swagger

Add an `@ApiTags` decorator to the `ArticlesController` class, to group all the `articles` endpoints together in Swagger:

```
// src/articles/articles.controller.ts

import { ApiTags } from '@nestjs/swagger';

@Controller('articles')
@ApiTags('articles')
export class ArticlesController {
  // ...
}
```



Update Swagger response types

If you look at the **Responses** tab under each endpoint in Swagger, you will find that the **Description** is empty. This is because Swagger does not know the response types for any of the endpoints. You're going to fix this using a few decorators.

First, you need to define an entity that Swagger can use to identify the shape of the returned `entity` object. To do this, update the `ArticleEntity` class in the `articles.entity.ts` file as follows:

```
// src/articles/entities/article.entity.ts

import { Article } from '@prisma/client';
import { ApiProperty } from '@nestjs/swagger';
```

COPY

```
  description: string | null;

  @ApiProperty()
  body: string;

  @ApiProperty()
  published: boolean;

  @ApiProperty()
  createdAt: Date;

  @ApiProperty()
  updatedAt: Date;
}
```

This is an implementation of the `Article` type generated by Prisma Client, with `@ApiProperty` decorators added to each property.

Now, it's time to annotate the controller route handlers with the correct response types. NestJS has a set of decorators for this purpose.

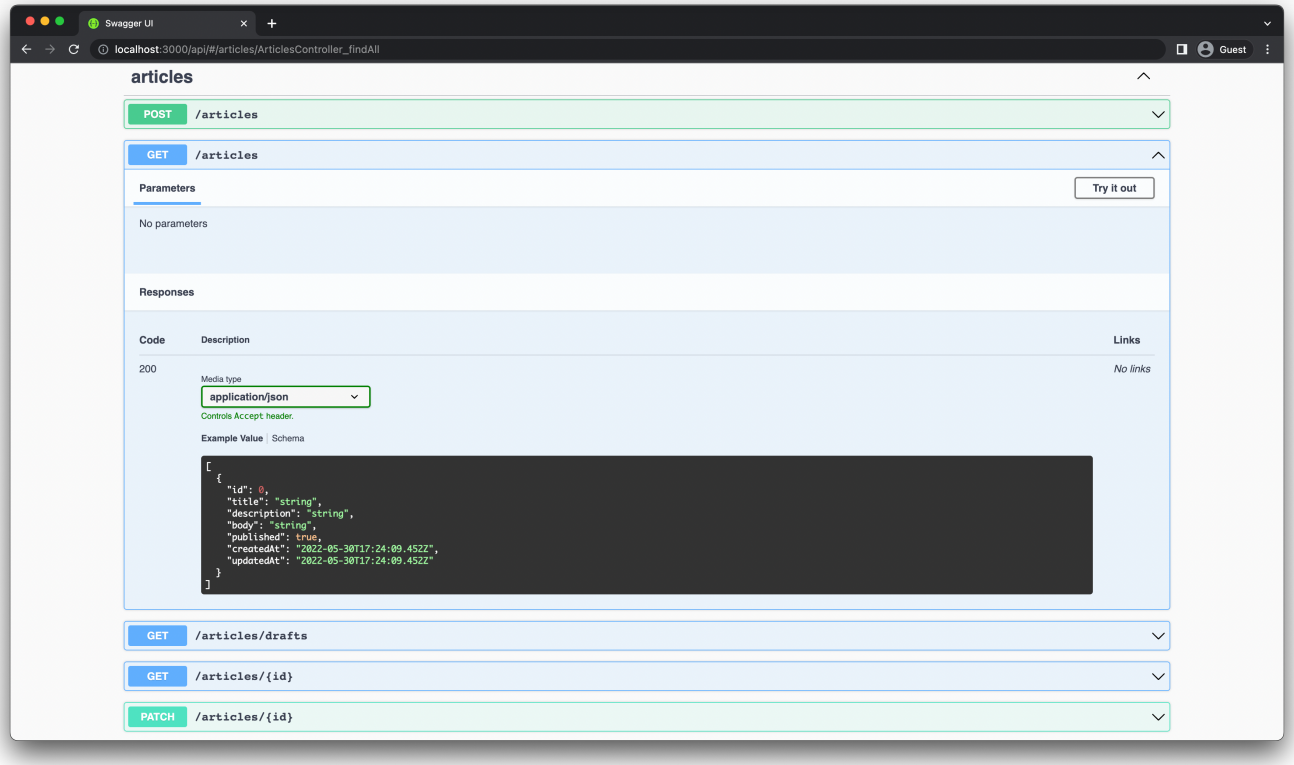
```
// src/articles/articles.controller.ts

+import { ApiCreatedResponse, ApiOkResponse, ApiTags } from '@nestjs/swagger';
+import { ArticleEntity } from './entities/article.entity';

@Controller('articles')
@ApiTags('articles')
export class ArticlesController {
  constructor(private readonly articlesService: ArticlesService) {}

  @Post()
+ @ApiCreatedResponse({ type: ArticleEntity })
```

COPY

```
  }

  @Get('drafts')
+ @ApiOkResponse({ type: ArticleEntity, isArray: true })
  findDrafts() {
    return this.articlesService.findDrafts();
  }

  @Get(':id')
+ @ApiOkResponse({ type: ArticleEntity })
  findOne(@Param('id') id: string) {
    return this.articlesService.findOne(+id);
  }

  @Patch(':id')
+ @ApiOkResponse({ type: ArticleEntity })
  update(@Param('id') id: string, @Body() updateArticleDto: UpdateArticleDto) {
    return this.articlesService.update(+id, updateArticleDto);
  }

  @Delete(':id')
+ @ApiOkResponse({ type: ArticleEntity })
  remove(@Param('id') id: string) {
    return this.articlesService.remove(+id);
  }
}
```

You added the `@ApiOkResponse` for `GET`, `PATCH` and `DELETE` endpoints and `@ApiCreatedResponse` for `POST` endpoints. The `type` property is used to specify the return type. You can find all the response decorators that NestJS provides in the NestJS docs.

Now, Swagger should properly define the response type for all endpoints on the API page.



Summary and final remarks

Congratulations! You've built a rudimentary REST API using NestJS. Throughout this tutorial you:

- Built a REST API with NestJS
- Smoothly integrated Prisma in a NestJS project
- Documented your REST API using Swagger and OpenAPI

One of the main takeaways from this tutorial is how easy it is to build a REST API with NestJS and Prisma. This is an incredibly productive stack for rapidly building well structured, type-safe and maintainable backend applications.

You can find the source code for this project on GitHub. Please feel free to raise an