# Nest.js Step-by-Step

By **Bilal Haidar**
Published in: **CODE Magazine: 2019 - July/August**
Last updated: March 30, 2021

Writing large-scale Node.js applications without a well-planned structure and code organization strategy will certainly end in disaster. Due to the nature of Node.js apps, with ever-changing application requirements coming in, new module files are being added to the solution and imported to a ridiculous extent.

One of the major goals of the Nest.js framework is to give back-end developers a modular code structure and TypeScript experience to help build and maintain their large-scale enterprise Node.js apps without any serious issues.

Heavily inspired by Angular, Nest is a modern Node.js framework built on top of Node.js and TypeScript, and used for building efficient, reliable, and scalable apps.

The brilliance behind Nest is that it makes use of Express.js but also provides compatibility with other libraries, like Fastify. The same applies to databases. I'm going to use PostgreSQL but Nest supports other databases, like MySQL, SQLite, and even MongoDB.

> Nest.js heavily inspired by Angular. Nest has modules and offers dependency injection right out of the box.

## What Am I Building?

This article is the first of a series of articles on the Nest framework. The ultimate goal of this series is to introduce you to Nest concepts and provide you with hands-on experience in writing Nest server-side apps so that you can start adopting it in your own projects.

Throughout this series, I'll be building a full stack app to manage To Do lists. I'll keep the app simple so that there's enough time to explore a variety of features available in the framework.

In a nutshell, I'm going to:

- Build a REST API to manage To Do lists and tasks
- Use the PostgreSQL database to store the data
- Authenticate users with JSON Web Tokens (JWT)
- Allow user registration and log in
- Build a front-end Angular app to allow users to create and manage their To Do lists

In this first installment, I'll start by generating a new Nest.js app and building the REST API based on an in-memory data storage.

You can find the source code for this article and the rest of this series online at www.CODEMag.com and associated with this article, and on GitHub at ttps://github.com/bhaidar/nestjs-todo-app.

## Related Articles

- NestJS Step-by-Step (Part 2)
- Nest.js Step-by-Step: Part 3 (Users and Authentication)
- NestJS Step-by-Step: Connecting NestJS with Angular (Part 4)

# Building Blocks of the Nest Framework

Let's go over the basic building blocks of the Nest framework.

## Modularization

In Nest.js, a module allows you to group together related controllers and service providers into a single code file. Simply put, a module in Nest.js is a TypeScript file decorated by `@Module()`. This decorator provides valuable metadata to the Nest.js framework to know exactly what controllers, service providers, and other related resources will be instantiated and used later by the app code.

Typically, you start your app with a single root module. As the application grows and expands, you start adding more modules to better organize the code. Additional modules could be `Users`, `Authentication`, and `Feature` modules for the different functions offered by the application. In addition, every app needs to have a `Shared` module to place all shared resources among the app controllers so that you don't repeat yourself and scatter the same code multiple times in various locations in your app.

Nest creates a single instance of any module available in the app. You can import the same `Shared` module across multiple other modules and Nest guarantees the creation of a single instance of all the providers shared by the module. What's important here is to configure the `Shared` module to export anything you want other modules to use.

In line with the concept of shared modules, Nest offers the **@Global()** decorator. This decorator allows you to make any modules you want to share with the rest of the application modules available without importing them explicitly.

The Nest module system also comes with a feature called `Dynamic` modules. `Dynamic` modules enable you to produce customizable modules. **Listing 1** is an example of a `Dynamic` module taken from the Nest docs.

Listing 1: Dynamic Module class

```
import { Module, DynamicModule } from '@nestjs/common';
import { createDatabaseProviders } from'./database.providers';
import { Connection } from './connection.provider';
@Module({
    providers: [Connection],
})
export class DatabaseModule {
    static forRoot(entities = [], options?): DynamicModule {
        const providers = createDatabaseProviders(options, entities);
        return {
```

```
            module: DatabaseModule, providers: providers,
            exports: providers,
```

The `Dynamic` module provides a `Connection` object. In addition, based on the options passed for the module's **forRoot()** static function, the module adds more providers and exports them to the module(s) where it's imported later on.

The official docs on Modules are available at https://docs.nestjs.com/modules.

## Controllers

A controller is the common element in all MVC Web frameworks, responsible for handling incoming HTTP requests, processing them (communicating with I/O or databases, etc.), and formulating a proper HTTP response to send back to the caller.

For example, if you request the URL `GET /todo`, Nest routes this request to a designated `Controller` class, typically, the `TodoController` class.

A Controller in Nest.js is decorated with the `@Controler(prefix?: string)` decorator. The prefix defines the REST API base endpoint of this controller. If you omit specifying a prefix value, Nest uses the name of the `Controller` class, excluding the word Controller.

A controller groups a set of `Route` handlers together that are responsible for processing the request under a base endpoint. Route handlers in other MVC frameworks are also known as `Actions`.

An `Action` or `Route` handler is a JavaScript function decorated with a suitable HTTP method decorator that enables Nest.js to select the correct handler based on the incoming `Request`.

```
import { Get, Controller }from '@nestjs/common';
@Controller('/api/todo')export class TodoController {  @Get()  getAllTodo(
```

To get a list of all To Do items in the database, send an HTTP Request to **GET /api/todo**.

Nest supports `@Get`, `@Post()`, `@Put()`, `@Delete()`, `@Header()`, `@UseFilters()`, `@UsePipes()`, `@All()`, and other `Route` handler decorators.

For a complete list of all route handlers available in Nest, check out the official documents available at http://docs.nestjs.com/controllers.

Similar to a component in Angular, once you create a new controller, you have to declare that controller in the module to enable the Nest Dependency Injection system to discover this controller and instantiate it once it's requested.

## Providers

A Provider or Service Provider is a JavaScript class that Nest.js uses to encapsulate related business logic and function into a single class. Nest collects the list of providers defined by all modules in the application, creates a single instance of each provider, and injects the provider instance to any controller requesting it.

Controllers provide the HTTP endpoint of your API and delegate the tasks (connecting to the database, etc.) to the providers or services. This pattern follows the Separation of Concerns and Single Responsibility patterns that belong to the family of SOLID principles (https://en.wikipedia.org/wiki/SOLID).

Nest.js offers the **@Injectable()** decorator that you decorate your service providers with. This decorator is optional, yet it's required when your service provider injects other service providers. The **@Injectable()** decorator helps Nest.js figure out the dependencies required by the service provider when instantiating it.

**Listing 2** is an example of how to define a Service Provider and then inject the provider into the Controller and define both the Controller and Provider inside the module.

Listing 2: Service Provider

```
@Controller( '/api/todo' )
export class TodoController {
    constructor(private todoService: TodoService) { }

    @Get()
    getAllTodo() {
```

```
        return this.todoService.getAllTodo();
    }
}

import { Module } from '@nestjs/common';
import { TodoController } from './todo.controller';
import { TodoService } from './todo.service';

@Module({
    controllers: [TodoController],
    providers: [TodoService],
})
    export class TodoModule
```

The official documents on `Providers` are available at https://docs.nestjs.com/providers.

## Dependency Injection/Inversion of Control

The Nest framework bakes its own implementation of the Dependency Injection system. For the most part, every modern framework makes use of the Dependency Injection/Inversion of Control concept to manage the different components that make up the application, and Nest.js is no different. Nest's backbone depends on Dependency Injections to instantiate and manage all of the dependencies in the application.

# Features for Further Exploration

Here's a short list of the other features in Nest.js framework and some brief discussion. In the articles to come, I'll make use of some of them and explore in more detail.

- **Middleware:** Nest.js middleware is a JavaScript function that has access to the `current request`, `response`, and `next()`` middleware in the pipeline. Nest.js Middleware resembles Express.js middleware. It's important to note that middleware runs before the route handler. The official site on Nest.js middleware is available here: https://docs.nestjs.com/middleware.
- **Exception Filters:** Nest.js comes with a built-in general-purpose global exception filter that handles all thrown exceptions in your application. Specifically, it handles exceptions

that are of type `HttpException` or any other exception class that it inherits from `HttpException`. The framework lets you extend the default exception handler and define your own exception filter, which gives you more control over the exception(s) and lets you decide whether you want to log the exception in a database or some other medium before returning a response to the client. The official documents on exception filters are available at: https://docs.nestjs.com/exception-filters.

- **Pipes:** If you're familiar with Angular, you'll notice that `Pipe` in Nest.js is similar to `Pipe` in Angular. A `Pipe` in Nest.js is a class annotated with the `@Injectable()` decorator. It's used to transform data from one format to another, to validate route handler parameters and to produce a new data format based on input parameters. Nest.js comes with two built-in pipes: `ValidationPipe` and `ParseIntPipe`. For more on pipes, go to: https://docs.nestjs.com/pipes.
- **Guards:** If you're familiar with Angular, you'll notice that `Guard` in Nest.js is similar to that in Angular. A `Guard` in Nest.js is a class annotated with the `@Injectable()` decorator. It extends the `CanActivate` class and implements the `canActivate()` method. A `Guard` decides whether the route handler executes the current request or not, based on the value returned from the `canActivate()` method. In general, `Guards` are used for authentication and authorization purposes. For more information on guards, visit: https://docs.nestjs.com/guards.
- **Interceptors:** `Interceptors` originate from the Aspect-Oriented Programming concepts. In Nest.js, an `Interceptor` is a class annotated with the `@Injectable()` decorator and implements `NestInterceptor`. An interceptor provides implementation for the `intercetpor()` method. An `interceptor` manipulates requests and responses. The official site on interceptors is available at: https://docs.nestjs.com/interceptors.
- **GraphQL:** GraphQL is a language for querying and mutating your data via APIs. Nest.js helps you build a `GraphQL` server for your application by defining a `GraphQL` schema, including resolvers and types. The `GraphQL` in Nest.js is available at: https://docs.nestjs.com/graphql/resolvers-map.
- **WebSockets:** `WebSockets` are used for real-time, bidirectional, and event-based communication. Nest.js packages `WebSockets` under the term `Gateway`. A `Gateway` is a class annotated with the `@WebSocketGateway()` decorator. By default, Nest.js `Gateways` use socket.io and are easily configured to use other libraries including the native `WebSocket` libraries. A `Gateway`, once defined, can communicate with a client in

a bidirectional manner. To delve further into `WebSockets` and all there is to know, go to: https://docs.nestjs.com/websockets/gateways.

This was a brief overview of the main concepts in the Nest framework. Of course, there are plenty of other components to discuss in Nest.js, but I'll save that for the next articles. For now, I'll start building a demo application and discuss the concepts described above and a few others along the way.

> The Nest team offers thorough documents on all aspects of the framework. Make sure to visit their website at http://docs.nestjs.com.

# Demo

Let's get down to the nitty gritty and start coding the application. There are a few hoops to jump through and hurdles to overcome, but each step will ensure that you're building a solidly functioning app.

## Step1

Install the Nest.js CLI by issuing the following command:

```
npm install -g @nestjs/cli
```

The Nest.js CLI is a tool to help you scaffold a new Nest.js application, create modules, controllers, services, and other components. The CLI is available at https://github.com/nestjs/nest-cli.

Now that you've installed the CLI globally on your computer, you'll use it in the next step to create your app.

## Step 2

Create a host folder for your application by issuing the following command:

```
mkdir nest-app
```

## Step 3

Go to the **nest-app** directory and create a new Nest.js app by issuing the following command:

```
nest new server
```

During the process, the CLI asks you a series of questions for additional information.

## Step 4

Navigate to the server directory and issue the following command to start the application:

```
cd server && yarn run start:dev
```

Now that the application is initialized, open a browser and navigate to http://localhost:3000. You'll see the famous **Hello World!** message. If you see this message, the application is up and running.

## Step 5

Let's explore the application files created by opening the solution inside your favorite coding editor, as shown in **Figure 1**. In my case, I've chosen to use Visual Studio Code.

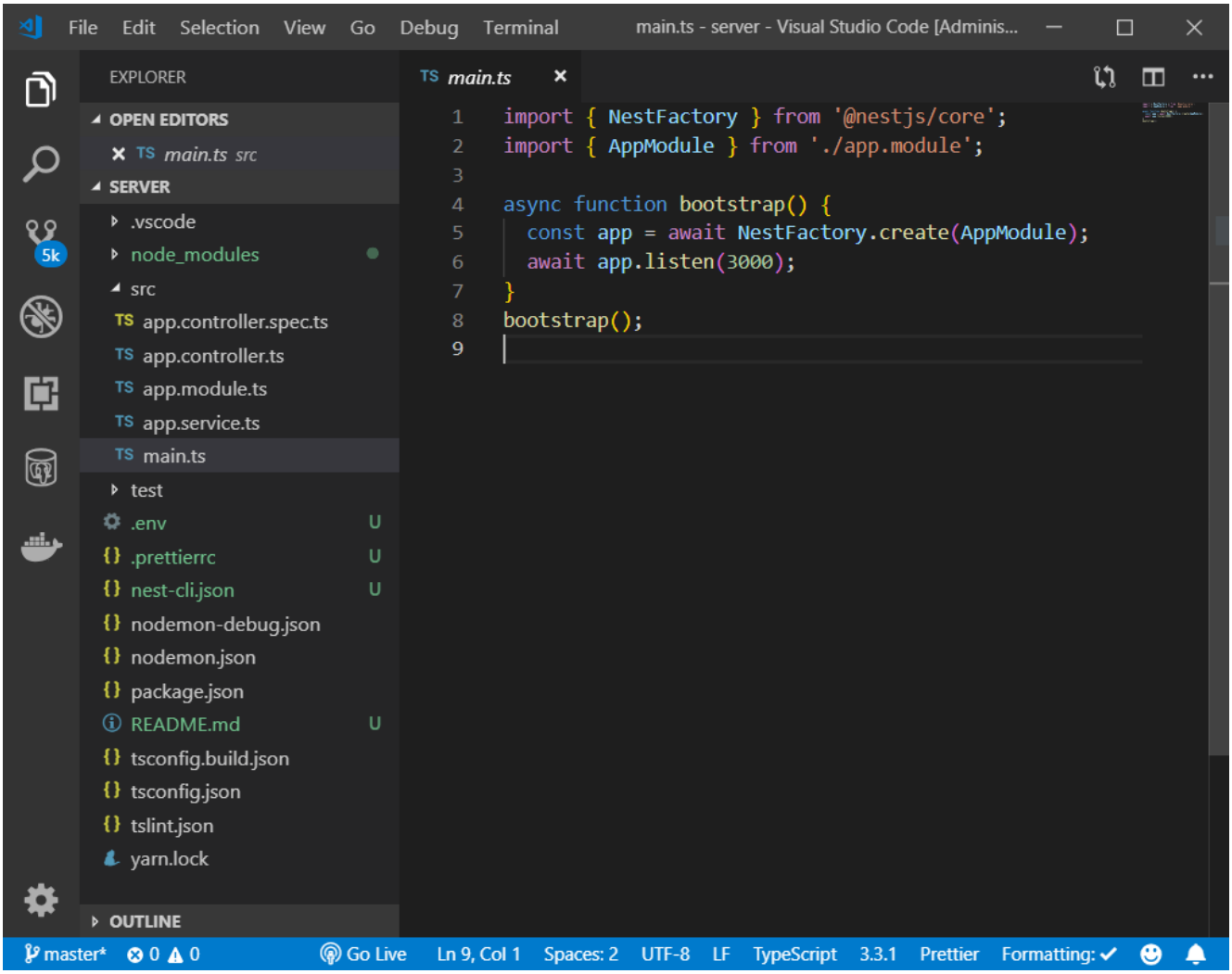


**Figure 1:** Nest.js-generated app files

The Nest.js CLI doesn't add a `.gitignore` file, so you need to start by creating this file.

Similar to Angular apps, Nest.js has a `main.ts` file that bootstraps your app, as you can see in **Figure 1**. You use the `NestFactory.create()` method to create a new app (**~ Express.js app**) that listens to incoming requests at port 3000 by default.

Nest.js CLI scaffolds a root `AppModule` together with an `AppController` and `AppService` classes.

## Step 6

Install the `dotenv NPM` package to easily manage your environment variables in your application. To install `dotenv` issue the following commands:

```
yarn add  dotenv
yarn add @types/dotenv -D
```

Next, create a new `.env` file at the root of the server app. Add the `PORT` variable and assign it a value of 4000.

```
PORT=4000
```

Add the following import at the top of the `main.ts` file:

```
import 'dotenv/config';
```

The import statement loads all the variables defined in the `.env` file and populates the Node.js `process.env` collection with those variables.

Let's make use of the `PORT` environment variable inside `main.ts` instead of hard-coding the value to 3000. In addition, Nest.js comes with a `Logger` class that you can use to log messages to the console. You'll be using this `Logger` to signal that the application has started on the 4000 port.

```
const port = process.env.PORT;

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

    Logger.log(`Server started running onhttp://localhost:${port}`, 'Boots
```

Now when you run the application, you'll see a message stating that the server is running on localhost on port 4000.

## Step 7

Create the To Do module by issuing the following CLI command:

```
nest generate module
```

This command creates a new module named `TodoModule`. In addition, it imports the `TodoModule` into the `AppModule`:

```
@Module({
    imports: [TodoModule],
    controllers: [AppController],
    providers: [AppService],

})
export class AppModule { }
```

## Step 8

Add the `Todo Entity` class under the `/src/todo/entity` directory:

```
export class TodoEntity {
    id: string;
    name: string;
    description?: string;
}
```

## Step 9

Add the following `Data Transfer Objects` (DTO) under the `/src/todo/dto` directory:

The server sends the client a `TodoDto` when the client requests or updates a single To Do item.

```
export class TodoDto {
    id: string;
    name: string;
    description?: string;
}
```

The client sends the server a `TodoCreateDto` when the client wants to create a new To Do item.

The `TodoCreateDto` class is received from the client when it requests to create a new To Do item.

```
export class TodoCreateDto {
    name: string;
    description?: string;
}
```

The server sends the client a list of `TodoDto` in the form of `TodoListDto` when the client requests all To Do items.

```
import { TodoDto } from './todo.dto';
export class TodoListDto {
    todos: TodoDto[];
}
}
```

Usually, you use DTO objects to communicate with the client rather than plain entities. You'll appreciate the use of DTOs when you add support for databases later in this series of articles. What's important to me is that with a DTO, I have the upper hand in deciding what information the client and server can exchange during the course of a single request/response.

## Step 10

Add a new `mapper.ts` file under the `/src/shared` directory with the following content:

```
export const toTodoDto = (data: TodoEntity): TodoDto => {
    const { id, name, description } = data;

    let todoDto: TodoDto = { id, name, description, };
    return todoDto;

};
export const toTodoDto = (data: TodoEntity): TodoDto => {
    const { id, name, description } = data;

    let todoDto: TodoDto = { id, name, description, };

    return todoDto;
};
```

This is a utility function that maps a `TodoEntity` into a `TodoDto`. Typically, you use some sort of a `Mapper` library, but I'm keeping things simple here and crafting my own mappers. During the course of the upcoming articles, I'll revisit this mapper and adjust it accordingly.

## Step 11

Add a new `utils.ts` file under the `/src/shared` directory with the following content:

```
export const toPromise = <T>(data: T): Promise<T> => {
    return new Promise<T>(resolve => { resolve(data);  });
};
```

This is a utility function that converts any data source into a Promise of data.

## Step 12

Create the `todo` controller by issuing the following CLI command:

```
nest generate controller todo
```

This command creates the `TodoController` class and imports it into the `TodoModule` class.

```
import { Controller } from '@nestjs/common';

@Controller('todo')
export class TodoController { }
```

The `TodoController` class is annotated with `@Controller()` decorator. Let's change the default prefix to **/api/todos** as follows:

```
@Controller('api/todos')
```

Let's add a few route handler functions to the TodoController. **Listing 3** shows the code for this controller.

Let's go through the labeled code lines in **Listing 3**:

Listing 3: TodoController class

```
@Controller("api/todos")
export class TodoController {
    /** 1 */
    constructor(private readonly todoService: TodoService) {}
    @Get()
    /** 2 */
    async findAll(): Promise<TodoListDto> {
        /** 3 */
        const todos = await this.todoService.getAllTodo();
```

```
        /** 4 */     return toPromise({ todos });
    }

    /** 5 */
    @Get(":id")
    async findOne(@Param("id") id: string): Promise<TodoDto> {
        return await this.todoService.getOneTodo(id);
    }
```

1. The controller injects the `TodoService` and redirects the work of the route handler to the service.
2. All route handlers are defined as `async` functions making use of the `async/await` pattern in TypeScript and ensuring fast-running handlers.
3. The `findAll()` route handler redirects its output by calling the service's `getAllTodo()` function to return all To Do items.
4. The route handler returns a `Promise<TodoDto>`, making use of the helper function `toPromise()`.
5. The `@Get(':id')` decorator annotates the `findOne()`` function and specifies a route parameter. For instance, a request with URL value of `/api/todos/1` is mapped to this route handler in hand.
6. The `@Post()` decorator annotates the `create()` function that creates a new To Do item.
7. The `@Body()` decorator extracts the payload of the function from the incoming request and converts the data into an instance of the `TodoCreateDto` object.
8. The `@Param(':id')` decorator extracts the `To Do` ID variable from the route parameters and injects it as an input parameter into the `update()` function.

Before you start writing your `TodoService`, let's add some mock data to use in the service.

## Step 13

Add mock data under `/src/mock/todos.mock.ts` file with some mock data, as you can see in **Listing 4** below.

Listing 4: Mock data sample

```
export const todos: TodoEntity[] = [
    {
        id: 'eac400ba-3c78-11e9-b210-d663bd873d93',
        name: 'Supermarket Todo list',
    },
    {
        id: 'eac40736-3c78-11e9-b210-d663bd873d93',
        name: 'Office Todo list',
    },
    {
        id: 'eac408d0-3c78-11e9-b210-d663bd873d93',
        name: 'Traveling Todo  list',
    },
    {
        id: 'eac40a7e-3c78-11e9-b210-d663bd873d93',
        name: 'Studying Todo list',
    },
    {
        id: 'eac40c90-3c78-11e9-b210-d663bd873d93'
```

## Step 14

Create the `TodoService` by issuing the following command:

```
nest generate service todo
```

This command creates the `TodoService` class and imports it into the `TodoModule` class.

Before going any further, let's install the `uuid NPM` package to help generate a unique ID value. You'll need to generate new IDs when creating new To Do items. To install the `uuid` package, issue the following commands:

```
yarn add uuid
yarn add @types/uuid -D
```

Let's implement the service shown in **Listing 5**.

Listing 5: TodoService class

```
import { todos } from 'src/mock/todos.mock';
import * as uuid from 'uuid';

@Injectable()
export class TodoService {
    /** 1 */
    todos: TodoEntity[] = todos;

    async getOneTodo(id: string): Promise<TodoDto> {
        /** 2 */
        const todo = this.todos.find(todo => todo.id === id);

        if (!todo) {
        /** 3 */
        throw new HttpException(`Todo item doesn't exist`, HttpStatus.BAD_
        }

        /** 4 */
```

Let's go through the labeled code lines in **Listing 5**:

1. Assigns the imported mock data to a local variable.
2. Finds a single `To Do` item by ID.
3. If the `To Do` item is not found, the code throws an `HttpException` with a status of Bad Request and status code of 400.
4. The `toTodoDto()` function converts a `TodoEntity` into a `TodoDto` object. Then, the `toPromise()` function resolves the `TodoDto` and returns the results back to the controller.

You can find the rest of the service in the accompanying source code of this article.

## Step 15

Start the app and test the To Do API by issuing the following command:

```
yarn run start:dev
```

Remember to attach the **Content-Type: application/json** to POST requests when using Nest.js framework.

In **Figure 2**, you create a new To Do item by sending a `POST /api/todos` request, with a To Do item in the request payload using Postman client

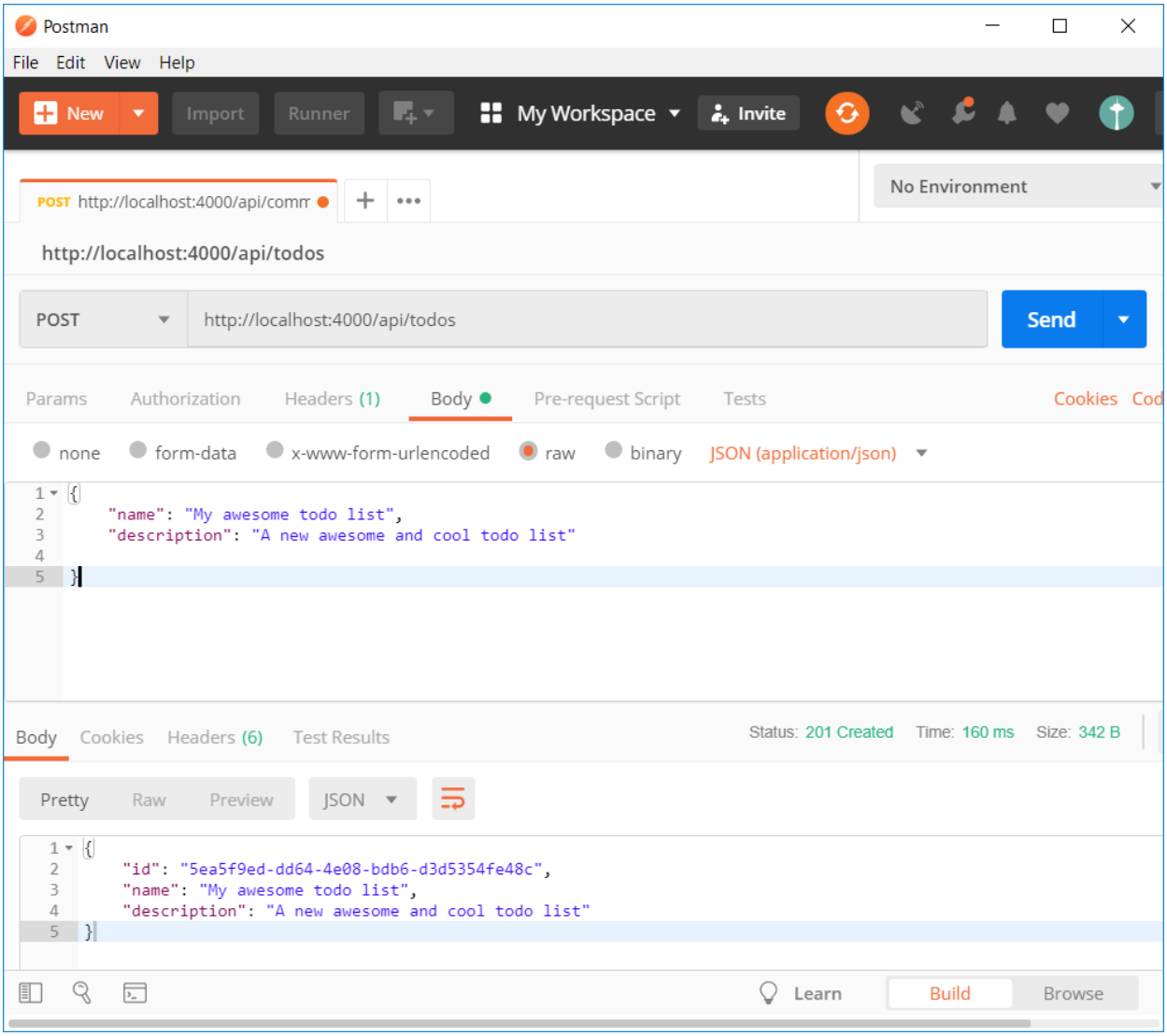


**Figure 2:** Create a TO DO item using a POST request.

In **Figure 3,** you verify that the new To Do item is now saved in the data by issuing a `Get /api/todos` request.

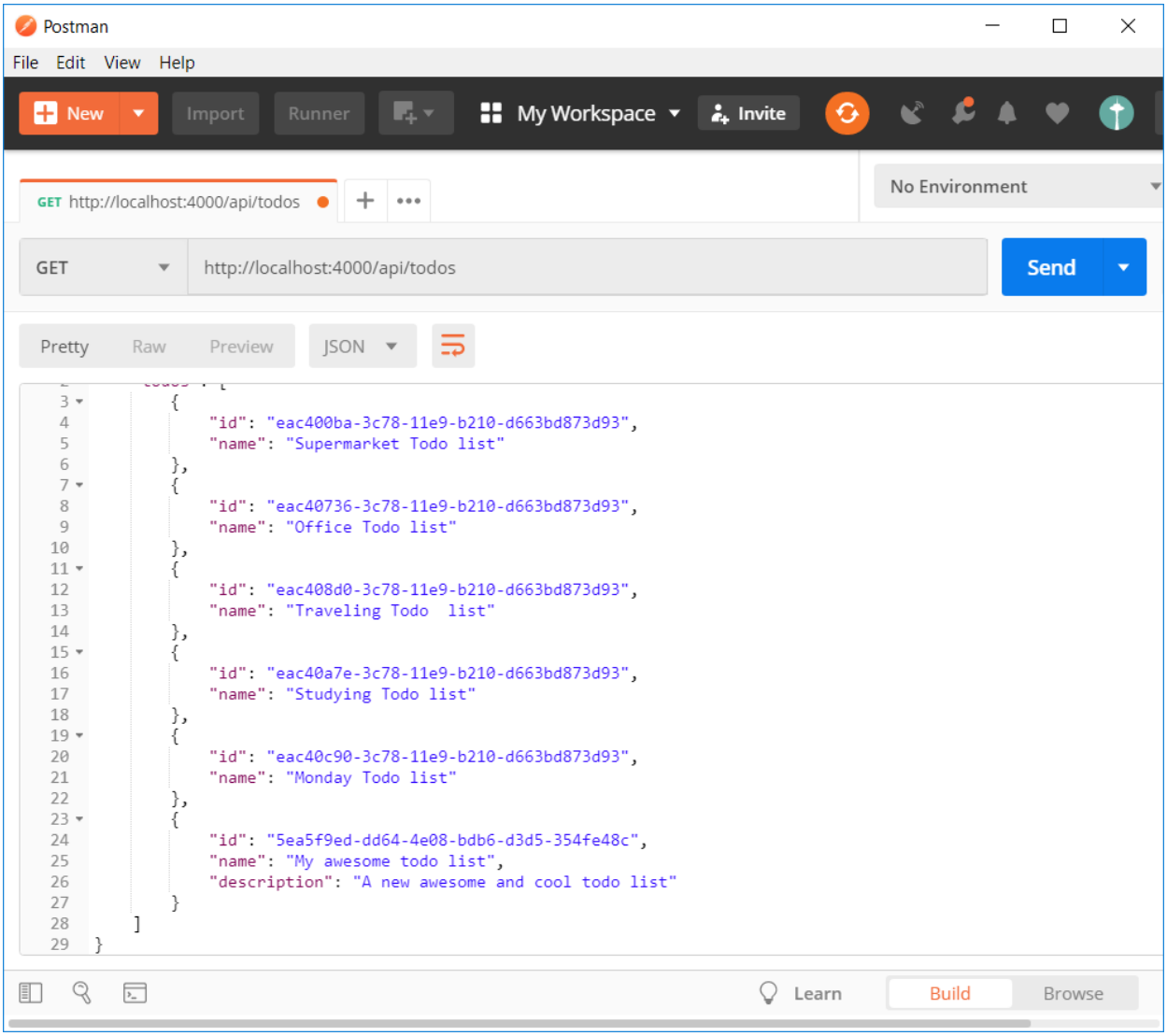


**Figure 3:** The GET request to verify the TO DO item is created.

You may continue testing the rest of the API function on your own.

I'll switch offline now and add the necessary code required to allow the user to create Tasks under a To Do list.

# What's Next?

In the rest of the series, you will further build and improve this app by:

- Adding a PostgreSQL database to store the data. Nest integrates with TypeORM, a well-established and mature JavaScript ORM, and I'll be showing you how to add it to your app and use it to write your queries and mutations against the database.

- Adding a user module together with authentication and authorization module.
- Adding a front-end Angular app so that users can access the app from a browser and start using it.
- Optional: Adding a GraphQL back-end server to serve clients using GraphQL queries and mutations.
- Optional: Adding WebSockets to the mix and providing a real-time experience to the clients.

# Conclusion

With the intention of providing some insight into the Nest framework capabilities, this first installment of my new Nest.js series sheds light on the building blocks and starts the buildup of the full stack To Do list application.

Stick around to discover how to make use of more features of Nest, to better develop a back-end Node.js application.

## Related Articles

- NestJS Step-by-Step (Part 2)
- Nest.js Step-by-Step: Part 3 (Users and Authentication)
- NestJS Step-by-Step: Connecting NestJS with Angular (Part 4)

## Nest is a Node.js Framework

Nest uses modern JavaScript, is built with TypeScript (preserves compatibility with pure JavaScript), and combines elements of OOP, FP, and FRP.

## Nest.js and Angular

If you're an Angular developer, the concept of Nest Service Provider sounds familiar to you and resembles the same concept of providers in Angular.