Ultimate Courses™

NestJS Build a RESTful CRUD API

by Kerry Ritter

Build your first **NestJS** app with the CLI from scratch, CRUD operations and integrate a UI.

Contents

Welcome

Follow @kerryritter

Hi! I'm Kerry Ritter, the author of this eBook and course author at Ultimate Courses[1]. I have been working in NestJS since it was released and I'm excited to write this eBook to help you get started on your journey with this fantastic framework.

NestJS has been exploding in popularity since its release in 2017 - and for good reason. NestJS provides a great way to build server-side Node applications with modern tooling like TypeScript and the latest ECMAScript. With architecture patterns that will be familiar to developers of many backgrounds - most notably Angular, Java, and ASP.NET - NestJS is fairly easy to get started and be productive with.

In this eBook, you will:

- Dive into NestJS fundamentals
- Discover how to create NestJS applications
- Build your first NestJS REST API
- Use your REST API with an existing user interface

Whether you're new to Node or new to server-side development, this eBook will guide you in successfully getting started with NestJS and TypeScript.

If by the end of this eBook you're as excited about NestJS as I am, I highly recommend you pre-order the NestJS Basics and NestJS Pro courses coming soon at Ultimate Courses[2].

Thank you for reading and I hope you find this eBook helpful. If you do, let me know in a tweet[3]!

[1]https://www.ultimatecourses.com
[2]https://www.ultimatecourses.com
[3]https://www.twitter.com/kerryritter

Introduction to NestJS

What is NestJS?

NestJS is a JavaScript application framework for building efficient, scalable Node.js server-side applications. It uses progressive JavaScript, is built with and fully supports TypeScript - yet still enables developers to code in pure JavaScript - and combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming).

That's a technical way of saying that NestJS enables you to build readable, maintainable, and enterprise-ready applications on the Node platform. For developers in the ASP.NET or Java worlds, this means you can now write applications in Node using tools and paradigms you might find quite familiar - tools such as decorator-driven (also known as attribute-driven) controller routing, inversion-of-control techniques, and MV* application architectures. For developers in the Angular world, this means you can use techniques you've used in your client-side Angular code on the server-side - techniques such as module containers, injectable services, pipes, guards, and more.

Unlike many other popular Node-based web frameworks, NestJS takes a fairly opinionated approach. NestJS provides refined, standardized architecture patterns that are supported by the CLI and encouraged by the framework design. The framework also provides a fair amount of tooling out of the box: a dependency injection container, end-to-end and unit test suite setups, and scripts for production builds, a development server, formatting, and linting. While not immediately packaged with tools for configuration (`@nestjs/config`) or database interactions (`@nestjs/typeorm` or `@nestjs/mongoose` being the popular two), the modularity pattern built into NestJS makes it very easy to bolt functionality such as these into your application rather quickly.

What is TypeScript?

As mentioned, NestJS is a JavaScript framework for NodeJS that is built using TypeScript. If you're unfamiliar with TypeScript, this can be a confusing statement - how can a framework written in one language be targeting another language? The explanation is a rather simple premise: TypeScript is a superset of JavaScript and all TypeScript code is "transpiled" to JavaScript code. Being a "superset language" means that all JavaScript code is inherently valid TypeScript code - all of the added syntactical sugar and functionality is optionally added on top of the existing JavaScript language. Transpilation is the process in which code written in one language is converted or compiled into another language. For TypeScript, this means the new features are converted into code that runs in pure JavaScript environments and abstract features like data typing is stripped out of the actual executable output. Take the following example:

```typescript
// Written TypeScript Code
class Dog {
  constructor(
    readonly name: string,
    readonly age: number,
  ) {
  }

  getDetailOutput(): string {
    return `${this.name} is ${this.age} years old.`;
  }
}

console.log(new Dog('Max', 10).getDetailOutput());
console.log(new Dog('Socket', 6).getDetailOutput());
```

```javascript
// Transpiled JavaScript Output
"use strict";
class Dog {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    getDetailOutput() {
        return `${this.name} is ${this.age} years old.`;
    }
}

console.log(new Dog('Max', 10).getDetailOutput());
console.log(new Dog('Socket', 6).getDetailOutput());
```

You may notice that in the TypeScript `constructor`, there are some techniques that you won't see in JavaScript code. The most obvious differences are the `: string` and `: number` declarations. These type declarations are a way to tell the transpiler and the intellisense tooling in your code-editor that you expect these parameters to be of a certain data type. So, if I try to do something like this, I will get an error. This type mismatch will cause the TypeScript transpilation to fail because the constructor expects a `string` and a `number`, not two `strings`.

```typescript
console.log(new Dog('Max', 'ten').getDetailOutput());
                        // ^ 'ten' is not a number!
```

You may also notice that the JavaScript code does not have any reference to input or output types. This is because typing is only a concept used in TypeScript by the TypeScript compiler and integrated intellisense tools; JavaScript run-times such as in Node or your browser cannot interpret this additional syntax, so the compiler removes them.

Similarly, the `getDetailOutput()` method has a return type declaration of `string`. This tells the consumer that this method will return a string, and the transpiler and intellisense tools will respond accordingly.

In addition to these type declarations, you may have noticed that the `readonly` modifier in the constructor. This is a future feature of JavaScript that is not currently available and is another aspect that makes using TypeScript such an enjoyable experience: TypeScript developers get to use many of the future JavaScript features today! As you may see in the two code examples, the `readonly` modifier is a short-hand way to automatically assign the `constructor` parameters to class-level variables on the instance of `Dog`. This and many other features are immediately available in the TypeScript language, but not yet available when using pure JavaScript and Node.

TypeScript has a plethora of functionality and features to offer to make your life as a developer easier. While the examples in this eBook will be written in TypeScript, we will be using the simpler aspects of the language; a deep understanding of TypeScript is not a prerequisite to being productive in this eBook! However, if you'd like to master the TypeScript language, Ultimate Courses offers courses that will take you from beginner to expert[4].

What about Express?

If you're familiar with the NodeJS world, you've likely heard of Express. It is one of the most installed and most depended-on NPM packages of all time. So why, then, do so many people opt to use NestJS?

Express is, by its own tagline, an "unopinionated, minimalist" framework. This means that Express offers a relatively small and concise API without putting too many restrictions on how to architect or organize your codebase. While this may be desirable, many prefer a more opinionated, structured, and streamlined methodology.

Express's unopinionated nature allows developers to create their own architecture and design. For some, this manifests into an MVC-esque controllers-based approach; for others, this leads to a more functional-style route-to-function approach. While this gives the developer many options when creating applications, it also means that there is no standard application architecture or design in the Express world. This leads to an ecosystem that has no particularly focused or guided way of solving core problems like database interactions, configuration, code extensibility, or modularity. Again this is not inherently bad or undesirable; NestJS's opinionated design is purely an alternative approach that provides the structure and standardization that many prefer.

However, it is important to note that using NestJS does not forgo the use of Express - quite the opposite! By default, Express is the web framework that powers NestJS under the hood. This means all the tooling available to the Express ecosystem - such as `helmet` for security, `multer` for file uploads, `cookie-parser` for cookie management - is available to you in NestJS. NestJS merely creates

[4]https://ultimatecourses.com/courses/typescript

structured and dependency injection-friendly ways to interact with the Express pipeline, such as by using abstract base classes for `@Injectable` middleware and decorators for tools like `@Pipes` and `@Guards`. While the implementation may not look exactly the same, at the end of the day you are still working with Express under the hood, just in a more opinionated and structured fashion.

Note that Express is the underlying HTTP framework used by NestJS by default: NestJS's design allows for changing out what underlying platform is being used to handle web requests. Another popular framework that is supported by NestJS is Fastify, a library focused on high-performance web applications. Leveraging NestJS allows you to swap out these underlying platforms with relative ease!

Review

Hopefully this section has left you with the following take-aways as you begin your journey into NestJS:

- NestJS is an opinionated NodeJS application framework.
- NestJS is built with TypeScript, but you can write your applications in TypeScript or JavaScript.
- NestJS uses Express under the hood and provides mechanisms to use Express paradigms like middleware and routing.

System Setup & Installing Prerequisites

Install NodeJS

NestJS is a framework that runs on NodeJS. To get started, you'll need to install the NodeJS runtime onto your machine. The simplest way to do this is by going to https://www.nodejs.org and downloading the LTS version for your operating system. It is also a recommended idea to use a tool called `nvm` (https://github.com/nvm-sh/nvm) or `nvm-windows` (https://github.com/coreybutler/nvm-windows); these tools are version managers for NodeJS that make it very easy to install, uninstall, or change the version of NodeJS that is being used on your machine.

Once installed, open your command-line tool of choice and verify your installation with these two simple commands to check the version of NodeJS and NPM. While NestJS only requires Node 10 or above, it is best practice to stay up-to-date with the current LTS version of Node.

```
node -v
npm -v
```

Install the NestJS CLI

NestJS provides a command-line interface that offers project scaffolding, resource generation, and scripts for building, testing, formatting, and linting. While not a requirement, it is highly recommended to leverage the CLI to simplify these processes. The NestJS CLI is installable via NPM as a global package:

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

Once installed, open your command-line tool of choice and verify your installation with this command to check the version of NestJS. This eBook leverages version 7 of NestJS, but should be compatible with future major versions.

```
nest -v
```

NestJS CLI Basics

The CLI tool allows you to simplify a number of processes that would otherwise be done by hand. While we won't cover all of the CLI, let's review some of the most helpful commands available.

nest new

This command kicks off a wizard that will scaffold a new NestJS web application. It automatically generates your project structure, required NPM dependencies, and provides scripts for building, linting, formatting, and testing.

nest start

This command handles the transpilation of your TypeScript or JavaScript code and kicks off the NestJS web server at `http://localhost:3000` by default. Note that passing the `--watch` flag enables the file watcher that will automatically transpile your code and restart the web server whenever a file in `src` changes.

nest generate

This command allows you to generate any of the NestJS elements, such as modules, controllers, services, pipes, guards, and more. In addition to the element you specified, it will also set up a `.spec` test suite file that is immediately ready for you to write your unit tests.

Postman

Postman is a fantastic tool that makes it easy to send web requests to APIs. In this eBook, we will be making web requests to our NestJS API, and using Postman will make testing these requests much simpler. The Postman API client can be downloaded at https://www.postman.com/downloads/.

Starting Your Application

Now that you have an idea of what NestJS is all about and you have installed the tools that you need, let's get started!

Scaffolding the project with the NestJS CLI.

To create a new NestJS application, open your command-line tool of choice, then find your preferred working directory. Once there, run the following command to scaffold a new application named "todo-list-api".

```
nest new todo-list-api
```

The CLI will generate some files for you and then ask you which package manager you prefer to use. In this eBook, we will be using `npm` and recommend you do the same. Once selected, the CLI will begin installing all of the required dependencies. You can now open the project at `todo-list-api` in your code editor of choice; I recommend Visual Studio Code[5] because of its lightweight nature and excellent TypeScript support.

Review of the default project structure.

At the root level of the project, you will see configuration files needed for the NestJS tooling, particularly for building, linting, and formatting. You are welcome to modify these to suit your preferences, but the default settings are a great starting point.

package.json default scripts

Most notably in your new project code base, you will see `package.json`. This is the file used by NPM to manage dependencies, provide script commands, and provide basic project definition. Inside this file, the NestJS CLI has set up useful scripts that we will use throughout this eBook. These scripts can be run with `npm run {script name}`.

npm run build

This command will transpile your TypeScript code into Node-ready JavaScript code. The output of this process can be found at `dist`. These contents will be what you deploy to your server.

[5]https://code.visualstudio.com/

npm run format

This useful command leverages the library `prettier` to format the code to a uniform style guide. This is a great tool, especially for teams, to create a cohesive and uniform code base. Typically you will want to run this before each commit or code change.

npm run start

This command runs the code in the `dist` folder created by the `npm run build` command to start the NestJS web server. This command is typically used on the server where your code is running.

npm run start:dev

This is similar to the `npm run start` command, but it uses the `--watch` flag to enable a file-watcher that automatically rebuilds your code and restarts the web server. This command is what you will want to use when developing on your machine.

Application Files

In NestJS applications, all of the application source code lives in the `src` folder. Opening this folder, you will see four TypeScript files and one test suite.

main.ts

`main.ts` is the entry point of your NestJS application; when the `nest start` is executed, it finds the `main.ts` file and executes it. This file is particularly small and straight-forward. Inside of the `bootstrap()` method, the `NestFactory` class generates an `INestApplication` (the `app` variable) by instantiating the `AppModule` class (outlined below).

Once you have your `INestApplication`, you can apply global middleware, make global application changes, and start your process - much like one would do when using Express. In our case, which is set up by NestJS by default, we want to start the Express web server. This is done with the `app.listen()` method. The application will run on whatever port you supply to the `listen` method which is `3000` by default.

app.controller.ts

The `AppController` is a basic NestJS controller that provides an endpoint at `http://localhost:3000`. By default, a service is injected into the class to demonstrate the dependency injection process.

app.controller.spec.ts

This spec file is a test suite for the `AppController` class. Suites like these are automatically generated whenever you create elements using the `nest generate` command and provide a great boilerplate to get started with your tests.

app.service.ts

The `AppService` is a simple NestJS injectable provider. In this default class we return a 'hello world' message, but in real-world applications you will see these service classes doing the bulk of the business logic work.

app.module.ts

The `AppModule` is your root module. Modules are used as the dependency injection container and is where all of your controllers and providers will be registered for use. As you will see, the `AppController` and `AppService` classes are registered here for the application to use.

It is common practice to keep this particular `AppModule` module going forward and import all subsequent modules in the `imports` array.

Serve the application

Now that you have seen the default setup of a NestJS application, let's serve the application and access the default route in your browser.

1. In your command-line tool of choice, run the `npm run start:dev` command in your application directory - you should see the `package.json` file in this directory. You'll see the NestJS process creating your `INestApplication`, wiring up all of the application routes, and start listening at `http://localhost:3000`.

```
[NestFactory] Starting Nest application...
[InstanceLoader] AppModule dependencies initialized +10ms
[RoutesResolver] AppController {}: +5ms
[RouterExplorer] Mapped {, GET} route +3ms
[NestApplication] Nest application successfully started +4ms
```

2. In your browser, go to `http://localhost:3000`. You will be greeted with the "Hello World!" message.

Congratulations - you ran your first NestJS application! In the next section, we will start extending this default application with our own routes and services.

Setting up the Todo Module

Understanding NestJS module structure

The NestJS application architecture takes a "per-feature" module approach, akin to what you may see in the Angular or Python Django worlds. With this design, all of the classes and functional layers required to make a single feature functional (i.e. CRUD actions for a database model) will be grouped inside of one folder and injected into a module.

ASP.NET, Java, and other communities often employ a pattern that groups classes by type (i.e. a `Controllers` folder, a `Services` folder, a `Models` folder). For example, an ASP.NET MVC API application structure may look like this:

```
Controllers/
|--- TodoController.cs
Services/
|--- TodoService.cs
Models/
|--- TodoEntity.cs
```

Following the NestJS structure, your app will instead look like this:

```
todo/
|--- todo.controller.ts
|--- todo.entity.ts
|--- todo.module.ts
|--- todo.service.ts
```

This may be a shift in thinking if you're accustomed to the former approach, but grouping functionality per feature can be a great paradigm that keeps related code easy to discover. This approach makes it rather painless to pick up and move large portions of functionality to new codebases or into packagable libraries, and also makes easy to split up code responsibilities to teams or team members.

In the above example, I noted three elements of a NestJS modules that are important to understand before going forward: modules, controllers, and providers.

Modules

Modules are where all of the controllers and providers are registered, along with all of the imported sub-modules which contain their own controllers and providers. Modules hook up the dependency injection containers and enable the resolving of required dependencies for controllers and providers. In NestJS, modules are created by decorating a class with the `@Module` decorator. A simple example of a Module class would look like:

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

The `@Module` decorator takes in an object with four properties:

- `imports`: Importing other `@Module` classes allows for using other sub-modules and creating the hierarchy of functionality. For instance, the `AppModule` will be importing the `TodoModule`, enabling the `TodoModule` functionality to be enabled at the root of the application. This is also how third-party modules such as `@nestjs/typeorm`, `@nestjs/config`, and others are wired into your application.
- `controllers`: The `controllers` array contains the `@Controller` classes that are registered to this module. By defining them in this array, they are made accessible via HTTP requests.
- `providers`: The `providers` array contains all `@Injectable` classes such as data services, instance factories, pipes, guards, middleware and interceptors.
- `exports`: This array allows for making providers and imports registered to this module available for use outside of the current module. For example, a `TodoService` registered to a `TodoModule` is only available classes inside the `TodoModule`. Once `TodoService` is added to the exports array, it is now available to outside modules.

Controllers

Controllers are responsible for interpreting input, passing the request details to services, and then formatting the service results in a way that can be exposed as responses. In the case of NestJS, "Controller" specifically refers to a class that handles HTTP requests.

In NestJS, controllers are decorated with the `@Controller` decorator. By passing a string into the `@Controller` argument, we define a base route path for all of the routes inside of the controller. The routes inside this class are notated with HTTP method decorators and path designations.

It is best practice to keep controllers relatively "thin" and avoid putting significant business logic, data logic, or otherwise processing logic in your controller route methods: that work belongs in a provider service. By keeping the controller's responsibility clearly defined as input and output transformation, it becomes easier to write, read, design, and test your code.

A simple example of a Controller class would look like:

```
@Controller('api/app')
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get('health')
  getHealth(): HealthCheck {
    return this.appService.getHealth();
  }
}
```

Providers

Providers is a blanket term in NestJS that encompasses service classes, repositories, factories, utilities classes, and anything of that nature. This also includes important NestJS elements like middleware and guards. These are marked with the `@Injectable()` decorator.

Create the `Todo` directory and files

To start building our custom `Todo` module, let's first create the empty files we will need. While the NestJS CLI has some excellent generation tools that we can use (which we will review later!), for this exercise we will create them by hand to demonstrate how these files all work together.

Inside of your application directory, make the following directory and files:

- todo/
- todo/todo.controller.ts
- todo/todo.interface.ts
- todo/todo.module.ts
- todo/todo.service.ts

Create the `Todo` interface

Inside of the `todo/todo.interface.ts` file, we will define the shape of the `Todo` JSON object. Note that interfaces are different from classes in that interfaces are "abstract" - there is no JavaScript code output and they are purely used for the TypeScript compiler and intellisense tooling - while classes are actual instantiations in memory.

To create an interface, use the TypeScript keyword `interface`.

```typescript
// todo.interface.ts
interface Todo {
}
```

Our `Todo` class will have three properties:

- an optional `id` (optional new `Todo` requests will auto-assign the `id`)
- a `label` string
- a `complete` boolean flag

We will add these to the interface as such:

```typescript
// todo.interface.ts
interface Todo {
  id?: number;
  label: string;
  complete: boolean;
}
```

Now that we've defined the structure, we will need to make this interface available to the other TypeScript files using the keyword `export`. Without this keyword, the interface is essentially private to the file in which it is defined. In the end, your `todo.interface.ts` file should look as such:

```typescript
export interface Todo {
  id?: number;
  label: string;
  complete: boolean;
}
```

Create the `TodoService` data service

Following the prescribed NestJS architecture pattern, we will create a `TodoService` class that handle the data logic for maintaining our `Todos`. Inside of the `todo.service.ts` file, we will define an exported TypeScript class, `TodoService`, and decorate it with the `@Injectable` decorator.

```
// todo.service.ts
import { Injectable } from '@nestjs/common';

@Injectable()
export class TodoService {
}
```

In this eBook, we will be using an in-memory array as our storage mechanism. This will not persist and will be cleared out every time the development server is restarted, but it will serve as a simple mechanism to start your NestJS journey. To create this array, we will define and instantiate a `storage` class-level variable. Once added, your `todo.service.ts` class should look as such:

```
import { Injectable } from '@nestjs/common';
import { Todo } from './todo.interface';

@Injectable()
export class TodoService {
  private storage: Todo[] = [];
}
```

Note that NestJS providers are singletons by default and any injection of the `TodoService` will share the same in-memory array.

Create the `TodoController` REST API Controller

We now need a Controller to handle our HTTP requests. To do so, we create and export a `TodoController` class that is decorated with the NestJS `@Controller` decorator with the base route path of "todo".

```
// todo.controller.ts
import { Controller } from '@nestjs/common';

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

To leverage the `TodoService` we created, we will use NestJS's dependency injection through constructor injection. Using TypeScript's constructor assignment feature, we can easily set the provider as a private, readonly class-level instance property.

```
// todo.controller.ts
import { Controller } from '@nestjs/common';
import { TodoService } from './todo.service';

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

NestJS will analyze the parameters in the constructor, find the types in the dependency injection container, and resolve them. We will now be able to use the `TodoService` in controller methods via `this.todoService`.

To validate our module, we will set up a single GET route that returns an empty array (note the added import). Routes are enabled using HTTP method decorators: `@Get`, `@Post`, `@Put`, `@Patch`, and `@Delete`. Just like the parameter passed to `@Controller`, the value passed to the method decorator defines the path of the route. In the case of this GET request, we will leave it as undefined, ultimately resolving to the base route path of `/todo`.

```
import { Controller, Get } from '@nestjs/common';
import { TodoService } from './todo.service';
import { Todo } from './todo.interface';

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

  @Get()
  findAll(): Todo[] {
    return [];
  }
}
```

Create the `TodoModule` module

With our Controller and Provider classes created, we now need to create a class and decorate it with the `@Module` decorator to register them to the NestJS application. To do so, we'll create a `TodoModule` class and decorate it with the `@Module` decorator.

```
// todo.module.ts
import { Module } from '@nestjs/common';

@Module({})
export class TodoModule {}
```

Now we can add the `TodoService` as a Provider to this module to make it available for injection.

```
// todo.module.ts
import { Module } from '@nestjs/common';
import { TodoService } from './todo.service';

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

With the `TodoService` now registered, we can register the `TodoController` which will resolve the service as a dependency.

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

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

Import `TodoModule` into `AppModule`

We now have a contained module of functionality, from the HTTP request to the data storage. However, the root module `AppModule` we saw in the previous chapters has no reference to `TodoModule`. As we know, our entrypoint `main.ts` creates the `INestApplication` from the `AppModule`, so without this reference, our `TodoModule` logic is unavailable.

To create this link between `AppModule` and `TodoModule`, open the `app.module.ts` file. In this `@Module` decorator, add `TodoModule` to the `imports` array.

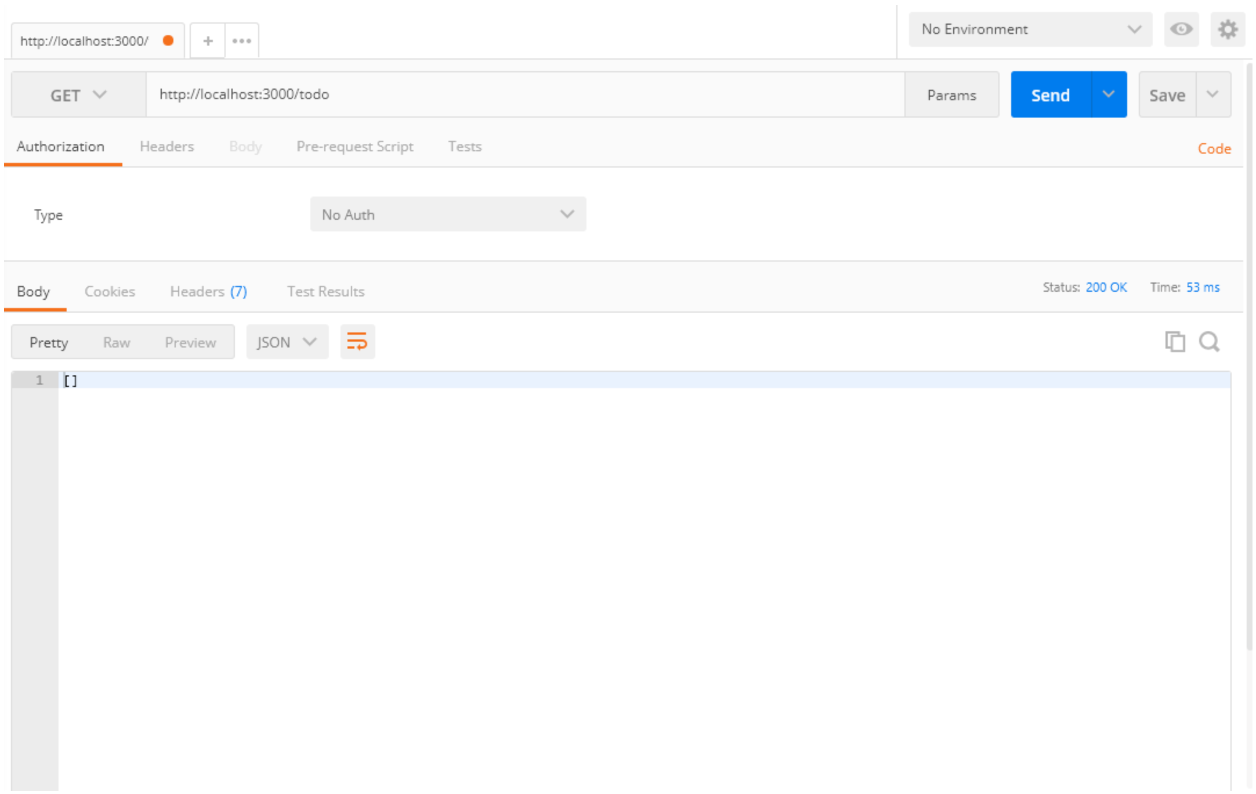
```
// app.module.ts
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { TodoModule } from './todo/todo.module';

@Module({
  imports: [TodoModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Make a request to the `TodoController`

If the development server is not still running, start it by running `npm run start:dev`. In the console output, you should see Nest wiring up the `TodoModule` and creating the `TodoController` route:

```
[NestFactory] Starting Nest application...
[InstanceLoader] AppModule dependencies initialized +9ms
[InstanceLoader] TodoModule dependencies initialized +1ms
[RoutesResolver] AppController {}: +4ms
[RouterExplorer] Mapped {, GET} route +2ms
[RoutesResolver] TodoController {/todo}: +0ms
[RouterExplorer] Mapped {/todo, GET} route +1ms
[NestApplication] Nest application successfully started +2ms
```

With your server still running, open the Postman application. Make a new request to the `TodoController` route at `http://localhost:3000/todo`. The controller will return a JSON response of an empty array, which we return in the `findAll()` method.



```
GET http://localhost:3000/todo
Type: No Auth
Status: 200 OK    Time: 53 ms
[]
```

We are now resolving HTTP requests to our `TodoController` and are ready to start adding in the CRUD actions to the controller and service.

Bonus: Using the CLI generator

In this exercise, we created these files by hand to demonstrate how all of the decorators and files work together. The Nest CLI simplifies this process quite a bit and we can accomplish the same module setup with the following commands:

```
nest generate module todo
nest generate controller todo
nest generate service todo
cd todo && nest generate interface todo
```

Implementing the Data Service and Controller

We are now ready to start implementing our CRUD actions. For all of these actions, we will be querying and mutating the `storage` array in our service `TodoService`.

findAll()

Add the service method

To begin, we will implement a method that will give us all of the items in storage. Our first step is to open the `todo.service.ts` file, add the `findAll()` method definition, and return the `storage` array.

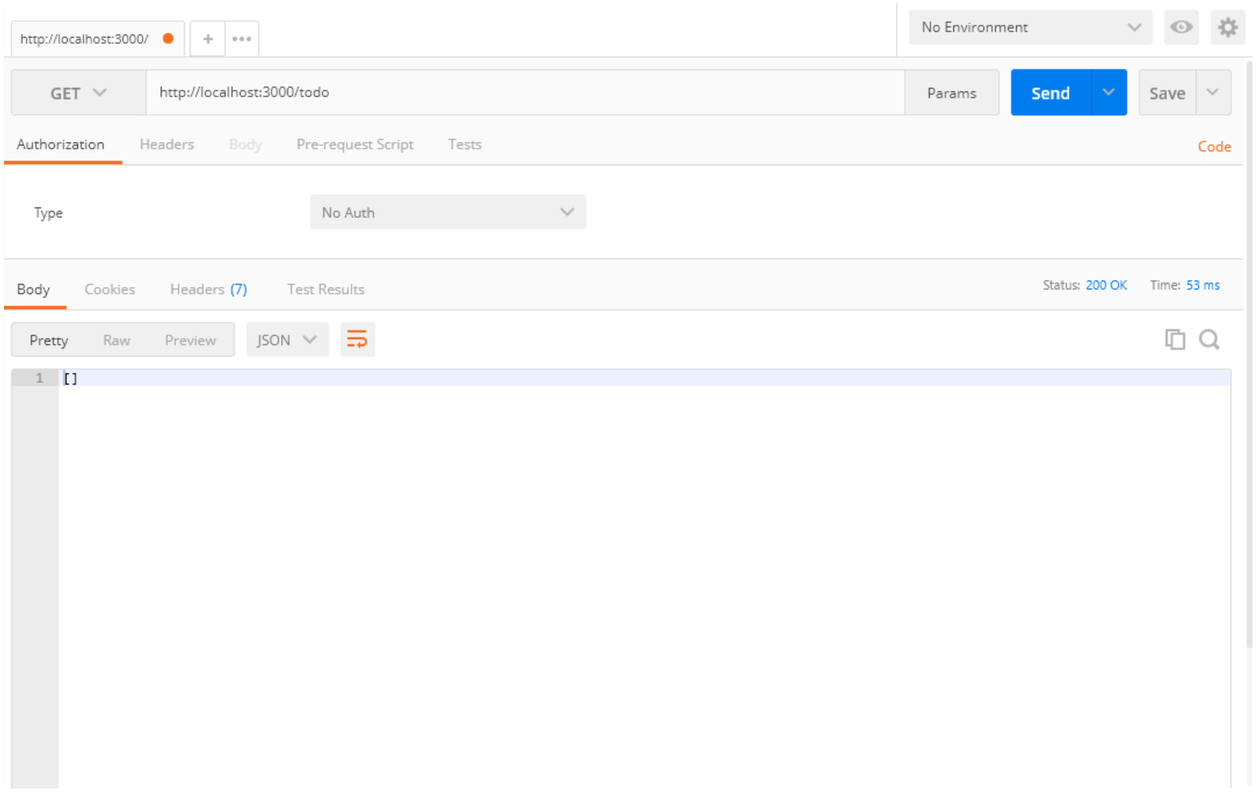
```
// todo.service.ts
import { Injectable } from '@nestjs/common';
import { Todo } from './todo.interface';

@Injectable()
export class TodoService {
  private storage: Todo[] = [];

  findAll(): Todo[] {
    return this.storage;
  }
}
```

Add the GET route

We have now exposed the private `storage` array and can access it through the controller. To call this, open the `todo.controller.ts` file and update the `findAll()` method to call the `TodoService`.

```
// todo.controller.ts
import {
  Controller,
  Get,
} from '@nestjs/common';
import { TodoService } from './todo.service';
import { Todo } from './todo.interface';

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

  @Get()
  findAll(): Todo[] {
    return this.todoService.findAll();
  }
}
```

Ensure that the development server is still running in your command-line tool. Open Postman and make a GET request to `http://localhost:3000/todo`. In response, we'll get an empty array (again).



```
[]
```

create()

Add the service method

To add a new `Todo` to our `storage` array, let's implement a `create()` method in our data service. This method will take in a `Todo` interface, calculate and set the optional `id` value, and add it to the storage array.

```typescript
// todo.service.ts
import { Injectable } from '@nestjs/common';
import { Todo } from './todo.interface';

@Injectable()
export class TodoService {
  private storage: Todo[] = [];

  create(todo: Todo): void {
    const currentMaxId = Math.max(...this.storage.map((t: Todo) => t.id));
    todo.id = currentMaxId + 1;
    this.storage.push(todo);
  }

  findAll(): Todo[] {
    return this.storage;
  }
}
```

Add the POST route

With the data service method in place, we now need to enable an HTTP endpoint in our controller to call it. To do this, we will create a `@Post` route on the `TodoController`. Following REST standards, we'll configure this route to be `POST /todo`.

With `@Post`, `@Put`, and `@Patch` routes, the HTTP content body is used to transfer data objects, such as JSON, XML, or text. In this example, we will be sending a JSON object of a `Todo` item. For NestJS to parse and interpret the content body, the decorator `@Body` should be applied to the route method parameter. NestJS will apply `JSON.parse()` to the content and provide you a JSON object to work with. Since we expect this payload to match our `Todo` interface, we will type it as such.

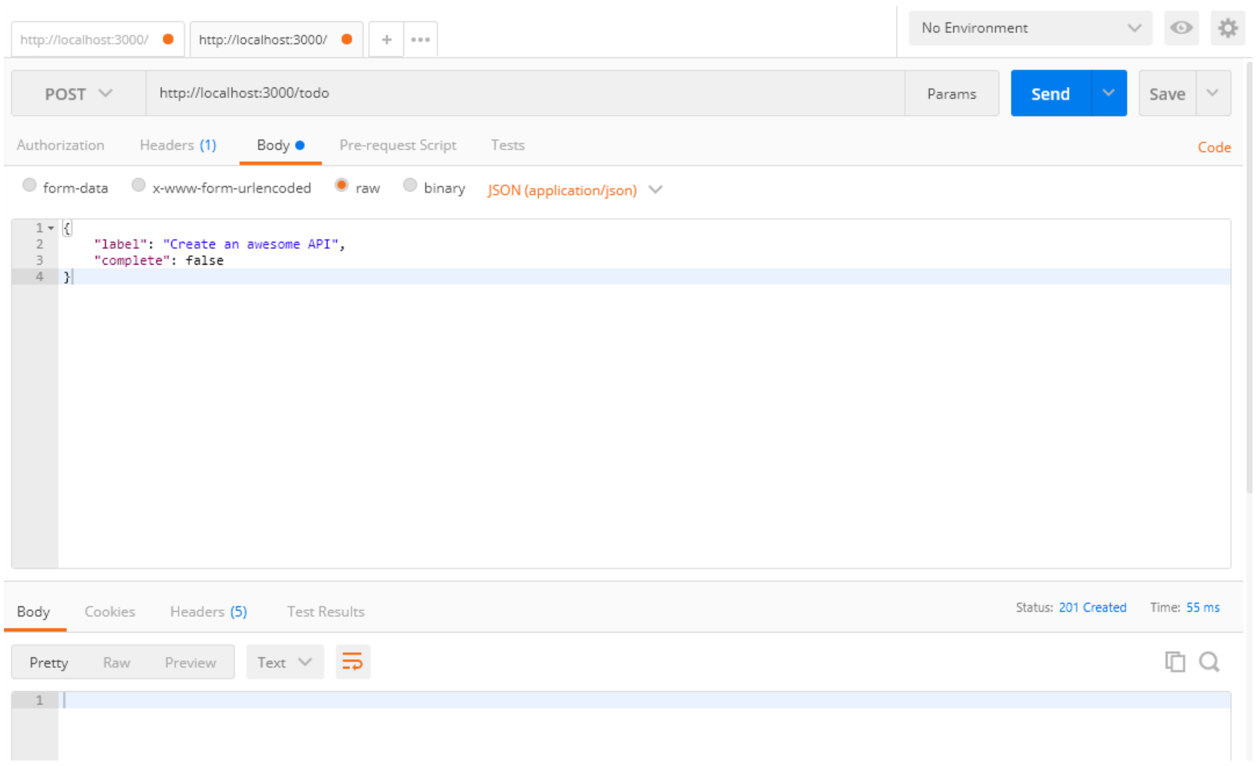
```typescript
// todo.controller.ts
import {
  Body,
  Controller,
  Get,
  Post,
} from '@nestjs/common';
import { Todo } from './todo.interface';
import { TodoService } from './todo.service';

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

  @Post()
  create(@Body() todo: Todo): void {
    return this.todoService.create(todo);
  }

  @Get()
  findAll(): Todo[] {
    return this.todoService.findAll();
  }
}
```

We can now send a POST request to `http://localhost:3000/todo` with a JSON payload to add it to the `storage` array. In Postman, do the following to set up your POST request:

1. Create a new request tab.
2. Change the GET value in the HTTP method dropdown to POST.
3. Enter the `http://localhost:3000/todo` value in the URL.
4. In the “**Body**” tab, select “**raw**”. In the dropdown to the right of “binary”, select “**JSON (application/json)**”.
5. In the textarea, supply the following body:

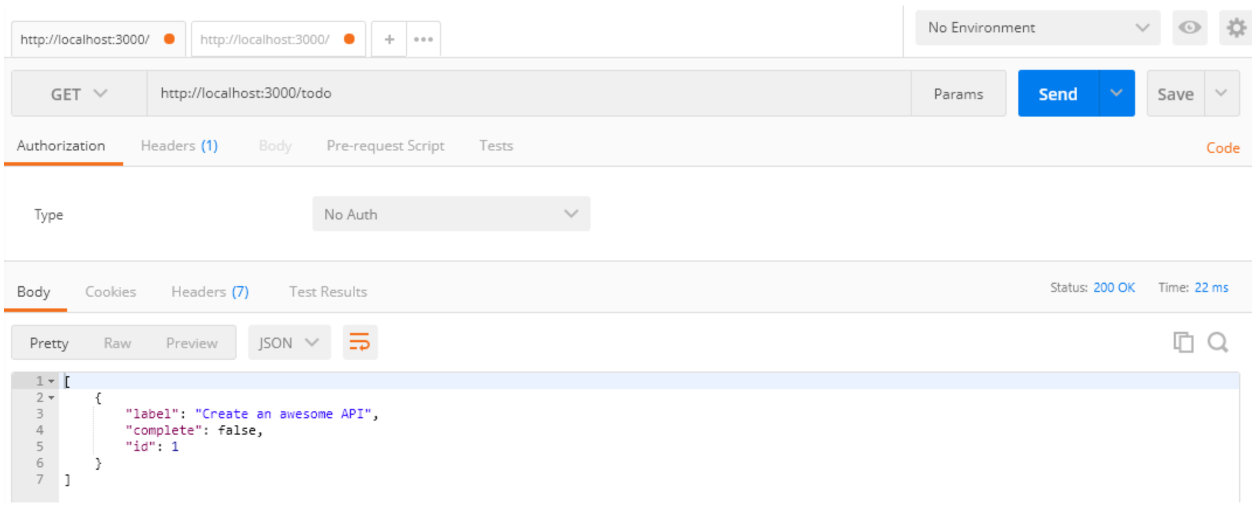
```json
{
        "label": "Create an awesome API",
        "complete": false
}
```

Send the request. Our response will be empty, but with a status of `201 Created`.



```
{
    "label": "Create an awesome API",
    "complete": false
}
```

We can now make a GET request to `http://localhost:3000/todo` again to see our new `Todo` item in our array with a set `id` of 1.



```
[
    {
        "label": "Create an awesome API",
        "complete": false,
        "id": 1
    }
]
```

Quick aside: Adding the NestJS Logger

You may have noticed that there was no indication in our command-line console that request was processed. While this is likely ideal in most scenarios, it would be very helpful for us to see that we are hitting the routes that we expect.

NestJS provides a `Logger` implementation that makes it easy to add logging to your application. It is best practice to use this `Logger` class over the `console` log methods because the log messages are formatted in a cohesive fashion, and the `Logger` allows you to swap out functionality or disable logging completely at a global level with simple configuration in your `main.ts` file.

To add logging to our controller, create a new instance of a `Logger` as a class-level variable on the `TodoController`. We will add a log statement to each of our methods as well.

```
import {
  Body,
  Controller,
  Get,
  Logger,
  Post,
} from '@nestjs/common';
import { Todo } from './todo.interface';
import { TodoService } from './todo.service';

@Controller('todo')
export class TodoController {
  private readonly logger = new Logger(TodoController.name);

  constructor(private readonly todoService: TodoService) {}

  @Post()
  create(@Body() todo: Todo): void {
    this.logger.log('Handling create() request...');
    return this.todoService.create(todo);
  }

  @Get()
  findAll(): Todo[] {
    this.logger.log('Handling findAll() request...');
    return this.todoService.findAll();
  }
}
```

Now when we make a request to either method, we will see items in our console such as:

```
[TodoController] Handling create() request...
[TodoController] Handling findAll() request...
```

findOne()

Add the service method

We will now add a method that allows us to retrieve a single `Todo` item from the `storage` array. To do so, in the `TodoService` add `findOne()` method that does a `find()` on the `storage` array.

```typescript
import { Injectable } from '@nestjs/common';
import { Todo } from './todo.interface';

@Injectable()
export class TodoService {

// ...

  findOne(id: number): Todo {
    return this.storage.find((t: Todo) => t.id === id);
  }
}
```

Add the GET route

We will now need to update the `TodoController` accordingly to enable this service method through an HTTP request. In `todo.controller.ts`, we will add a new `@Get` method, but this time we will take in a route parameter in the `@Get` decorator.

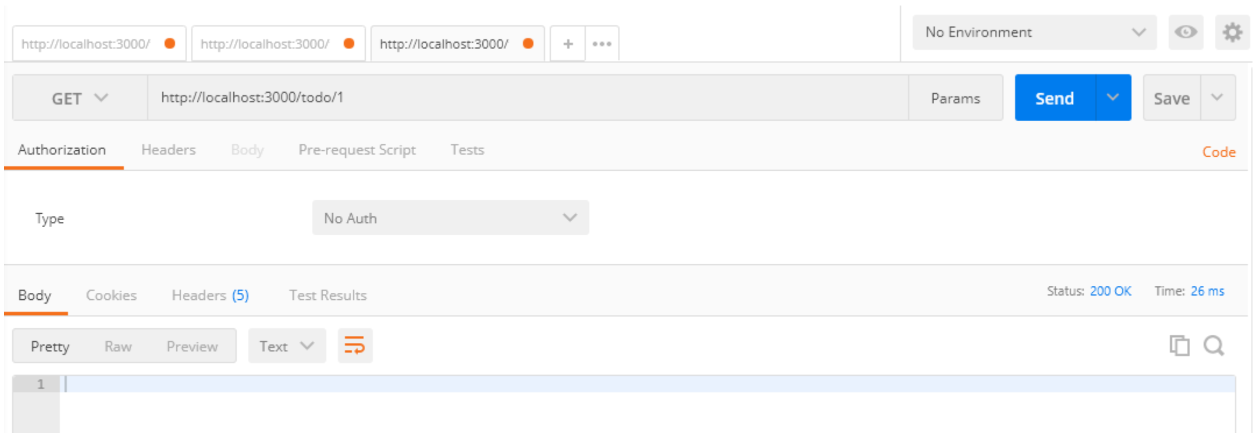
Route parameters are variables inside of a given path. Routers use pattern matching to interpret and parse these paths and pluck out the variables in the provided route pattern. To access this variable in your NestJS route method, the `@Param` decorator is applied to the method argument, supplying the name of the route parameter.

In our API, we want to supply the `Todo` object's `id` as a route parameter, so we use the route pattern of `:id` to indicate a parameter variable with an `id`, and then use the `@Param('id')` decorator to hook the value into the method argument.

```
import {
  Body,
  Controller,
  Get,
  Logger,
  Param,
  Post,
} from '@nestjs/common';
import { Todo } from './todo.interface';
import { TodoService } from './todo.service';

@Controller('todo')
export class TodoController {
  // ...

@Get(':id')
  findOne(@Param('id') id: number): Todo {
    this.logger.log('Handling findOne() request with id=' + id + '...');
    return this.todoService.findOne(id);
  }
}
```

Since the `storage` array was cleared when we restarted our development server, we will do the following to test:

1. Send the POST request outlined in the `create()` section to create a new `Todo` item.
2. Send a GET request to `http://localhost:3000/todo` to see all the items in `storage`, including your new `Todo`.
3. Send a GET request to `http://localhost:3000/todo/1`, our new route, to retrieve the `Todo` item directly.



Uh oh, the response is empty!

Our command-line console says we hit the route with expected ID (`[TodoController] Handling findOne() request with id=1...`). So, what happened?

It is important to understand that by default, `@Param` values are strings. While we gave it the type of `number`, recall that TypeScript types are abstract and have no impact on the actual executing code; this is to say that the executing code had no way of knowing it should have parsed the route parameter `id` to a number. This is where the `@Pipe` decorator comes in.

About NestJS Pipes

Pipes in NestJS transform request parameters at a method level. While you can create your own, the `@nestjs/common` package comes with very helpful pipes that cover most common use-cases.

- `ValidationPipe`: provides request validation using the `class-validator` library.
- `ParseIntPipe`: parses the route parameter to a `number` type.
- `ParseBoolPipe`: parses the route parameter to a `boolean` type.
- `ParseArrayPipe`: parses the route parameter to an array of a given type.
- `ParseUUIDPipe`: parses the route parameter to a UUID of a provided version.
- `DefaultValuePipe`: allows for a default value if no value is supplied in the route.

Parsing the `id` route parameter

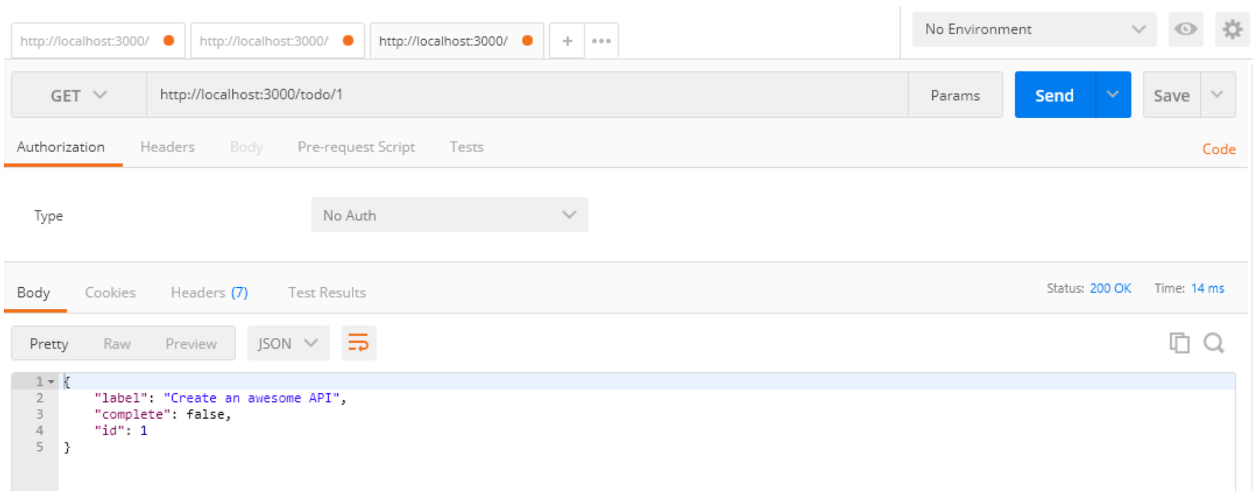
To fix our `findOne()` endpoint, we need to add the `ParseIntPipe` to our `@Param` pipe.

```
import {
  Body,
  Controller,
  Get,
  Logger,
  Param,
  ParseIntPipe,
  Post,
} from '@nestjs/common';
import { Todo } from './todo.interface';
import { TodoService } from './todo.service';

@Controller('todo')
export class TodoController {
  // ...

@Get(':id')
```

```
  findOne(@Param('id', ParseIntPipe) id: number): Todo {
    this.logger.log('Handling findOne() request with id=' + id + '...');
    return this.todoService.findOne(id);
  }
}
```

This will parse the value to a number and now our `.find()` method in our data service will find an exact equality against the `Todo` item we POST. Since the `storage` array was cleared when we restarted our development server, we will do the following to test:

1. Send the POST request outlined in the `create()` section to create a new `Todo` item.
2. Send a GET request to `http://localhost:3000/todo` to see all the items in `storage`, including your new `Todo`.
3. Send a GET request to `http://localhost:3000/todo/1`, our new route, to retrieve the `Todo` item directly.



```
{
    "label": "Create an awesome API",
    "complete": false,
    "id": 1
}
```

With this pipe in place, we're now able to retrieve the `Todo` item.

update()

Add the service method

We will now create a method to update an item in our `storage` array. This method will find the item in the array and replace it with an object in the request body.

```
// todo.service.ts
import { Injectable } from '@nestjs/common';
import { Todo } from './todo.interface';

@Injectable()
export class TodoService {
  // ...

  update(id: number, todo: Todo): void {
    const index = this.storage.findIndex((t: Todo) => t.id === id);
    this.storage[index] = todo;
  }
}
```

Add the PUT route

With this method in place, we will expose the method through a PUT request. To do so, we'll need to use our learnings about the `@Body` decorator, the `@Param` decorator, and pipes to create the route we need. The route will supply the `id` while the content body will provide the payload of the `Todo`.

```
import {
  Body,
  Controller,
  Get,
  Logger,
  Param,
  ParseIntPipe,
  Post,
  Put,
} from '@nestjs/common';
import { Todo } from './todo.interface';
import { TodoService } from './todo.service';

@Controller('todo')
export class TodoController {
  // ...

  @Put(':id')
  update(@Param('id', ParseIntPipe) id: number, @Body() todo: Todo): void {
    this.logger.log('Handling update() request with id=' + id + '...');
    return this.todoService.update(id, todo);
  }
}
```

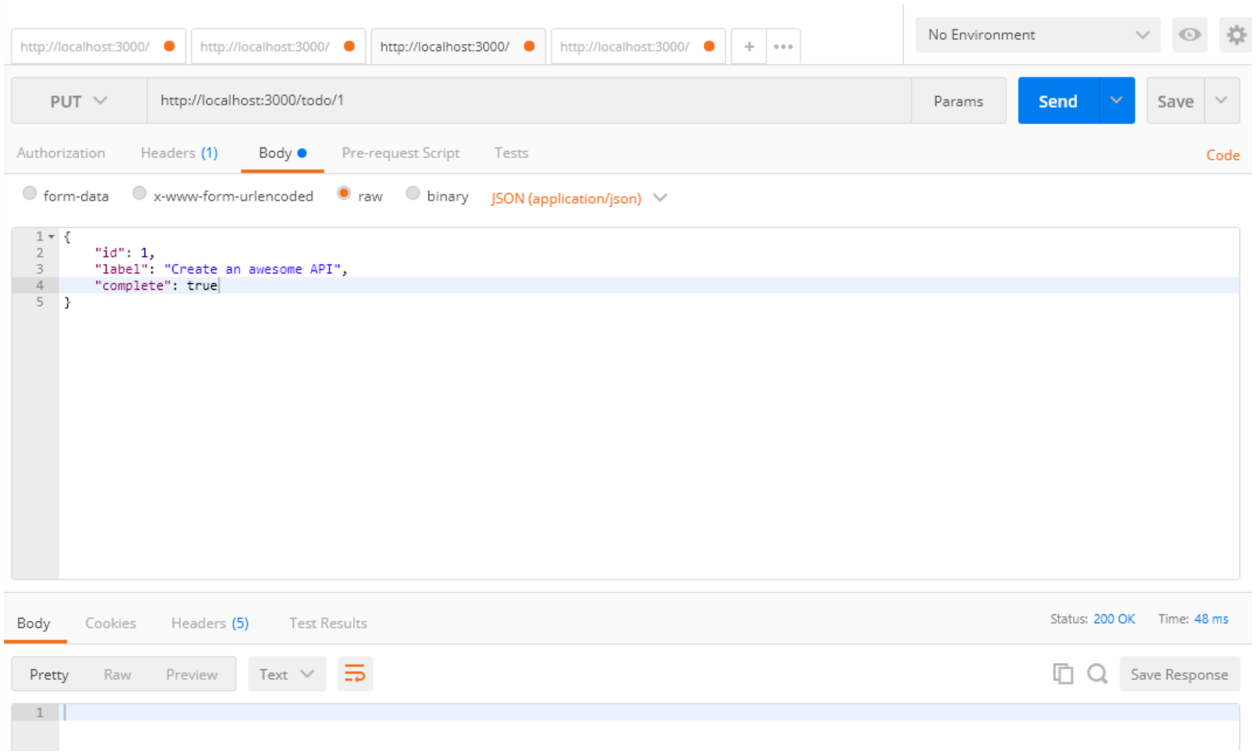
Since the `storage` array was cleared when we restarted our development server, we will do the following to test:

1. Send the POST request outlined in the `create()` section to create a new `Todo` item.
2. Send a GET request to `http://localhost:3000/todo` to see all the items in `storage`, including your new `Todo`.

With the `storage` array set up, let's now send a PUT request to update the `"complete"` flag to `true`.

1. Create a new request tab.
2. Change the GET value in the HTTP method dropdown to PUT.
3. Enter the `http://localhost:3000/todo/1` value in the URL.
4. In the "**Body**" tab, select "**raw**". In the dropdown to the right of "binary", select "**JSON (application/json)**".
5. In the textarea, supply the following body:

```json
{
  "id": 1,
  "label": "Create an awesome API",
  "complete": true
}
```



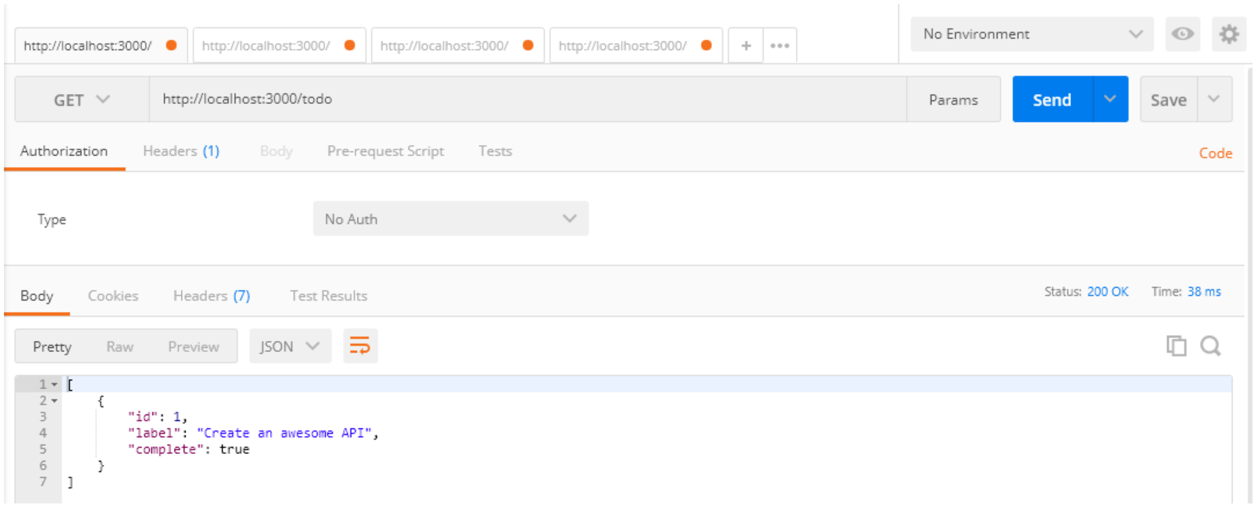
PUT http://localhost:3000/todo/1

raw — JSON (application/json)

```json
{
    "id": 1,
    "label": "Create an awesome API",
    "complete": true
}
```

Status: 200 OK Time: 48 ms

To validate the value has changed, send a GET request to `http://localhost:3000/todo` to see all of the items in the `storage` array.



```
[
    {
        "id": 1,
        "label": "Create an awesome API",
        "complete": true
    }
]
```

remove()

Add the service method

At the end of our `Todo` process, we'll want to remove items from the `storage` array. Following our process, we'll add a new `remove()` method to the `TodoService` that finds the item in the `storage` array and splices it out of memory.

```
// todo.service.ts
import { Injectable } from '@nestjs/common';
import { Todo } from './todo.interface';

@Injectable()
export class TodoService {
  // ...
  remove(id: number): void {
    const index = this.storage.findIndex((t: Todo) => t.id === id);
    this.storage.splice(index, 1);
  }
}
```

Add the DELETE route

DELETE requests look quite like GET requests in that they are not accompanied by a content body. To create this endpoint, we'll add the `@Delete` decorator to a new method that calls our data service.

As with the GET and PUT routes, we'll need to make sure `ParseIntPipe` pipe is in place.

```
import {
  Body,
  Controller,
  Delete,
  Get,
  Logger,
  Param,
  ParseIntPipe,
  Post,
  Put,
} from '@nestjs/common';
import { Todo } from './todo.interface';
import { TodoService } from './todo.service';

@Controller('todo')
export class TodoController {
  // ...

  @Delete(':id')
  remove(@Param('id', ParseIntPipe) id: number): void {
    this.logger.log('Handling remove() request with id=' + id + '...');
    return this.todoService.remove(id);
  }
}
```

We can now test the whole CRUD workflow via Postman.

1. Send the POST request outlined in the `create()` section to create a new `Todo` item.
2. Send a GET request to `http://localhost:3000/todo` to see all the items in `storage`, including your new `Todo`.
3. Send the PUT request outlined in the `update()` section to set the `Todo` item to `"completed": true`.
4. Send a GET request to `http://localhost:3000/todo` to see the updated item in `storage`.
5. Send a DELETE request to `http://localhost:3000/todo/1` to remove the item in `storage`.
6. Send a GET request to `http://localhost:3000/todo` to see the now emptied `storage`.

With logging enabled, your command-line console will look something like this:

```
[TodoController] Handling create() request...
[TodoController] Handling findAll() request...
[TodoController] Handling update() request with id=1...
[TodoController] Handling findAll() request...
[TodoController] Handling remove() request with id=1...
[TodoController] Handling findAll() request...
```

Ready for integration!

The `TodoController` now has full CRUD operations and we can hook it up to a user interface to allow users to modify the `storage` array. In the next chapter, we'll point a pre-existing user interface on StackBlitz at our local server and watch the full-stack application come to life.

Integrating with a User Interface

It’s time for the fun part: seeing everything come to life in a fully integrated full-stack application. We have prepared a Todo list app that you can run against your local development server: https://stackblitz.com/edit/js-17i3zg

To test with this user interface, start up your development web server with `npm run start:dev`. In the user interface, try to create a new `Todo` item by typing in the “What’s next?” input and hitting enter.

Uh oh, nothing happened!

Enable CORS

If you check out your browser console, you’ll see some errors about a missing “Access-Control-Allow-Origin” header. These errors are due to cross-origin reference sharing (CORS) restrictions that prevent web applications from making XHR requests to other domains. These restrictions are a security feature browsers have put in place using what are known as pre-flight requests and API response headers.

Pre-flight requests are OPTION HTTP requests that are executed before the actual XHR request. Your browser automatically injects this request and if the restrictions specified in the response headers are met by the request parameters, then the actual request will continue as expected. By default, these restrictions prevent requests from any other origin; in our case, our server at `localhost` is preventing XHR requests from `stackblitz.com`.

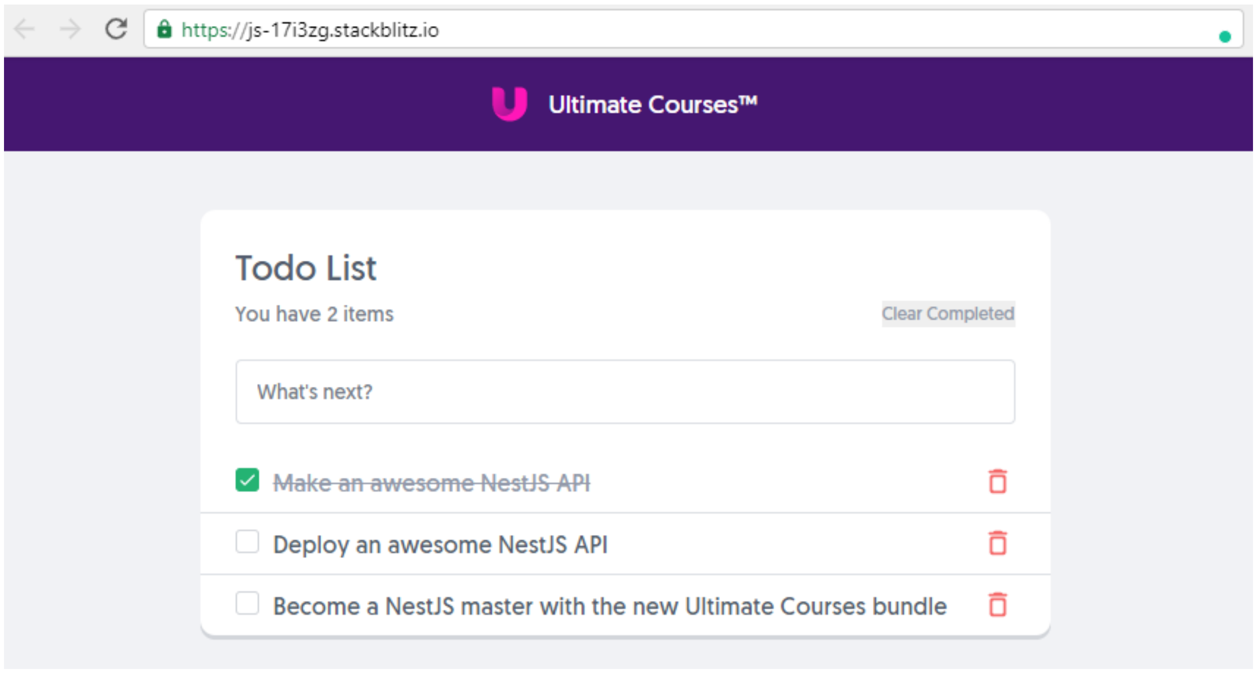
NestJS makes opening these CORS restrictions simple. In your `main.ts` file, call `app.enableCors();` in the `bootstrap()` method. This will add headers to your controller route responses that enable cross-domain requests.

```
// main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors();
  await app.listen(3000);
}
bootstrap();
```

Test the Full-Stack Application

With CORS enabled, you should be able to create, edit, check-off, and remove items in the todo list. You can test your API by doing the following actions:

1. Create a new todo item.
2. Double-click a todo item to change the label.
3. Check off a todo item.
4. Remove a todo item.
5. Clear all completed todo items.



~~Make an awesome NestJS API~~
Deploy an awesome NestJS API
Become a NestJS master with the new Ultimate Courses bundle

Summary

This example is a great demonstration of building loosely-coupled REST APIs and user interfaces. By building REST APIs with standardized route patterns and clearly indicating data transfer contracts, we are able to separate our concerns between back-end processing and front-end user interactions in a way that is maintainable and scalable.

This pattern can be leveraged to create very stable and maintainable applications, from small applications to large products. Creating a full-stack TypeScript application with NestJS and the front-end framework of your choice is a recipe for success.

Through this eBook, I hope to have demonstrated how the NestJS framework makes it easy to create server-side codebases that scale. Where do you go from here? I recommend you pre-order the NestJS Basics and NestJS Pro courses coming soon at Ultimate Courses[6], which will teach you all the fundamentals, how to get started with database management, how to build more complex REST APIs, and how to fully master the NestJS framework.

[6]https://ultimatecourses.com