BLEEDING EDGE PRESS

Nest.js:

A Progressive Node.js Framework

Greg Magolan, Patrick Housley, Adrien de Peretti, Jay Bell, David Guijarro

Table of Contents

Preface

What is Nest.js?

There are so many available web frameworks, and with the advent of Node.js, even more have been released. JavaScript frameworks go in and out of style very quickly as web technologies change and grow. Nest.js is a good starting point for many developers that are looking to use a modern web framework because it uses a language that is very similar to that of the most used language on the web to this day, JavaScript. Many developers were taught programming using languages such as Java or C/C++, which are both strict languages, so using JavaScript can be a little awkward and easy to make mistakes given the lack of type safety. Nest.js uses TypeScript, which is a happy medium. It is a language that provides the simplicity and power of JavaScript with the type safety of other languages you may be used to. The type safety in Nest.js is only available at compile time, because the Nest.js server is compiled to a Node.js Express server that runs JavaScript. This is still a major advantage, however, since it allows you to better design programs error free prior to runtime.

Node.js has a rich ecosystem of packages in NPM (Node Package Manager). With over 350,000 packages, it's the world's largest package registry. With Nest.js making use of Express, you have access to each and every one of these packages when developing Nest applications. Many even have type definitions for their packages that allow IDE's to read the package and make suggestions/auto fill in code that may not be possible when crossing JavaScript code with TypeScript code. One of the largest benefits of Node.js is the huge repository of modules that are available to pull from instead of having to write your own. Nest.js includes some of these modules already as part of the Nest platform, like `@nestjs/mongoose`, which uses the NPM library `mongoose`. Prior to 2009, JavaScript was mainly a front-end language, but after the release of Node.js in 2009, it spurred the development of many JavaScript and TypeScript projects like: Angular, React, and Vue, among others. Angular was a heavy inspiration for the development of Nest.js because both use a Module/Component system that allows for reusability. If you are not familiar with Angular, it is a TypeScript-based front-end framework that can be used cross-platform to develop responsive web apps and native apps, and it functions a lot like Nest does. The two also pair very well together with Nest providing the ability to run a Universal server to serve pre-rendered Angular web pages to speed up website delivering times using Server-Side Rendering (SSR) mentioned above.

About the example

This book will reference a working Nest.js project that is hosted on GitHub at (https://github.com/backstopmedia/nest-book-example). Throughout the book, code snippets and chapters will reference parts of the code so that you can see a working example of what you are learning about. The example Git repository can be cloned within your command prompt.

```
git clone https://github.com/backstopmedia/nest-book-example.git
```

This will create a local copy of the project on your computer, which you can run locally by building the project with Docker:

```
docker-compose up
```

Once your Docker container is up and running on port localhost:3000, you will want to run the migration before doing anything else. To do this run:

```
docker ps
```

To get the ID of your running Docker container:

```
docker exec -it [ID] npm run migrate up
```

This will run the database migrations so that your Nest.js app can read and write to the database with the correct schema.

If you don't want to use Docker, or cannot use Docker, you can build the project with your choice of package managers such as `npm` or `yarn`:

```
npm install
```

or

```
yarn
```

This installs the dependencies in your `node_modules` folder. Then run:

```
npm start:dev
```

Or the following to start your Nest.js server:

```
yarn start:dev
```

These will run `nodemon`, which will cause your Nest.js application to restart if any changes are made, saving you from having to stop, rebuild, and start your application again.

About the authors

- **Greg Magolan** is a Senior Architect, Full-Stack Engineer, and Angular Consultant at Rangle.io. He has been developing enterprise software solutions for over 15 years working for companies suchas Agilent Technologies, Electronic Arts, Avigilon, Energy Transfer Partners, FunnelEnvy, Yodel andACM Facility Safety.
- **Jay Bell** is the CTO of Trellis. He is a Senior Angular developer that uses Nest.js in production to develop industry leading software to help non-profits and charities in Canada. He is a serial entrepreneur that has developed software in a large range of industries from helping combat wildfires with drones to building mobile apps.
- **David Guijarro** is a Front-End Developer at Car2go Group GmbH. He has a wide experience working within the JavaScript ecosystem. He has successfully built and led multi-cultural, multi-functional teams.
- **Adrien de Peretti** is a Full-Stack JavaScript Developer. He is passionate about new technologies and is constantly looking for new challenges, and is especially interested in the field of Artificial Intelligence and Robotics. When he is not front of his computer, Adrien is in nature and playing different sports.
- **Patrick Housley** is a Lead Technologist at VML. He is an IT professional with over six years of experience in the technology industry and is capable of analyzing complex issues spanning multiple technologies while providing detailed resolutions and explanations. He has strong front-end development skills with experience leading development teams in maintenance and greenfield projects.

Chapter 1. Introduction

Every web developer relies heavily on one web framework or another (sometimes more if their services have different requirements) and companies will rely on many frameworks, but each has its own pros and cons. These frameworks provide just that, a frame for developers to build on top of, providing the basic functionality that any web framework must provide in order to be considered as a good choice for a developer or company to use in their tech stack. In this book, we will talk about many of those parts of the framework you would expect to see in a progressive framework such as Nest. These include:

1. Dependency Injection
2. Authentication
3. ORM
4. REST API
5. Websockets
6. Microservices
7. Routing
8. Explanation of Nest specific tools
9. OpenApi (Swagger) Documentation
10. Command Query Responsibility Segregation (CQRS)
11. Testing
12. Server-side rendering with Universal and Angular.

Nest provides more of these features because it is a modern web framework built on top of a Node.js Express server. By leveraging the power of modern ES6 JavaScript for flexibility and TypeScript to enforce type safety during compile time, Nest helps bring scalable Node.js servers to a whole new level when designing and building server-side applications. Nest combines three different techniques into a winning combination that allows for highly testable, scalable, loosely coupled and maintainable applications. These are:

1. Object-Oriented Programming (OOP): A model that builds around objects instead of actions and reusability rather than niche functionality.
2. Functional Programming (FP): The designing of determinate functionality that does not rely upon global states, ie. a function f(x)

returns the same result every time for some set parameters that do not change.

3. Functional Reactive Programming (FRP): An extension of FP from above and Reactive programming. Functional Reactive Programming is at its core Functional Programming that accounts for a flow across time. It is useful in applications such as UI, simulations, robotics and other applications where the exact answer for a specific time period may differ from that of another time period.

Topics discussed

Each of the topics below will be discussed in more detail in the following chapters.

Nest CLI

New in version 5 of Nest there is a CLI that allows for command line generation of projects and files. The CLI can be installed globally with:

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

Or through Docker with:

```
docker pull nestjs/cli:[version]
```

A new Nest project can be generated with the command:

```
nest new [project-name]
```

This process will create the project from a typescript-starter and will ask for the `name`, `description`, `version` (defaults to 0.0.0), and `author` (this would be your name). After this process is finished you will have a fully setup Nest project with the dependencies installed in your `node_modules` folder. The `new`command will also ask what package manager you would like to use, in the same way that either `yarn`or `npm` can be used. Nest gives you this choice during creation.

The most used command from the CLI will be the `generate` (g) command, this will allow you to create new `controllers`, `modules`, `servies` or any other components that Nest supports. The list of available components is:

1. `class` (cl)
2. `controller` (co)
3. `decorator` (d)
4. `exception` (e)
5. `filter` (f)
6. `gateway` (ga)
7. `guard` (gu)
8. `interceptor` (i)
9. `middleware` (mi)
10. `module` (mo)
11. `pipe` (pi)
12. `provider` (pr)
13. `service` (s)

Note that the string in the brackets is the alias for that specific command. This means that instead of typing:

```
nest generate service [service-name]
```

In your console, you can enter:

```
nest g s [service-name]
```

Lastly, the Nest CLI provides the `info` (i) command to display information about your project. This command will output information that looks something like:

```
[System Information]
OS Version     : macOS High Sierra
NodeJS Version : v8.9.0
YARN Version    : 1.5.1
[Nest Information]
microservices version : 5.0.0
websockets version    : 5.0.0
testing version       : 5.0.0
common version        : 5.0.0
core version          : 5.0.0
```

Dependency Injection

Dependency Injection is the technique of supplying a dependent object, such as a module or component, with a dependency like a service, thereby injecting it into the component's constructor. An example of this taken from the sequelize chapter is below. Here we are injecting the `UserRespository` service into the constructor of the `UserService`, thereby providing access to the User Database repository from inside the `UserService` component.

```
@Injectable()
export class UserService implements IUserService {
     constructor(@Inject('UserRepository') private readonly
UserRepository: typeof User) { }
     ...
}
```

In turn this `UsersService` will be injected into the UsersController in the `src/users/users.controller.ts` file, which will provide access to the `UsersService` from the routes that point to this controller. More about Routes and Dependency injection in later chapters.

Authentication

Authentication is one of the most important aspects of developing. As developers, we always want to make sure that users can only access the resources they have permission to access. Authentication can take many forms, from showing your drivers license or passport to providing a username and password for a login portal. In recent years these authentication methods have expanded out to become more complicated, but we still need the same server-side logic to make sure that these authenticated users are always who they say they are and persist this authentication so they do not need to reauthenticate for every single call to a REST API or Websocket because that would provide a pretty terrible user experience. The chosen library for this is ironically named Passport as well, and is very well known and used in the Node.js ecosystem. When integrated into Nest it uses a JWT (JSON Web Token) strategy. Passport is a Middleware that the HTTP call is passed through before hitting the endpoint at the controller. This is the `AuthenticationMiddleware` written for the example project that extends `NestMiddleware`, authenticating each user based on the email in the request payload.

```
@Injectable()
export class AuthenticationMiddleware implements NestMiddleware {
    constructor(private userService: UserService) { }
```

```
    async resolve(strategy: string): Promise<ExpressMiddleware> {
        return async (req, res, next) => {
            return passport.authenticate(strategy, async (/*...*/args:
any[]) => {
                const [, payload, err] = args;
                 if (err) {
                     return
res.status(HttpStatus.BAD_REQUEST).send('Unable to authenticate the user.');
                 }

                const user = await this.userService.findOne({
                     where: { email: payload.email }
                });
                 req.user = user;
                 return next();
             })(req, res, next);
        };
    }
}
```

Nest also implements Guards, which are decoratorated with the same `@Injectable()` as other providers. Guards restrict certain endpoints based on what the authenticated user has access to. Guards will be discussed more in the Authentication chapter.

ORM

An ORM is an Object-relational mapping and is one of the most important concepts when dealing with communication between a server and a database. An ORM provides a mapping between objects in memory (Defined classes such a `User` or `Comment`) and Relational tables in a database. This allows you to create a Data Transfer Object that knows how to write objects stored in memory to a database, and read the results from an SQL or another query language, back into memory. In this book, we will talk about three different ORMs: two relational and one for a NoSQL database. TypeORM is one of the most mature and popular ORMs for Node.js and thus has a very wide and flushed out feature set. It is also one of the packages that Nest provides its own packages for: `@nestjs/typeorm`. It is incredibly powerful and has support for many databases like MySQL, PostgreSQL, MariaDB, SQLite, MS SQL Server,

Oracle, and WebSQL. Along with TypeORM, Sequelize is also another ORM for relational data.

If TypeORM is one of the most popular ORMs, then Sequelize is THE most popular in the Node.js world. It is written in plain JavaScript but has TypeScript bindings through the `sequelize-typescript`and `@types/sequelize` packages. Sequelize boasts strong transaction support, relations, read replication and many more features. The last ORM covered in this book is one that deals with a non-relational, or NoSQL, database. The package `mongoose` handles object relations between MongoDB and JavaScript. The actual mapping between the two is much closer than with relational databases, as MongoDB stores its data in JSON format, which stands for JavaScript Object Notation. Mongoose is also one of the packages that has a `@nestjs/mongoose` package and provides the ability to query the database through query chaining.

REST API

REST is one of the main design paradigms for creating APIs. It stands for Representative State Transfer, and uses JSON as a transfer format, which is in line with how Nest stores objects, thus it is a natural fit for consuming and returning HTTP calls. A REST API is a combination of many techniques that are talked about in this book. They are put together in a certain way; a client makes an HTTP call to a server. That server will Route the call to the correct Controller based on the URL and HTTP verb, optionally passing it through one or more Middlewares prior to reaching the Controller. The Controller will then hand it off to a Service for processing, which could include communication with a Database through an ORM. If all goes well, the server will return an OK response to the client with an optional body if the client requested resources (GET request), or just a 200/201 HTTP OK if it was a POST/PUT/DELETE and there is no response body.

WebSockets

WebSockets are another way to connect to and send/receive data from a server. With WebSockets, a client will connect to the server and then subscribe to certain channels. The clients can then push data to a subscribed channel. The server will receive this data and then broadcast it to every client that is subscribed to that specific channel. This allows multiple clients to all receive real-time updates without having to make API calls manually, potentially flooding the server with GET requests. Most chat apps use WebSockets to allow

for real-time communication, and everyone in a group message will receive the message as soon as one of the other members sends one. Websockets allow for more of a streaming approach to data transfer than traditional Request-Response API's, because Websockets broadcast data as it's received.

Microservices

Microservices allow for a Nest application to be structured as a collection of loosely coupled services. In Nest, microservices are slightly different, because they are an application that uses a different transport layer other than HTTP. This layer can be TCP or Redis pub/sub, among others. Nest supports TCP and Redis, although if you are married to another transport layer it can be implemented by using the `CustomTransportStrategy` interface. Microservices are great because they allow a team to work on their own service within the global project and make changes to the service without affecting the rest of the project since it is loosely coupled. This allows for continuous delivery and continuous integration independent of other teams microservices.

GraphQL

As we saw above, REST is one paradigm when designing APIs, but there is a new way to think about creating and consuming APIs: GraphQL. With GraphQL, instead of each resource having its own URL pointing to it, a URL will accept a query parameter with a JSON object in it. This JSON object defines the type and format of the data to return. Nest provides functionality for this through the `@nestjs/graphql` package. This will include the `GraphQLModule` in the project, which is a wrapper around the Apollo server. GraphQL is a topic that could have an entire book written about it, so we don't go into it any further in this book.

Routing

Routing is one of the core principles when discussing web frameworks. Somehow the clients need to know how to access the endpoints for the server. Each of these endpoints describes how to retrieve/create/manipulate data that is stored on the server. Each `Component` that describes an API endpoint must have a `@Controller('prefix')` decorator that describes the API prefix for this component's set of endpoints.

```
@Controller('hello')
export class HelloWorldController {
   @Get('world')
   printHelloWorld() {
      return 'Hello World';
   }
}
```

The above Controller is the API endpoint for `GET /hello/world` and will return an `HTTP 200 OK` with `Hello World` in the body. This will be discussed more in the Routing chapter where you will learn about using URL params, Query params, and the Request object.

Nest specific tools

Nest provides a set of Nest.js specific tools that can be used throughout the application to help with writing reusable code and following SOLID principles. These decorators will be used in each of the subsequent chapters, as they define a specific functionality:

1. @Module: The definition for this reusable package of code within the project, it accepts the following parameters to define its behavior. · · *Imports: These are the modules that contain the components used within this module.* · · Exports: These are the components that will be used in other modules, that import this module. · · *Components: These are the components that will be available to be shared across at least this module through the Nest Injector.* · · Controllers: The controllers created within this module, these will define the API endpoints based on the routes defined.
2. @Injectable: Almost everything in Nest is a provider that can be injected through constructors. Providers are annotated with `@Injectable()`. .. *Middleware: A function that is run before a request is passed to the route handler. In this chapter, we will talk about the difference between Middleware, Async Middlewares and Functional Middleware.* .. Interceptor: Similar to Middleware, they bind extra logic before and after the execution of a method, and they can both transform or completely override a function. Interceptors are inspired by Aspect-Oriented Programming (AOP). .. *Pipe: Similar to part of an Interceptors functionality, Pipe transforms input data to the desired output.* ..Guard: A smarter and

more niche Middleware, Guards have the singular purpose of determining if a request should be handled by the router handler or not. ..* Catch: Tell an `ExceptionFilter` what exception to look for and then bind data to it.

3. @Catch: Binds metadata to the exception filter and tells Nest that a filter is looking only for the exceptions listed in the `@Catch`.

Note: In Nest Version 4 not everything under `@Injectable()` listed above uses the `@Injectable()` decorator. Components, Middlewares, Interceptors, Pipes, and Guards each have their own decorator. In Nest Version 5 these have all been combined to `@Injectable()` to reduced the differences between Nest and Angular.

OpenAPI (Swagger)

Documentation is very important when writing a Nest server, and is especially so when creating an API that will be consumed by others, otherwise the developer writing the clients that will eventually be consuming the API do not know what to send or what they get back. One of the most popular documentation engines out there is Swagger. Like with others, Nest provides a dedicated module for the OpenAPI (Swagger) spec, `@nestjs/swagger`. This module provides decorators to help describe the inputs/outputs and endpoints of your API. This documentation is then accessible through an endpoint on the server.

Command Query Responsibility Segregation (CQRS)

Command Query Responsibility Segregation (CQRS) is the idea that each method should either be one that performs an action (command) or requests data (query), but not both. In the context of our sample app, we would not have the database access code directly within the Controller for an endpoint, but rather create a Component (Database Service) that has a method such as `getAllUsers()` that will return all the users that the Controllers Service can call, thus separating the question and the answer into different Components.

Testing

Testing your Nest server will be imperative so that once it is deployed their are no unforseen issure and it all runs smoothly. There are two different kinds of tests you will learn about in this book: Unit Tests and E2E Tests (End-to-end Tests). Unit Testing is the art of testing small snippets or blocks of code, and this could be as granular as testing individual functions or writing a test for a `Controller`, `Interceptor`, or any other `Injectable`. There are many popular unit testing frameworks out there, and `Jasmine` and `Jest` are two popular ones. Nest provides special packages, `@nestjs/testing` specifically, for writing unit tests in `*.spec.ts` and `*.test.ts` classes.

E2E Testing is the other form of testing that is commonly used and is different from unit testing only in that it tests entire functionality rather than individual functions or components, which is where the name end-to-end testing came from. Eventually applications will become so large that it is hard to test absolutely every piece of code and endpoint. In this case you can use E2E tests to test the application from beginning to the end to make sure everything works along the way. For E2E testing a Nest application can use the `Jest` library again to mock up components. Along with `Jest` you can use the `supertest` library to simulate HTTP requests.

Testing is a very important part of writing applications and should not be ignored. This is a chapter that will be relevant no matter what language or framework you end up working with. Most large scale development companies have entire teams dedicated to writing tests for the code that is pushed to production applications, and these are called QA developers.

Server-side rendering with Angular Universal

Angular is a client side application development framework and Angular Universal is a technology that allows our Nest server to pre-render the webpages and serve them to the client, which has numerous benefits that will be discussed in the Server-side Rendering with Angular Universal chapter. Nest and Angular pair very well together due to both using TypeScript and Node.js. Many of the packages that can be used in the Nest server can also be used in the Angular app because they both compile to JavaScript.

Summary

Throughout this book, you will go through each of the above topics in more detail, continuously building on top of prior concepts. Nest provides a clean

well-organized framework that implements each of these concepts in a simple yet efficient way that is consistent across all modules because of the modular design of the framework.

Chapter 2. Overview

In this chapter we'll take an overview of Nest.js and look at the core concepts that you'll need to build a Nest.js application.

Controllers

Controllers in Nest are responsible for handling incoming requests and returning responses to the client. Nest will route incoming requests to handler functions in controller classes. We use the `@Controller()` decorator to create a controller class.

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

@Controller('entries')
export class EntryController {
     @Get()
     index(): Entry[] {
          const entries: Entry[] = this.entriesService.findAll();
          return entries;
     }
```

We'll go over the details of routing and handling requests in the **Routing and Request Handling**chapter.

Providers

Providers in Nest are used to create services, factories, helpers, and more that can be injected into controllers and other providers using Nest's built-in dependency injection. The `@Injectable()` decorator is used to create a provider class.

The `AuthenticationService` in our blog application, for example, is a provider that injects and uses the `UsersService` component.

```
@Injectable()
export class AuthenticationService {
     constructor(private readonly userService: UserService) {}
```

```
    async validateUser(payload: {
        email: string;
        password: string;
    }): Promise<boolean> {
        const user = await this.userService.findOne({
            where: { email: payload.email }
        });
        return !!user;
    }
}
```

We'll talk more about dependency injection in the the **Dependency Injection** chapter.

Modules

A Nest.js application is organized into modules. If you're familiar with modules in Angular, then the module syntax Nest uses will look very familiar.

Every Nest.js application will have a **root module**. In a small application, this may be the only module. In a larger application, it makes sense to organize your application into multiple modules that split up your code into features and related capabilities.

A module in Nest.js is a class with a `@Module()` decorator.
The `@Module()` decorator takes a single object that describes module using the following properties.

Property	Description
`components`	The components to be instantiated that may be shared across this m to be available to other modules
`controllers`	The controllers that are created by this module
`imports`	The list of modules to import that export components that are require
`exports`	The list of components from this module to be made available to othe

In our example application, the root Module is named `AppModule` and the application is split up into a number of sub-modules that handle the major parts

of the application such as authentication, comments, database access, blog entries and users.

```
@Module({
      components: [],
      controllers: [],
      imports: [
            DatabaseModule,
            AuthenticationModule.forRoot('jwt'),
            UserModule,
            EntryModule,
            CommentModule,
            UserGatewayModule,
            CommentGatewayModule
      ],
      exports: [],
})
export class AppModule implements NestModule {}
```

The AppModule imports the modules that are needed for the application. The root module in our application doesn't need to have any `exports` since no other modules import it.

The root module also doesn't have any `components` or `controllers`, as these are all organized within the sub-modules they are related to. The `EntryModule`, for example, includes both `components` and `controllers` that are related to blog entries.

```
@Module({
      components: [entryProvider, EntryService],
      controllers: [EntryController],
      imports: [],
      exports: [EntryService],
})
export class EntryModule implements NestModule {}
```

Modules in Nest.js are singletons by default. This means that you can share the same instance of an exported component, such as the `EntryService` above, between modules without any effort.

Bootstrapping

Every Nest.js application needs to be bootstrapped. This is done by by using the `NestFactory` to create the root module and calling the `listen()` method.

In our example application, the entry point is `main.ts` and we use the async / await pattern to create the `AppModule` and call `listen()`:

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

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

Middleware

Nest.js middleware is either a function or a class decorated with the `@Injectable()` decorator that implements the `NestMiddleware` interface. Middleware is called **before** route handlers. These functions have access to the **request** and **response** object, and they can makes changes to the request and response object.

One or more middleware functions can be configured for a route, and a middleware function can choose to pass the execution to the next middleware function on the stack or to end the request-response cycle.

If a middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function or to the request handler if it is the last function on the stack. Failing to do so will leave the request hanging.

The `AuthenticationMiddleware` in our blog application, for example, is responsible for authenticating a user that is accessing the blog.

```
import {
      MiddlewareFunction,
      HttpStatus,
      Injectable,
      NestMiddleware
} from '@nestjs/common';
```

```
import * as passport from 'passport';
import { UserService } from '../../modules/user/user.service';

@Injectable()
export class AuthenticationMiddleware implements NestMiddleware {
     constructor(private userService: UserService) {}

     async resolve(strategy: string): Promise<MiddlewareFunction> {
          return async (req, res, next) => {
               return passport.authenticate(strategy, async (...args: any[])
=> {
                    const [, payload, err] = args;
                    if (err) {
                         return res
                              .status(HttpStatus.BAD_REQUEST)
                              .send('Unable to authenticate the user.');
                    }

                    const user = await this.userService.findOne({
                         where: { email: payload.email }
                    });
                    req.user = user;
                    return next();
               })(req, res, next);
          };
     }
}
```

If authentication fails, a 400 response is sent back to the client. If authentication is successful, then `next()` is called and the request will continue down the middleware stack until it reaches the request handler.

Middleware is configured on routes in the `configure()` function of a Nest.js module. For example, the `AuthenticationMiddle` above is configured in the `AppModule` as shown here.

```
@Module({
     imports: [
          DatabaseModule,
          AuthenticationModule.forRoot('jwt'),
          UserModule,
```

```
            EntryModule,
            CommentModule,
            UserGatewayModule,
            CommentGatewayModule,
            KeywordModule
        ],
        controllers: [],
        providers: []
})
export class AppModule implements NestModule {
        public configure(consumer: MiddlewareConsumer) {
            const userControllerAuthenticatedRoutes = [
                { path: '/users', method: RequestMethod.GET },
                { path: '/users/:id', method: RequestMethod.GET },
                { path: '/users/:id', method: RequestMethod.PUT },
                { path: '/users/:id', method: RequestMethod.DELETE }
            ];

            consumer
                .apply(AuthenticationMiddleware)
                .with(strategy)
                .forRoutes(
                    ...userControllerAuthenticatedRoutes,
                    EntryController,
                    CommentController
                );
        }
}
```

You can apply middleware to all routes on a controller, as is done for the `EntryController` and `CommentController`. You can also apply middleware to specific routes by their path, as is done for the subset of routes from the `UserController`.

Guards

Guards are classes that are decorated with the `@Injectable()` decorator and implement the `CanActivate`interface. A guard is responsible for determining if a request should be handled by a route handler or route. Guards are executed **after** every middleware, but **before** pipes. Unlike middleware, guards

have access to the `ExecutionContext` object, so they know exactly what is going to evaluated.

In our blog application, we use the `CheckLoggedInUserGuard` in the `UserController` to only allow a user to access and access their own user information.

```
import { Injectable, CanActivate, ExecutionContext } from
'@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class CheckLoggedInUserGuard implements CanActivate {
     canActivate(
          context: ExecutionContext
     ): boolean | Promise<boolean> | Observable<boolean> {
          const req = context.switchToHttp().getRequest();
          return Number(req.params.userId) === req.user.id;
     }
}
```

The `@UseGuards` decorator is used to apply a guard to a route. This decorator can be used on a controller class to apply the guard to all routes in that controller, or it can be used on individual route handlers in a controller as seen in the `UserController`:

```
@Controller('users')
export class UserController {
     constructor(private readonly userService: UserService) { }

     @Get(':userId')
     @UseGuards(CheckLoggedInUserGuard)
     show(@Param('userId') userId: number) {
          const user: User = this.userService.findById(userId);
          return user;
     }
```

Summary

In this chapter we covered Nest.js controllers, providers, modules, bootstrapping, and middleware. In the next chapter we will go over Nest.js authentication.

Chapter 3. Nest.js authentication

Nest.js, using version 5 the `@nestjs/passport` package, allows you to implement the authentication strategy that you need. Of course you can also do this manually using `passport`.

In this chapter you will see how to use passport by integrating it into your Nest.js project. We also cover what a strategy is, and how to configure the strategy to use with passport.

We will also manage restriction access using an authentication middleware, and see how guards can check data before the user accesses the handlers. In addition, we'll show how to use the passport package provided by Nest.js in order to cover both possibilities.

As an example, we will use the following repository files:

- `/src/authentication`
- `/src/user`
- `/shared/middlewares`
- `/shared/guards`

Passport

Passport is a well known library that is popular and flexible to use. In fact, passport is flexible middleware that can be fully customized. Passport allows different ways to authenticate a user like the following:

- `local strategy` that allows you to authenticate a user just with it's own data `email` and `password` in most cases.
- `jwt strategy` that allows you to authenticate a user by providing a token and verifying this token using `jsonwebtoken`. This strategy is used a lot.

Some strategies use the social network or Google in order to authenticate the user with a profile such as `googleOAuth`, `Facebook`, or even `Twitter`.

In order to use passport you have to install the following package: `npm i passport`. Before you see how to implement the authentication, you must implement the `userService` and the `userModel`.

Manual implementation

In this section, we will implement the authentication manually using passport without using the Nest.js package.

Implementation

In order to configure passport, three things need to be configured:

- The authentication strategy
- The application middleware
- The session, which is optional

Passport uses the strategy to authenticate a request, and the verification of the credential is delegated to the strategies in some of the requests.

Before using passport, you must configure the strategy, and in this case we will use the `passport-jwt`strategy.

Before anything else, you must install the appropriate packages:

- `npm i passport-jwt @types/passport-jwt`
- `npm i jsonwebtoken @types/jsonwebtoken`

AUTHENTICATION MODULE

In order to have a working example, you must implement some modules, and we will start with `AuthenticationModule`. The `AuthenticationModule` will configure the strategy using the jwt strategy. To configure the strategy we will extend the `Strategy` class provided by the `passport-jwt` package.

Strategy

Here is an example of a strategy extending the `Strategy` class in order to configure it and use it in passport.

```
@Injectable()
export default class JwtStrategy extends Strategy {
    constructor(private readonly authenticationService:
AuthenticationService) {
        super({
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
```

```
            passReqToCallback: true,
            secretOrKey: 'secret'
        }, async (req, payload, next) => {
            return await this.verify(req, payload, next);
        });
        passport.use(this);
    }

    public async verify(req, payload, done) {
        const isValid = await
this.authenticationService.validateUser(payload);
        if (!isValid) {
            return done('Unauthorized', null);
        } else {
            return done(null, payload);
        }
    }
}
```

The constructor allows you to pass some configuration parameters to the extended `Strategy` class. In this case we are using only three parameters:

- `jwtFromRequest` option accepts a function in order to extract the token from the request. In our case we are using the `ExtractJwt.fromAuthHeaderAsBearerToken()` function provided by the `passport-jwt`package. This function will pick the token from the header of the request using the `Authorization`header, and pick the token that follows the `bearer` word.
- `passReqToCallback` parameter takes a boolean in order to tell if you want to get the `req` in the verify method that you will see later.
- `secretOrKey` parameter takes a string or a buffer in order to verify the token signature.

Other parameters are available to configure the strategy, but to implement our authentication we don't need them.

Also, after passing the different previous parameters, we pass a callback function called `verify`. This function is asynchronous, and has the purpose to verify if the token passed and if the payload obtained from the token is valid or not. This function executes our `verify` method, which calls the `authenticationService` in order to validate the user with the payload as a parameter.

If the user is valid, we return the payload, otherwise we return an error to indicate that the payload is not valid.

Authentication service

As shown in the previous section, in order to verify the payload that you get from the token, call the `validateUser` method provided by the `AuthenticationService`.

In fact, this service will implement another method in order to generate the token for the logged in user. The service can be implemented as the following example.

```
@Injectable()
export class AuthenticationService {
    constructor(private readonly userService: UserService) { }

    createToken(email: string, ttl?: number) {
          const expiresIn = ttl || 60 * 60;
          const secretOrKey = 'secret';
          const user = { email };
          const token = jwt.sign(user, secretOrKey, { expiresIn });
          return {
               expires_in: expiresIn,
               access_token: token,
          };
    }

    async validateUser(payload: { email: string; password: string }):
Promise<boolean> {
          const user = await this.userService.findOne({
               where: { email: payload.email }
          });
          return !!user;
    }
}
```

The service injects the `UserService` in order to find the user using the payload pass to the `validateUser`method. If the email in the payload allows you to find the user, and if that user has a valid token, she can continue the authentication process.

In order to provide a token for the user who try to logged in, implement the `createToken` method, which takes as parameters an `email` and an optional `ttl`. The `ttl` (Time to live) will configure the token to be valid for a period. The value of the `ttl` is expressed in seconds, and the default value that we have defined in `60 * 60`, which means 1 hour.

Authentication controller

In order to process the authentication of the user, implement the controller and provide a handler for the login endpoint.

```
@Controller()
export class AuthenticationController {
    constructor(
          private readonly authenticationService: AuthenticationService,
          private readonly userService: UserService) {}

    @Post('login')
    @HttpCode(HttpStatus.OK)
    public async login(@Body() body: any, @Res() res): Promise<any> {
         if (!body.email || !body.password) {
              return res.status(HttpStatus.BAD_REQUEST).send('Missing email
or password.');
         }

         const user = await this.userService.findOne({
              where: {
                   email: body.email,
                    password: crypto.createHmac('sha256',
body.password).digest('hex')
              }
         });
         if (!user) {
              return res.status(HttpStatus.NOT_FOUND).send('No user found
with this email and password.');
         }

         const result = this.authenticationService.createToken(user.email);
         return res.json(result);
     }
}
```

The controller provides the login handler, which is accessible by a call on the `POST /login` route. The purpose of this method is to validate the credentials provided by the user in order to find him in the database. If the user is found, create the appropriate token that will be returned as a response with the `expiresIn`value corresponding to our previously defined `ttl`. Otherwise the request will be rejected.

Module

We have now defined our service and strategy in order to configure passport and provide some method to create a token and validate a payload. Let's define `AuthenticationModule`, which is similar to the following example.

```
@Module({})
export class AuthenticationModule {
    static forRoot(strategy?: 'jwt' | 'OAuth' | 'Facebook'):
DynamicModule {
         strategy = strategy ? strategy : 'jwt';
          const strategyProvider = {
               provide: 'Strategy',
               useFactory: async (authenticationService:
AuthenticationService) => {
                    const Strategy = (await import
(`./passports/${strategy}.strategy`)).default;
                    return new Strategy(authenticationService);
               },
               inject: [AuthenticationService]
          };
          return {
               module: AuthenticationModule,
               imports: [UserModule],
               controllers: [AuthenticationController],
               providers: [AuthenticationService, strategyProvider],
               exports: [strategyProvider]
          };
     }
}
```

As you can see, the module is not defined as a normal module, so it has no components or controller defined in the `@Module()` decorator. In fact, this module is a dynamic module. In order to provide a multiple strategy, we can implement a static method on the class in order to call it when we import the module in

another one. This method `forRoot` takes as a parameter the name of the strategy that you want to use and will create a `strategyProvider` in order to be added to the components list in the returned module. This provider will instanciate the strategy and provide the `AuthenticationService` as a dependency.

Let's continue by creating something to protect, such as the `UserModule`.

USER MODULE

The `UserModule` provides a service, a controller, and a model (see the sequelize chapter for the User model). We create some methods in the `UserService` in order to manipulate the data concerning the user. These methods are used in the `UserController` in order to provide some features to the user of the API.

All of the features can't be used by the user or restricted in the data that is returned.

User service

Let's examine an example of the `UserService` and some methods in order to access and manipulate the data. All of the methods describe in this part will be used in the controller, and some of them are restricted by the authentication.

```
@Injectable()
export class UserService() {
     // The SequelizeInstance come from the DatabaseModule have a look
to the Sequelize chapter
     constructor(@Inject('UserRepository') private readonly
UserRepository: typeof User,
                    @Inject('SequelizeInstance') private readonly
sequelizeInstance) { }

     /* ... */
}
```

The service injects the `UserRepository` that we have described in the Sequelize chapter in order to access the model and the data store in the database. We also inject the `SequelizeInstance`, also described in the Sequelize chapter, in order to use the transaction.

The `UserService` implements the `findOne` method to find a user with a criteria passing in the `options`parameter. The `options` parameter can look like this:

```
{
    where: {
        email: 'some@email.test',
        firstName: 'someFirstName'
    }
}
```

Using this criteria, we can find the corresponding user. This method will return only one result.

```
@Injectable()
export class UserService() {
    /* ... */

    public async findOne(options?: object): Promise<User | null> {
        return await this.UserRepository.findOne<User>(options);
    }

    /* ... */
}
```

Let's implement the `findById` method, which takes as a parameter an ID in order to find a unique user.

```
@Injectable()
export class UserService() {
    /* ... */

    public async findById(id: number): Promise<User | null> {
        return await this.UserRepository.findById<User>(id);
    }

    /* ... */
}
```

Then we need a way to create a new user in the database passing the user respecting the `IUser` interface. This method, as you can see, uses a `this.sequelizeInstance.transaction` transaction to avoid reading the data before everything is finished. This method passes a parameter to the `create` function, which is `returning` in order to get the instance user that has been created.

```
@Injectable()
export class UserService() {
     /* ... */

     public async create(user: IUser): Promise<User> {
          return await this.sequelizeInstance.transaction(async
transaction => {
               return await this.UserRepository.create<User>(user, {
                    returning: true,
                    transaction,
               });
          });
     }

     /* ... */
}
```

Of course, if you can create a user, you also need the possibility to update it with the following method following the `IUser` interface. This method too will return the instance of the user that has been updated.

```
@Injectable()
export class UserService() {
     /* ... */

     public async update(id: number, newValue: IUser): Promise<User |
null> {
          return await this.sequelizeInstance.transaction(async
transaction => {
               let user = await this.UserRepository.findById<User>(id, {
transaction });
               if (!user) throw new Error('The user was not found.');

               user = this._assign(user, newValue);
               return await user.save({
                    returning: true,
                    transaction,
               });
          });
     }
```

```
    /* ... */
}
```

In order to make a round in all of the methods, we will implement the `delete` method to remove a user completely from the database.

```
@Injectable()
export class UserService() {
    /* ... */

    public async delete(id: number): Promise<void> {
        return await this.sequelizeInstance.transaction(async
transaction => {
            return await this.UserRepository.destroy({
                where: { id },
                transaction,
            });
        });
    }

    /* ... */
}
```

In all of the previous examples, we have define a complete `UserService` that allowed us to manipulate the data. We have the possibility to create, read, update, and delete a user.

User model

If you wish to see the implementation of the user model, you can refer to the Sequelize chapter.

User controller

Now that we have created our service and model, we need to implement the controller to handle all the requests from the client. This controller provides at least a create, read, update and delete handler that should be implemented like the following example.

```
@Controller()
export class UserController {
    constructor(private readonly userService: UserService) { }
```

```
    /* ... */
}
```

The controller injects the `UserService` in order to use the methods implemented in the `UserService`.

Provide a `GET users` route that allows access to all users from the database, and you will see how we don't want the user accessing the data of all of the users, just only for himself. This is why we are using a guard that only allows a user to acces his own data.

```
@Controller()
export class UserController {
    /* ... */

    @Get('users')
    @UseGuards(CheckLoggedInUserGuard)
    public async index(@Res() res) {
        const users = await this.userService.findAll();
        return res.status(HttpStatus.OK).json(users);
    }

    /* ... */
}
```

The user has access to a route that allows you to create a new user. Of course, if you want, the user can register into the logged in application, which we must allow for those without a restriction.

```
@Controller()
export class UserController {
    /* ... */

    @Post('users')
    public async create(@Body() body: any, @Res() res) {
        if (!body || (body && Object.keys(body).length === 0)) throw new
Error('Missing some information.');

        await this.userService.create(body);
        return res.status(HttpStatus.CREATED).send();
    }
```

```
    /* ... */
}
```

We also provide a `GET users/:id` route that allows you to get a user by his ID. Of course a logged in user should not be able to access the data from another user even from this route. This route is also protected by a guard in order to allow the user access to himself and not another user.

```
@Controller()
export class UserController {
    /* ... */

    @Get('users/:id')
    @UseGuards(CheckLoggedInUserGuard)
    public async show(@Param() id: number, @Res() res) {
        if (!id) throw new Error('Missing id.');

        const user = await this.userService.findById(id);
        return res.status(HttpStatus.OK).json(user);
    }

    /* ... */
}
```

A user can have the idea to update some of his own information, which is why we provide a way to update a user through the following `PUT users/:id` route. This route is also protected by a guard to avoid a user updating another user.

```
@Controller()
export class UserController {
    /* ... */
    @Put('users/:id')
    @UseGuards(CheckLoggedInUserGuard)
    public async update(@Param() id: number, @Body() body: any, @Res()
res) {
        if (!id) throw new Error('Missing id.');

        await this.userService.update(id, body);
        return res.status(HttpStatus.OK).send();
    }
```

Use deletion to finish the last handler. This route has to also be protected by a guard to avoid a user from deleting another user. The only user that can be deleted by a user is himself.

```
    @Delete('users/:id')
    @UseGuards(CheckLoggedInUserGuard)
    public async delete(@Param() id: number, @Res() res) {
        if (!id) throw new Error('Missing id.');

        await this.userService.delete(id);
        return res.status(HttpStatus.OK).send();
    }
}
```

We have implemented all of the methods that we need in this controller. Some of them are restricted by a guard in order to apply some security and avoid a user from manipulating the data from another user.

Module

To finish the implementation of the `UserModule`, we have to set up the module of course. This module contains a service, a controller, and a provider that allows you to inject the user model and provides a way to manipulate the stored data.

```
@Module({
    imports: [],
    controllers: [UserController],
    providers: [userProvider, UserService],
    exports: [UserService]
})
export class UserModule {}
```

This module is imported like the `AuthenticationModule` into the main `AppModule` in order to use it in the app and be accessible.

APP MODULE

The `AppModule` imports three modules for our example.

- `DatabaseModule` accesses the sequelize instance and accesses the database.
- `AuthenticationModule` allows you to log into a user and use the appropriate strategy.

- `UserModule` exposes some endpoints that can be requested by the client.

In the end, the module should looks like the following example.

```
@Module({
    imports: [
          DatabaseModule,
          // Here we specify the strategy
          AuthenticationModule.forRoot('jwt'),
          UserModule
     ]
})
export class AppModule implements NestModule {
    public configure(consumer: MiddlewaresConsumer) {
         consumer
              .apply(AuthenticationMiddleware)
              .with(strategy)
              .forRoutes(
                   { path: '/users', method: RequestMethod.GET },
                    { path: '/users/:id', method: RequestMethod.GET },
                    { path: '/users/:id', method: RequestMethod.PUT },
                    { path: '/users/:id', method: RequestMethod.DELETE }
              );
     }
}
```

As you can see in this example, we have applied the `AuthenticationMiddleware` to the routes that we want to protect from a non-logged in user.

This middleware has the purpose of applying the passport middleware `passport.authenticate`, which verifies the token provided by the user and stores in the header the request as the `Authorization` value. This middleware will take the strategy parameter to correspond to the strategy that should be applied, which for us is `strategy = 'jwt'`.

This middleware is applied on almost all of the routes of the `UserController`, except for the `POST /users`that allows you to create a new user.

Authentication middleware

As seen in the previous section, we have applied the `AuthenticationMiddleware`, and we have seen that passport is middleware to authenticate the user. This middleware will execute the `passport.authenticate` method using the strategy `jwt`, taking a callback function that will return the results of the authentication method. As a result we can receive the payload corresponding to the token or an error in case the authentication doesn't work.

```
@Injectable()
export class AuthenticationMiddleware implements NestMiddleware {
     constructor(private userService: UserService) { }

     async resolve(strategy: string): Promise<ExpressMiddleware> {
          return async (req, res, next) => {
               return passport.authenticate(strategy, async (...args: any[])
=> {
                    const [,  payload, err] = args;
                    if (err) {
                         return
res.status(HttpStatus.BAD_REQUEST).send('Unable to authenticate the user.');
                    }

                    const user = await this.userService.findOne({ where:
{ email: payload.email }});
                    req.user = user;
                    return next();
               })(req, res, next);
          };
     }
}
```

If the authentication work we will be able to store the user into the request `req` in order to be use by the controller or the guard. the middleware implement the interface `NestMiddleware` in order to implement the resolve function. It also inject the `UserService` in order to find the user authenticated.

Managing restrictions with guards

Nest.js comes with a guard concept. This injectable has a single responsibility, which is to determine if the request has to be handled by the route handler.

The guard is used on a class that implements the `canActivate` interface in order to implement the `canActivate` method.

The guards are executed after every middleware and before any pipes. The interest of doing this is to separate the restriction logic of the middleware and reorganize this restriction.

Imagine using a guard to manage the access to a specific route and you want this route to only be accessible to the logged in user. To do that we have implemented a new guard, which has to return 'true' if the user accessing the route is the same as the one belonging to the resource that the user want to access. With this kind of guard, you avoid a user to access another user.

```
@Injectable()
export class CheckLoggedInUserGuard implements CanActivate {
     canActivate(context: ExecutionContext): boolean | Promise<boolean>
| Observable<boolean> {
          const request = context.switchToHttp().getRequest();
          return Number(req.params.userId) === req.user.id;
     }
}
```

As you can see, you get the handler from the context that corresponds to the route handler on the controller where the guard is applied. You also get the `userId` from the request parameters to compare it from to the logged in user register into the request. If the user who wants to access the data is the same, then he can access the references in the request parameter, otherwise he will receive a `403 Forbidden`.

To apply the guard to the route handler, see the following example.

```
@Controller()
@UseGuards(CheckLoggedInUserGuard)
export class UserController {/*...*/}
```

Now that we have protected all of our route handlers of the userController, they are all accessible except for the `delete` one, because the user has to be an `admin` to access it. If the user does not have the appropriate role, they will receive a `403 Forbidden` response.

Nest.js passport package

The `@nestjs/passport` package is an extensible package that allows you to use any strategy from passport into Nest.js. As seen in the previous section, it is possible to implement the authentication manually, but if you want to do it in a quicker way and have the strategy wrapped, then use the good package.

In this section, you will see the usage of the package using `jwt` as shown in the previous section. To use it you have to install the following package:

```
npm install --save @nestjs/passport passport passport-jwt jsonwebtoken
```

To use the package you will have the possibility to use the exact same `AuthenticationService` that you have implemented in the previous section, but remember to follow the next code sample.

```
@Injectable()
export class AuthenticationService {
    constructor(private readonly userService: UserService) { }

    createToken(email: string, ttl?: number) {
          const expiresIn = ttl || 60 * 60;
          const secretOrKey = 'secret';
          const user = { email };
          const token = jwt.sign(user, secretOrKey, { expiresIn });
          return {
               expires_in: expiresIn,
               access_token: token,
          };
    }

    async validateUser(payload: { email: string; password: string }):
Promise<boolean> {
          const user = await this.userService.findOne({
               where: { email: payload.email }
          });
          return !!user;
    }
}
```

To instanciate the jwt strategy, you will also have to implement the `JwtStrategy`, but now you only need to pass the options because the passport is wrapped by the package and will apply the strategy to passport automatically under the hood.

```
@Injectable()
export default class JwtStrategy extends PassportStrategy(Strategy) {
    constructor(private readonly authenticationService:
AuthenticationService) {
        super({
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
            passReqToCallback: true,
            secretOrKey: 'secret'
        });
    }

    public async validate(req, payload, done) {
        const isValid = await
this.authenticationService.validateUser(payload);
        if (!isValid) {
            return done('Unauthorized', null);
        } else {
            return done(null, payload);
        }
    }
}
```

As you can see, in this new implementation of the `JwtStrategy` you don't need to implement the callback anymore. This is because you now extend the `PassportStrategy(Strategy)` where `Strategy` is the imported member from the `passport-jwt` library. Also, the `PassportStrategy` is a mixin that will call the `validate` method that we've implemented and named according to the abstract member of this mixin class. This method will be called by the strategy as the validation method of the payload.

Another feature provided by the package is the `AuthGuard` that can be used with `@UseGuards(AuthGuard('jwt'))` to enable the authentication on a specific controller method instead of using the middleware that we have implemented in the previous section.

The `AuthGuard` takes as parameters the name of the strategy that you want to apply, which in our example is `jwt`, and can also take some other parameters that follow the `AuthGuardOptions` interface. This interface defines three options that can be used:

- `session` as a boolean

- `property` as a string to define the name of the property that you want to be add into the request to attach to the authenticated user
- `callback` as a function that allows you to implement your own logic

By default the `session` is set to false and the `property` is set to user. By default, The callback will return the `user` or an `UnauthorizedException`. And that's it, you can now authenticate the user on any controller method and get the user from the request.

The only thing you have to do is to create the `AuthModule` as the following sample:

```
@Module({
   imports: [UserModule],
   providers: [AuthService, JwtStrategy],
})
export class AuthModule {}
```

And as you can see, it isn't in your hands to create a provider to instanciate the strategy, because it's now wrapped into the package.

Summary

In this chapter you have learned what a passport is and strategies to configure the different parts of the passport in order to authenticate the user and store it into the request. You have also seen how to implement the different modules, `AuthenticationModule` and the `UserModule`, in order to be logged into the user and provide some endpoints accessible by the user. Of course, we have restricted the access to some data that applies the `AuthenticationMiddleware` and the `CheckLoggedInUserGuard` for more security.

You have also seen the new `@nestjs/passport` package, which allows you to implement in faster ways a few classes as `AuthenticationService` and `JwtStrategy`, and be able to authenticate any user on any controller method using the `AuthGuard` provided by the package.

In the next chapter you will learn about the Dependency Injection pattern.

Chapter 4. Dependency Injection system of Nest.js

This chapter provides an overview of the Dependency Injection (DI) pattern, which is frequently used today by the biggest frameworks. It is a way to keep code clean and easier to use. By using this pattern you end up with fewer coupled components and more reusable ones, which helps accelerate the development process time.

Here we examine the method that used the injection before the pattern existed, and how the injection changed in time to use Nest.js injection with a modern approach using TypeScript and decorators. You will also see snippets that show the advantage of this type of pattern, and modules provided by the framework.

Nest.js is based on Angular in terms of architecture, and is used to create testable, scalable, loosely-coupled and easily maintainable applications. As is the case with Angular, Nest.js has its own dependency injection system, which is part of the `core` of the framework, meaning that Nest.js is less dependent on a third-party library.

Overview of Dependency Injection

Since `Typescript 1.5` introduces the notion of the decorator, you can do `meta-programing` using the added metadata provided by using a decorator on different objects or properties, such as `class`, `function`, `function parameters` or `class property`. The meta-programing is the ability to write some code or program using the metadata describing an object. This type of program allows you to modify the functioning of a program using its own metadata. In our case this metadata is of interest to us, because it helps inject some object into another object, whose name is Dependency Injection.

By using the decorator, you can add metadata on any object or property linked to those decorators. This will define, for example, the type of object that takes the decorator, but it can also define all of the parameters needed by a function that are described in its metadata. To get or define metadata on any object, you can also use the `reflect-metadata` library in order to manipulate them.

Why use Dependency Injection

The real interest in using Dependency Injection is that the objects will be less coupled between the dependent and its dependencies. With the framework that provides the injector system, you can manage your objects without thinking about the instanciation of them, because that is managed by the injector, which is there to resolve the dependencies of every dependent object.

This means that it is easier to write tests and mock dependencies, which are much cleaner and more readable.

How it works without Dependency Injection

Let's imagine an `AuthenticationService` that needs a `UserService` to be injected.

Here is the `UserService`:

```
export class UserService() {
    private users: Array<User> = [{
        id: 1,
        email: 'userService1@email.com',
        password: 'pass'
    }];

    public findOne({ where }: any): Promise<User> {
        return this.users
        .filter(u => {
            return u.email === where.email &&
            u.password === where.password;
        });
    }
}
```

And the `AuthenticationService`, which instantiates the `UserService` that is needed:

```
export class AuthenticationService {
    public userService: UserService;

    constructor() {
        this.userService = new UserService();
    }
```

```
    async validateAUser(payload: { email: string; password: string }):
Promise<boolean> {
        const user = await this.userService.findOne({
            where: payload
        });
        return !!user;
    }
}
const authenticationService = new AuthenticationService();
```

As you can see, you have to manage all of the related dependencies in the class itself to be used inside the `AuthenticationService`.

The disadvantage of this is mostly the inflexibility of the `AuthenticationService`. If you want to test this service, you have to think about its own hidden dependencies, and of course, you can't share any services between different classes.

How it works with a manual Dependency Injection

Let's see now how you can pass dependencies through the constructor using the previous `UserService`.

```
// Rewritten AuthenticationService
export class AuthenticationService {
    /*
      Declare at the same time the public
      properties belongs to the class
    */
    constructor(public userService: UserService) { }
}
// Now you can instanciate the AutheticationService like that
const userService = new UserService();
const authenticationService = new AuthenticationService(userService);
```

You can easily share the `userService` instance through all of the objects, and it is no longer the `AuthenticationService`, which has to create a `UserService` instance.

This makes life easier because the injector system will allow you to do all of this without needing to instantiate the dependencies. Let's see this using the previous class in the next section.

Dependency Injection pattern today

Today, to use Dependency Injection, you just have to use the decorator system provided by Typescript and implemented by the framework that you want to use. In our case, as you will see in the Tools chapter, Nest.js provides some decorators that will do almost nothing except add some metadata on the object or property where they will be used.

This metadata will help make the framework aware that those objects can be manipulated, injecting the needed dependencies.

Here is an example of the usage of the `@Injectable()` decorator:

```
@Injectable()
export class UserService { /*...*/ }

@Injectable()
export class AuthenticationService {
     constructor(private userService: UserService) { }
}
```

This decorator will be transpiled and will add some metadata to it. This means that you have accessed `design:paramtypes` after using a decorator on the class, which allows the injector to know the type of the arguments that are dependent on the `AuthenticationService`.

Generally, if you would like to create your own class decorator, this one will take as parameter the `target` that represents the `type` of your class. In the previous example, the type of the `AuthenticationService` is the `AuthenticationService` itself. The purpose of this custom class decorator will be to register the target in a `Map` of services.

```
export Component = () => {
     return (target: Type<object>) => {
          CustomInjector.set(target);
     };
}
```

Of course, you have seen how to register a services into a Map of service, so let's look at how this could be a custom injector. The purpose of this injector will be to register all of the services into a Map, and also to resolve all the dependencies of an object.

```
export const CustomInjector = new class {
  protected services: Map<string, Type<any>> = new Map<string,
Type<any>>();

  resolve<T>(target: Type<any>): T {
    const tokens = Reflect.getMetadata('design:paramtypes', target) ||
[];
    const injections = tokens.map(token =>
CustomInjector.resolve<any>(token));
    return new target(/*...*/injections);
  }

  set(target: Type<any>) {
    this.services.set(target.name, target);
  }
};
```

So, if you would like to instanciate our `AuthenticationService`, which depends on the super `UserService` class, you should call the injector in order to resolve the dependencies and return this instance of the wanted object.

In the following example, we will resolve through the injector the `UserService` that will be passed into the constructor of the `AuthenticationService` in order to be able to instanciate it.

```
const authenticationService =
CustomInjector.resolve<AuthenticationService>(AuthenticationService);
const isValid = authenticationService.validateUser(/* payload */);
```

Nest.js Dependency Injection

From the `@nestjs/common` you have access to the decorators provided by the framework and one of them is the `@Module()` decorator. This decorator is the main decorator to build all of your modules and work with the Nest.js Dependency Injection system between them.

Your application will have at least one module, which is the main one. The application can use only one module (the main one) in the case of a small app. Nonetheless, as your app grows, you will have to create several modules to arrange your app for the main module.

From the main module, Nest will know all of the related modules that you have imported, and then create the application tree to manage all of the Dependency Injections and the scope of the modules.

To do this, the `@Module()` decorator respects the `ModuleMetadata` interface, which defines the properties allowed to configure a module.

```
export interface ModuleMetadata {
      imports?: any[];
      providers?: any[];
      controllers?: any[];
      exports?: any[];
      modules?: any[]; // this one is deprecated.
}
```

To define a module, you have to register all of the services stored in `providers` that will be instantiated by the Nest.js `injector`, as well as the `controllers` that can inject the providers, which are services, registered into the module or those exported by another module through the `exports` property. In such a case, these have to be registered in `imports`.

It is not possible to access an injectable from another module if it has not been exported by the module itself, and if the exporting module hasn't been imported into the concerned module, which has to use the external services.

How does Nest.js create the Dependency injection tree?

In the previous section, we talked about the main module, generally called `AppModule`, which is used to create the app from `NestFactory.create`. From here, Nest.js will have to register the module itself, and it will also go through each module imported to the main module.

Nest.js will then create a `container` for the entire app, which will contain all of the `module`, `globalModule`, and `dynamicModuleMetadata` of the entire application.

After it has created the container, it will initialize the app and, during the initialization, it will instantiate an `InstanceLoader` and a `DependenciesScanner -> scanner.ts`, via which Nest.js will have the possibility to scan every module and metadata related to it. It does this to resolve all of the dependencies and generate the instance of all modules and services with their own injections.

If you want to know the details of the engine, we recommend that you go deep into the two classes: `InstanceLoader` and `DependenciesScanner`.

To have a better understanding of how this works, take a look at an example.

Imagine that you have three modules:

- ApplicationModule
- AuthenticationModule
- UserModule

The app will be created from the `ApplicationModule`:

```
@Module({
     imports: [UserModule, AuthenticationModule]
})
export class ApplicationModule {/*...*/}
```

This imports the `AuthenticationModule`:

```
@Module({
     imports: [UserModule],
     providers: [AuthenticationService]
})
export class AuthenticationModule {/*...*/}

@Injectable()
export class AuthenticationService {
     constructor(private userService: UserService) {}
}
```

And the `UserModule`:

```
@Module({
     providers: [UserService],
     exports: [UserService]
})
export class UserModule {/*...*/}

@Injectable()
export class UserService {/*...*/}
```

In this case, the `AuthenticationModule` must import the `UserModule`, which exports the `UserService`.

We have now built our application's architecture module and have to create the app, which will be allowed to resolve all of the dependencies.

```
const app = await NestFactory.create(ApplicationModule);
```

Essentially, when you create the app, Nest.js will:

- Scan the module.
    - Store the module and an empty scope array (for the main module). The scope will then be populated with the module, which imports this scanned module.
    - Look at the related modules through the `modules` metadata.
- Scan for the modules dependencies as services, controllers, related modules, and exports to store them in the module.
- Bind all of the global modules in each module to the related module.
- Create all of the dependencies by resolving the prototype, creating an instance for each one. For dependencies that have dependencies themselves, Nest.js will resolve them in the same way and include these in the previous level.

What about the global module?

Nest.js also provides a `@Global()` decorator, allowing Nest to store them in a global `Set` of modules, which will be added to the related `Set` of the module concerned.

This type of module will be registered with the `__globalModule__` metadata key and added to the globalModule set of the container. They will then be added to the related `Set` of the module concerned. With a global module, you are allowed to inject components from the module into another module without importing it into the targeted module. This avoids having to import a module, which is possibly used by all of the modules, into all of the modules.

Here is an example:

```
@Module({
     imports: [DatabaseModule, UserModule]
})
export class ApplicationModule {/*...*/}
@Global()
```

```
@Module({
      providers: [databaseProvider],
      exports: [databaseProvider]
})
export class DatabaseModule {/*...*/}
@Module({
      providers: [UserService],
      exports: [UserService]
})
export class UserModule {/*...*/}

@Injectable()
export class UserService {
      // SequelizeInstance is provided by the DatabaseModule store as a
global module
      constructor(@Inject('SequelizeInstance') private readonly
sequelizeInstance) {}
}
```

With all the previous information, you should now be familiar with the mechanism of the Nest.js dependency injection and have a better understanding of how they work together.

The difference between Nest.js and Angular DI

Even if Nest.js is widely based on Angular, there is a major difference between them. In Angular, each service is a singleton, which is the same as Nest.js, but there is a possibility to ask Angular to provide a new instance of the service. To do that in Angular, you can use the `providers` property of the `@Injectable()` decorator to have a new instance of a provider registered in the module and available only for this component. That can be useful to have to avoid overwriting some properties through different components.

Summary

So to recap, we have seen in this chapter how it was unflexible and hard to test an object without using the Dependecy Injection. Also, we have learned more about the evolution of the method to implement the dependencies into the dependent, first by implementing the dependencies into the dependent, then

changing the method by passing them manually into the constructor to arrive with the injector system. This then resolves the dependencies, injecting them in the constructor automatically by resolving a tree, which is how Nest.js uses this pattern.

In the next chapter we will see how Nest.js uses TypeORM, an Object Relational Mapping (ORM) that works with several different relational databases.

Chapter 5. TypeORM

Almost every time you use Nest.js in the real world, you need some kind of persistence for your data. That is, you need to save the data that the Nest.js app receives somewhere, and you need to read data from somewhere so that you can then pass that data as a response to the requests that the Nest.js app receives.

That “somewhere” will be, most of the time, a database.

TypeORM is a Object Relational Mapping (ORM) that works with several different relational databases. An Object Relational Mapping is a tool that converts between objects (such as “Entry” or “Comment,” since we’re building a blog) and tables in a database.

The result of this conversion is an entity (called Data Transfer Object) that knows how to read data from the database to memory (so you can use the data as a response for a request,) as well as how to write to the database from memory (so that you are able to store data for later).

TypeORM is conceptually similar to Sequelize. TypeORM is also written in TypeScript and uses decorators extensively, so it’s a great match for Nest.js projects.

We will obviously focus on using TypeORM together with Nest.js, but TypeORM can also be used in both the browser and the server side, with traditional JavaScript as well as TypeScript.

TypeORM allows you to use both the data mapper pattern, as well as the active record pattern. We will focus on the active record pattern as it greatly reduces the amount of boilerplate code needed to use in the context of a typical Nest.js architecture, like the one explained throughout the book.

TypeORM can also work with MongoDB, though in this case using a dedicated NoSQL ORM such as Mongoose is a more common approach.

What database to use

TypeORM supports the following databases:

- MySQL

- MariaDB
- PostgreSQL
- MS SQL Server
- sql.js
- MongoDB
- Oracle (experimental)

Considering that in this book we are already using PostgreSQL with Sequelize and MongoDB with Mongoose, we decided to use MariaDB with TypeORM.

About MariaDB

MariaDB is an open source, community-driven project led by some of the original developers of MySQL. It was forked from MySQL when Oracle acquired the latter with the intention of keeping it free and open under the GNU General Public License.

The original idea of the project was to act as a drop-in replacement for MySQL. This remains largely true for version up to 5.5, while MariaDB kept its version numbers in sync with the MySQL ones.

Nevertheless, newer versions, starting with versions 10.0, have slightly diverted from this approach. It's still true, though, that MariaDB still focuses on being highly compatible with MySQL and sharing the same API.

Getting started

TypeORM is of course distributed as an npm package. You need to run `npm install typeorm @nestjs/typeorm`.

You also need a TypeORM database driver; in this case, we will install the MySQL/MariaDB one with `npm install mysql`.

TypeORM depends on `reflect-metadata` as well, but luckily we had it previously installed as Nest.js depends on it too, so there's nothing else for us to do. Keep in mind that you will need to install this dependency too if you're using TypeORM outside of a Nest.js context.

NOTE: If you haven't yet, it's always a good idea to install Node.js: `npm install --save-dev @types/node`.

Start the database

In order to get a database to connect to, we will use Docker Compose, with the official MariaDB Docker image, to set up our local development environment. We will point to the `latest` Docker image tag, which at the time of writing, corresponds to version 10.2.14.

```
version: '3'

volumes:
  # for persistence between restarts
  mariadb_data:

services:
  mariadb:
    image: mariadb:latest
    restart: always
    ports:
      - "3306:3306"
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: nestbook
      MYSQL_USER: nest
      MYSQL_PASSWORD: nest
    volumes:
        - mariadb_data:/var/lib/mysql

  api:
    build:
      context: .
      dockerfile: Dockerfile
      args:
        - NODE_ENV=development
    depends_on:
      - mariadb
    links:
      - mariadb
    environment:
      PORT: 3000
    ports:
      - "3000:3000"
```

```
    volumes:
      - .:/app
      - /app/node_modules
    command: >
      npm run start:dev
```

Connect to the database

Now that we have a database to connect TypeORM, let's configure the connection.

We have several ways of configuring TypeORM. The most straightforward one, which is great for getting started, is creating a `ormconfig.json` file in the project root folder. This file will get grabbed automagically by TypeORM on startup.

Here is an example configuration file that suits our usecase (i.e. using Docker Compose with the configuration previously proposed).

ormconfig.json

```
{
   "type": "mariadb",
   "host": "mariadb",
   "port": 3306,
   "username": "nest",
   "password": "nest",
   "database": "nestbook",
   "synchronize": true,
   "entities": ["src/**/*.entity.ts"]
}
```

Some notes on the configuration file:

- The properties `host`, `port`, `username`, `password` and `database` need to match the ones specified earlier in the `docker-compose.yml` file; otherwise, TypeORM will not be able to connect to the MariaDB Docker image.
- The `synchronize` property tells TypeORM whether to create or update the database schema whenever the application starts, so that the schemas match the entities declared in the code. Setting this property to `true` can easily lead to loss of data, so **make sure**

you know what you're doing before enabling this property in production environments.

Initialize TypeORM

Now that the database is running and you are able to successfully establish a connection between it and our Nest.js app, we need to instruct Nest.js to use TypeORM as a module.

Thanks to the `@nest/typeorm` package we previously installed, using TypeORM inside our Nest.js application is as easy as importing the `TypeOrmModule` in our main app module (probably the `app.module.ts`file.)

```
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [
    TypeOrmModule.forRoot(),
    ...
  ]
})

export class AppModule {}
```

Modelling our data

Probably the best thing about using an ORM is that you can take advantage of the modelling abstraction that they provide: basically, they allow us to think about our data and to shape it with properties (including types and relations), generating "object types" (and plugging them to databases tables) that we can then use and manipulate as direct interfaces.

This abstraction layer saves you from writing database-specific code like queries, joins, etc. A lot of people love not having to struggle with selects and the like; so this abstraction layer comes in handy.

Our first entity

When working with TypeORM, this object abstractions are named *entities*.

An entity is basically a class that is mapped to a database table.

With that said, let's create our first entity, which we will name `Entry`. We will use this entity to store entries (posts) for our blog. We will create a new file at `src/entries/entry.entity.ts`; that way TypeORM will be able to find this entity file since earlier in our configuration we specified that entity files will follow the `src/**/*.entity.ts` file naming convention.

```
import { Entity } from 'typeorm';

@Entity()
export class Entry {}
```

The `@Entity()` decorator from the `typeorm` npm package is used to mark the `Entry` class as an entity. This way, TypeORM will know that it needs to create a table in our database for these kinds of objects.

The `Entry` entity is still a bit too simple: we haven't defined a single property for it. We will probably need things like a title, a body, an image and a date for our blog entries, right? Let's do it!

```
import { Entity, Column } from 'typeorm';

@Entity()
export class Entry {
  @Column() title: string;

  @Column() body: string;

  @Column() image: string;

  @Column() created_at: Date;
}
```

Not bad! Each property we define for our entity is marked with a `@Column` decorator. Again, this decorator tells TypeORM how to treat the property: in this case, we are asking for each property to be stored in a column of the database.

Sadly, this entity will not work with this code. This is because each entity needs to have at least one primary column, and we didn't mark any column as such.

Our best bet is to create an `id` property for each entry and store that on a primary column.

```
import { Entity, Column, PrimaryColumn } from 'typeorm';

@Entity()
export class Entry {
   @PrimaryColumn() id: number;

   @Column() title: string;

   @Column() body: string;

   @Column() image: string;

   @Column() created_at: Date;
}
```

Ah, that's better! Our first entity is working now. Let's use it!

Using our models

When having to connect requests to data models, the typical approach in Nest.js is building dedicated services, which serve as the "touch point" with each model, and to build controllers, which link the services to the requests reaching the API. Let's follow the `model -> service -> controller` approach in the following steps.

The service

In a typical Nest.js architecture, the application heavy-lifting is done by the services. In order to follow this pattern, create a new `EntriesService`, using it to interact with the `Entry` entity.

So, let's create a new file at: **`src/entries/entries.service.ts`**

```
import { Component } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';

import { Entry } from './entry.entity';
```

```
@Component()
export class EntriesService {
  constructor(
    // we create a repository for the Entry entity
    // and then we inject it as a dependency in the service
    @InjectRepository(Entry) private readonly entry: Repository<Entry>
  ) {}

  // this method retrieves all entries
  findAll() {
    return this.entry.find();
  }

  // this method retrieves only one entry, by entry ID
  findOneById(id: number) {
    return this.entry.findOneById(id);
  }

  // this method saves an entry in the database
  create(newEntry: Entry) {
    this.entry.save(newEntry);
  }
}
```

The most important part of the service is creating a TypeORM repository with `Repository<Entry>`, and then injecting it in our constructor with `@InjectRepository(Entry)`.

By the way, in case you're wondering, repositories are probably the most commonly used design pattern when dealing with ORMs, because they allow you to abstract the database operations as object collections.

Coming back to the latest service code, once you have created and injected the Entry repository, use it to `.find()` and `.save()` entries from the database, among other things. These helpful methods are added when we create a repository for the entity.

Now that we have taken care of both the data model and the service, let's write the code for the last link: the controller.

The controller

Let's create a controller for exposing the Entry model to the outside world through a RESTful API. The code is really simple, as you can see.

Go ahead and create a new file at: **`src/entries/entries.controller.ts`**

```
import { Controller, Get, Post, Body, Param } from '@nestjs/common';

import { EntriesService } from './entry.service';

@Controller('entries')
export class EntriesController {
  constructor(private readonly entriesSrv: EntriesService) {}

  @Get()
  findAll() {
    return this.entriesSrv.findAll();
  }

  @Get(':entryId')
  findOneById(@Param('entryId') entryId) {
    return this.entriesSrv.findOneById(entryId);
  }

  @Post()
  create(@Body() entry) {
    return this.entriesSrv.create(entry);
  }
}
```

As usual, we are using Nest.js dependency injection to make the `EntryService` available in our `EntryController`.

Building a new module

The last step for our new entity endpoint to work is to include the entity, the service, and the controller in the app module. Instead of doing this directly, we will follow the "separated modules" approach and create a new module for our entries, importing all of the necessary pieces there and then importing the whole module in the app module.

So, let's create a new file named: **src/entries/entries.module.ts**

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

import { Entry } from './entry.entity';
import { EntriesController } from './entry.controller';
import { EntriesService } from './entry.service';

@Module({
  imports: [TypeOrmModule.forFeature([Entry])],
  controllers: [EntriesController],
  components: [EntriesService],
})
export class EntriesModule {}
```

Remember when we included the `TypeOrmModule` in the `AppModule` as one of the first steps of this chapter? We used the `TypeOrmModule.forRoot()` formula there. However, we are using a different one here: `TypeOrmModule.forFeature()`.

This distinction coming from the Nest.js TypeORM implementation allows us to separate different functionalities ("features") in different modules. This way you can adapt your code to some of the ideas and best practices exposed in the Architecture chapter of this book.

Anyway, let's import the new `EntriesModule` into the `AppModule`. If you neglect this step, your main app module won't be aware of the existence of the `EntriesModule` and your app will not work as intended.

src/app.module.ts

```
import { TypeOrmModule } from '@nestjs/typeorm';
import { EntriesModule } from './entries/entries.module';

@Module({
  imports: [
    TypeOrmModule.forRoot(),
    EntriesModule,
    ...
  ]
})

export class AppModule {}
```

That's it! Now you can fire requests to `/entities` and the endpoint will invoke writes and reads from the database.

It's time to give our database a try! We will fire some requests to the endpoints that we previously linked to the database and see if everything works as expected.

We will start with a GET request to the `/entries` endpoint. Obviously, since we haven't created any entries yet, we should receive an empty array as a response.

```
> GET /entries HTTP/1.1
> Host: localhost:3000
< HTTP/1.1 200 OK

[]
```

Let's create a new entry.

```
> GET /entries HTTP/1.1
> Host: localhost:3000
| {
|   "id": 1,
|   "title": "This is our first post",
|   "body": "Bla bla bla bla bla",
|   "image": "http://lorempixel.com/400",
|   "created_at": "2018-04-15T17:42:13.911Z"
| }

< HTTP/1.1 201 Created
```

Success! Let's retrieve the new entry by ID.

```
> GET /entries/1 HTTP/1.1
> Host: localhost:3000
< HTTP/1.1 200 OK
```

```
{
  "id": 1,
  "title": "This is our first post",
  "body": "Bla bla bla bla bla",
  "image": "http://lorempixel.com/400",
  "created_at": "2018-04-15T17:42:13.911Z"
}
```

Yes! Our previous POST request triggered a write in the database and now this last GET request is triggering a read from the database, and returning the data previously saved!

Let's try to retrieve all entries once again.

```
> GET /entries HTTP/1.1
> Host: localhost:3000
< HTTP/1.1 200 OK

[{
  "id": 1,
  "title": "This is our first post",
  "body": "Bla bla bla bla bla",
  "image": "http://lorempixel.com/400",
  "created_at": "2018-04-15T17:42:13.911Z"
}]
```

We just confirmed that requests to the `/entries` endpoint successfully executed reads and writes in our database. This means that our Nest.js app is usable now, since the basic functionality of almost any server application (that is, storing data and retrieving it on demand) is working properly.

Improving our models

Even though we are now reading from and writing to the database through our entity, we only wrote a basic, initial implementation; we should review our code to see what can be improved.

Let's now go back to the entity file, `src/entries/entry.entity.ts`, and figure out what kind of improvements we can do there.

Auto-generated IDs

All of the database entries need to have a unique ID. At this point, we are simply relying on the ID sent by the client when creating the entity (when sending the POST request,) but this is less than desirable.

Any server-side application will be connected to multiple clients, and all of those clients have no way of knowing which ID's are already in use, so it would be impossible for them to generate and send a unique ID with each POST request.

TypeORM provides a couple of ways of generating unique ID's for entities. The first one is using the `@PrimaryGeneratedColumn()` decorator. By using it, you no longer need to include an ID in the body of the POST request, nor do you need to manually generate an ID for an entry before saving it. Instead, the ID is automagically generated by TypeORM whenever you ask for a new entry to be saved to the database.

Our code looks something like the following:

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Entry {
   @PrimaryGeneratedColumn() id: number;

   @Column() title: string;

   @Column() body: string;

   @Column() image: string;

   @Column() created_at: Date;
}
```

It's worth mentioning that these unique ID's will be generated in a sequential way, which means that each ID will be one number higher than the highest already present in the database (the exact method for generating the new ID will depend on the database type.)

TypeORM can go one step further, though: if you pass the `"uuid"` argument to the `@PrimaryGeneratedColumn()` decorator, the generated value will then look like a random collection of letters and numbers with some dashes, making sure they're unique (at least *reasonably* unique.)

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Entry {
   @PrimaryGeneratedColumn('uuid') id: string;

   @Column() title: string;

   @Column() body: string;

   @Column() image: string;

   @Column() created_at: Date;
}
```

Also, remember to change the type of `id` from `number` to `string`!

When was the entry created?

In the original entity definition, the `created_at` field was also expected to be received from the client. We can, however, improve this easily with some more TypeORM magic decorators.

Let's use the `@CreateDateColumn()` decorator to dynamically generate the insertion date for each entry. In other words, you don't need to set the date from the client or create it manually before saving the entry.

Let's update the entity:

```
import {
   Entity,
   Column,
```

```
  CreateDateColumn,
  PrimaryGeneratedColumn,
} from 'typeorm';

@Entity()
export class Entry {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column() title: string;

  @Column() body: string;

  @Column() image: string;

  @CreateDateColumn() created_at: Date;
}
```

Nice, isn't it? How about knowing also when the entry was last modified, as well as how many revisions have been done to it? Again, TypeORM makes both easy to do, and requires no additional code on our side.

```
import {
  Entity,
  Column,
  PrimaryGeneratedColumn,
  CreateDateColumn,
  UpdateDateColumn,
  VersionColumn,
} from 'typeorm';

@Entity()
export class Entry {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column() title: string;

  @Column() body: string;

  @Column() image: string;

  @CreateDateColumn() created_at: Date;
```

```
    @UpdateDateColumn() modified_at: Date;

    @VersionColumn() revision: number;
}
```

Our entity will now automagically handle for us the modification date, as well as the revision number, on each subsequent save operations. You can track changes made to each instance of the entity without having to implement a single line of code!

Column types

When defining columns in our entities using decorators, as exposed above, TypeORM will infer the type of database column from the used property type. This basically means that when TypeORM finds a line like the following

```
@Column() title: string;
```

This maps the `string` property type to a `varchar` database column type.

This will work just fine a lot of the time, but in some occasions we might find ourselves in the position of being more explicit about the type of columns to be created in the database. Fortunately, TypeORM allows this kind of custom behavior with very little overhead.

To customize the column type, pass the desired type as a string argument to the `@Column()` decorator. A specific example would be:

```
@Column('text') body: string;
```

The exact column types that can be used depend on the type of database you are using.

COLUMN TYPES FOR MYSQL / MARIADB

`int`, `tinyint`, `smallint`, `mediumint`, `bigint`, `float`, `double`, `dec`, `decimal`, `numeric`, `date`, `datetime`, `timestamp`, `time`, `year`, `char`, `varchar`, `nvarchar`, `text`, `tinytext`, `mediumtext`, `blob`, `longtext`, `tinyblob`, `mediumblob`, `longblob`, `enum`, `json`, `binary`, `geometry`, `point`, `linestring`, `polygon`, `multipoint`, `multilinestring`, `multipolygon`, `geometrycollection`

COLUMN TYPES FOR POSTGRES

int, int2, int4, int8, smallint, integer, bigint, decimal, numeric, real, float, flo
at4, float8, double precision, money, character
varying, varchar, character, char, text, citext, hstore, bytea, bit, varbit, bit
varying, timetz, timestamptz, timestamp, timestamp without time zone, timestamp
with time zone, date, time, time without time zone, time with time
zone, interval, bool, boolean, enum, point, line, lseg, box, path, polygon, circle, c
idr, inet, macaddr, tsvector, tsquery, uuid, xml, json, jsonb, int4range, int8range
, numrange, tsrange, tstzrange, daterange

COLUMN TYPES FOR SQLITE / CORDOVA / REACT-NATIVE

int, int2, int8, integer, tinyint, smallint, mediumint, bigint, decimal, numeric, f
loat, double, real, double precision, datetime, varying
character, character, native character, varchar, nchar, nvarchar2, unsigned big
int, boolean, blob, text, clob, date

COLUMN TYPES FOR MSSQL

int, bigint, bit, decimal, money, numeric, smallint, smallmoney, tinyint, float, re
al, date, datetime2, datetime, datetimeoffset, smalldatetime, time, char, varchar,
text, nchar, nvarchar, ntext, binary, image, varbinary, hierarchyid, sql_variant,
timestamp, uniqueidentifier, xml, geometry, geography

COLUMN TYPES FOR ORACLE

char, nchar, nvarchar2, varchar2, long, raw, long
raw, number, numeric, float, dec, decimal, integer, int, smallint, real, double
precision, date, timestamp, timestamp with time zone, timestamp with local
time zone, interval year to month, interval day to
second, bfile, blob, clob, nclob, rowid, urowid

If you're not ready to commit yourself to one specific database type and you'd like to keep your options open for the future, it might not be the best idea to use a type that's not available in every database.

NoSQL in SQL

TypeORM has still one last trick in the hat: a `simple-json` column type that can be used in every supported database. With it, you can directly save Plain Old JavaScript Objects in one of the relational database columns. Yes, mindblowing!

Let's put it to use with a new `author` property in the entity.

```
import {
  Entity,
  Column,
  PrimaryGeneratedColumn,
  CreateDateColumn,
  UpdateDateColumn,
  VersionColumn,
} from 'typeorm';

@Entity()
export class Entry {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column() title: string;

  @Column('text') body: string;

  @Column() image: string;

  @Column('simple-json') author: { first_name: string; last_name:
string };

  @CreateDateColumn() created_at: Date;

  @UpdateDateColumn() modified_at: Date;

  @VersionColumn() revision: number;
}
```

The `simple-json` column type allows you to directly store even complex JSON trees without needing to define a model for them first. This can come handy in situations where you appreciate a bit more flexibility than the traditional relational database structure allows.

Relationships between data models

If you followed the chapter up to this point, you will have a way of saving new blog entries to your database through your API and then reading them back.

The next step is to create a second entity to handle comments in each blog entry and then create a relationship between entries and comments in such a way that one blog entry can have several comments that belong to it.

Let's create the `Comments` entity then.

src/comments/comment.entity.ts

```
import {
  Entity,
  Column,
  PrimaryGeneratedColumn,
  CreateDateColumn,
  UpdateDateColumn,
  VersionColumn,
} from 'typeorm';

@Entity()
export class Comment {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column('text') body: string;

  @Column('simple-json') author: { first_name: string; last_name:
string };

  @CreateDateColumn() created_at: Date;

  @UpdateDateColumn() modified_at: Date;

  @VersionColumn() revision: number;
}
```

You have probably noticed that the `Comment` entity is quite similar to the `Entry` entity.

The next step will be to create a "one-to-many" relationship between entries and comments. For that, include a new property in the `Entry` entity with a `@OneToMany()` decorator.

src/entries/entry.entity.ts

```
import {
```

```
  Entity,
  Column,
  PrimaryGeneratedColumn,
  CreateDateColumn,
  UpdateDateColumn,
  VersionColumn,
  OneToMany,
} from 'typeorm';

import { Comment } from '../comments/comment.entity';

@Entity()
export class Entry {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column() title: string;

  @Column('text') body: string;

  @Column() image: string;

  @Column('simple-json') author: { first_name: string; last_name:
string };

  @OneToMany(type => Comment, comment => comment.id)
  comments: Comment[];

  @CreateDateColumn() created_at: Date;

  @UpdateDateColumn() modified_at: Date;

  @VersionColumn() revision: number;
}
```

“One-to-many” relationships have to be bi-directional, so you need to add an inverse relationship “many-to-one” in the `comment` entity. This way, both will get properly “tied up.”

src/comments/comment.entity.ts

```
import {
  Entity,
```

```
    Column,
    PrimaryGeneratedColumn,
    CreateDateColumn,
    UpdateDateColumn,
    VersionColumn,
    ManyToOne,
} from 'typeorm';

import { Entry } from '../entries/entry.entity';

@Entity()
export class Comment {
    @PrimaryGeneratedColumn('uuid') id: string;

    @Column('text') body: string;

    @Column('simple-json') author: { first_name: string; last_name:
string };

    @ManyToOne(type => Entry, entry => entry.comments)
    entry: Entry;

    @CreateDateColumn() created_at: Date;

    @UpdateDateColumn() modified_at: Date;

    @VersionColumn() revision: number;
}
```

The second argument that we're passing to both the `@OneToMany()` and the `@ManyToOne()` decorators is used to specify the inverse relationship that we're also creating on the other related entity. In other words, in the `Entry` we are saving the related `Comment` entity in a property named `comments`. That's why, in the `Comment` entity definition, we pass `entry => entry.comments` as a second argument to the decorator, to the point where in `Entry` the comments be stored.

NOTE: Not all relationships *need* to be bi-directional. "One-to-one" relationships can very well be both uni-directional or bi-directional. In the case of uni-directional "one-to-one" relationships, the owner of the relationship is the one declaring it, and the other entity wouldn't need to know anything about the first one.

That's it! Now each of our entries can have several comments.

How to store related entities

If we talk about code, the most straightforward way of saving a comment that belongs to an entry would be to save the comment and then save the entry with the new comment included. Create a new `Comments` service to interact with the entity, and then modify the `Entry` controller to call that new `Comments` service.

Let's see how. It's not as hard as it sounds!

This would be our new service:

src/comments/comments.service.ts

```
import { Component } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';

import { Comment } from './comment.entity';

@Component()
export class CommentsService {
  constructor(
    @InjectRepository(Comment) private readonly comment:
Repository<Comment>
  ) {}

  findAll() {
    return this.comment.find();
  }

  findOneById(id: number) {
    return this.comment.findOneById(id);
  }

  create(comment: Comment) {
    return this.comment.save(comment);
  }
}
```

The code sure looks familiar, doesn't it? It's very similar to the `EntriesService` that we already had, since we are providing quite the same functionality for both comments and entries.

This would be the modified `Entries` controller:

src/entries/entries.controller.ts

```
import { Controller, Get, Post, Body, Param } from '@nestjs/common';

import { EntriesService } from './entries.service';
import { CommentsService } from '../comments/comments.service';

import { Entry } from './entry.entity';
import { Comment } from '../comments/comment.entity';

@Controller('entries')
export class EntriesController {
  constructor(
    private readonly entriesSrv: EntriesService,
    private readonly commentsSrv: CommentsService
  ) {}

  @Get()
  findAll() {
    return this.entriesSrv.findAll();
  }

  @Get(':entryId')
  findOneById(@Param('entryId') entryId) {
    return this.entriesSrv.findOneById(entryId);
  }

  @Post()
  async create(@Body() input: { entry: Entry; comments: Comment[] }) {
    const { entry, comments } = input;
    entry.comments: Comment[] = [];
    await comments.forEach(async comment => {
      await this.commentsSrv.create(comment);
      entry.comments.push(comment);
    });
    return this.entriesSrv.create(entry);
```

```
    }
}
```

In short, the new `create()` method:

- Receives both a blog entry and an array of comments that belong to that entry.
- Creates a new empty array property (named `comments`) inside the blog entry object.
- Iterates over the received comments, saving each one of them and then pushing them one by one to the new `comments` property of `entry`.
- Finally, saves the `entry`, which now includes a "link" to each comment inside its own `comments`property.

SAVING RELATED ENTITIES THE EASIER WAY

The code we wrote last works, but it's not very convenient.

Fortunately, TypeORM provides us with a easier way to save related entities, though: enabling "cascades".

Setting `cascade` to `true` in our entity will mean that we'll no longer need to separately save each related entity; rather, saving the owner of the relationship to the database will save those related entities at the same time. This way, our previous code can be simplified.

First of all, let's modify our `Entry` entity (which is the owner of the relationship) to enable cascade.

src/entries/entry.entity.ts

```
import {
  Entity,
  Column,
  PrimaryGeneratedColumn,
  CreateDateColumn,
  UpdateDateColumn,
  VersionColumn,
  OneToMany,
} from 'typeorm';

import { Comment } from '../comments/comment.entity';
```

```
@Entity()
export class Entry {
   @PrimaryGeneratedColumn('uuid') id: string;

   @Column() title: string;

   @Column('text') body: string;

   @Column() image: string;

   @Column('simple-json') author: { first_name: string; last_name:
string };

   @OneToMany(type => Comment, comment => comment.id, {
      cascade: true,
   })
   comments: Comment[];

   @CreateDateColumn() created_at: Date;

   @UpdateDateColumn() modified_at: Date;

   @VersionColumn() revision: number;
}
```

This was really easy: we just added a `{cascade: true}` object as third argument for the `@OneToMany()`decorator.

Now, we will refactor the `create()` method on the `Entries` controller.

src/entries/entries.controller.ts

```
import { Controller, Get, Post, Body, Param } from '@nestjs/common';

import { EntriesService } from './entries.service';

import { Entry } from './entry.entity';
import { Comment } from '../comments/comment.entity';

@Controller('entries')
export class EntriesController {
```

```
    constructor(private readonly entriesSrv: EntriesService) {}

    @Get()
    findAll() {
      return this.entriesSrv.findAll();
    }

    @Get(':entryId')
    findAll(@Param('entryId') entryId) {
      return this.entriesSrv.findOneById(entryId);
    }

    @Post()
    async create(@Body() input: { entry: Entry; comments: Comment[] }) {
      const { entry, comments } = input;
      entry.comments = comments;
      return this.entriesSrv.create(entry);
    }
}
```

Please compare the new controller with our previous implementations; we were able to get rid of the dependency on the `Comments` service, as well as an iterator on the `create()` method. This makes our code shorter and cleaner, which is always good as it reduces the risk of introducing bugs.

In this section we found out how to save entities that are related one to another, while saving their relationship as well. This is a crucial step for the success of our related entities. Nice job!

Retrieving related entities in bulk

Now that we know how to save an entity and include its relationships, we'll take a look on how to read both an entity from the database, as well as all their related entities.

The idea in this case is that, when we request a blog entry (only one) from the database, we also get the comments that belong to it.

Of course, since you're familiar with blogs in general (they've been around for a while, right?), you will be aware that not all blogs load both the blog entry and the comments at the same time; many of them load the comments only when you reach the bottom of the page.

To demonstrate the functionality, however, we will assume that our blogging platform will retrieve both the blog entry and the comments at the same time.

We will need to modify the `Entries` service to achieve this. Again, it's going to be quite easy!

src/entries/entries.service.ts

```
import { Component } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';

import { Entry } from './entry.entity';

@Component()
export class EntriesService {
  constructor(
    @InjectRepository(Entry) private readonly entry: Repository<Entry>
  ) {}

  findAll() {
    return this.entry.find();
  }

  findOneById(id: number) {
    return this.entry.findOneById(id, { relations: ['comments'] });
  }

  create(newEntry: Entry) {
    this.entry.save(newEntry);
  }
}
```

We only added a `{ relations: ['comments'] }` as second argument to the `findOneById()` method of the `Entry` repository. The `relations` property of the options object is an array, so we can retrieve as many relationships we need to. Also, it can be used with any `find()` related method (that is, `find()`, `findByIds()`, `findOne()` and so on.)

Lazy relationships

When working with TypeORM, regular relationships (like the ones we have written so far) are *eager*relationships. This means that when we read entities from the database, the `find*()` methods will return the related entities as well, without us needing to write joins or manually read them.

We can also configure our entities to treat relationships as *lazy*, so that the related entities are not retrieved from the database until we say so.

This is achieved by declaring the type of the field that holds the related entity as a `Promise` instead of a direct type. Let's see the difference in code:

```
// This relationship will be treated as eager
@OneToMany(type => Comment, comment => comment.id)
comments: Comment[];

// This relationship will be treated as lazy
@OneToMany(type => Comment, comment => comment.id)
comments: Promise<Comment[]>;
```

Of course, using lazy relationships means that we need to change the way we save our entity to the database. The next code block demonstrates how to save lazy relationships. Pay attention to the `create()` method.

src/entries/entries.controller.ts

```
import { Controller, Get, Post, Body, Param } from '@nestjs/common';

import { EntriesService } from './entries.service';
import { CommentsService } from '../comments/comments.service';

import { Entry } from './entry.entity';
import { Comment } from '../comments/comment.entity';

@Controller('entries')
export class EntriesController {
  constructor(
    private readonly entriesSrv: EntriesService,
    private readonly commentsSrv: CommentsService
  ) {}

  @Get()
```

```
    findAll() {
      return this.entriesSrv.findAll();
    }

    @Get(':entryId')
    findAll(@Param('entryId') entryId) {
      return this.entriesSrv.findOneById(entryId);
    }

    @Post()
    async create(@Body() input: { entry: Entry; comments: Comment[] }) {
      const { entry, comments } = input;
      const resolvedComments = [];
      await comments.forEach(async comment => {
        await this.commentsSrv.create(comment);
        resolvedComments.push(comment);
      });
      entry.comments = Promise.resolve(resolvedComments);
      return this.entriesSrv.create(entry);
    }
}
```

We made the `create()` method "lazy" by:

1. Initializing a new `resolvedComments` empty array.
2. Going through all of the comments received in the request, saving each one and then adding it to the `resolvedComments` array.
3. When all comments are saved, we assign a promise to the `comments` property of `entry`, and then immediately resolve it with the array of comments built in step 2.
4. Save the `entry` with the related comments as an already resolved promise.

The concept of assigning an immediately resolved promise as value of an entity before saving is not easy to digest. Still, we need to resort to this because of the asynchronous nature of JavaScript.

That said, be aware that TypeORM support for lazy relationships is still in the experimental phase, so use them with care.

Other kinds of relationships

So far we've explored "one-to-many" relationships. Obviously, TypeORM supports "one-to-one" and "many-to-many" relationships as well.

One-to-one

Just in case you're not familiar with this kind of relationships, the idea behind it is that one instance of an entity, and only one, belongs to one instance, and only one, of another entity.

To give a more specific example, let's imagine that we were going to create a new `EntryMetadata` entity to store new things that we want to keep track of, like, let's say, the number of likes a blog entry got from readers and a shortlink for each blog entry.

Let's start by creating a new entity called `EntryMetadata`. We will put the file in the `/entry` folder, next to the `entry.entity.ts` file.

src/entries/entry_metadata.entity.ts

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()
export class EntryMetadata {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column() likes: number;

  @Column() shortlink: string;
}
```

The entity we just created is quite simple: it only has the regular `uuid` property, as well as two other properties for storing `likes` for an entry, and also a `shortlink` for it.

Now let's tell TypeORM to include one instance of the `EntryMetadata` entity in each instance of the `Entry` entity.

src/entries/entry.entity.ts

```
import {
  Entity,
```

```
  Column,
  PrimaryGeneratedColumn,
  CreateDateColumn,
  UpdateDateColumn,
  VersionColumn,
  OneToMany,
  OneToOne,
  JoinColumn,
} from 'typeorm';

import { EntryMetadata } from './entry-metadata.entity';
import { Comment } from '../comments/comment.entity';

@Entity()
export class Entry {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column() title: string;

  @Column('text') body: string;

  @Column() image: string;

  @Column('simple-json') author: { first_name: string; last_name:
string };

  @OneToOne(type => EntryMetadata)
  @JoinColumn()
  metadata: EntryMetadata;

  @OneToMany(type => Comment, comment => comment.id, {
    cascade: true,
  })
  comments: Comment[];

  @CreateDateColumn() created_at: Date;

  @UpdateDateColumn() modified_at: Date;

  @VersionColumn() revision: number;
}
```

You might have noticed the `@JoinColumn()` decorator. Using this decorator in "one-to-one" relationships is required by TypeORM.

BI-DIRECTIONAL ONE-TO-ONE RELATIONSHIPS

At this point, the relationship between `Entry` and `EntryMetadata` is uni-directional. In this case, it is probably enough.

Let's say, however, that we want to have the possibility of accessing an instance of `EntryMetadata` directly and then fetch the `Entry` instance it belongs to. Well, we can't do that right now; not until we make the relationship bi-directional.

So, just for the sake of demonstration, we will include the inverse relationship in the `EntryMetadata` instance to the `Entry` instance, so that you know how it works.

src/entries/entry_metadata.entity.ts

```
import { Entity, PrimaryGeneratedColumn, Column, OneToOne } from
'typeorm';

import { Entry } from './entry.entity';

@Entity()
export class EntryMetadata {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column() likes: number;

  @Column() shortlink: string;

  @OneToOne(type => Entry, entry => entry.metadata)
  entry: Entry;
}
```

Make sure you don't include the `@JoinColumn()` decorator on this second entry. That decorator should only be used in the owner entity; in our case, in `Entry`.

The second adjustment we need to make is pointing to the location of the related entity in our original `@OneToOne()` decorator. Remember, we just saw that this needs to be done by passing a second argument to the decorator, like this:

src/entries/entry.entity.ts

```
import {
  Entity,
  Column,
  PrimaryGeneratedColumn,
  CreateDateColumn,
  UpdateDateColumn,
  VersionColumn,
  OneToMany,
  OneToOne,
  JoinColumn,
} from 'typeorm';

import { EntryMetadata } from './entry-metadata.entity';
import { Comment } from '../comments/comment.entity';

@Entity()
export class Entry {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column() title: string;

  @Column('text') body: string;

  @Column() image: string;

  @Column('simple-json') author: { first_name: string; last_name:
string };

  @OneToOne(type => EntryMetadata, entryMetadata => entryMetadata.entry)
  @JoinColumn()
  metadata: EntryMetadata;

  @OneToMany(type => Comment, comment => comment.id, {
    cascade: true,
  })
  comments: Comment[];

  @CreateDateColumn() created_at: Date;

  @UpdateDateColumn() modified_at: Date;
```

```
    @VersionColumn() revision: number;
}
```

That's it! Now we have a beautiful, working bi-directional one-to-one relationship between the `Entry`and the `EntryMetadata` entities.

By the way, if you're wondering how could we save and then retrieve this two related entities, I've got good news for you: it works the same way that we saw with one-to-many relationships. So, either do it by hand as exposed earlier in this chapter, or (my personal favorite) use "cascades" for saving them, and `find*()` to retrieve them!

Many-to-many

The last type of relationship that we can establish for our entities is known as "many-to-many." This means that multiple instances of the owning entity can include multiple instances of the owned entity.

A good example might be us wanting to include "tags" to our blog entries. An entry might have several tags, and a tag can be used in several blog entries, right. That makes the relationship fall under the "many-to-many" typology.

We will save some code here, because these relationships are declared exactly the same way than the "one-to-one" relationships, only changing the `@OneToOne()` decorator to `@ManyToMany()`.

Advanced TypeORM

Let's take a look at security.

Security first

If you went through the Sequelize chapter in this same book, you might be familiar with the concept of lifecycle hooks. In that chapter, we are using a `beforeCreate` hook to encrypt the users' passwords before we save them to our database.

In case you're wondering if such a thing exists also in TypeORM, the answer is yes! Though the TypeORM documentation refers to them as "listeners" instead.

So, to demonstrate its functionality, let's write a very simple `User` entity with a username and a password, and we will make sure to encrypt the password before we save it to the database. The specific listener we will be using is called `beforeInsert` in TypeORM.

```
@Entity
export class User {
   @PrimaryGeneratedColumn('uuid') id: string;

   @Column() username: string;

   @Column() password: string;

   @BeforeInsert()
   encryptPassword() {
      this.password = crypto.createHmac('sha256',
this.password).digest('hex');
   }
}
```

Other listeners

In general, a listener is a method that gets triggered upon a specific event within TypeORM, be it write-related or read-related. We just learned about the `@BeforeInsert()` listener, but we have a few other ones we can take advantage of:

- `@AfterLoad()`
- `@BeforeInsert()`
- `@AfterInsert()`
- `@BeforeUpdate()`
- `@AfterUpdate()`
- `@BeforeRemove()`
- `@AfterRemove()`

Composing and extending entities

TypeORM offers two different ways of reducing code duplication between entities. One of them follows the composition pattern, while the other follows the inheritance pattern.

Even though a lot of authors defend favoring composition over inheritance, we will expose here the two possibilities and let the reader decide which one fits better his/her own particular needs.

EMBEDDED ENTITIES

The way of composing entities in TypeORM is using an artifact known as embedded entity.

Embedded entities are basically entities with some declared table columns (properties) that can be included inside other bigger entities.

Let's go with the example: after reviewing the code we wrote earlier for the entities of both `Entry` and `Comment`, we can easily see that there are (among others) three duplicated properties: `created_at`, `modified_at` and `revision`.

It would be a great idea to create an "embeddable" entity to hold those three properties and then embed them into both our original entities. Let's see how.

We will first create a `Versioning` entity (the name is not great, I know, but should work for you to see the idea) with those three duplicated properties.

src/common/versioning.entity.ts

```
import { CreateDateColumn, UpdateDateColumn, VersionColumn } from
'typeorm';

export class Versioning {
  @CreateDateColumn() created_at: Date;

  @UpdateDateColumn() modified_at: Date;

  @VersionColumn() revision: number;
}
```

Notice that we're not using the @Entity decorator in this entity. This is because it's not a "real" entity. Think of it as an "abstract" entity, i.e. an entity that we will never instantiate directly, but we rather will use to embed it in other instantiable entities in order to give them some reusable functionality. Or, in other words, composing entities from smaller pieces.

So, now we will embed this new "embeddable" entity into our two original entities.

src/entries/entry.entity.ts

```
import {
  Entity,
  Column,
  PrimaryGeneratedColumn,
  OneToMany,
  OneToOne,
  JoinColumn,
} from 'typeorm';

import { EntryMetadata } from './entry-metadata.entity';
import { Comment } from '../comments/comment.entity';
import { Versioning } from '../common/versioning.entity';

@Entity()
export class Entry {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column() title: string;

  @Column('text') body: string;

  @Column() image: string;

  @Column('simple-json') author: { first_name: string; last_name:
string };

  @OneToOne(type => EntryMetadata, entryMetadata => entryMetadata.entry)
  @JoinColumn()
  metadata: EntryMetadata;

  @OneToMany(type => Comment, comment => comment.id, {
    cascade: true,
  })
  comments: Comment[];

  @Column(type => Versioning)
  versioning: Versioning;
}
```

src/comments/comment.entity.ts

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

import { Versioning } from '../common/versioning.entity';

@Entity()
export class Comment {
  @PrimaryGeneratedColumn('uuid') id: string;

  @Column('text') body: string;

  @Column('simple-json') author: { first_name: string; last_name:
string };

  @Column(type => Versioning)
  versioning: Versioning;
}
```

Even in this really simple case, we've reduced the two original entities from three different properties to only one! In both the `Entry` entity and the `Comment` entity, the `versioning` column will be actually replaced by the properties inside the `Versioning` embedded entity when we invoke any of their reading or writing methods.

ENTITY INHERITANCE

The second choice that TypeORM offers for reusing code between our entities is using entity inheritance.

If you're already familiar with TypeScript, entity inheritance is quite easy to understand (and implement) when you take into account that entities are nothing more (and nothing less!) than regular TS classes with some decorators on top.

For this particular example, let's imagine that our Nest.js-based blog has been online for some time, and that it has become quite a success. Now we would like to introduce sponsored blog entries so that we can make a few bucks and invest them in a few more books.

The thing is, sponsored entries are going to be a lot like regular entries, but with a couple of new properties: sponsor name and sponsor URL.

In this case, we might decide, after quite some thought, to extend our original `Entry` entity and create a `SponsoredEntry` out of it.

src/entries/sponsored-entry.entity.ts

```
import { Entity, Column } from 'typeorm';

import { Entry } from './entry.entity';

@Entity()
export class SponsoredEntry extends Entry {
  @Column() sponsorName: string;

  @Column() sponsorUrl: string;
}
```

That's about it. Any new instance we create from the `SponsoredEntry` entity will have the same columns from the extended `Entry` entity, plus the two new columns we defined for `SponsoredEntry`.

Caching

TypeORM brings a caching layer out of the box. We can take advantage of it with only a little overhead. This layer is specially useful if you are designing an API that expects a lot of traffic and/or you need the best performance you can get.

Both cases would benefit increasingly from the cache because we use more complex data retrieval scenarios, such as complex `find*()` options, lots of related entities, etc.

The caching needs to be explicitly activated when connecting to the database. In our case so far, this will be the `ormconfig.json` file that we created at the beginning of the chapter.

ormconfig.json

```
{
  "type": "mariadb",
  "host": "db",
  "port": 3306,
  "username": "nest",
  "password": "nest",
  "database": "nestbook",
  "synchronize": true,
```

```
    "entities": ["src/**/*.entity.ts"],
    "cache": true
}
```

After activating the caching layer on the connection, we will need to pass the `cache` option to our `find*()` methods, like in the following example:

```
this.entry.find({ cache: true });
```

The line of code above will make the `.find()` method to return the cached value if it's present and not expired, or the value from the corresponding database table otherwise. So, even if the method is fired three thousand times within the expiration time window (see below), only one database query would be actually executed.

TypeORM uses a couple of defaults when dealing with caches:

1. The default cache lifetime is 1,000 milliseconds (i.e. 1 second.) In case we need to customize the expiration time, we just need to pass the desired lifetime as value to the `cache` property of the options object. In the case above, `this.entry.find({ cache: 60000 })` would set a cache TTL of 60 seconds.
2. TypeORM will create a dedicated table for the cache in the same database you're already using. The table will be named `query-result-cache`. This is not bad, but it can greatly improved if we have a Redis instance available. In that cache, we will need to include our Redis connection details in the `ormconfig.json` file:

ormconfig.json

```
{
    "type": "mariadb",
    ...
    "cache": {
        "type": "redis",
        "options": {
            "host": "localhost",
            "port": 6379
        }
    }
}
```

This way we can easily improve the performance of our API under heavy load.

Building a query

The TypeORM's repository methods for retrieving data from our database greatly isolates the complexity of querying away from us. They provide a very useful abstraction so that we don't need to bother with actual database queries.

However, apart from using these various `.find*()` methods, TypeORM also provides a way of manually executing queries. This greatly improves flexibility when accessing our data, at the cost of demanding us to write more code.

The TypeORM tool for executing queries is the `QueryBuilder`. A very basic example could involve refactoring our good old `findOneById()` method in the `EntriesService` so that it uses the `QueryBuilder`.

src/entries/entries.service.ts

```
import {getRepository} from "typeorm";
...

findOneById(id: number) {
   return getRepository(Entry)
      .createQueryBuilder('entry')
      .where('entry.id = :id', { id })
      .getOne();
}

...
```

Another slightly more complex scenario would be to build a join in order to also retrieve the related entities. We will come back once again to the `findOneById()` method we just modified to include the related comments.

src/entries/entries.service.ts

```
import {getRepository} from "typeorm";
...

findOneById(id: number) {
   return getRepository(Entry)
      .createQueryBuilder('entry')
      .where('entry.id = :id', { id })
```

```
        .leftJoinAndSelect('entry.comments', 'comment')
        .getOne();
}

...
```

Building our model from a existing database

Up until this point, we have started with a "clean" database, then created our models, leaving to TypeORM the task of transforming the models into database columns.

This is the "ideal" situation, but... What if we found ourselves in the opposite situations? What if we already had a database with tables and columns populated?

There's a nice open source project we can use for that: typeorm-model-generator. It's packed as a command line tool and can be run with `npx`.

NOTE: In case you're not familiar with it, `npx` is a command that comes out of the box with `npm` > 5.2 and that allows us to run npm modules from the command line without having to install them first. To use it, you just need to prepend `npx` to the regular commands from the tool. We would use `npx ng new PROJECT-NAME` on our command line, for example, if we wanted to scaffold a new project with Angular CLI.

When it's executed, typeorm-model-generator will connect to the specified database (it supports roughly the same ones that TypeORM does) and will generate entities following the settings we pass as command line arguments.

Since this is a useful tool for only some very specific use cases, we will leave the configuration details out of this book. However, if you find yourself using this tool, go ahead and check its GitHub repository.

Summary

TypeORM is a very useful tool and enables us to do a lot of heavy lifting when dealing with databases, while greatly abstracting things like data modelling, queries, and complex joins, thus simplifying our code.

It's also very suitable for being used in Nest.js-based projects thanks to the great support the framework provides through the `@nest/typeorm` package.

Some of the things that we covered in this chapter are:

- The database types supported by TypeORM and some hints on how to choose one.
- How to connect TypeORM to your database.
- What is an entity and how to create your first one.
- Storing and retrieving data from your database.
- Leveraging TypeORM to make it easier working with metadata (ID's, creation and modification dates...).
- Customizing the type of columns in your database to match your needs.
- Building relationships between your different entities and how to handle them when reading from and writing to the database.
- More advanced procedures like reusing code through composition or inheritance; hooking into lifecycle events; caching; and building queries by hand.

All in all, we really think the more familiar you grow with Nest.js, the more likely you start to feel comfortable writing TypeORM code, since they both look alike in a few aspects as their extensive use of TypeScript decorators.

In the next chapter we cover Sequelize, which is a promise-based ORM.

Chapter 6. Sequelize

Sequelize is a promise-based ORM working for Node.js v4 and later. This ORM supports many dialects, such as:

- `PostgreSQL`
- `MySQL`
- `SQLite`
- `MSSQL`

This provides a solid support for transactions. With Sequelize you have the possibility of using `sequelize-typescript`, which provides decorators to put in your entity and manages all the fields of your model, with types and constraints.

Also, Sequelize comes from many hooks providing you with the significant advantage of being able to check and manipulate your data at any level of the transaction.

In this chapter, we will see how to configure your database using `postgresql` and how to configure the connection to your database. After that we will see how to implement our first entity, which will be a simple `User` entity and then how to create a provider for this entity in order to inject the entity into a `UserService`. We will also see the migration system through `umzung`, and how to create our first migration file.

You can have a look on the on the `src/modules/database`, `src/modules/user`, `/src/shared/config`, and `/src/migrations /migrate.ts` of the repository.

Configure Sequelize

In order to be able to use Sequelize, we have first to set up the connection between sequelize and our database. In order to do that, we will create the `DatabaseModule`, which will contain the provider of the sequelize instance.

In order to set up this connection, we will define a configuration file, which will have as properties all you need to be connected to your database. This configuration will have to implement the `IDatabaseConfig` interface in order to void to forget some parameters.

```
export interface IDatabaseConfigAttributes {
    username: string;
```

```
        password: string;
        database: string;
        host: string;
        port: number;
        dialect: string;
        logging: boolean | (() => void);
        force: boolean;
        timezone: string;
}

export interface IDatabaseConfig {
        development: IDatabaseConfigAttributes;
}
```

This configuration should be set up as the following example, and set the parameters through the environment variable or the default value.

```
export const databaseConfig: IDatabaseConfig = {
        development: {
                username: process.env.POSTGRES_USER ||
'postgres',
                password: process.env.POSTGRES_PASSWORD || null,
                database: process.env.POSTGRES_DB || 'postgres',
                host: process.env.DB_HOST || '127.0.0.1',
                port: Number(process.env.POSTGRES_PORT) || 5432,
                dialect: 'postgres',
                logging: false,
                force: true,
                timezone: '+02:00',
        }
};
```

After the configuration, you have to create the appropriate provider, which will have the purpose to create the instance of sequelize using the right configuration. In our case we just set up the environment configuration, but you can set up all the configuration with the same pattern, you just need to change the values.

This instance is for you to be aware about the different model that should be provided. In order to tell sequelize which model we need, we use the `addModels` method on the instance and pass an array of model. Of course, in the following section we will see how to implement a new model.

```
export const databaseProvider = {
      provide: 'SequelizeInstance',
      useFactory: async () => {
            let config;
            switch (process.env.NODE_ENV) {
                  case 'prod':
                  case 'production':
                  case 'dev':
                  case 'development':
                  default:
                        config = databaseConfig.development;
            }

            const sequelize = new Sequelize(config);
            sequelize.addModels([User]);
            return sequelize;
      }
};
```

This provider will return the instance of Sequelize. This instance will be useful to use the transaction provided by Sequelize. Also, in order to be able to inject it, we have provided in the `provide` parameter, the name of the token `SequelizeInstance`, which will be used to inject it.

Sequelize also provides a way to immediately synchronize your model and your database using `sequelize.sync()`. This synchronisation should not be used in production mode, because it recreates a new database and removes all of the data each time.

We have now set up our Sequelize configuration, and we need to set up the `DatabaseModule` as shown in the following example:

```
@Global()
@Module({
      providers: [databaseProvider],
      exports: [databaseProvider],
})
export class DatabaseModule {}
```

We defined the `DatabaseModule` as a `Global` in order to be added into all the modules as a related module, letting you inject the provider `SequelizeInstance` into any module as following:

```
@Inject('SequelizeInstance`) private readonly sequelizeInstance
```

We now have a complete working module to access our data in the database.

Create a model

After having set up the sequelize connection, we have to implement our model. As seen in the previous section, we tell Sequelize that we will have the `User` model using this method`sequelize.addModels([User]);`.

You now see all of the required features to set up it.

@Table

This decorator will allow you to configure our representation of the data, and here are some parameters:

```
{


    timestamps:  true,
    paranoid:  true,
    underscored:  false,
    freezeTableName:  true,
    tableName:  'my_very_custom_table_name'
}
```

The `timestamp` parameter will tell you that you want to have an `updatedAt` and `deletedAt` columns. The `paranoid` parameter allows you to soft delete data instead of removing it to lose your data. If you pass `true`, Sequelize will expected a `deletedAt` column in oder to set the date of the remove action.

The `underscored` parameter will automatically transform all of the camelcase columns into underscored columns.

The `freezTableName` will provide a way to avoid Sequelize to pluralize the name of the table.

The `tableName` allows you to set the name of the table.

In our case we only use `timestamp: true, tableName: 'users'` in order to get the `updatedAt` and `createdAt` column and name the table as `users`.

@column

This decorator will help define our column. You can also not pass any parameter, so in this case Sequelize will try to infer the column type. The types that can be inferred are `string`, `boolean`, `number`, `Date`and `Blob`.

Some parameter allows us to define some constraints on the column. Let's imagine the `email` column, where we would like this email as a string and that cannot be null, so the email has to be unique. Sequelize can recognize an email, but we have to tell it how to validate the email passing
the `validate#isUnique` method.

Take a look at the following example.

```
@Column({
      type: DataType.STRING,
      allowNull: false,
      validate: {
            isEmail: true,
            isUnique: async (value: string, next: any): Promise<any> => {
                  const isExist = await User.findOne({ where: { email:
value }});
                  if (isExist) {
                        const error = new Error('The email is already
used.');
                        next(error);
                  }
                  next();
            },
      },
})
```

In the previous example, we passed some options, but we could also use some decorator as `@AllowNull(value: boolean)`, `@Unique` or even `@Default(value: any)`.

To set an `id` column, the `@PrimaryKey` and `@AutoIncrement` decorators are an easy way to set up the constraint.

Create the User model

Now that we have seen some useful decorator, let's create our first model, the `User`. In order to do that, we will create the class that has to extend from the base class `Model<T>`, and this class takes a template value for the class itself.

```
export class User extends Model<User> {...}
```

We now add the `@Table()` decorator in order to configure our model. This decorator takes options corresponding to the interface `DefineOptions` and as we described in the ***@Table section*** we will pass as options the timestamp as true and the name of the table.

```
@Table({ timestamp: true, tableName: 'users' } as IDefineOptions)
export class User extends Model<User> {...}
```

Now we have to define some columns for our model. To do this, `sequelize-typescript` provides the `@Column()` decorator. This decorator allows us to provide some options to configure our field. You can pass the data type `DataType.Type` directly.

```
@Column(DataTypes.STRING)
public email: string;
```

You can also use the options shown in the ***@Column section*** in order to validate and ensure the data of the email.

```
@Column({
      type: DataType.STRING,
      allowNull: false,
      validate: {
            isEmail: true,
            isUnique: async (value: string, next: any): Promise<any> => {
                  const isExist = await User.findOne({
                        where: { email: value }
                  });
                  if (isExist) {
                        const error = new Error('The email is already
used.');
                        next(error);
                  }
                  next();
            },
```

```
    },
})
public email: string;
```

You now know how to set up a column, so let's set up the rest of the model with the column that we need for a simple user.

```
@Table(tableOptions)
export class User extends Model<User> {
    @PrimaryKey
    @AutoIncrement @Column(DataType.BIGINT)
    public id: number;

    @Column({
        type: DataType.STRING,
        allowNull: false,
    })
    public firstName: string;

    @Column({
        type: DataType.STRING,
        allowNull: false,
    })
    public lastName: string;

    @Column({
        type: DataType.STRING,
        allowNull: false,
        validate: {
            isEmail: true,
            isUnique: async (value: string, next: any): Promise<any>
=> {
                const isExist = await User.findOne({
                    where: { email: value }
                });
                if (isExist) {
                    const error = new Error('The email is already
used.');
                    next(error);
                }
                next();
```

```
            },
        },
    })
    public email: string;

    @Column({
        type: DataType.TEXT,
        allowNull: false,
    })
    public password: string;

    @CreatedAt
    public createdAt: Date;

    @UpdatedAt
    public updatedAt: Date;

    @DeletedAt
    public deletedAt: Date;
}
```

In all the added columns, you can see the password of type `TEXT`, but of course, you cannot store a password as a plain text, so we have to hash it in order to protect it. To do that, use the lifeCycle hooks provided by Sequelize.

LifeCycle hooks

Sequelize come with many lifeCycle hooks that allow you to manipulate and check the data along the process of creating, updating, or deleting a data.

Here are some interesting hooks from Sequelize.

```
beforeBulkCreate(instances, options)
beforeBulkDestroy(options)
beforeBulkUpdate(options)

beforeValidate(instance, options)
afterValidate(instance, options)

beforeCreate(instance, options)
beforeDestroy(instance, options)
```

```
beforeUpdate(instance, options)
beforeSave(instance, options)
beforeUpsert(values, options)

afterCreate(instance, options)
afterDestroy(instance, options)
afterUpdate(instance, options)
afterSave(instance, options)
afterUpsert(created, options)

afterBulkCreate(instances, options)
afterBulkDestroy(options)
afterBulkUpdate(options)
```

In this case, we need to use the `@BeforeCreate` decorator in order to hash the password and replace the original value before storing it in the database.

```
@Table(tableOptions)
export class User extends Model<User> {
      ...
      @BeforeCreate
      public static async hashPassword(user: User, options: any) {
            if (!options.transaction) throw new Error('Missing
transaction.');

            user.password = crypto.createHmac('sha256',
user.password).digest('hex');
      }
}
```

The `BeforeCreate` previously written allows you to override the `password` property value of the user in order to override it before the insertion of the object into the database, and to ensure a minimum of security.

Injecting a model into a service

Our first `User` model is now setup. Of course, we will have to inject it into a service or even a controller. To inject a model anywhere else, we must first create the appropriate provider in order to give it to the module.

This provider will define the key to use in order to inject it and take as a value the `User` model that we have implemented before.

```
export const userProvider = {
     provide: 'UserRepository',
     useValue: User
};
```

To inject it in into a service we will use the `@Inject()` decorator, which can take the string defined in the previous example `UserRepository`.

```
@Injectable()
export class UserService implements IUserService {
     constructor(@Inject('UserRepository') private readonly
UserRepository: typeof User) { }
     ...
}
```

After injecting the model into the service, you will be able to use it to access and manipulate the data as you want. For example, you can execute `this.UserRepository.findAll()` to register the data in your database.

Finally, we have to set up the module to take as providers, the `userProvider` that provides access to the model and the `UserService`. The `UserService` can be exported to be used in another module by importing the `UserModule`.

```
@Module({
     imports: [],
     providers: [userProvider, UserService],
     exports: [UserService]
})
export class UserModule {}
```

Usage of Sequelize transaction

You might remark this line, `if (!options.transaction) throw new Error('Missing transaction.');`, in the `hashPassword` method decorated with the `@BeforeCreate`. As said before, Sequelize provides a strong support of the transaction. So for each action or process of action, you can use a transaction. To use the Sequelize transaction, take a look at the following example of a `UserService`.

```
@Injectable()
export class UserService implements IUserService {
     constructor(@Inject('UserRepository') private readonly
UserRepository: typeof User,
                    @Inject('SequelizeInstance') private readonly
sequelizeInstance) { }
     ...
}
```

We have injected both the model and the Sequelize instance that we talked about earlier in this chapter.

To use a transaction to wrap some access to the database, you can do the following:

```
public async create(user: IUser): Promise<User> {
     return await this.sequelizeInstance.transaction(async transaction =>
{
          return await this.UserRepository.create<User>(user, {
               returning: true,
               transaction,
          });
     });
}
```

We use the `sequelizeInstance` to create a new transaction and pass it to the `create` method of the `UserRepository`.

Migration

With Sequelize you have a way to sync your model and your database. The thing is, this synchronization will remove all of your data in order to recreate all of the tables representing the model. So, this feature is useful in testing, but not in a production mode.

In order to manipulate your database, you have the possibility to use `umzung`, a framework agnostic library and migration tool for Nodejs. It is not related to any database, but provides an API in order to migrate or rollback the migration.

When you are using the command `npm run migrate up`, which executes `ts-node migrate.ts`, you can pass `up/down` as a parameter. In order to track all of the

migration already applied, a new table will be created with the default name `SequelizeMeta`, and all of the applied migrations will be stored in this table.

Our migration file can be found in the repository as the root with the name `migrate.ts`. Also, all of the migrations files will be stored in the `migrations` folder of the repository example.

Configuring the migration script

In order to configure the umzung instance, you will be able to set some options:

- `storage`, which correspond to the `sequelize` string key for us
- `storageOptions`, which will take Sequelize, and it is in this option that you can change the default name of the table of the column used to store the name of the migrations applied throughout the `modelName`, `tableName` and `columnName` properties.

Some other configurations are able, in order to set the up method name and the down method name, to pass a logging function. The `migrations` property will allow you to provide some params to pass to the method up/down and the path of the migrations to apply with the appropriate pattern.

```
const umzug = new Umzug({
    storage: 'sequelize',
    storageOptions: { sequelize },

    migrations: {
        params: [
            sequelize,
            sequelize.constructor, // DataTypes
        ],
        path: './migrations',
        pattern: /\.ts$/
    },

    logging: function () {
        console.log.apply(null, arguments);
    }
});
```

Create a migration

To execute the migration script, provide the migration that you want to apply. Imagine that you want to create the `users` table using migration. You must set an `up` and a `down` method.

```
export async function up(sequelize) {
     // language=PostgreSQL
     sequelize.query(`
          CREATE TABLE "users" (
               "id" SERIAL UNIQUE PRIMARY KEY NOT NULL,
               "firstName" VARCHAR(30) NOT NULL,
               "lastName" VARCHAR(30) NOT NULL,
               "email" VARCHAR(100) UNIQUE NOT NULL,
               "password" TEXT NOT NULL,
               "birthday" TIMESTAMP,
               "createdAt" TIMESTAMP NOT NULL,
               "updatedAt" TIMESTAMP NOT NULL,
               "deletedAt" TIMESTAMP
          );
     `);

     console.log('*Table users created!*');
}

export async function down(sequelize) {
     // language=PostgreSQL
     sequelize.query(`DROP TABLE users`);
}
```

In each method, the parameter will be `sequelize`, which is the instance used in the configuration file. Throughout this instance you can use the que query method in order to write our SQL query. In the previous example, the function `up` will execute the query to create the `users` table. The `down` method has the purpose to drop this table in case of a rollback.

Summary

In this chapter you have seen how to set up the connection to the database by instanciating a Sequelize instance, using the factory in order to inject the instance directly in another place.

Also, you have seen decorators provided by sequelize-typescript in order to set up a new model. You have also seen how to add some constraints on the columns and how to use the lifeCycle hooks to hash a password before saving it. Of course, the hooks can be used to validate some data or check some information before doing anything else. But you also have seen how to use the `@BeforeCreate` hook. You are therefore ready to use a Sequelize transaction system.

Finally, you have seen how to configure umzung to execute migrations, and how to create your first migration in order to create the users table.

In the next chapter you will learn how to use Mongoose.

Chapter 7. Mongoose

Mongoose is the third and last database mapping tool that we will be covering in this book. It is the best known MongoDB mapping tool in the JavaScript world.

A word about MongoDB

When MongoDB was initially released, in 2009, it took the database world by storm. At that point the vast majority of databases in use were relational, and MongoDB quickly grew to be the most popular non-relational database (also known as "NoSQL".)

NoSQL databases difer from relational databases (such as MySQL, PostgreSQL, etc.) in that they model the data they store in ways other than tables related one to another.

MongoDB, specifically, is a "document-oriented database." It saves data in "documents" encoded in BSON format ("Binary JSON", a JSON extension that includes various data types specific for MongoDB). The MongoDB documents are grouped in "collections."

Traditional relational databases separate data in tables and columns, similar to a spreadsheet. On the other hand, document-oriented databases store complete data objects in single instances of the database, similar to a text file.

While relational databases are heavily structured, document-oriented ones are much more flexible, since developers are free to use non-predefined structures in our documents, and even completely change our data structure from document instance to document instance.

This flexibility and lack of defined structure means that is usually easier and faster to "map" (transform) our objects in order to store them in the database. This brings reduced coding overhead and faster iterations to our projects.

A word about Mongoose

Mongoose is technically not an ORM (Object Relational Mapping) though it's commonly referred to as one. Rather, it is an ODM (Object Document Mapping) since MongoDB itself is based in documents instead of relational

tables. The idea behind ODM's and ORM's is the same, though: providing an easy-to-use solution for data modelling.

Mongoose works with the notion of "schemas." A schema is simply an object that defines a collection (a group of documents) and the properties and allowed types of values that the document instances will have (i.e. what we would call "their shape.").

Mongoose and Nest.js

Just like we saw in the TypeORM and the Sequelize chapters, Nest.js provides us with a module that we can use with Mongoose.

Getting started

As a first step, we need to install the Mongoose npm package, as well as the Nest.js/Mongoose npm package.

Run `npm install --save mongoose @nestjs/mongoose` in your console, and `npm install --save-dev @types/mongoose` inmediately after.

Set up the database

Docker Compose is the easiest way to get started with MongoDB. There's an official MongoDB image in the Docker registry we recommend that you use. The latest stable version at the moment of writing this is `3.6.4`.

Let's create a Docker Compose file to build and start both the database we will be using, as well as our Nest.js app, and link them together so that we can access the database later from our code.

```
version: '3'

volumes:
  mongo_data:

services:
  mongo:
    image: mongo:latest
    ports:
    - "27017:27017"
```

```
    volumes:
    - mongo_data:/data/db
  api:
    build:
      context: .
      dockerfile: Dockerfile
      args:
        - NODE_ENV=development
    depends_on:
      - mongo
    links:
      - mongo
    environment:
      PORT: 3000
    ports:
      - "3000:3000"
    volumes:
      - .:/app
      - /app/node_modules
    command: >
      npm run start:dev
```

We're pointing to the `latest` tag of the MongoDB image, which is an alias that resolves to the most recent stable version. If you're feeling adventurous, feel free to change the tag to `unstable`... though be aware that things might break!

Start the containers

Now that your Docker Compose file is ready, fire up the containers and start working!

Run `docker-compose up` in your console to do it.

Connect to the database

Our local MongoDB instance is now running and ready to accept connections.

We need to import the Nest.js/Mongoose module that we installed a couple of steps ago into our main app module.

```
import { MongooseModule } from '@nestjs/mongoose';
```

```
@Module({
  imports: [
    MongooseModule.forRoot(),
    ...
  ],
})
export class AppModule {}
```

We are adding the `MongooseModule` to the `AppModule` and we're relying on the `forRoot()` method to properly inject the dependency. You might find the `forRoot()` method familiar if you read the chapter about TypeORM, or if you are familiar with Angular and its official routing module.

There's a *captcha* with the code above: it won't work, because there's still no way for Mongoose or the `MongooseModule` to figure out *how* to connect to our MongoDB instance.

THE CONNECTION STRING

If you check in the Mongoose documentation or make a quick search on Google, you'll see that the usual way of connecting to a MongoDB instance is by using the `'mongodb://localhost/test'` string as an argument for the `.connect()` method in Mongoose (or even in the Node MongoDB native client.)

That string is what is known as a "connection string." The connection string is what tells any MongoDB client how to connect to the corresponding MongoDB instance.

The bad news here is that, in our case, the "default" example connection string will not work, because we are running our database instance inside a container linked from another container, a Node.js one, which is the one that our code runs in.

The good news, though, is that we can use that Docker Compose link to connect to our database, because Docker Compose establishes a virtual network connection between both containers, the MongoDB one and the Node.js one.

So, the only thing that we need to do is changing the example connection string to

```
'mongodb://mongo:27017/nest'
```

where `mongo` is the name of our MongoDB container (we specified this is the Docker Compose file), `27017` is the port that the MongoDB container is exposing (27017 being the default for MongoDB), and `nest` is the collection we will store our documents on (you're free to change it to your heart's content.)

THE RIGHT ARGUMENT FOR THE `FORROOT()` METHOD

Now that we have adjusted our connection string, let's modify our original `AppModule` import.

```
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [
    MongooseModule.forRoot('mongodb://mongo:27017/nest'),
    ...
  ],
})
export class AppModule {}
```

The connection string is now added as an argument to the `forRoot()` method, so Mongoose is aware of *how* to connect to the database instance and will start successfully.

Modelling our data

We already mentioned before that Mongoose works with the concept of "schemas."

Mongoose schemas play a similar role to TypeORM entities. However, unlike the latter, the former are not classes, but rather plain objects that inherit from the `Schema` prototype defined (and exported) by Mongoose.

In any case, schemas need to be instantiated into "models" when you are ready to use them. We like to think about schemas as "blueprints" for objects, and about "models" as object factories.

Our first schema

With that said, let's create our first entity, which we will name `Entry`. We will use this entity to store entries (posts) for our blog. We will create a new file

at `src/entries/entry.entity.ts`; that way TypeORM will be able to find this entity file since earlier in our configuration we specified that entity files will follow the `src/**/*.entity.ts` file naming convention.

Let's create our first schema. We will use it as a blueprint for storing our blog entries. We will also place the schema next to the other blog entries related files, grouping our files by "domain" (that is, by functionality.)

NOTE: You're free to organize your schemas as you see fit. We (and the official Nest.js documentation) suggest storing them near the module where you use each one of them. In any case, you should be good with any other structural approach as long as you correctly import your schema files when you need them.

src/entries/entry.schema.ts

```
import { Schema } from 'mongoose';

export const EntrySchema = new mongoose.Schema({
   _id: Schema.Types.ObjectId,
   title: String,
   body: String,
   image: String,
   created_at: Date,
});
```

The schema we just wrote is:

1. Creating an object with the properties we need for our blog entries.
2. Instantiating a new `mongoose.Schema` type object.
3. Passing our object to the constructor of the `mongoose.Schema` type object.
4. Exporting the instantiated `mongoose.Schema`, so that it can be used elsewhere.

NOTE: Storing the ID of our objects in a property called `_id`, starting with underscore, it's a useful convention when working with Mongoose; it'll make it possible later to rely on the Mongoose `.findById()` model method.

INCLUDING THE SCHEMA INTO THE MODULE

The next step is to "notify" the Nest.js `MongooseModule` that you intend to use the new schema we created. For that, we need to create an "Entry" module (in case we didn't have one just yet) like the following:

src/entries/entries.module.ts

```
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

import { EntrySchema } from './entry.schema';

@Module({
  imports: [
    MongooseModule.forFeature([{ name: 'Entry', schema: EntrySchema }]),
  ],
})
export class EntriesModule {}
```

Quite similarly to what we did in the TypeORM chapter, we now need to use the `forFeature()` method of the `MongooseModule` in order to define the schemas that it needs to register to be used by models in the scope of the module.

Again, the approach is heavily influenced by Angular modules like the router, so it maybe looks familiar to you!

If not, note that this way of handling dependencies greatly increases the decoupling between functional modules in our apps, enabling us to easily include, remove and reuse features and functionality just by adding or removing modules to the imports in the main `AppModule`.

INCLUDE THE NEW MODULE INTO THE MAIN MODULE

And, talking about the `AppModule`, don't forget to import the new `EntriesModule` into the root `AppModule`, so that we can successfully use the new functionality we are writing for our blog. Let's do it now!

```
import { MongooseModule } from '@nestjs/mongoose';

import { EntriesModule } from './entries/entries.module';

@Module({
```

```
    imports: [
      MongooseModule.forRoot('mongodb://mongo:27017/nest'),
      EntriesModule,
      ...
    ],
})
export class AppModule {}
```

Using the schema

As mentioned before, we will use the schema we just defined to instantiate a new data model that we will be able to use in our code. Mongoose models are the ones that do the heavy lifting in regards to mapping objects to database documents, and also abstract common methods for operating with the data, such as `.find()` and `.save()`.

If you've come from the TypeORM chapter, models in Mongoose are very similar to repositories in TypeORM.

When having to connect requests to data models, the typical approach in Nest.js is building dedicated services, which serve as the "touch point" with each model, and controllers. This links the services to the requests reaching the API. We follow the `data model -> service -> controller` approach in the following steps.

The interface

Before we create our service and controller, we need to write a small interface for our blog entries. This is because, as mentioned before, Mongoose schemas are not TypeScript classes, so in order to properly type the object to use it later, we will need to define a type for it first.

src/entries/entry.interface.ts

```
import { Document } from 'mongoose';

export interface Entry extends Document {
  readonly _id: string;
  readonly title: string;
  readonly body: string;
  readonly image: string;
```

```
  readonly created_at: Date;
}
```

Remember to keep your interface in sync with your schemas so that you don't run into issues with the shape of your objects later.

The service

Let's create a service for our blog entries that interact with the `Entry` model.

src/entries/entries.service.ts

```
import { Component } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model, Types } from 'mongoose';

import { EntrySchema } from './entry.schema';
import { Entry } from './entry.interface';

@Component()
export class EntriesService {
  constructor(
    @InjectModel(EntrySchema) private readonly entryModel: Model<Entry>
  ) {}

  // this method retrieves all entries
  findAll() {
    return this.entryModel.find().exec();
  }

  // this method retrieves only one entry, by entry ID
  findById(id: string) {
    return this.entryModel.findById(id).exec();
  }

  // this method saves an entry in the database
  create(entry) {
    entry._id = new Types.ObjectId();
    const createdEntry = new this.entryModel(entry);
    return createdEntry.save();
  }
```

```
}
```

In the code above, the most important bit happens inside the constructor: we are using the `@InjectModel()` decorator to instantiate our model, by passing the desired schema (in this case, `EntrySchema`) as a decorator argument.

Then, in that same line of code, we are injecting the model as a dependency in the service, naming it as `entryModel` and assigning a `Model` type to it; from this point on, we can take advantage of all the goodies that Mongoose models offer to manipulate documents in an abstract, simplified way.

On the other hand, it's worth mentioning that, in the `create()` method, we are adding an ID to the received entry object by using the `_id` property (as we previously defined on our schema) and generating a value using Mongoose's built-in `Types.ObjectId()` method.

The controller

The last step we need to cover in the model -> service -> controller chain is the controller. The controller will make it possible to make an API request to the Nest.js app and either write to or read from the database.

This is how our controller should look like:

src/entries/entries.controller.ts

```
import { Controller, Get, Post, Body, Param } from '@nestjs/common';

import { EntriesService } from './entry.service';

@Controller('entries')
export class EntriesController {
  constructor(private readonly entriesSrv: EntriesService) {}

  @Get()
  findAll() {
    return this.entriesSrv.findAll();
  }

  @Get(':entryId')
  findById(@Param('entryId') entryId) {
    return this.entriesSrv.findById(entryId);
  }
```

```
  @Post()
  create(@Body() entry) {
    return this.entriesSrv.create(entry);
  }
}
```

As usual, we are using Nest.js Dependency Injection to make the `EntryService` available in our `EntryController`. Then we route the three basic requests we are expecting to listen to (`GET` all entries, `GET` one entry by ID and `POST` a new entry) to the corresponding method in our service.

The first requests

At this point, our Nest.js API is ready to listen to requests (both `GET` and `POST`) and operate on the data stored in our MongoDB instance based on those requests. In other words, we are ready to read from and write to our database from the API.

Let's give it a try.

We will start with a GET request to the `/entries` endpoint. Obviously, since we haven't created any entries yet, we should receive an empty array as a response.

```
> GET /entries HTTP/1.1
> Host: localhost:3000
< HTTP/1.1 200 OK


[]
```

Let's create a new entry by sending a `POST` request to the `entries` endpoint and including in the request body a JSON object that matches the shape of our previously defined `EntrySchema`.

```
> GET /entries HTTP/1.1
> Host: localhost:3000
| {
|   "title": "This is our first post",
```

```
|   "body": "Bla bla bla bla bla",
|   "image": "http://lorempixel.com/400",
|   "created_at": "2018-04-15T17:42:13.911Z"
| }


< HTTP/1.1 201 Created
```

Yes! Our previous `POST` request triggered a write in the database. Let's try to retrieve all entries once again.

```
> GET /entries HTTP/1.1
> Host: localhost:3000
< HTTP/1.1 200 OK


[{
  "id": 1,
  "title": "This is our first post",
  "body": "Bla bla bla bla bla",
  "image": "http://lorempixel.com/400",
  "created_at": "2018-04-15T17:42:13.911Z"
}]
```

We just confirmed that requests to our `/entries` endpoint successfully execute reads and writes in our database. This means that our Nest.js app is usable now, since the basic functionality of almost any server application (that is, storing data and retrieving it on demand) is working properly.

Relationships

While it's true that MongoDB is not a relational database, it's also true that it allows "join-like" operations for retrieving two (or more) related documents at once.

Fortunately for us, Mongoose includes a layer of abstraction for these operations and allows us to set up relationships between objects in a clear, concise way. This is provided by using `ref`s in schemas' properties, as well as the `.populate()` method (that triggers something known as the "population" process; more on it later.)

Modelling relationships

Let's go back to our blog example. Remember that so far we only had a schema that defined our blog entries. We will create a second schema that will allow us to create comments for each blog entry, and save them to the database in a way that allows us later to retrieve both a blog entry as well as the comments that belong to it, all in a single database operation.

So, first, we create a `CommentSchema` like the following one:

src/comments/comment.schema.ts

```
import * as mongoose from 'mongoose';

export const CommentSchema = new mongoose.Schema({
   _id: Schema.Types.ObjectId,
   body: String,
   created_at: Date,
   entry: { type: Schema.Types.ObjectId, ref: 'Entry' },
});
```

This schema is, at this point, a "stripped-down version" of our previous `EntrySchema`. It's actually dictated by the intended functionality, so we shouldn't care too much about that fact.

Again, we are relying on the `_id` named property as a naming convention.

One notable new thing is the `entry` property. It will be used to store a reference to the entry each comment belongs to. The `ref` option is what tells Mongoose which model to use during population, which in our case is the `Entry` model. All `_id`'s we store here need to be document `_id`'s from the Entry model.

NOTE: We will ignore the `Comment` interface for brevity; it's simple enough for you to be able to complete it on your own. Don't forget to do it!

Second, we need to update our original `EntrySchema` in order to allow us to save references to the `Comment` instances that belong to each entry. See the following example on how to do it:

src/entries/entry.schema.ts

```
import * as mongoose from 'mongoose';

export const EntrySchema = new mongoose.Schema({
  _id: Schema.Types.ObjectId,
  title: String,
  body: String,
  image: String,
  created_at: Date,
  comments: [{ type: Schema.Types.ObjectId, ref: 'Comment' }],
});
```

Note that the `comments` property that we just added is *an array of objects*, each of which have an ObjectId as well as a reference. The key here is including an *array* of related objects, as this array enables what we could call a "one-to-many" relationship, if we were in the context of a relational database.

In other words, each entry can have multiple comments, but each comment can only belong to one entry.

Saving relationships

Once our relationship is modelled, we need to provide a method for saving them into our MongoDB instance.

When working with Mongoose, storing a model instance and its related instances requires some degree of manually nesting methods. Fortunately, `async/await` will make the task much easier.

Let's modify our `EntryService` to save both the receive blog entry and a comment associated with it; both will be sent to the `POST` endpoint as different objects.

src/entries/entries.service.ts

```
import { Component } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model, Types } from 'mongoose';
```

```
import { EntrySchema } from './entry.schema';
import { Entry } from './entry.interface';

import { CommentSchema } from './comment.schema';
import { Comment } from './comment.interface';

@Component()
export class EntriesService {
  constructor(
    @InjectModel(EntrySchema) private readonly entryModel:
Model<Entry>,
    @InjectModel(CommentSchema) private readonly commentModel:
Model<Comment>
  ) {}

  // this method retrieves all entries
  findAll() {
    return this.entryModel.find().exec();
  }

  // this method retrieves only one entry, by entry ID
  findById(id: string) {
    return this.entryModel.findById(id).exec();
  }

  // this method saves an entry and a related comment in the database
  async create(input) {
    const { entry, comment } = input;

    // let's first take care of the entry (the owner of the
relationship)
    entry._id = new Types.ObjectId();
    const entryToSave = new this.entryModel(entry);
    await entryToSave.save();

    // now we are ready to handle the comment
    // this is how we store in the comment the reference
    // to the entry it belongs to
    comment.entry = entryToSave._id;
```

```
    comment._id = new Types.ObjectId();
    const commentToSave = new this.commentModel(comment);
    commentToSave.save();

    return { success: true };
  }
}
```

The modified `create()` method is now:

1. Assigning an ID to the entry.
2. Saving the entry while assigning it to a `const`.
3. Assigning an ID to the comment.
4. Using the ID of the entry we created before as value of the `entry` property of the comment. *This is the reference we mentioned before.*
5. Saving the comment.
6. Returning a success status message.

This way we make sure that, inside the comment, we are successfully storing a reference to the entry the comment belongs to. By the way, note that we store the reference by entry's ID.

The next step should obviously be providing a way of reading from the database the related items we now are able to save to it.

Reading relationships

As covered a few sections ago, the method that Mongoose provides for retrieving related documents from a database at once is called "population," and it's invoked with the built-in `.populate()` method.

We'll see how to use this method by changing the `EntryService` once again; at this point, we will deal with the `findById()` method.

src/entries/entries.service.ts

```
import { Component } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model, Types } from 'mongoose';

import { EntrySchema } from './entry.schema';
```

```
import { Entry } from './entry.interface';

import { CommentSchema } from './comment.schema';
import { Comment } from './comment.interface';

@Component()
export class EntriesService {
  constructor(
    @InjectModel(EntrySchema) private readonly entryModel:
Model<Entry>,
    @InjectModel(CommentSchema) private readonly commentModel:
Model<Comment>
  ) {}

  // this method retrieves all entries
  findAll() {
    return this.entryModel.find().exec();
  }

  // this method retrieves only one entry, by entry ID,
  // including its related documents with the "comments" reference
  findById(id: string) {
    return this.entryModel
      .findById(id)
      .populate('comments')
      .exec();
  }

  // this method saves an entry and a related comment in the database
  async create(input) {
    ...
  }
}
```

The `.populate('comments')` method that we just included will transform the `comments` property value from an array of IDs to an array of actual documents that correspond with those IDs. In other words, their ID value is replaced with the Mongoose document returned from the database by performing a separate query before returning the results.

Summary

NoSQL databases are a powerful alternative to “traditional” relational ones. MongoDB is arguably the best known of the NoSQL databases in use today, and it works with documents encoded in a JSON variant. Using a document-based database such as MongoDB allows developers to use more flexible, loosely-structured data models and can improve iteration time in a fast-moving project.

The well known Mongoose library is an adaptor for MongoDB that works in Node.js and that abstracts quite a lot of complexity when it comes to querying and saving operations.

Over this chapter we’ve covered quite some aspects of working with Mongoose and Nest.js, like:

- How to start up a local MongoDB instance with Docker Compose.
- How to import the @nestjs/mongoose module in our root module and connect to our MongoDb instance.
- What are schemas and how to create one for modelling our data.
- Setting up a pipeline that allows us to write to and read from our MongoDB database as a reaction of requests made to our Nest.js endpoints.
- How to establish relationships between different types of MongoDB documents and how to store and retrieve those relationships in an effective way.

In the next chapter we cover web sockets.

Chapter 8. Web sockets

As you have seen, Nest.js provides a way to use web sockets into your app through the `@nestjs/websockets` package. Also, inside the framework the usage of the `Adapter` allows you to implement the socket library that you want. By default, Nest.js comes with it's own adapter, which allows you to use `socket.io`, a well known library for web sockets.

You have the possibility to create a full web socket app, but also, add some web socket features inside your Rest API. In this chapter, we will see how to implement the web socket over a Rest API using the decorators provided by Nest.js, but also how to validate an authenticated user using specific middleware.

The advantage of the web socket is to be able to have some real-time features in an application depending on your needs. For this chapter you can have a look at the `/src/gateways` files from the repository, but also `/src/shared/adapters` and `/src/middlewares`.

Imagine the following `CommentGatewayModule`, which looks like this:

```
@Module({
    imports: [UserModule, CommentModule],
    providers: [CommentGateway]
})
export class CommentGatewayModule { }
```

Import the `UserModule` in order to have access to the `UserService`, which will be useful later, as well as the `CommentModule`. Of course, we will create the `CommentGateway`, which is used as an injectable service.

WebSocketGateway

To implement your first module using the Nest.js web socket, you will have to use the `@WebSocketGateway` decorator. This decorator can take an argument as an object to provide a way to configure how to use the adapter.

The implementation of the arguments respect the interface `GatewayMetadata`, allowing you to provide:

- `port`, which must be use by the adapter
- `namespace`, which belongs to the handlers

- `middlewares` that have to be applied on the gateway before accessing the handlers

All the parameters are optional.

To use it, you have to create you first gateway class, so imagine a `UserGateway`:

```
@WebSocketGateway({
    middlewares: [AuthenticationGatewayMiddleware]
})
export class UserGateway { /*....*/ }
```

By default, without any parameters, the socket will use the same port as your express server (generally `3000`). As you can see, in the previous example we used a `@WebSocketGateway`, which uses the default port `3000` without namespace and with one middleware that we will see later.

Gateways

The gateways in the class using the decorator previously seen contain all of the handlers that you need to provide the results of an event.

Nest.js comes with a decorator that allows you to access the server instance `@WebSocketServer`. You have to use it on a property of your class.

```
export class CommentGateway {
    @WebSocketServer() server;

    /* ... */
}
```

Also, throughout the gateway, you have access to the injection of injectable services. So, in order to have access of the comment data, inject the `CommentService` exported by the `CommentModule`, which has been injected into this module.

```
export class CommentGateway {
    /* ... */

    constructor(private readonly commentService: CommentService) { }

    /* ... */
```

```
}
```

The comment service allows you to return the appropriate result for the next handlers.

```
export class CommentGateway {
     /* ... */

     @SubscribeMessage('indexComment')
     async index(client, data): Promise<WsResponse<any>> {
          if (!data.entryId) throw new WsException('Missing entry id.');

          const comments = await this.commentService.findAll({
               where: {entryId: data.entryId}
          });

          return { event: 'indexComment', data: comments };
     }

     @SubscribeMessage('showComment')
     async show(client, data): Promise<WsResponse<any>> {
          if (!data.entryId) throw new WsException('Missing entry id.');
          if (!data.commentId) throw new WsException('Missing comment
id.');

          const comment = await this.commentService.findOne({
               where: {
                    id: data.commentId,
                    entryId: data.entryId
               }
          });

          return { event: 'showComment', data: comment };
     }
}
```

We now have two handlers, the `indexComment` and the `showComment`. To use the `indexComment` handler we expect an `entryId` in order to provide the appropriate comment, and for the `showComment` we expect an `entryId`, and of course a `commentId`.

As you have seen, to create the event handler use the `@SubscribeMessage` decorator provide by the framework. This decorator will create the `socket.on(event)` with the event corresponding to the string passed as a parameter.

Authentication

We have set up our `CommentModule`, and now we want to authenticate the user using the token (have a look to the Authentication chapter). In this example we use a mutualised server for the REST API and the Websocket event handlers. So, we will mutualise the authentication token in order to see how to validate the token received after a user has been logged into the application.

It is important to secure the websocket in order to avoid the access of data without logging into the application.

As shown in the previous part, we have used middleware named `AuthenticationGatewayMiddleware`. The purpose of this middleware is to get the token from the web socket `query`, which is brought with the `auth_token` property.

If the token is not provided, the middleware will return a `WsException`, otherwise we will use the `jsonwebtoken` library (have a look to the Authentication chapter) to verify the token.

Let's set up the middleware:

```
@Injectable()
export class AuthenticationGatewayMiddleware implements
GatewayMiddleware {
     constructor(private readonly userService: UserService) { }
     resolve() {
          return (socket, next) => {
               if (!socket.handshake.query.auth_token) {
                    throw new WsException('Missing token.');
               }

               return jwt.verify(socket.handshake.query.auth_token,
'secret', async (err, payload) => {
                    if (err) throw new WsException(err);
```

```
                    const user = await this.userService.findOne({ where:
{ email: payload.email }});
                    socket.handshake.user = user;
                    return next();
                });
            }
        }
}
```

The middleware used for the web socket is almost the same as the REST API. Implementing the `GatewayMiddleware` interface with the `resolve` function is now nearly the same. The difference is that you have to return a function, which takes `socket` and the `next` function as its parameters. The socket contains the `handshake` with the `query` sent by the client, and all of the parameters provided, in our case, the `auth_token`.

Similar to the classic authentication middleware (have a look to the Authentication chapter), the socket will try to find the user with the given payload, which here contains the email, and then register the user into the handshake in order to be accessible into the gateway handler. This is a flexible way to already have the user in hand without finding it again in the database.

Adapter

As mentioned in the beginning of this chapter, Nest.js comes with it own adapter, which uses `socket.io`. But the framework needs to be flexible and it can be used with any third party library. In order to provide a way to implement another library, you have the possibility to create your own adapter.

The adapter has to implement the `WebSocketAdapter` interface in order to implement the following methods. For example, we will use `ws` as a socket library in our new adapter. To use it, we will have to inject the `app` into the constructor as follows:

```
export class WsAdapter implements WebSocketAdapter {
    constructor(private app: INestApplication) { }

    /* ... */
}
```

By doing this, we can get the `httpServer` in order to use it with the `ws`. After that, we have to implement the `create` method in order to create the socket server.

```
export class WsAdapter implements WebSocketAdapter {
     /* ... */

     create(port: number) {
          return new WebSocket.Server({
               server: this.app.getHttpServer(),
               verifyClient: ({ origin, secure, req }, next) => {
                    return (new
WsAuthenticationGatewayMiddleware(this.app.select(UserModule).
                    get(UserService))).resolve()(req, next);
               }
          });
     }

     /* ... */
}
```

As you can see, we have implemented the `verifyClient` property, which takes a method with `{ origin, secure, req }` and `next` values. We will use the `req`, which is the `IncomingMessage` from the client and the `next` method in order to continue the process. We use the `WsAuthenticationGatewayMiddleware` to authenticate the client with the token, and to inject the appropriate dependencies we select the right module and the right service.

The middleware in this case processes the authentication:

```
@Injectable()
export class WsAuthenticationGatewayMiddleware implements
GatewayMiddleware {
     constructor(private userService: UserService) { }
     resolve() {
          return (req, next) => {
               const matches = req.url.match(/token=([^&].*)/);
               req['token'] = matches && matches[1];

               if (!req.token) {
                    throw new WsException('Missing token.');
               }
```

```
                return jwt.verify(req.token, 'secret', async (err, payload)
=> {
                    if (err) throw new WsException(err);

                    const user = await this.userService.findOne({ where:
{ email: payload.email }});
                    req.user = user;
                    return next(true);
                });
            }
        }
}
```

In this middleware, we have to manually parse the URL to get the token and verify it with `jsonwebtoken`. After that, we have to implement the `bindClientConnect` method to bind the connection event to a callback that will be used by Nest.js. It is a simple method, which takes in arguments to the server with a callback method.

```
export class WsAdapter implements WebSocketAdapter {
    /* ... */

    bindClientConnect(server, callback: (...args: any[]) => void) {
        server.on('connection', callback);
    }

    /* ... */
}
```

To finish our new custom adapter, implement the `bindMessageHandlers` in order to redirect the event and the data to the appropriate handler of your gateway. This method will use the `bindMessageHandler`in order to execute the handler and return the result to the `bindMessageHandlers` method, which will return the result to the client.

```
export class WsAdapter implements WebSocketAdapter {
    /* ... */

        bindMessageHandlers(client: WebSocket, handlers:
MessageMappingProperties[], process: (data) => Observable<any>) {
            Observable.fromEvent(client, 'message')
```

```
                    .switchMap((buffer) => this.bindMessageHandler(buffer,
handlers, process))
                    .filter((result) => !!result)
                    .subscribe((response) =>
client.send(JSON.stringify(response)));
          }

          bindMessageHandler(buffer, handlers: MessageMappingProperties[],
process: (data) => Observable<any>): Observable<any> {
               const data = JSON.parse(buffer.data);
               const messageHandler = handlers.find((handler) =>
handler.message === data.type);
               if (!messageHandler) {
                    return Observable.empty();
               }
               const { callback } = messageHandler;
               return process(callback(data));
          }

     /* ... */
}
```

Now, we have created our first custom adapter. In order to to use it, instead of the Nest.js `IoAdapter`, we have to call the `useWebSocketAdapter` provided by the `app: INestApplication` in your `main.ts` file as follows:

```
app.useWebSocketAdapter(new WsAdapter(app));
```

We pass in the adapter, the `app` instance, to be used as we have seen in the previous examples.

Client side

In the previous section, we covered how to set up the web socket on the server side and how to handle the event from the client side.

Now we will see how to set up your client side, in order to use the Nest.js `IoAdapter` or our custom `WsAdapter`. In order to use the `IoAdapter`, we have to get the `socket.io-client` library and set up our first HTML file.

The file will define a simple script to connect the socket to the server with the token of the logged in user. This token we will be used to determine if the user is well connected or not.

Check out the following code:

```
<script>
      const socket = io('http://localhost:3000',  {
            query: 'auth_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
          eyJlbWFpbCI6InRlc3QzQHRlc3QuZnIiLCJpYXQiOjE1MjQ5NDk3NTgs
          ImV4cCI6MTUyNDk1MzM1OH0.QH_jhOWKockuV-w-vIKMgT_eLJb3dp6a
          ByDbMvEY5xc'
      });
</script>
```

As you see, we pass at the socket connection a token `auth_token` into the query parameter. We can pick it from the socket handshake and then validate the socket.

To emit an event, which is also easy, see the following example:

```
socket.on('connect', function () {
      socket.emit('showUser', { userId: 4 });
      socket.emit('indexComment', { entryId: 2 });
      socket.emit('showComment', { entryId: 2, commentId: 1 });
});
```

In this example, we are waiting for the `connect` event to be aware when the connection is finished. Then we send three events: one to get the user, then an entry, and the comment of the entry.

Using the following on event, we are able to get the data sent by the server as a response to our previously emitted events.

```
socket.on('indexComment', function (data) {
      console.log('indexComment', data);
});
socket.on('showComment', function (data) {
      console.log('showComment', data);
});
socket.on('showUser', function (data) {
      console.log('showUser', data);
});
```

```
socket.on('exception', function (data) {
      console.log('exception', data);
});
```

Here we show in the console all of the data responded by the server, and we have also implemented an event `exception` in order to catch all exceptions that the server can return.

Of course, as we have seen in the authentication chapter, the user can't access the data from another user.

In cases where we would like to use the custom adapter, the process is similar. We will open the connection to the server using the `WebSocket` as follows:

```
const ws = new WebSocket("ws://localhost:3000?token=eyJhbGciO
iJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InRlc3QzQHRlc3QuZnIiL
CJpYXQiOjE1MjUwMDc2NjksImV4cCI6MTUyNTAxMTI2OX0.GQjWzdKXAFTAtO
kpLjId7tPliIpKy5Ru50evMzf15YE");
```

We open the connection here on the localhost with the same port as our HTTP server. We also pass the token as a query parameter in order to pass the `verifyClient` method, which we have seen with the `WsAuthenticationGatewayMiddleware`.

Then we will wait for the return of the server, to be sure that the connection is successful and usable.

```
ws.onopen = function() {
      console.log('open');
      ws.send(JSON.stringify({ type: 'showComment', entryId: 2, commentId:
1 }));
};
```

When the connection is usable, use the `send` method in order to send the type of event we want to handle, which is here with `showComment` and where we pass the appropriate parameters, just like we did with the socket.io usage.

We will use the `onmessage` in order to get the data returned by the server for our previous sent event. When the `WebSocket` receives an event, a `message` event is sent to the manager that we can catch with the following example.

```
ws.onmessage = function(ev) {
      const _data = JSON.parse(ev.data);
```

```
        console.log(_data);
};
```

You can now use this data as you'd like in the rest of your client app.

Summary

In this chapter you learned how to set up the server side, in order to use the:

- `socket.io` library provided by the Nest.js `IoAdapter`
- `ws` library with a custom adapter

You also set up a gateway in order to handle the events sent by the client side.

You've seen how to set up the client side to use the `socket.io-client` or `WebSocket` client to connect the socket to the server. This was done on the same port as the HTTP server, and you learned how to send and catch the data returned by the server or the exception in case of an error.

And finally, you learned how to set up the authentication using middleware in order to check the socket token provided and identify if the user is authenticated or not to be able to access the handler in the cases of an `IoAdapter` or a custom adapter.

The next chapter will cover microservices with Nest.js.

Chapter 9. Microservices

Using Nest.js microservices, we are able to extract out part of our application's business logic and execute it within a separate Nest.js context. By default, this new Nest.js context does not execute in a new thread or even a new process. For this reason, the name "microservices" is a bit misleading. In fact, if you stick with the default TCP transport, users may find that requests take longer to complete. However, there are benefits to offloading some pieces of your application to this new microservice context. To cover the basics, we will stick with the TCP transport, but look for some real-world strategies where Nest.js microservices can improve application performance in the Advanced Architecture section of this chapter. To see a working example, remember you can clone the accompanying Git repository for this book:

```
git clone https://github.com/backstopmedia/nest-book-example.git
```

Server bootstrap

To get started, make sure `@nestjs/microservices` is installed within your project. This module provides the client, server, and accompanying utilities needed to convert a Nest.js API application into a microservices application. Finally, we will modify our blog application's bootstrap to enable microservices.

```
async function bootstrap() {
      const app = await NestFactory.create(AppModule);
      app.connectMicroservice({
            transport: Transport.TCP,
            options: {
                  port: 5667
            }
      });

      await app.startAllMicroservicesAsync();
      await app.listen(3001);
}
```

The `connectMicroservice` method instructs the NestApplication to setup a new NestMicroservice context. The object provides the options for setting up the NestMicroservice context. Here, we are keeping things simple and using the standard TCP transport provided with Nest.js. A call

to `startAllMicroservicesAsync` starts up the NestMicroservice context. Be sure to do this before calling `listen` on the NestApplication.

Configuration

The configuration parameters passed to `connectMicroservice` depends on the transport we use. A transport is a combination of a client and server that work in unison to transmit microservice requests and responses between the NestApplication and NestMicroservice contexts. Nest.js comes with a number of built-in transports and provides the ability to create custom transports. The available parameters depend on the transport we use. For now, we will use the TCP transport, but will cover other transports later. The possible options for the TCP transport are:

- **host**: The host that is running the NestMicroservice context. The default is to assume `localhost` but this can be used if the NestMicroservice is running as a separate project on a different host such as a different Kubernetes pod.
- **port**: The port that the NestMicroservice context is listening on. The default is to assume `3000`, but we will use a different port to run our NestMicroservice context.
- **retryAttempts**: In the context of the TCP transport, this is the number of times the server will attempt to re-establish itself after it has received a `CLOSE` event.
- **retryDelay**: Works in conjunction with `retryAttempts` and delays the transports retry process by a set amount of time in milliseconds.

First microservice handler

For our first microservice handler, let's convert the UserController index method into a microservice handler. To do this, we copy the method and make a few simple modifications. Instead of annotating the method with `Get`, we will use `MessagePattern`.

```
@Controller()
export class UserController {

    @Get('users')
    public async index(@Res() res) {
```

```
        const users = await this.userService.findAll();
        return res.status(HttpStatus.OK).json(users);
    }

    @MessagePattern({cmd: 'users.index'})
    public async rpcIndex() {
        const users = await this.userService.findAll();
        return users;
    }
}
```

A message pattern provides Nest.js the means for determining which microservice handler to execute. The pattern can be a simple string or a complex object. When a new microservice message is sent, Nest.js will search through all registered microservice handlers to find the handler that matches the message pattern exactly.

The microservice method itself can perform the same business logic that a normal controller handler can to respond in almost the same manor. Unlike a normal controller handler, a microservice handler has no HTTP context. In fact, decorators like `@Get`, `@Body`, and `@Req` make no sense and should not be used in a microservice controller. To complete the processing of a message, a simple value, promise, or RxJS Observable can be returned from the handler.

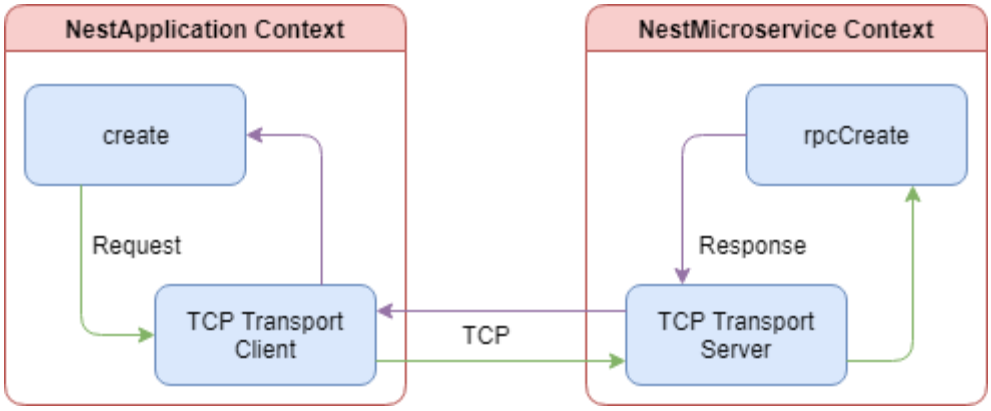
Sending data

The previous microservice handler was very contrived. It is more likely that microservice handlers will be implemented to perform some processing on data and return some value. In a normal HTTP handler, we would use `@Req` or `@Body` to extract the data from the HTTP request's body. Since microservice handlers have no HTTP context, they take input data as a method parameter.

```
@Controller()
export class UserController {
    @Client({transport: Transport.TCP, options: { port: 5667 }})
    client: ClientProxy

    @Post('users')
    public async create(@Req() req, @Res() res) {
        this.client.send({cmd: 'users.index'}, {}).subscribe({
```

```
                next: users => {
                    res.status(HttpStatus.OK).json(users);
                },
                error: error => {

res.status(HttpStatus.INTERNAL_SERVER_ERROR).json(error);
                }
            });
    }

    @MessagePattern({cmd: 'users.create'})
    public async rpcCreate(data: any) {
        if (!data || (data && Object.keys(data).length === 0)) throw new
Error('Missing some information.');

        await this.userService.create(data);
    }
}
```

In this example, we used the `@Client` decorator to provide Nest.js Dependency Injection a place to inject an instance of the microservice client. The client decorator takes the same configuration object passed to `connectMicroservice` when bootstrapping the application. The client is how the NestApplication context communicates with the NestMicroservice context. Using the client, we modified the original `@Post('users')` API to offload processing of creating a new user to NestMicroservice context.



This diagram shows a simplified view of the data flow when a new user is created. The client makes a TCP connection with the microservice context and offloads the processing of the database operations. The `rpcCreate` method will return a successful response with some data or an exception. While the

microservice message is being processed, the normal controller handler will wait for the response.

Take note that the microservice client `send` method returns an Observable. If you want to wait for the response from the microservice, simply subscribe to the Observable and use the response object to send the results. Alternatively, Nest.js treats Observables as first class citizens and can they can be returned from the handler. Nest.js will take care of subscribing to the Observable. Keep in mind, you lose a little control over the response status code and body. However, you can regain some of that control with exceptions and exception filters.

Exception filters

Exception filters provide a means to transform exceptions thrown from microservice handlers into meaningful objects. For example, our `rpcCreate` method currently throws an error with a string but what happens when the UserService throws an error or possibly the ORM. This method could throw a number of different errors and the only means that calling method knows what happened is to parse the error string. That's simply unacceptable, so let's fix it.

Start with creating a new exception class. Notice that our microservice exception extends RpcException and does not pass a HTTP status code in the constructor. These are the only differences between microservice exceptions and normal Nest.js API exceptions.

```
export class RpcValidationException extends RpcException {
     constructor(public readonly validationErrors: ValidationError[]) {
          super('Validation failed');
     }
}
```

We can now change the `rpcCreate` method to throw this exception when the data is not valid.

```
@MessagePattern({cmd: 'users.create'})
public async rpcCreate(data: any) {
     if (!data || (data && Object.keys(data).length === 0)) throw new
RpcValidationException();

     await this.userService.create(data);
```

```
}
```

Finally, create an exception filter. Microservice exception filters differ from their normal API counterparts by extending RpcExceptionFilter and returning an ErrorObservable. This filter will catch the RpcValidationException we created and throw an object containing a specific error code.

Note the `throwError` method is from the RxJS version 6 package. If you are still using RxJS version 5, use `Observable.throw`.

```
@Catch(RpcValidationException)
export class RpcValidationFilter implements RpcExceptionFilter {
     public catch(exception: RpcValidationException): ErrorObservable {
          return throwError({
               error_code: 'VALIDATION_FAILED',
               error_message: exception.getError(),
               errors: exception.validationErrors
          });
     }
}
```

All we have left is to act on our new exception when it occurs. Modify the `create` method to catch any exceptions thrown from the microservice client. In the catch, check if the `error_code` field has a value of `VALIDATION_FAILED`. When it does, we can return a `400` HTTP status code back to the user. This will allow the user's client, the browser, to treat the error differently, possibly showing the user some messaging and allowing them to fix the data entered. This provides a much better user experience compared to throwing all errors back to the client as `500` HTTP status code.

```
@Post('users')
public async create(@Req() req, @Res() res) {
     this.client.send({cmd: 'users.create'}, body).subscribe({
          next: () => {
               res.status(HttpStatus.CREATED).send();
          },
          error: error => {
               if (error.error_code === 'VALIDATION_FAILED') {
                    res.status(HttpStatus.BAD_REQUEST).send(error);
               } else {

res.status(HttpStatus.INTERNAL_SERVER_ERROR).send(error);
```

```
                }
            }
        });
    }
```

Pipes

The most common pipe used with and provided by Nest.js is the ValidationPipe. This pipe, however, cannot be used with microservice handlers because it throws exceptions extending HttpException. All exceptions thrown in a microservice must extend RpcException. To fix this, we can extend the ValidationPipe, catch the HttpException, and throw an RpcException.

```
@Injectable()
export class RpcValidationPipe extends ValidationPipe implements
PipeTransform<any> {
    public async transform(value: any, metadata: ArgumentMetadata) {
        try {
            await super.transform(value, metadata);
        } catch (error) {
            if (error instanceof BadRequestException) {
                throw new RpcValidationException();
            }

            throw error;
        }

        return value;
    }
}
```

Before we can use the ValidationPipe, we have to create a class that describes the format of the data our microservice method expects.

```
class CreateUserRequest {
    @IsEmail()
    @IsNotEmpty()
    @IsDefined()
    @IsString()
    public email: string;
```

```
    @Length(8)
    @Matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)\S+$/)
    @IsDefined()
    @IsString()
    public password: string;

    @IsNotEmpty()
    @IsDefined()
    @IsString()
    public firstName: string;

    @IsNotEmpty()
    @IsDefined()
    @IsString()
    public lastName: string;
}
```

The new request class uses the `class-validator` NPM package to validate the object being passed to our microservice method from the Nest.js microservice module. The class contains all the properties with specific decorators to describe what those properties should contain. For example, the `email` property should be an email address, cannot be empty, must be defined, and must be a string. Now we just need to hook it up to our `rpcCreate` method.

```
@MessagePattern({cmd: 'users.create'})
@UsePipes(new RpcValidationPipe())
@UseFilters(new RpcValidationFilter())
public async rpcCreate(data: CreateUserRequest) {
    await this.userService.create(data);
}
```

Since microservice handlers do not make use of the `@Body` decorator, we will need to use `@UsePipes` to make use of our new RpcValidationPipe. This will instruct Nest.js to validation the input data against it's class type. Just like you would for APIs, use validation classes and the RpcValidationPipe to offload input validation out of your controller or microservice method.

Guards

Guards in microservices serve the same purpose as they do in normal APIs. They determine if a specific microservice handler should handle a request. Up to now, we have used guards to protect API handlers from unauthorized access. We should do the same thing to our microservice handlers. Although in our application, our microservice handler is only called from our already protected API handler, we should never assume that will always be the case.

```
@Injectable()
export class RpcCheckLoggedInUserGuard implements CanActivate {
     canActivate(context: ExecutionContext): boolean | Promise<boolean>
| Observable<boolean> {
          const data = context.switchToRpc().getData();
          return Number(data.userId) === data.user.id;
     }
}
```

The new guard looks exactly like the API `CheckLoggedInUserGuard` guard. The difference is in the parameters that are passed to the `canActivate` method. Since this guard is being executed in the context of a microservice, it will be given a microservice `data` object instead of the API request object.

We use the new microservice guard the same way we did our API guard. Simply decorate out microservice handler with `@UseGuards` and our guard will now protect our microservice from misuse. Let's make a new microservice for retrieving the current user's information.

```
@Get('users/:userId')
@UseGuards(CheckLoggedInUserGuard)
public async show(@Param('userId') userId: number, @Req() req, @Res() res)
{
     this.client.send({cmd: 'users.show'}, {userId, user:
req.user}).subscribe({
          next: user => {
               res.status(HttpStatus.OK).json(user);
          },
          error: error => {
               res.status(HttpStatus.INTERNAL_SERVER_ERROR).send(error);
          }
     });
}

@MessagePattern({cmd: 'users.show'})
```

```
@UseGuards(RpcCheckLoggedInUserGuard)
public async rpcShow(data: any) {
     return await this.userService.findById(data.userId);
}
```

The `show` API handler now offloads the heavy lifting of accessing the database to the NestMicroservice context. The guard on the microservice handler ensures, if the handler is somehow invoked outside of the `show` API handler, it will still protect the user data from being exposed to unauthorized requests. But there is still a problem. This example returns the entire user object from the database, including the hashed password. This is a security vulnerability best solved by interceptors.

Interceptors

Microservice interceptors function no differently from normal API interceptors. The only difference is that the interceptor is passed the data object sent to the microservice handler instead of an API request object. This means you can actually write interceptors once and use them in both contexts. Just like API interceptors, microservice interceptors are executed before the microservice handler and must return an Observable. To secure our `rpcShow` microservice endpoint, we will create a new interceptor that will expect a `User` database object and remove the `password` field.

```
@Injectable()
export class CleanUserInterceptor implements NestInterceptor {
     intercept(context: ExecutionContext, stream$: Observable<any>):
Observable<any> {
          return stream$.pipe(
               map(user => JSON.parse(JSON.stringify(user))),
               map(user => {
                    return {
                         ...user,
                         password: undefined
                    };
               })
          );
     }
}
@MessagePattern({cmd: 'users.show'})
@UseGuards(RpcCheckLoggedInUserGuard)
```

```
@UseInterceptors(CleanUserInterceptor)
public async rpcShow(data: any) {
    return await this.userService.findById(data.userId);
}
```

The response from the `rpcShow` microservice handler will now have the `password` field removed. Notice in the interceptor we had to convert the `User` database object to and from JSON. This may differ depending on the ORM you make use of. With Sequelize, we need to get the raw data from the database response. This is because the response from the ORM is actually a class containing many different ORM methods and properties. By converting it to JSON and back, we can use the spread operator with `password: undefined` to delete the `password` field.

Built-in transports

The TCP transport is only one of several transports Nest.js has built-in. Using the TCP transport, we had to bind our NestMicroservice context to an additional port, taking up yet another port on the server, and ensuring our NestMicroservice context was running before starting the NestApplication context. Other built-in transports can overcome these limitations and add additional benefits.

Redis

Redis is a simple in-memory data store that can be used as a pub-sub message broker. The Redis transport makes use of the redis NPM package and a Redis server to pass messages between the NestApplication and NestMicroservice contexts. To use the Redis transport, we need to update our `bootstrap`method to use the correct NestMicroservice configuration.

```
async function bootstrap() {
    const app = await NestFactory.create(AppModule);
    app.connectMicroservice({
        transport: Transport.REDIS,
        options: {
            url: process.env.REDIS_URL
        }
    });

    await app.startAllMicroservicesAsync();
```

```
    await app.listen(3001);
}
```

You would also have to update all locations where you have use the `@Client` decorator to the same settings. Instead, let's centralize this configuration so we are not duplicating code and we can switch out the transport easier.

```
export const microserviceConfig: RedisOptions = {
    transport: Transport.REDIS,
    options: {
        url: process.env.REDIS_URL
    }
};
```

The Redis transport can take the following options:

- **url**: The url of the Redis server. The default is `redis://localhost:6379`.
- **retryAttempts**: The number of times the microservice server and client will attempt to reconnect to the Redis server when the connection is lost. This is used to create a `retry_strategy` for the `redis`NPM package.
- **retryDelay**: Works in conjunction with `retryAttempts` and delays the transports retry process by a set amount of time in milliseconds.

Now we can update the applications `bootstrap` to use the `microserviceConfig` object we have created.

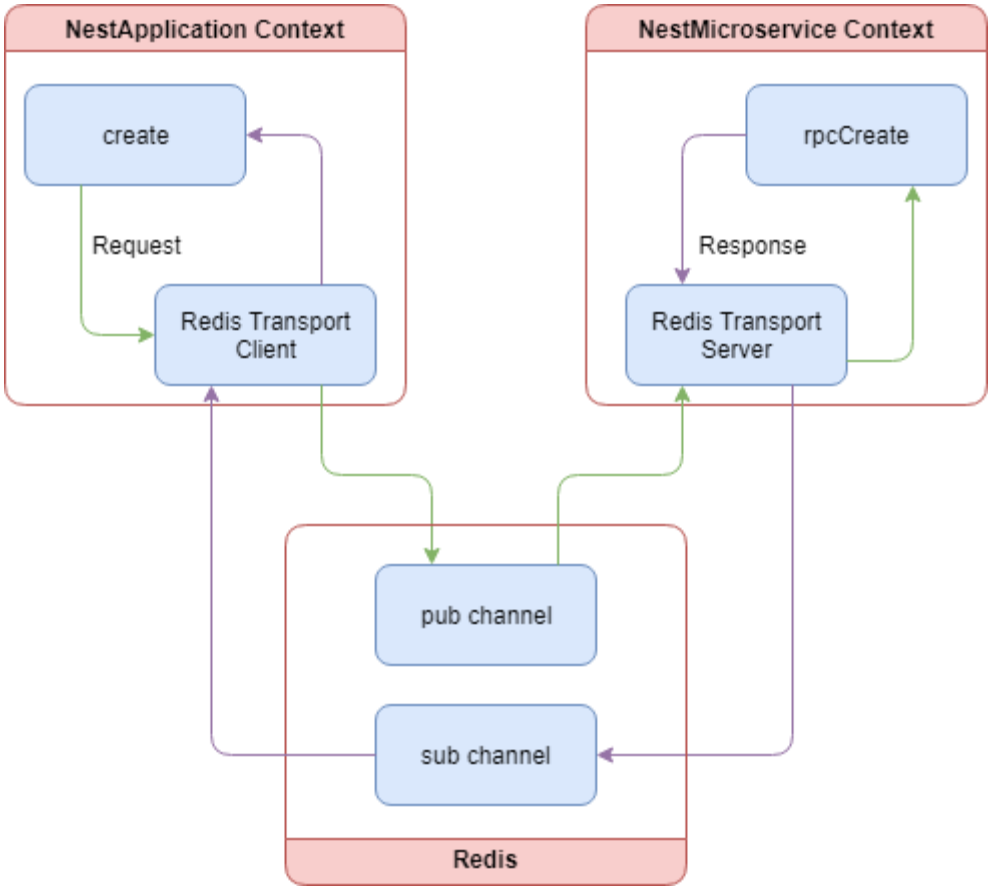
```
async function bootstrap() {
    const app = await NestFactory.create(AppModule);
    app.connectMicroservice(microserviceConfig);

    await app.startAllMicroservicesAsync();
    await app.listen(3001);
}
```

Finally, update the `@Client` decorator in the UserController.

```
@Controller()
export class UserController {
```

```
    @Client(microserviceConfig)
    client: ClientProxy
}
```

Start up a Redis server, such as the redis docker image and the application and all our microservice transaction will now be processing through the Redis server. The below diagram shows a simplified view of the data flow when a new user is created and we are using the Redis transport.



Both the client and the server make a connection with the Redis server. When `client.send` is called, the client alters the message pattern on the fly to create pub and sub channels. The server consumes the message and removes the message pattern modification to find the correct microservice handler. Once processing is complete in the microservice handler, the pattern is modified again to match the sub channel. The client consumes this new message, unsubscribes from the sub channel, and passes the response back to the caller.

MQTT

MQTT is a simple message protocol designed to be used when network bandwidth is a premium. The MQTT transport make use of the mqtt NPM

package and a remote MQTT server to pass messages between the NestApplication and NestMicroservice contexts. The data flow and how the microservice client and server operate are almost identical to the Redis transport. To make use of the MQTT transport, let's update the microserviceConfig configuration object.

```
export const microserviceConfig: MqttOptions = {
     transport: Transport.MQTT,
     options: {
          url: process.env.MQTT_URL
     }
};
```

The MQTT transport can take several options, all of which are detailed in the Github repository for the `mqtt` NPM package. Most notably, the transport defaults the `url` option to `mqtt://localhost:1883` and there is no connection retrying. If the connection to the MQTT server is lost, microservice messages will no longer be passed.

Start up a MQTT server, such as the eclipse-mosquitto docker image, and the application and all our microservice transaction will now be processing through the MQTT server.

NATS

NATS is an open source message broker server that touts extremely high throughput. The NATS transport make use of the nats NPM package and a remote NATS server to pass messages between the NestApplication and NestMicroservice contexts.

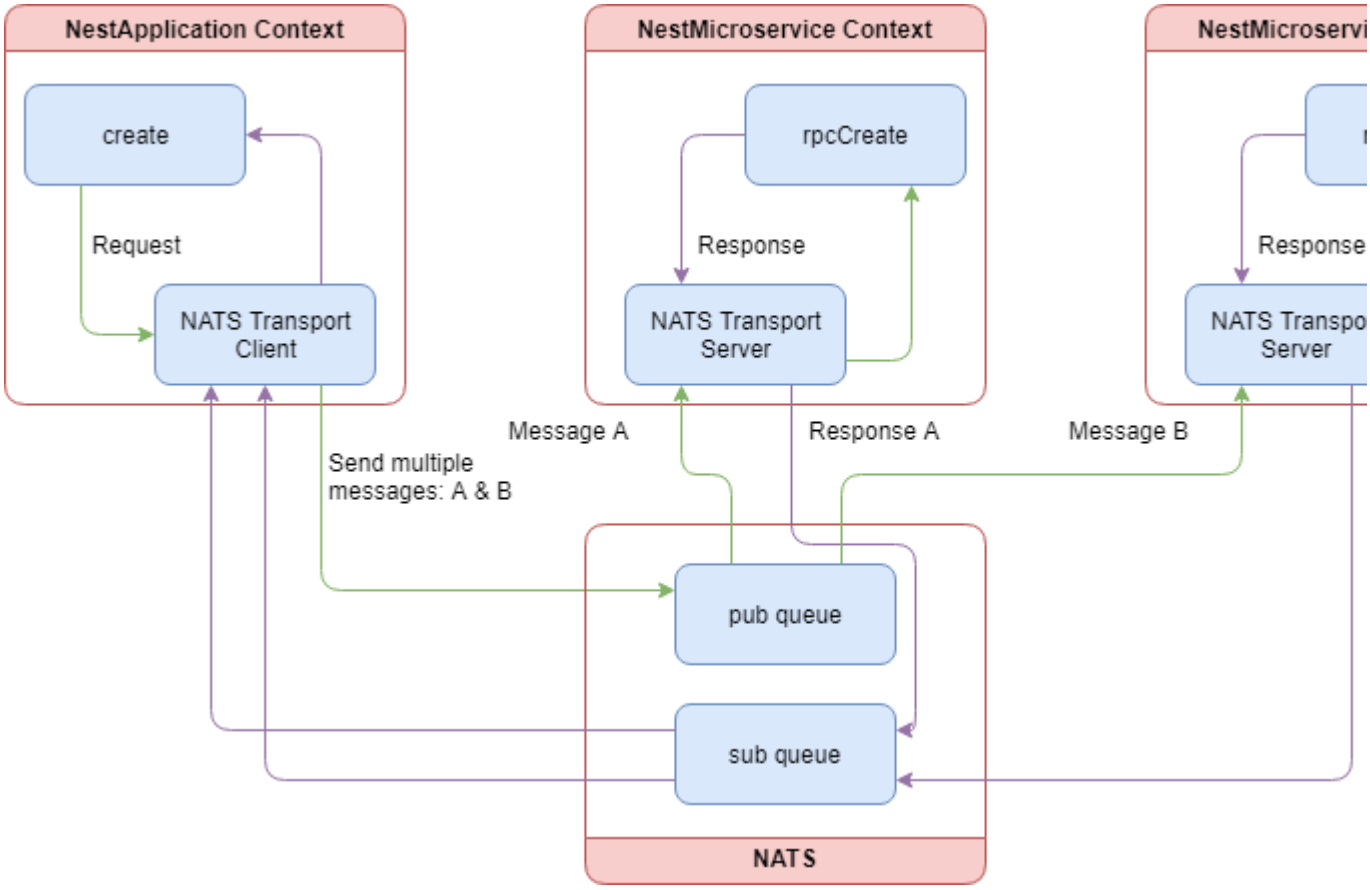
```
export const microserviceConfig: MqttOptions = {
     transport: Transport.NATS,
     options: {
          url: process.env.NATS_URL
     }
};
```

The NATS transport can take the following options:

- **url**: The url of the NATS server. The default is `nats://localhost:4222`.

- **name/pass**: The username and password used to authenticate the Nest.js application with the NATS server.
- **maxReconnectAttempts**: The number of times the server and client will attempt to reconnect to the NATS server when the connection is lost. The default is to attempt to reconnect 10 times.
- **reconnectTimeWait**: Works in conjunction with `maxReconnectAttempts` and delays the transports retry process by a set amount of time in milliseconds.
- **servers**: An array of `url` strings all of which are NATS servers. This allows the transport to take advantage of a cluster of NATS servers.
- **tls**: A boolean indicating if TLS should be used when connecting to the NATS server. **Note** this defaults to false meaning all messages are passed in clear text. An object can also be provided instead of a boolean, and can contain the standard Node TLS settings like the client certificate.

Start up a NATS server, such as the nats docker image, and the application and all our microservice transaction will now be processing through the NATS server. The below diagram shows a simplified view of the data flow when a new user is created and we are using the NATS transport.



Both the client and the server make a connection with the NATS server. When `client.send` is called, the client alters the message pattern on the fly to create pub and sub queues. One of the most notable differences between the Redis transport and the NATS transport is that the NATS transport makes use of queue groups. This means we can now have multiple NestMicroservice contexts and the NATS server will load balance messages between them. The server consumes the message and removes the message pattern modification to find the correct microservice handler. Once processing is complete in the microservice handler, the pattern is modified again to match the sub channel. The client consumes this new message, unsubscribes from the sub channel, and passes the response back to the caller.

gRPC

gRPC is a remote procedural call client and server designed to be used with Googles Protocol Buffers. gRPC and protocol buffers are extensive subjects worthy of their own books. For that reason, we will stick to discussing the setup and use of gRPC within a Nest.js application. To get started, we will need the grpc NPM package. Before we can write any code for our Nest.js application, we must write a Protocol Buffer file.

```
syntax = "proto3";

package example.nestBook;

message User {
    string firstName = 1;
    string lastName = 2;
    string email = 3;
}

message ShowUserRequest {
    double userId = 1;
}

message ShowUserResponse {
    User user = 1;
}

service UserService {
    rpc show (ShowUserRequest) returns (ShowUserResponse);
}
```

The above code snippet describes a single gRPC service called `UserService`. This will typically map to a service or controller within your own project. The service contains a single method, `show`, that takes in an object with a `userId` and returns an object with a `user` property. The `syntax` value indicates to the gRPC package which format of the protocol buffers language we are using.
The `package` declaration acts as a namespace for everything we define in our proto file. This is most useful when importing and extending other proto files.

Note: We kept the proto file pretty simple so we can focus on configuring Nest.js to use gRPC microservices.

Like all other transports, we now need to configure the NestMicroservice context and the microservice client in our controller.

```
export const microserviceConfig: GrpcOptions = {
      transport: Transport.GRPC,
      options: {
            url: '0.0.0.0:5667',
            protoPath: join(__dirname, './nest-book-example.proto'),
            package: 'example.nestBook'
      }
};
```

The gRPC transport can take the following options:

- **url**: The url of the gRPC server. The default is `localhost:5000`.
- **credentials**: A `ServerCedentials` object from the `grpc` NPM package. The default is to use the `grpc.getInsecure` method to retrieve a default credential object. This will disable TLS encryption. In order to setup a secure communication channel, use `grpc.createSsl` and provide the root CA, private, and public certificates. More information about credentials can be found here.
- **protoPath**: The absolute path to the proto file.
- **root**: An absolute path to the root of where all your proto files are found. This is an optional option and is most likely not necessary if you are not importing other proto files within your own. If this option is defined, it will be prepended to the `protoPath` option.
- **package**: The name of the package to be used with the client and server. This should match the package name givin in the proto file.

We will need to make some changes to our controller before we can really use the gRPC transport.

```
@Controller()
export class UserController implements OnModuleInit {
      @Client(microserviceConfig)
      private client: ClientGrpc;
      private protoUserService: IProtoUserService;

      constructor(
            private readonly userService: UserService
```

```
    ) {
    }

    public onModuleInit() {
        this.protoUserService =
this.client.getService<IProtoUserService>('UserService');
    }
}
```

Notice that we still have the `client` property decorated with `@Client`, but we have a new type `ClientGrpc` and a new property `protoUserService`. The client injected when using the gRPC transport no longer contains a `send` method. Instead, it has a `getService` method that we must use to retrieve the service we defined in our proto file. We use the `onModuleInit` lifecycle hook so the gRPC service is retrieved immediately after Nest.js has instantiated our modules and before any clients try to use the controller APIs. The `getService` method is a generic and doesn't actually contain any method definitions. Instead, we need to provide our own.

```
import { Observable } from 'rxjs';

export interface IProtoUserService {
    show(data: any): Observable<any>;
}
```

We could be a little more explicit with our interface but this gets the point across. Now the `protoUserService` property in our controller will have a `show` method allowing us to call the `show` gRPC service method.

```
@Get('users/:userId')
@UseGuards(CheckLoggedInUserGuard)
public async show(@Param('userId') userId: number, @Req() req, @Res() res)
{
    this.protoUserService.show({ userId: parseInt(userId.toString(), 10)
}).subscribe({
        next: user => {
            res.status(HttpStatus.OK).json(user);
        },
        error: error => {
            res.status(HttpStatus.INTERNAL_SERVER_ERROR).json(error);
        }
    });
```

```
}

@GrpcMethod('UserService', 'show')
public async rpcShow(data: any) {
      const user =  await this.userService.findById(data.userId);
      return {
            user: {
                  firstName: user.firstName,
                  lastName: user.lastName,
                  email: user.email
            }
      };
}
```

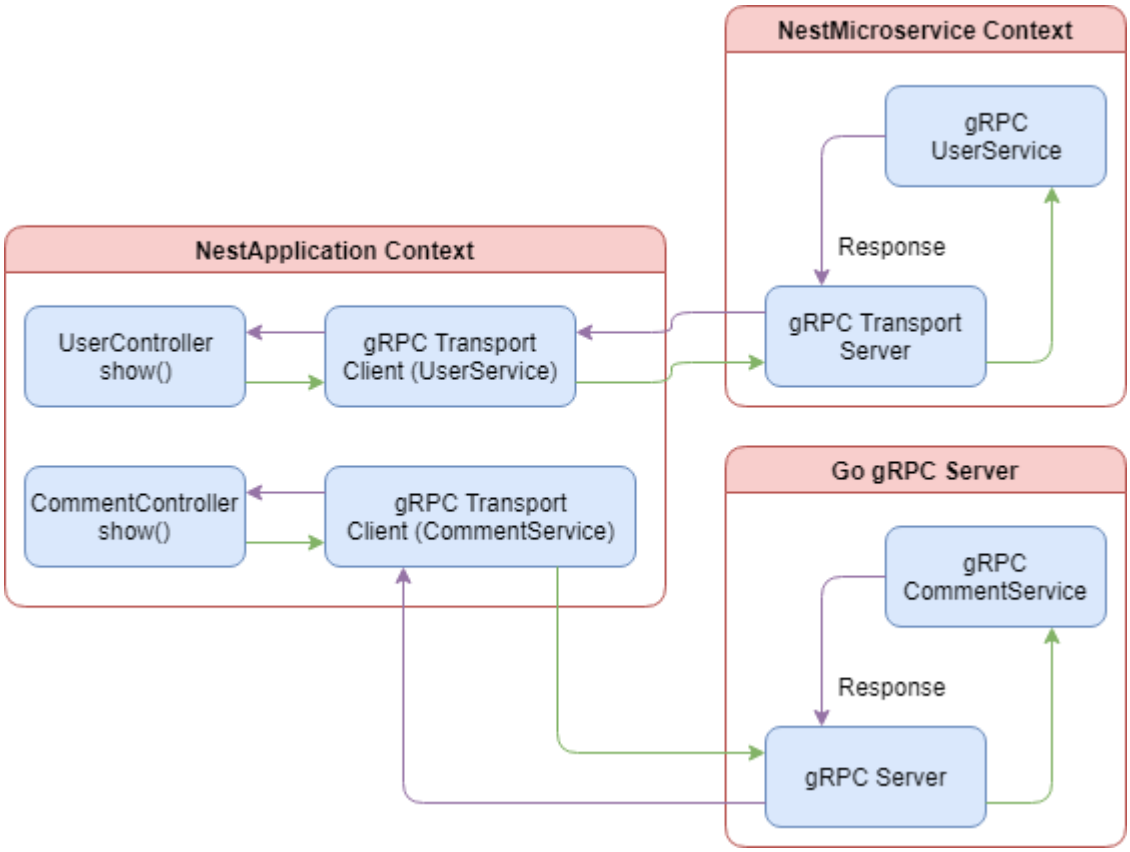
The controller's `show` API method gets updated to use the `protoUserService.show`. This will call the `rpcShow` method, but through the gRPC microservice transport. The `rpcShow` method contains a different decorator, `@GrpcMethod`, instead of `@MessagePattern`. This is required for all gRPC microservice handlers since the microservice is no longer matching a pattern, but instead is calling a defined gRPC service method. In fact, that is the mapping for the two optional parameters to the `@GrpcMethod` decorator: service name and service method.

```
export class UserController implements OnModuleInit {
      @GrpcMethod()
      public async rpcShow(data: any) {
      }
}
```

In the above example, we did not defined the service name and service method when calling the `@GrpcMethod` decorator. Nest.js will automatically map these values to the method and class name. In this example, this is equivalent to `@GrpcMethod('UserController', 'rpcShow')`.

You may have noticed that we are using `0.0.0.0:5667` as the URL of our gRPC server. When we start up the Nest.js application, it will create a gRPC server on the localhost and listen on port `5667`. On the surface, this may look like a more complex version of the TCP transport. However, the power of the gRPC transport is directly derived from the language and platform agnostic nature of protocol buffers. This means we can create a Nest.js application that exposes microservices using gRPC that may be used by any other language or platform as long is it also uses protocol buffers to connect to our microservices. We can

also create Nest.js applications that connect to microservices that may be exposed in another language like Go.



When using the gRPC transport to connect to services at two or more different URLs, we need to create an equal number of gRPC client connections, one for each server. The above diagram shows how processing would look if we offloaded the crud operations for comments in our example blog application to a Go server. We use a gRPC client to connect to the user microservices hosted in our Nest.js application and a separate one to connect to the comment microservices hosted in the Go application.

The same setup can be obtained by using any of the other transports. However, you would have to write the extra code to serialize and deserialize the messages between the Nest.js application and the Go application hosting the microservice. By using the gRPC transport, protocol buffers take care of that for you.

Custom transport

A custom transport allows you to define a new microservice client and server for communicating between the NestApplication and NestMicroservice contexts. You may want to create a custom transport strategy for a number of

reasons: you or your company already have a message broker service that is does not have a built-in Nest.js transport, or you need to customize how one of the built-in transports works. For the purpose of our example, we will work through implementing a new RabbitMQ transport.

```
export class RabbitMQTransportServer extends Server implements
CustomTransportStrategy {
      private server: amqp.Connection = null;
      private channel: amqp.Channel = null;

      constructor(
            private readonly url: string,
            private readonly queue: string
      ) {
            super();
      }
}
```

Nest.js requires all custom transports to implement the `CustomTransportStrategy` interface. This forces us to define our own `listen` and `close` methods. In our example, we connect to the RabbitMQ server and listen on a specific channel. Closing the server is as simple as disconnecting from the RabbitMQ server.

```
public async listen(callback: () => void) {
      await this.init();
      callback();
}

public close() {
      this.channel && this.channel.close();
      this.server && this.server.close();
}

private async init() {
      this.server = await amqp.connect(this.url);
      this.channel = await this.server.createChannel();
      this.channel.assertQueue(`${this.queue}_sub`, { durable: false });
      this.channel.assertQueue(`${this.queue}_pub`, { durable: false });
}
```

By extending the Nest.js `server` class, our custom transport comes pre-equipped with the RxJS handling of messages that makes Nest.js so great. However, our custom transport isn't really handling messages at this point. We need to add the logic for how messages will be sent and received through RabbitMQ to our custom transport.

```
public async listen(callback: () => void) {
      await this.init();
      this.channel.consume(`${this.queue}_sub`, this.handleMessage.bind(this),
{
            noAck: true,
      });
      callback();
}

private async handleMessage(message: amqp.Message) {
      const { content } = message;
      const packet = JSON.parse(content.toString()) as ReadPacket &
PacketId;
      const handler = this.messageHandlers[JSON.stringify(packet.pattern)];

      if (!handler) {
            return this.sendMessage({
                  id: packet.id,
                  err: NO_PATTERN_MESSAGE
            });
      }

      const response$ = this.transformToObservable(await
handler(packet.data)) as Observable<any>;
      response$ && this.send(response$, data => this.sendMessage({
            id: packet.id,
            ...data
      }));
}

private sendMessage(packet: WritePacket & PacketId) {
      const buffer = Buffer.from(JSON.stringify(packet));
      this.channel.sendToQueue(`${this.queue}_pub`, buffer);
}
```

The custom transport will now listen for incoming messages on the `sub` channel and send responses on the `pub` channel. The `handleMessage` method decodes the message's content byte array and uses the embedded pattern object to find the correct microservice handler to service the message. For example, the `{cmd: 'users.create'}` will be handled by the `rpcCreate` handler. Finally, we call the handler, transform the response into an Observable, and pass that back into the Nest.js `Server` class. Once a response is provided, it will be passed on to our `sendMessage` method and out through the `pub` channel.

Since a server is useless without a client, we will need to create one of those too. The RabbitMQ client must extend the Nest.js `ClientProxy` class and provide an override for the `close`, `connect`, and `publish`methods.

```
export class RabbitMQTransportClient extends ClientProxy {
     private server: amqp.Connection;
     private channel: amqp.Channel;
     private responsesSubject: Subject<amqp.Message>;

     constructor(
          private readonly url: string,
          private readonly queue: string) {
          super();
     }

     public async close() {
          this.channel && await this.channel.close();
          this.server && await this.server.close();
     }

     public connect(): Promise<void> {
          return new Promise(async (resolve, reject) => {
               try {
                    this.server = await amqp.connect(this.url);
                    this.channel = await this.server.createChannel();

                    const { sub, pub } = this.getQueues();
                    await this.channel.assertQueue(sub, { durable: false
});
                    await this.channel.assertQueue(pub, { durable: false
});
```

```
                    this.responsesSubject = new Subject();
                    this.channel.consume(pub, (message) => {
this.responsesSubject.next(message); }, { noAck: true });
                    resolve();
                } catch (error) {
                    reject(error);
                }
            });
        }

        protected async publish(partialPacket: ReadPacket, callback:
(packet: WritePacket) => void) {
        }

        private getQueues() {
            return { pub: `${this.queue}_pub`, sub: `${this.queue}_sub` };
        }
}
```

In our example, we created a new connection to the RabbitMQ server and the specified `pub` and `sub`channels. The client uses the channels in an opposite configuration compared to the server. The client sends messages through the `sub` channel and listens for responses on the `pub` channel. We also make use of the power of RxJS by piping all responses into a Subject to make processing simpler in the `publish` method. Let's implement the `publish` method.

```
protected async publish(partialPacket: ReadPacket, callback: (packet:
WritePacket) => void) {
        if (!this.server || !this.channel) {
            await this.connect();
        }

        const packet = this.assignPacketId(partialPacket);
        const { sub } = this.getQueues();

        this.responsesSubject.asObservable().pipe(
            pluck('content'),
            map(content => JSON.parse(content.toString()) as WritePacket &
PacketId),
            filter(message => message.id === packet.id),
            take(1)
```

```
        ).subscribe(({err, response, isDisposed}) => {
            if (isDisposed || err) {
                callback({
                    err,
                    response: null,
                    isDisposed: true
                });
            }

            callback({err, response});
        });

        this.channel.sendToQueue(sub, Buffer.from(JSON.stringify(packet)));
    }
```

The `publish` method starts off with assigning a unique ID to the message and subscribes to the response subject for sending the response back to the microservice caller. Finally, `sendToQueue` is called to send the message as a byte array to the `sub` channel. Once a response is received, the subscription to the response subject is fired. The first thing the subscription stream does is extract the `content` of the response and verify that the message ID matches the one that was assigned when `publish` was initially called. This keeps the client from processing a message response that does not belong to the specific `publish` execution context. Put simply, the client will receive every microservice response, even those that might be for a different microservice or a different execution of the same microservice. If the IDs match, the client checks fir errors and uses the `callback` to send the response back to the microservice caller.

Before we can use our new transport, we will need to update the microservice configuration object we created earlier.

```
export const microserviceConfig = {
    url: process.env.AMQP_URL
};

export const microserviceServerConfig: (channel: string) =>
CustomStrategy = channel => {
    return {
        strategy: new RabbitMQTransportServer(microserviceConfig.url,
channel)
    }
```

```
};
```

We now have a method that will instantiate our custom transport server. This is used in the `bootstrap`of our application to connect our NestMicroservice context to the RabbitMQ server.

```
async function bootstrap() {
     const app = await NestFactory.create(AppModule);
     app.connectMicroservice(microserviceServerConfig('nestjs_book'));

     await app.startAllMicroservicesAsync();
     await app.listen(3001);
}
```

The last piece of our custom transport is in our controller. Since we are using a custom transport, we can no longer use the `@ClientProxy` decorator. Instead, we have to instantiate our custom transport our selves. You could do this in the constructor as so:

```
@Controller()
export class UserController {
     client: ClientProxy;

     constructor(private readonly userService: UserService) {
          this.client = new
RabbitMQTransportClient(microserviceConfig.url, 'nestjs_book');
     }
}
```

Wait! You have now created a hard binding between the controller and the custom transport client. This makes it more difficult to migrate to a different strategy in the future and very difficult to test. Instead, let's make use of Nest.js's fabulous Dependency Injection to create our client. Start off with creating a new module to house and expose our custom transport client.

```
const ClientProxy = {
  provide: 'ClientProxy',
  useFactory: () => new RabbitMQTransportClient(microserviceConfig.url,
'nestjs_book')
};

@Module({
```

```
    imports: [],
    controllers: [],
    components: [ClientProxy],
    exports: [ClientProxy]
})
export class RabbitMQTransportModule {}
```

In our example, we gave our component the injection token `'ClientProxy'`. This was just to keep things simple, and you can call it whatever you like. The import part is to make sure the injection token used to register the component is also the one used in the `@Inject` decorator we place in our controller's constructor.

```
@Controller()
export class UserController {

    constructor(
        private readonly userService: UserService,
        @Inject('ClientProxy')
        private readonly client: ClientProxy
    ) {
    }
```

Our controller will now have a microservice client injected in at run time allowing the API handlers to communicate with the microservice handlers. Even better, the client can now be overridden in tests with a mock. Startup a RabbitMQ server, such as the rabbitmq docker image , and setup the `AMQP_URL`environment variable, ie `amqp://guest:guest@localhost:5672`, and all microservice requests will be processed through the RabbitMQ server.

The data flow and how the microservice client and server operate in our RabbitMQ example are almost identical to the NATS transport. Just like with NATS, RabbitMQ provides the ability to have multiple NestMicroservice contexts consuming messages. RabbitMQ will work to load balance between all the consumers.

Hybrid application

When we first started our microservice implementation in this chapter, we modified the bootstrap method to call `connectMicroservice`. This is a special method that converts our Nest.js application into a hybrid application. This

simply means our application now contains multiple context types. Simple enough but this has some implications that you should be aware of. Specifically, using the hybrid application approach, you will no longer be able to attach global filters, pipes, guards, and interceptors for the NestMicroservice context. This is because the NestMicroservice context is immediately bootstrapped, but not connected, in a hybrid application. To get around this limitation, we can create our two contexts independently.

```
async function bootstrap() {
      const app = await NestFactory.create(AppModule);
      const rpcApp = await NestFactory.createMicroservice(AppModule,
microserviceServerConfig('nestjs_book'));
      rpcApp.useGlobalFilters(new RpcValidationFilter());

      await rpcApp.listenAsync();
      await app.listen(process.env.PORT || 3000);
}
```

Now that we are creating the two application contexts independently, we can make use of globals for the NestMicroservice context. To test this, we can update the `rpcCreate` handler to remove the `RpcValidationFilter`. Executing the application at this point should still result in validation errors being returned when the request to the `create` API does not contain required fields.

```
@MessagePattern({cmd: 'users.create'})
public async rpcCreate(data: CreateUserRequest) {
      if (!data || (data && Object.keys(data).length === 0)) throw new
RpcValidationException();
      await this.userService.create(data);
}
```

We can extend this approach of bootstrapping our application to split even more of our application into separate contexts. This still does not make use of multiple processes or threads, but employing some more advanced architecture design, we can gain those benefits.

Advanced architecture design

So far we have covered everything needed to setup and start writing and using microservices in Nest.js. Along the way we describe some of the drawbacks Nest.js' implementation of microservices has. Most notably, since the

microservices does not run in a separate thread or process, you may not be gaining much in the way of performance when using Nest.js microservices.

However, that is not to say you can't get these benefits. Nest.js just doesn't provide the tools out of the box. In most material found on the subject of running a NodeJS application in production, the one thing that is typically always covered and recommended is the use of the NodeJS `cluster` module. We can do the same thing with our Nest.js application.

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

      await app.listen(process.env.PORT || 3000);
}

async function bootstrapRpc() {
      const rpcApp = await NestFactory.createMicroservice(AppModule,
microserviceServerConfig('nestjs_book'));
      rpcApp.useGlobalFilters(new RpcValidationFilter());

      await rpcApp.listenAsync();
}

if (cluster.isMaster) {
      const appWorkers = [];
      const rpcWorkers = [];

      for (let i = 0; i < os.cpus().length; i++) {
            const app = cluster.fork({
                  APP_TYPE: 'NestApplication'
            });
            const rpc = cluster.fork({
                  APP_TYPE: 'NestMicroservice'
            });

            appWorkers.push(app);
            rpcWorkers.push(rpc);
      }

      cluster.on('exit', function(worker, code, signal) {
            if (appWorkers.indexOf(worker) > -1) {
```

```
                const index = appWorkers.indexOf(worker);
                const app = cluster.fork({
                    APP_TYPE: 'NestApplication'
                });
                appWorkers.splice(index, 1, app);
            } else if (rpcWorkers.indexOf(worker) > -1) {
                const index = rpcWorkers.indexOf(worker);
                const rpc = cluster.fork({
                    APP_TYPE: 'NestMicroservice'
                });
                rpcWorkers.splice(index, 1, rpc);
            }
        });
    } else {
        if (process.env.APP_TYPE === 'NestApplication') {
            bootstrapApp();
        } else if (process.env.APP_TYPE === 'NestMicroservice') {
            bootstrapRpc();
        }
    }
```

Now, not only does our NestApplication and NestMicroservice contexts run on their own threads, they are also clustered according to the number of CPUs available on the server. For each CPU, a separate NestApplication and NestMicroservice context will be created. The NestApplication context threads will share the main application port. Finally, since we are using RabbitMQ, having multiple NestMicroservice contexts running, we have multiple subscribers waiting for microservice messages. RabbitMQ will take care of load balancing message distribution between all of our NestMicroservice instances. We have made our application more resilient and better equipped to handle a heavier load of users than what it was at the beginning of this chapter.

Summary

At the beginning of this chapter, we stated “microservice” was a misleading name for this part of Nest.js. In fact, that could still be the case, but it really depends on a number of factors. Our initial example using the TCP transport could hardly qualify as a microservice by all conventional definitions. Both the NestApplication and NestMicroservice context were executing from the same process, meaning a catastrophic failure in one could bring both down.

After highlighting all the transports, Nest.js comes with out-of-the-box, and we re-implemented our microservices in the example blog application using a custom RabbitMQ transport. We even went as far as running the NestApplication and NestMicroservice contexts in their own thread. This was a major step in the right direction for fulfilling the “microservice” name.

Although we didn’t cover specifics in this book, it should now be apparent that you’re not limited to using microservices defined in the same Nest.js project or repository. Using transports like Redis and RabbitMQ, we could create and use multiple Nest.js projects for the sole purpose of executing a NestMicroservice context. All of these projects can be running independently inside a Kubernetes cluster and accessed by passing messages via Redis or RabbitMQ. Even better, we can use the built-in gRPC transport to communicate with microservices wrote in other languages and deployed to other platforms.

In the next chapter we will learn about routing and request handling in Nest.js.

Chapter 10. Routing and request handling in Nest.js

Routing and request handling in Nest.js is handled by the controllers layer. Nest.js routes requests to handler methods, which are defined inside controller classes. Adding a routing decorator such as `@Get()` to a method in a controller tells Nest.js to create an endpoint for this route path and route every corresponding request to this handler.

In this chapter, we'll go over the various aspects of routing and request handling in Nest.js using the EntryController from our blog application as a basis for some examples. We'll be looking at different approaches that you can use to write request handlers, so not all examples shown will match code from our blog application.

Request handlers

A basic GET request handler for the `/entries` route registered in the EntryController could look like this:

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

@Controller('entries')
export class EntryController {
    @Get()
    index(): Entry[] {
        const entries: Entry[] = this.entriesService.findAll();
        return entries;
    }
```

The `@Controller('entries')` decorator tells Nest.js to add an `entries` prefix to all routes registered in the class. This prefix is optional. An equivalent way to setup this route would be as follows:

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

@Controller()
export class EntryController {
    @Get('entries')
    index(): Entry[] {
```

```
        const entries: Entry[] = this.entriesService.findAll();
        return entries;
    }
```

Here, we don't specify a prefix in the `@Controller()` decorator, but instead use the full route path in the `@Get('entries')` decorator.

In both cases, Nest.js will route all GET requests to `/entries` to the `index()` method in this controller. The array of entries returned from the handler will be **automatically** serialized to JSON and sent as the response body, and the response status code will be 200. This is the standard approach of generating a response in Nest.js.

Nest.js also provides the `@Put()`, `@Delete()`, `@Patch()`, `@Options()`, and `@Head()` decorators to create handlers for other HTTP methods. The `@All()` decorator tells Nest.js to route all HTTP methods for a given route path to the handler.

Generating responses

Nest.js provides two approaches for generating responses.

Standard approach

Using the standard and recommended approach, which has been available since Nest.js 4, Nest.js will **automatically** serialize the JavaScript object or array returned from the handler method to JSON and send it in the response body. If a string is returned, Nest.js will just send the string without serializing it to JSON.

The default response status code is 200, expect for POST requests, which uses 201. The response code for can easily be changed for a handler method by using the `@HttpCode(...)` decorator. For example:

```
@HttpCode(204)
@Post()
create() {
  // This handler will return a 204 status response
}
```

Express approach

An alternate approach to generating responses in Nest.js is to use a response object directly. You can ask Nest.js to inject a response object into a handler method using the `@Res()` decorator. Nest.js uses express response objects].

You can rewrite the response handler seen earlier using a response object as shown here.

```
import { Controller, Get, Res } from '@nestjs/common';
import { Response } from 'express';

@Controller('entries')
export class EntryController {
      @Get()
      index(@Res() res: Response) {
            const entries: Entry[] = this.entriesService.findAll();
            return res.status(HttpStatus.OK).json(entries);
      }
}
```

The express response object is used directly to serialize the entries array to JSON and send a 200 status code response.

The typings for the `Response` object come from express. Add the `@types/express` package to your `devDependencies` in `package.json` to use these typings.

Route parameters

Nest.js makes it easy to accept parameters from the route path. To do so, you simply specify route parameters in the path of the route as shown below.

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

@Controller('entries')
export class EntryController {
      @Get(':entryId')
      show(@Param() params) {
            const entry: Entry = this.entriesService.find(params.entryId);
            return entry;
```

```
    }
}
```

The route path for the above handler method above is `/entries/:entryId`, with the `entries` portion coming from the controller router prefix and the `:entryId` parameter denoted by a colon. The `@Param()`decorator is used to inject the params object, which contains the parameter values.

Alternately, you can inject individual param values using the `@Param()` decorator with the parameter named specified as shown here.

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

@Controller('entries')
export class EntryController {
    @Get(':entryId')
    show(@Param('entryId') entryId) {
        const entry: Entry = this.entriesService.findOne(entryId);
        return entry;
    }
}
```

Request body

To access the body of a request, use the `@Body()` decorator.

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

@Controller('entries')
export class EntryController {
    @Post()
    create(@Body() body: Entry) {
        this.entryService.create(body);
    }
}
```

Request object

To access the client request details, you can ask Nest.js to inject the request object into a handler using the `@Req()` decorator. Nest.js uses express request objects.

For example,

```
import { Controller, Get, Req } from '@nestjs/common';
import { Request } from 'express';

@Controller('entries')
export class EntryController {
     @Get()
     index(@Req() req: Request): Entry[] {
          const entries: Entry[] = this.entriesService.findAll();
          return entries;
     }
```

The typings for the `Request` object come from express. Add the `@types/express` package to your `devDependencies` in `package.json` to use these typings.

Asynchronous handlers

All of the examples shown so far in this chapter assume that handlers are synchronous. In a real application, many handlers will need to be asynchronous.

Nest.js provides a number of approaches to write asynchronous request handlers.

Async/await

Nest.js has support for async request handler functions.

In our example application, the `entriesService.findAll()` function actually returns a `Promise<Entry[]>`. Using async and await, this function could be written as follows.

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

@Controller('entries')
export class EntryController {
```

```
    @Get()
    async index(): Promise<Entry[]> {
        const entries: Entry[] = await this.entryService.findAll();
        return entries;
    }
```

Async functions have to return promises, but using the async/await pattern in modern JavaScript, the handler function can appear to be synchronous. Next, we'll resolve the returned promise and generate the response.

Promise

Similarly, you can also just return a promise from a handler function directly without using async/await.

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

@Controller('entries')
export class EntryController {
    @Get()
    index(): Promise<Entry[]> {
        const entriesPromise: Promise<Entry[]> =
this.entryService.findAll();
        return entriesPromise;
    }
```

Observables

Nest.js request handlers can also return RxJS Observables.

For example, if `entryService.findAll()` were to return an Observable of entries instead of a Promise, the following would be completely valid.

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

@Controller('entries')
export class EntryController {
    @Get()
    index(): Observable<Entry[]> {
        const entriesPromise: Observable<Entry[]> =
this.entryService.findAll();
```

```
        return entriesPromise;
    }
```

There is no recommended way to write asynchronous request handlers. Use whichever method you are most comfortable with.

Error responses

Nest.js has an exception layer, which is responsible for catching unhandled exceptions from request handlers and returning an appropriate response to the client.

A global exception filter handles all exception thrown from request handlers.

HttpException

If an exception thrown from a request handler is a `HttpException`, the global exception filter will transform it to the a JSON response.

For example, you can throw an `HttpException` from the `create()` handler function if the body is not valid as shown.

```
import { Body, Controller, HttpException, HttpStatus, Post } from
'@nestjs/common';

@Controller('entries')
export class EntryController {
    @Post()
    create(@Body() entry: Entry) {
        if (!entry) throw new HttpException('Bad request',
HttpStatus.BAD_REQUEST);
        this.entryService.create(entry);
    }
}
```

If this exception is thrown, the response would look like this:

```
{
    "statusCode": 400,
    "message": "Bad request"
```

```
}
```

You can also completely override the response body by passing an object to the `HttpException` constructor as follows.

```
import { Body, Controller, HttpException, HttpStatus, Post } from
'@nestjs/common';

@Controller('entries')
export class EntryController {
      @Post()
      create(@Body() entry: Entry) {
            if (!entry) throw new HttpException({ status:
HttpStatus.BAD_REQUEST, error: 'Entry required' });
            this.entryService.create(entry);
      }
}
```

If this exception is thrown, the response would look like this:

```
{
      "statusCode": 400,
      "error": "Entry required"
}
```

Unrecognized exceptions

If the exception is not recognized, meaning it is not `HttpException` or a class that inherits from `HttpException`, then the client will receive the JSON response below.

```
{
      "statusCode": 500,
      "message": "Internal server error"
}
```

Summary

With the help of using the EntryController from our example blog application, this chapter has covered aspects of routing and request handling in Nest.js. You

should now understand various approaches that you can use to write request handlers.

In the next chapter we detail the OpenAPI specification, which is a JSON schema that can be used to construct a JSON or YAML definition of a set of restful APIs.

Chapter 11. OpenAPI (Swagger) Specification

The OpenAPI specification, most notably known by it's former name Swagger, is a JSON schema that can be used to construct a JSON or YAML definition of a set of restful APIs. OpenAPI itself is language agnostic, meaning the underlying APIs can be constructed in any language with any tool or framework the developer would like. The sole concern of an OpenAPI document is to describe the inputs and outputs, among other things, of API endpoints. In this respect, an OpenAPI document acts as a documentation tool allowing developers to easily describe their public APIs in a format that is widely known, understood, and supported.

The OpenAPI document, however, is not just limited to being documentation. Many tools have been developed that are capable of using an OpenAPI document to auto-generate client projects, server stubs, an API explorer UI for visually inspecting the OpenAPI document, and even server generators. Developers can find such tools as the Swagger Editor, Codegen, and UI at https://swagger.io.

While some tools exist to generate an OpenAPI document, a number of developers maintain such documents either as individual JSON or YAML files. They can break their document up into smaller pieces using OpenAPI reference mechanics. In Nest.js, a separate module is available for developers to use to generate an OpenAPI document for their application. Instead of writing your OpenAPI document by hand, Nest.js will use the decorators you provide in your controllers to generate as much information that it can about the APIs within your project. Of course, it won't get everything out of the box. For that, the Nest.js swagger module provides additional decorators that you can use to fill in the gaps.

In this chapter, we will explore using the Nest.js Swagger module to generate a swagger version 2 document. We will begin with configuring the Nest.js Swagger module. We will setup our blog example application to expose the swagger document using the Swagger UI and begin exploring how the Nest.js decorators you are used to using already affect the swagger document. We will also explore the new decorators the swagger module provides. By the end of this chapter, you will have a complete understanding of how Nest.js produces a swagger document. Before getting started, be sure you run `npm install @nestjs/swagger` in your project. To see a working example, remember you can clone the accompanying Git repository for this book:

```
git clone https://github.com/backstopmedia/nest-book-example.git
```

Document Settings

Each swagger document can contain a basic set of properties such as the title of the application. This information can be configured using the various public methods found on the `DocumentBuilder` class. These methods all return the document instance allowing you to chain as many of the methods as you need. Be sure to finish your configuration before calling the `build` method. Once the `build` method has been called, the document settings are no longer modifiable.

```
const swaggerOptions = new DocumentBuilder()
      .setTitle('Blog Application')
      .setDescription('APIs for the example blog application.')
      .setVersion('1.0.0')
      .setTermsOfService('http://swagger.io/terms/')
      .setContactEmail('admin@example.com')
      .setLicense('Apache 2.0', 'http://www.apache.org/licenses/LICENSE-
2.0.html')
      .build();
```

These methods are used to configure the `info` section of the swagger document. The swagger specification requires the `title` and `version` fields to be provided, but Nest.js will default these values to an empty string and `"1.0.0"`, respectively. If your project has terms of service and a license, you can use `setTermsOfService` and `setLicense` to provide a URL to those resources within your application.

Swagger documents can also contain server information. Users, developers, and the UI can use this information to understand how to access the APIs described by the document.

```
const swaggerOptions = new DocumentBuilder()
      .setHost('localhost:3000')
      .setBasePath('/')
      .setSchemes('http')
      .build();
```

The `setHost` should contain only the server and port to access the APIs. If, in your application, you use `setGlobalPrefix` to configure a base path for the Nest.js application, set the same value in the swagger document

using `setBasePath`. The swagger specification uses a `schemes` array to describe the transfer protocol used by the APIs. While the swagger specification supports the `ws` and `wss` protocols as well as multiple values, Nest.js limits the value to either `http` or `https`. Metadata and external documentation can also be added to provide users of the swagger document additional details regarding how the APIs work.

```
const swaggerOptions = new DocumentBuilder()
    .setExternalDoc('For more information', 'http://swagger.io')
    .addTag('blog', 'application purpose')
    .addTag('nestjs', 'framework')
    .build();
```

Use the first parameter of `setExternalDoc` to describe the external documentation and a URL to the documentation as the second parameter. An endless number of tags can be added to the document using `addTag`. The only requirement is the first parameter to `addTag` be unique. The second parameter should describe the tag. The last document setting is how user's authenticate with the APIs.

Documenting authentication

The swagger specification supports three types of authentication: basic, API key, and Oauth2. Nest.js provides two different methods that can be used to auto-configure the swagger document authentication information with the possibility for some settings to be overridden. Keep in mind, this is describing how users will authenticate with your application.

```
const swaggerOptions = new DocumentBuilder()
    .addBearerAuth('Authorization', 'header', 'apiKey')
    .build();
```

If your application is using `basic` authentication, the username and password as a base64 encoded string, or JSON web tokens (JWT), you will make use of the `addBearerAuth` configuration method. The example above uses the defaults Nest.js will use if no parameters are passed and establishes that the APIs use an API key like a JWT in the authorization header. The first parameter should contain the key/header where the authentication key should be provided. This same configuration should be used if users will be using an application key to access the APIs. Application keys are typically used by public API provides like Google Maps to limit access to APIs and associate an API call to a specific billing account.

```
const swaggerOptions = new DocumentBuilder()
```

```
    .addBearerAuth('token', 'query', 'apiKey')
    .addBearerAuth('appId', 'query', 'apiKey')
    .build();
```

This example describes two query parameters that must be included when calling APIs that require authentication. The second parameter describes where the authentication key should be provided, either as a header, query, or body parameter. The third parameter is the type of authentication. When using `addBearerAuth`, use `apiKey` or `basic`. In addition to basic and API key authentication, swagger also supports documenting an Oauth2 authentication flow.

```
const swaggerOptions = new DocumentBuilder()
    .addOAuth2('password', 'https://example.com/oauth/authorize',
'https://example.com/oauth/token', {
      read: 'Grants read access',
      write: 'Grants write access',
      admin: 'Grants delete access'
    })
    .build();
```

The first parameter to the `addOAuth2` method is the OAuth2 flow the APIs use for authentication. In this example, we use the `password` flow to indicate the user should send a username and password to the API. You can also use `implicit`, `application`, and `accessCode` flow. The second and third parameters are the URLs where the user will authorize access to the APIs and request a refresh token, respectively. The last parameter is an object of all the scopes with descriptions that are available in the application.

For the blog application, we will keep the configuration simple and store the configuration in a new file in the `shared/config` directory. Having a central location will let us write the configuration once and implement multiple times.

```
export const swaggerOptions = new DocumentBuilder()
    .setTitle('Blog Application')
    .setDescription('APIs for the example blog application.')
    .setVersion('1.0.0')
    .setHost('localhost:3000')
    .setBasePath('/')
    .setSchemes('http')
    .setExternalDoc('For more information', 'http://swagger.io')
    .addTag('blog', 'application purpose')
```

```
    .addTag('nestjs', 'framework')
    .addBearerAuth('Authorization', 'header', 'apiKey')
    .build();
```

Our first implementation will use the configuration and the Nest.js swagger module to produce two new endpoints in our application: one to serve the swagger UI application and one to serve the swagger document as raw JSON.

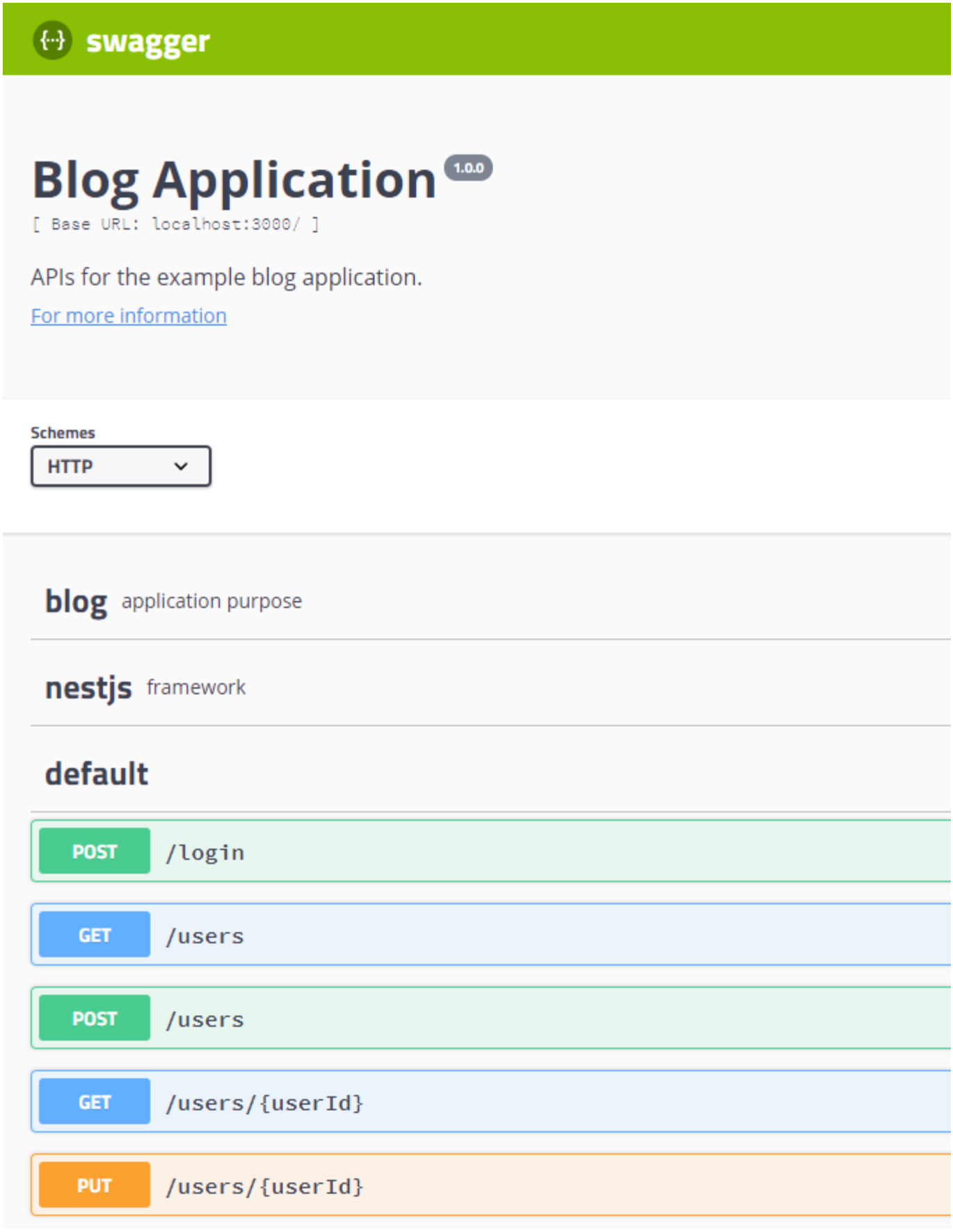
Swagger UI

The swagger module is unlike most other Nest.js modules. Instead of being imported into your application's primary app module, the swagger module is configured within the main bootstrap of your application.

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

    const document = SwaggerModule.createDocument(app, swaggerOptions);
    SwaggerModule.setup('/swagger', app, document);

    await app.listen(process.env.PORT || 3000);
}
```

After declaring the Nest application and before calling the `listen` method, we use the swagger document options configured in the last section and `SwaggerModule.createDocument` to create the swagger document. The swagger module will inspect all controllers within the application and use decorators to construct the swagger document in memory.

Once we have created the swagger document, we setup and instruct the swagger module to serve the swagger UI at a specified path, `SwaggerModule.setup('/swagger', app, document)`. Under the hood, the swagger module makes use of the `swagger-ui-express` NodeJS module to turn the swagger document into a full web UI application.


swagger
Blog Application
1.0.0
[Base URL: localhost:3000/]
APIs for the example blog application.
For more information
Schemes
HTTP
blog application purpose
nestjs framework
default
POST /login
GET /users
POST /users
GET /users/{userId}
PUT /users/{userId}

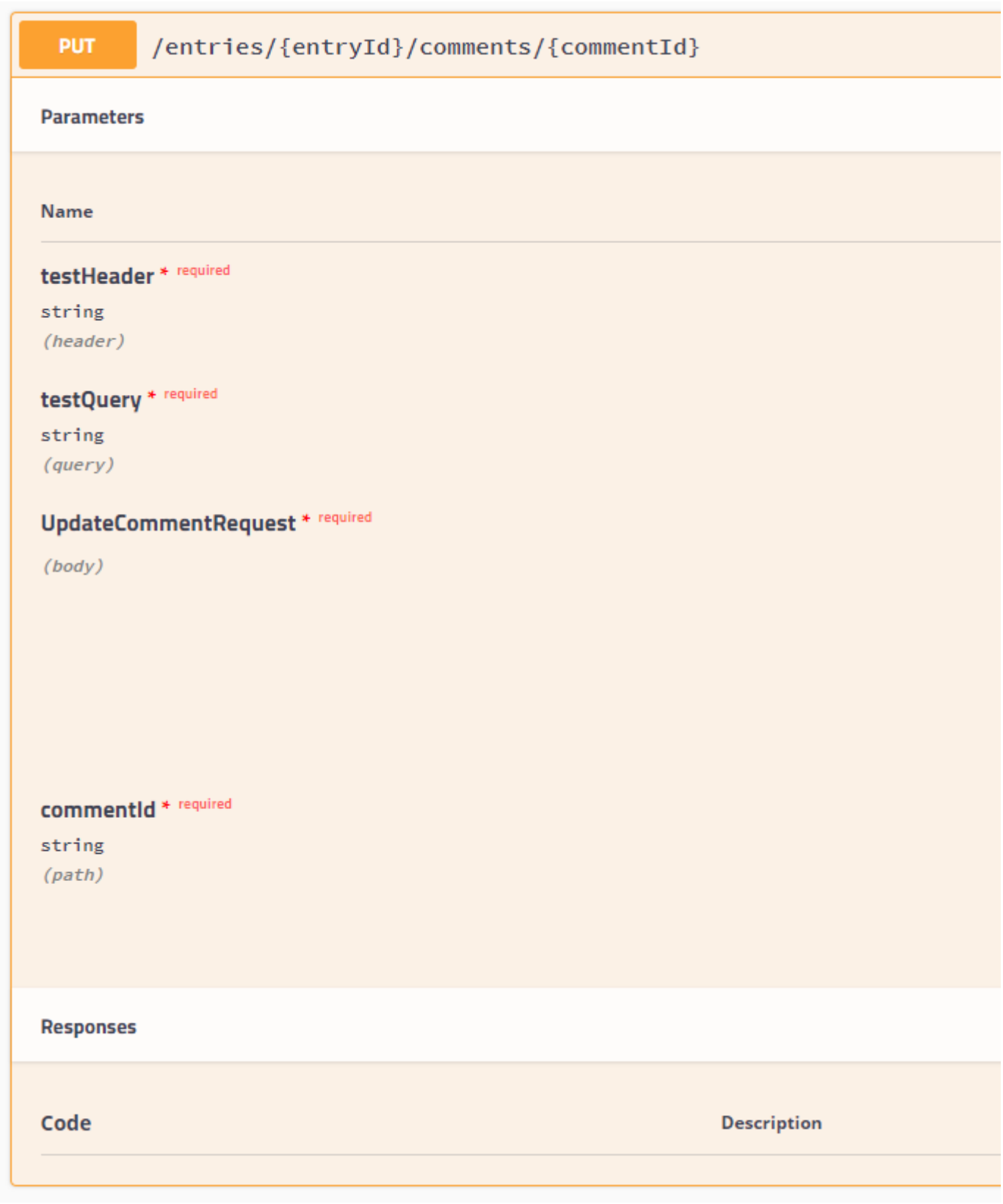
The above figure shows a basic Swagger UI application using our example blog application. The JSON used to produce the UI is also available by appending `-json` to the path we configured for the UI. In our example, accessing `/swagger-json` will return the swagger document. This can be used with code generators like Swagger Codegen. For more information about Swagger UI and Swagger Codegen, refer to https://swagger.io.

If you have followed along with the book and created the blog application, you may find that the Swagger UI produced does not contain a lot of information about the APIs in the application. Since the swagger document is built using Typescript decorator metadata, you may need to alter your types or make use of the additional decorators found in the Nest.js swagger module.

API input decorators

The Nest.js swagger module can produce a swagger document using the `@Body`, `@Param`, `@Query`, and `@Headers` decorators. However, depending on how you write your API controllers, the swagger document can contain very little information. The swagger module will use the types associated with the decorated parameters to describe the parameters an API expects within the swagger document. To depict this, we will modify the comment PUT API to use all four decorators and show how that affects the swagger document by reviewing the swagger UI application.

```
@Controller('entries/:entryId')
export class CommentController {
      @Put('comments/:commentId')
      public async update(
            @Body() body: UpdateCommentRequest,
            @Param('commentId') comment: string,
            @Query('testQuery') testQuery: string,
            @Headers('testHeader') testHeader: string
      ) {
      }
}
```



From the example, we can see the header of this API card uses a combination of the `@Controller` and `@Put` decorators to construct the path to the API. The parameters section is built using the `@Body`, `@Param`, `@Query`, and `@Headers` query

params. The types we provide to the decorated parameters is used in the Swagger UI as a hint to the user regarding what is expected in the parameter.

PUT /entries/{entryId}/comments/{commentId}

Parameters

Name	Description
testHeader * required string *(header)*	testHeader
testQuery * required string *(query)*	testQuery

UpdateCommentRequest * required
(body)
	Example Value | Model
	{
	"body": "string
	}
	Cancel
	Parameter content type
	application/json

commentId * required string *(path)*	commentId

Execute

Responses

Code	Description

Clicking the **Try it out** button in the header of the API card changes the card into a set of inputs. This allows the user to fill in the required and optional parameters of the API and execute the API call. We will cover the remaining sections of the API card later. For now, let's review the basic parameter decorators in more detail.

@Body

You may have noticed in our example, the parameter we decorated with `@Body` had a type of `UpdateCommentRequest`. Your application may or may not have this class already. If not, let's write it now.

```
export class UpdateCommentRequest {
    @ApiModelPropertyOptional()
    public body: string;
}
```

The request class is very basic and makes use of the first decorator we will cover from the Nest.js swagger module, `@ApiModelPropertyOptional`. This decorator informs the swagger module that the `body`property of the request class is an optional property that can be included in the request body when calling the API. This decorator is actually a shortcut for the `@ApiModelProperty` decorator. We could write our request class as:

```
export class UpdateCommentRequest {
    @ApiModelProperty({ required: false })
    public body: string;
}
```

However, if a property is optional, use the `@ApiModelPropertyOptional` decorator to save yourself some typing. Both decorators can take several additional properties in an object passed to the decorator that will further define the data model for the request body.

- **description**: A string that can be used to describe what the model property should contain or what it is used for.
- **required**: A boolean indicating if the model property is required. This only applies to the `@ApiModelProperty` decorator.
- **type**: The Nest.js swagger module will use the type associated with the model property or you can pass the **type** as any string or class value. If you use the **isArray** property, the **type** property should also be used. This property can also be

used to pass any of the data types defined in the swagger specification.

- **isArray**: A boolean indicating if the model property should take an array of values. If the model does take an array of values, be sure to include this value in the decorator or the Nest.js swagger module will not know to represent the model property as an array.
- **collectionFormat**: Maps to the swagger specification **collectionFormat** setting. This is used to depict how a model properties array values should be formatted. For the request body, this property should probably not be used. Possible values are:
    - **csv**: comma separated values `foo,bar`
    - **ssv**: space separated values `foo bar`
    - **tsv**: tab separated values `foo\tbar`
    - **pipes**: pipe separated values `foo|bar`
    - **multi**: corresponds to multiple parameter instances instead of multiple values for a single instance foo=bar&foo=baz. This is valid only for parameters in “query” or “formData”.
- **default**: The default value to be used for the model property in the swagger document. This value will also be used in the example provided in the Swagger UI. The type of this value will depend on the type of the model property but could be a string, number, or even an object.
- **enum**: If your model properties type is an enum, pass the same enum to the decorator using this property so the Nest.js swagger module can inject those enum values into the swagger document.
- **format**: If you use the **type** property with a data type described in the swagger specification, you may also need to pass the format for that data type. For example, a field that takes a number with multiple precision points, values after the decimal, the **type** would be `integer` but the **format** may be either `float` or `double`.
- **multipleOf**: A number indicating that the value passed in the model property should have a remainder of zero using the modulus operator. Setting this property in the decorator is only valid if the model properties type is `number` or the **type** provided to the decorator is `integer`.

- **maximum**: A number indicating that the value passed in the model property should be less than or equal to the given value to be valid. Setting this property in the decorator is only valid if the model properties type is `number` or the **type** provided to the decorator is `integer`. This property should not be used with **exclusiveMaximum**.
- **exclusiveMaximum**: A number indicating that the value passed in the model property should be less than the given value to be valid. Setting this property in the decorator is only valid if the model properties type is `number` or the **type** provided to the decorator is `integer`. This property should not be used with **maximum**.
- **minimum**: A number indicating that the value passed in the model property should be greater than or equal to the given value to be valid. Setting this property in the decorator is only valid if the model properties type is `number` or the **type** provided to the decorator is `integer`. This property should not be used with **exclusiveMinimum**.
- **exclusiveMinimum**: A number indicating that the value passed in the model property should be less than the given value to be valid. Setting this property in the decorator is only valid if the model properties type is `number` or the **type** provided to the decorator is `integer`. This property should not be used with **minimum**.
- **maxLength**: A number indicating that the value passed in the model property should a character length less than or equal to the given value to be valid. Setting this property in the decorator is only valid if the model properties type is `string` or the **type** provided to the decorator is `string`.
- **minLength**: A number indicating that the value passed in the model property should a character length more than or equal to the given value to be valid. Setting this property in the decorator is only valid if the model properties type is `string` or the **type** provided to the decorator is `string`.
- **pattern**: A string containing a JavaScript compatible regular expression. The value passed in the model property should match the regular expression to be valid. Setting this property in the decorator is only valid if the model properties type is `string` or the **type** provided to the decorator is `string`.
- **maxItems**: A number indicating that the value passed in the model property should an array length less than or equal to the

given value to be valid. Setting this property in the decorator is only valid if the the **isArray** is also provided with a value of `true`.

- **minItems**: A number indicating that the value passed in the model property should an array length more than or equal to the given value to be valid. Setting this property in the decorator is only valid if the the **isArray** is also provided with a value of `true`.
- **uniqueItems**: A number indicating that the value passed in the model property should contain a set of unique array values. Setting this property in the decorator is only valid if the the **isArray** is also provided with a value of `true`.
- **maxProperties**: A number indicating that the value passed in the model property should contain a number of properties less than or equal to the given value to be valid. Setting this property in the decorator is only valid if the model property type is a class or object.
- **minProperties**: A number indicating that the value passed in the model property should contain a number of properties more than or equal to the given value to be valid. Setting this property in the decorator is only valid if the model property type is a class or object.
- **readOnly**: A boolean indicating the model property **MAY** be sent in the API response body, but should not be provided in the request body. Use this if you will be using the same data model class to represent the request and response bodies of an API.
- **xml**: A string containing XML that represent the format of the model property. Only use if the model property will contain XML.
- **example**: An example value to place in the swagger document. This value will also be used in the example provided in the Swagger UI and takes precedence over the **default** decorator property value.

The property that has been decorated with the `@Body` decorator should always have a type that is a class. Typescript interfaces cannot be decorated and do not provide the same metadata that a class with decorators can. If, in your application, any of your APIs have a property with the `@Body` decorator and an interface type, the Nest.js swagger module will not be able to correctly create the swagger document. In fact, the Swagger UI will most likely note display the body parameter at all.

@Param

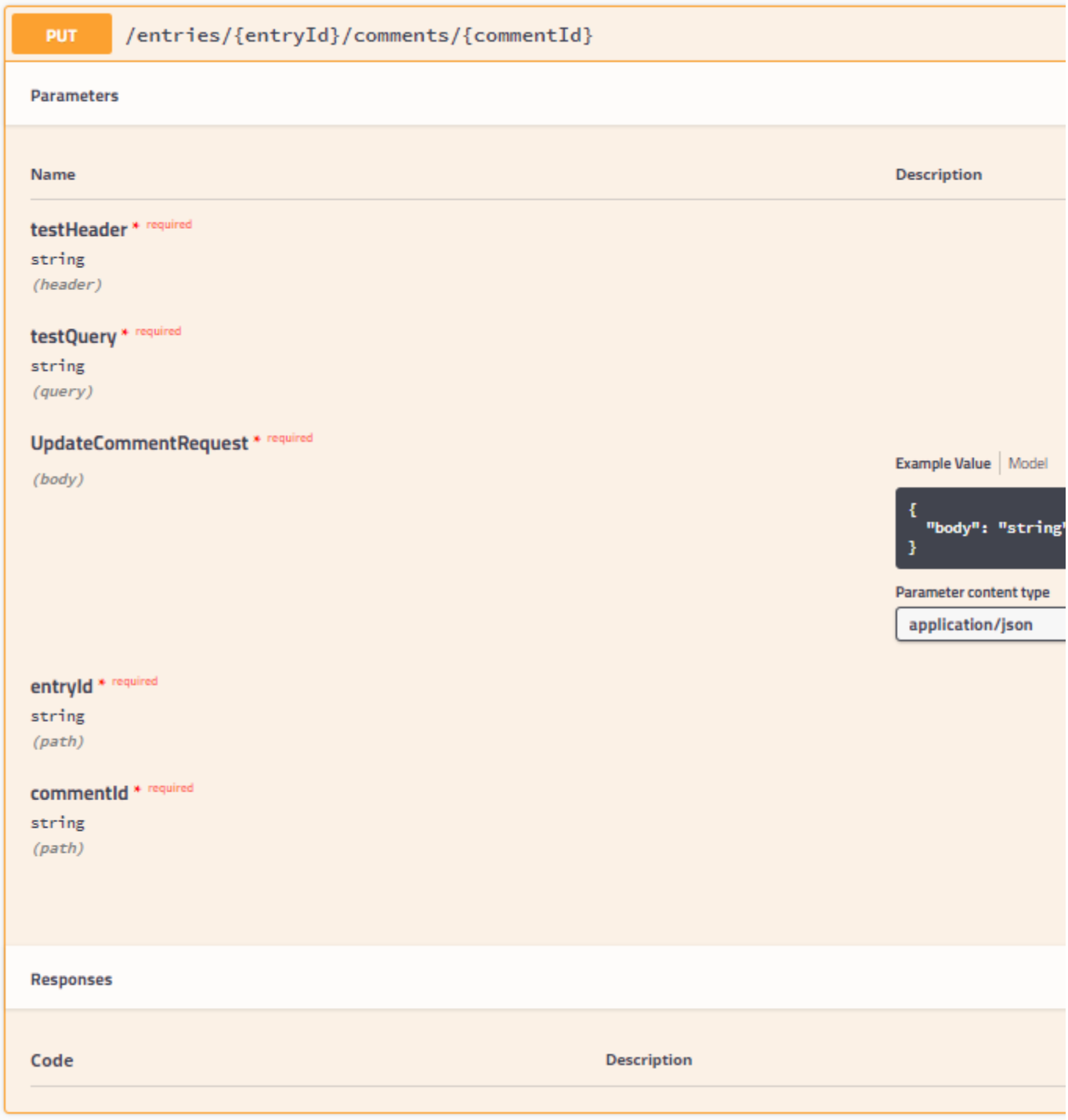
The `@Param` decorator in our example contained a string value indicating which URL parameter to use for the `comment` parameter of our controller method. When the Nest.js swagger module encounters this decorator with the provided string, it is able to determine the name of the URL parameter and includes it in the swagger document along with the type provided for the method parameter. However, we could have also written the controller method without passing a string to the `@Param` decorator to get an object containing all of the URL parameters. If we do this, Nest.js will only be able to determine the names and types of the URL parameters if we use a class as the type for the `comment` parameter or use the `@ApiImplicitParam` decorator provided by the Nest.js swagger module on the controller method. Let's create a new class to describe our URL params and see how it affects the swagger UI.

```
export class UpdateCommentParams {
      @ApiModelProperty()
      public entryId: string;

      @ApiModelProperty()
      public commentId: string;
}
```

In the `UpdateCommentParams` class, we have created a single property and used the `@ApiModelProperty`decorator so the Nest.js swagger module knows to include the properties with their types in the swagger document. Do not try to split the `entryId` out into it's own class and extend it because the Nest.js swagger module will not be able to pickup the properties of the extended class. It is also important that the names of the properties used in the class matches the names used in the `@Controller` and `@Put` decorators. We can change our comment to use the new class.

```
@Put('comments/:commentId')
public async update(
      @Body() body: UpdateCommentRequest,
      @Param() params: UpdateCommentParams,
      @Query('testQuery') testQuery: string,
      @Headers('testHeader') testHeader: string
) {
}
```

We have changed the controller so that all path parameters are provided to the controller method's `params` parameter as an object.



The swagger UI has been updated to show the comment put API takes two required URL parameters: `entryId` and `commentId`. If you will be writing APIs that use a single parameter in your method controller to hold all of the URL parameters, your preferred method of informing the Nest.js swagger module is what you should expect as URL parameters. Using a class as the type for your URL parameters not only informs the Nest.js swagger module of the URL parameters, it also helps in writing your application by providing type checking and code auto-completion.

If, however, you don't want to make a new class to use as the type for your URL parameters, use an interface, or one or more of the URL parameters are in a Nest.js guard, or middleware, or in a custom decorator, but not in the controller method. You can still inform the Nest.js swagger module about the URL parameters using the `@ApiImplicitParam` decorator.

```
@Put('comments/:commentId')
@ApiImplicitParam({ name: 'entryId' })
public async update(
      @Body() body: UpdateCommentRequest,
      @Param('commentId') comment: string,
      @Query('testQuery') testQuery: string,
      @Headers('testHeader') testHeader: string
) {
}
```

If a path param is required to reach the controller method, but the controller method does not use the param specifically, the Nest.js swagger module will not include it in the swagger document unless the controller method is decorated with the `@ApiImplicitParam` decorator. Use the decorator once for each path parameter that is necessary to reach the controller method, but it isn't used in the controller itself.

```
@Put('comments/:commentId')
@ApiImplicitParam({ name: 'entryId' })
@ApiImplicitParam({ name: 'commentId' })
public async update(
      @Body() body: UpdateCommentRequest,
      @Query('testQuery') testQuery: string,
      @Headers('testHeader') testHeader: string
) {
}
```

For example, the above controller, being a part of the comment controller, requires two path parameters: `entryId` and `commentId`. Since the controller does not contain any `@Param` decorators in the method parameters, `@ApiImplicitParam` is used to describe both path params.

The `@ApiImplicitParam` decorator can take several additional properties in an object passed to the decorator that will further define the URL parameter in the swagger document.

- **name**: A string containing the name of the URL parameter. This decorator property is the only one required.
- **description**: A string that can be used to describe what the URL parameter should contain or what it is used for.
- **required**: A boolean indicating if the URL parameter is required.
- **type**: A string containing one of the types defined in the swagger specification. Classes and objects should not be used.

@Query

The `@Query` decorator in our example contained a string value indicating which query parameter to use for the `testQuery` parameter of our controller method. When the Nest.js swagger module encounters this decorator with the provided string, it is able to determine the name of the query parameter and includes it in the swagger document along with the type provided for the method parameter. However, we could have also wrote the controller method without passing a string to the `@Query` decorator to get an object containing all the query parameters. If we do this, Nest.js will only be able to determine the names and types of the query parameters if we use a class as the type for the `testQuery` parameter or use the `@ApiImplicitQuery` decorator provided by the Nest.js swagger module on the controller method. Let's create a new class to describe our query params and see how it affects the Swagger UI.

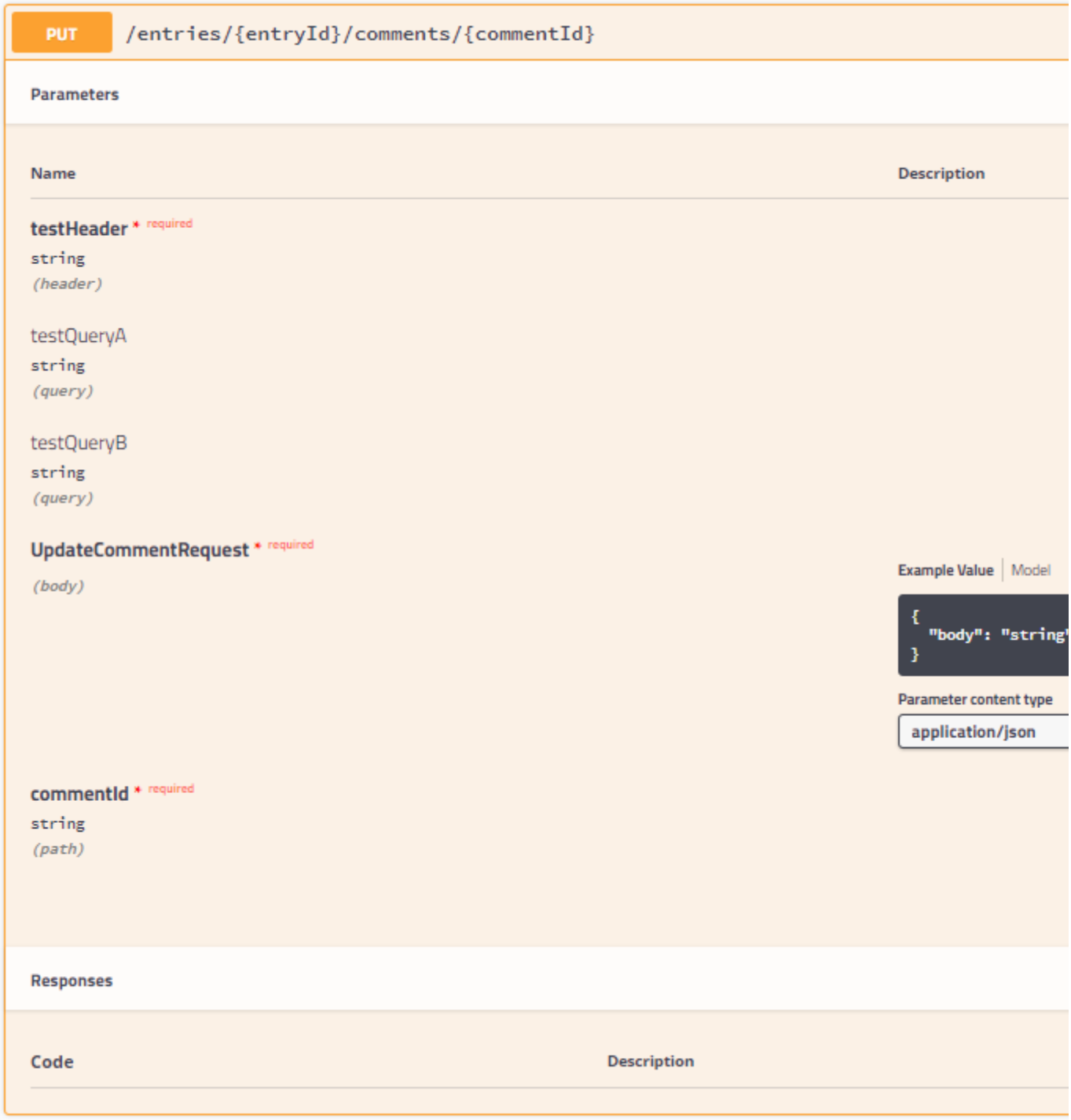
```
export class UpdateCommentQuery {
    @ApiModelPropertyOptional()
    public testQueryA: string;

    @ApiModelPropertyOptional()
    public testQueryB: string;
}
```

In the `UpdateCommentQuery` class, we have created two properties and used the `@ApiModelPropertyOptional` decorator so the Nest.js swagger module knows to include these properties with their types in the swagger document. We can change our comment and put the controller method to use the new class.

```
@Put('comments/:commentId')
public async update(
    @Body() body: UpdateCommentRequest,
    @Param('commentId') comment: string,
```

```
        @Query() queryParameters: UpdateCommentQuery,
        @Headers('testHeader') testHeader: string
) {
}
```

We have changed the controller so that all query parameters are provided to the controller method's `queryParameters` parameter as an object.



The swagger UI has been updated to show the comment, and the `put` API takes two optional query parameters: `testQueryA` and `testQueryB`. If you will be writing APIs that will use a single parameter in your method controller to hold all of the query parameters, this should be your preferred method of informing the Nest.js swagger module you are expecting as query parameters. Using a class as the type for your query parameters not only informs the Nest.js swagger module of the query parameters, it also helps in writing your application by providing type checking and code auto-completion.

However, if you do not wish to make a new class to use as the type for your query parameters, you use an interface, or the query parameters are used in a Nest.js guard or middleware in a custom decorator, but not in the controller method. You can still inform the Nest.js swagger module about the query parameters using the `@ApiImplicitQuery` decorator.

```
@Put('comments/:commentId')
@ApiImplicitQuery({ name: 'testQueryA' })
@ApiImplicitQuery({ name: 'testQueryB' })
public async update(
      @Param('commentId') comment: string,
      @Body() body: UpdateCommentRequest,
      @Query() testQuery: any,
      @Headers('testHeader') testHeader: string
) {
}
```

If a query param is required to reach the controller method, but the controller method does not use the query param specifically, the Nest.js swagger module will not include it in the swagger document unless the controller method is decorated with the `@ApiImplicitQuery` decorator. Use the decorator once for each query parameter that is necessary to reach the controller method, but is not not used in the controller itself.

```
@Put('comments/:commentId')
@ApiImplicitQuery({ name: 'testQueryA' })
@ApiImplicitQuery({ name: 'testQueryB' })
public async update(
      @Param('commentId') comment: string,
      @Body() body: UpdateCommentRequest,
      @Headers('testHeader') testHeader: string
) {
}
```

For example, the above controller requires two query parameters: `testQueryA` and `testQueryB`. Since the controller does not contain any `@Query` decorators in the method parameters, `@ApiImplicitQuery` is used to describe both query params.

The `@ApiImplicitQuery` decorator can take several additional properties in an object passed to the decorator that will further define the query parameter in the swagger document.

- **name**: A string containing the name of the query parameter. This decorator property is the only one required.
- **description**: A string that can be used to describe what the query parameter should contain or what it is used for.
- **required**: A boolean indicating if the query parameter is required.
- **type**: A string containing one of the types defined in the swagger specification. Classes and objects should not be used.
- **isArray**: A boolean indicating if the model property should take an array of values. If the model does take an array of values, be sure to include this value in the decorator or the Nest.js swagger module will not know to represent the model property as an array.
- **collectionFormat**: Maps to the swagger specification **collectionFormat** setting. This is used to depict how a model properties array values should be formatted. Possible values are:
  - **csv**: comma separated values `foo,bar`
  - **ssv**: space separated values `foo bar`
  - **tsv**: tab separated values `foo\tbar`
  - **pipes**: pipe separated values `foo|bar`
  - **multi**: corresponds to multiple parameter instances instead of multiple values for a single instance foo=bar&foo=baz. This is valid only for parameters in “query” or “formData”.

@Headers

The `@Headers` decorator in our example contained a string value indicating which request header value to use for the `testHeader` parameter of our controller method. When the Nest.js swagger module encounters this decorator with the

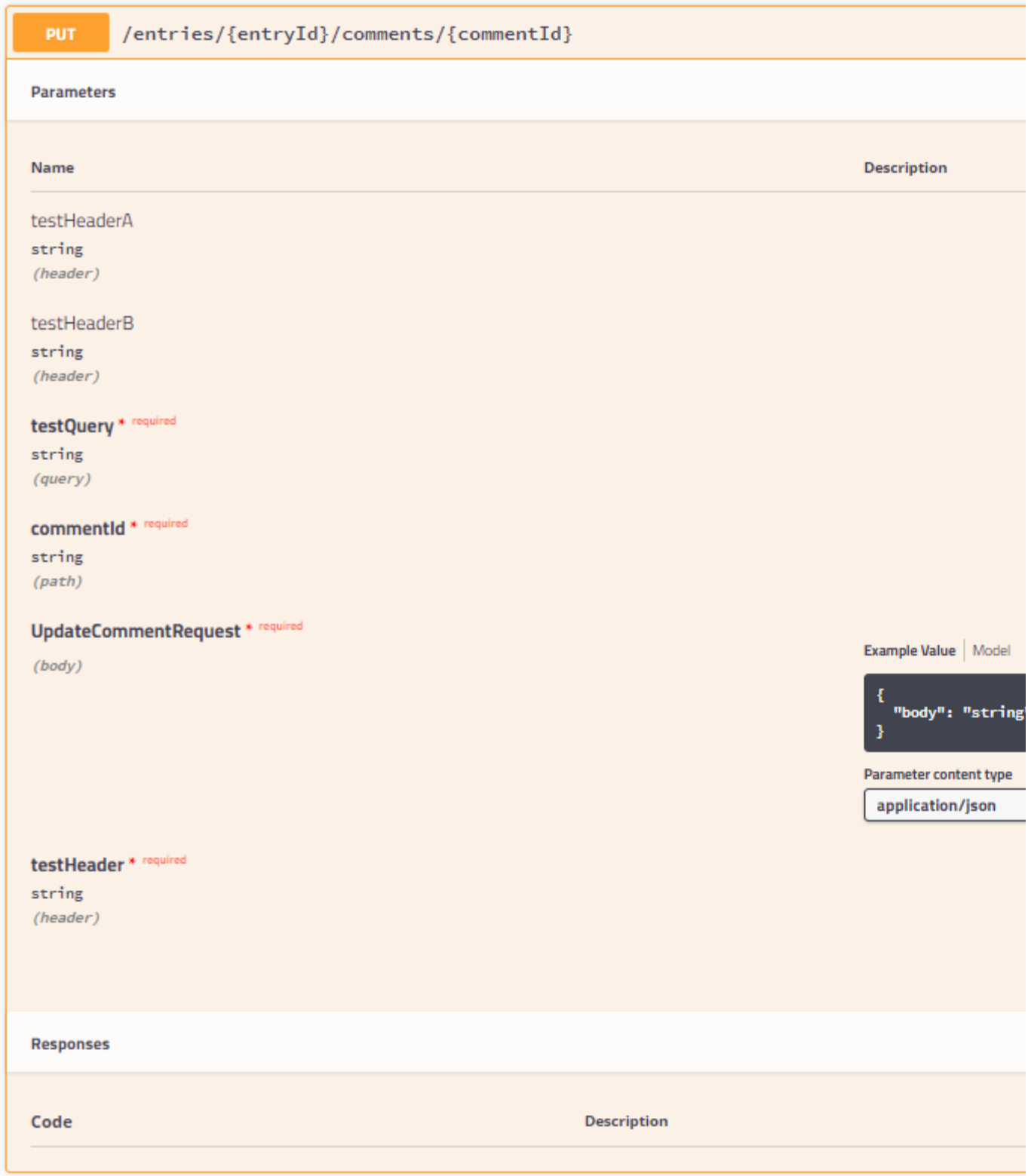
provided string, it is able to determine the name of the request header and includes it in the swagger document along with the type provided for the method parameter. However, we could have also written the controller method without passing a string to the `@Headers` decorator to get an object containing all the request headers. If we do this, Nest.js will only be able to determine the names and types of the request headers if we use a class as the type for the `testHeader` parameter or use the `@ApiImplicitHeader` decorator provided by the Nest.js swagger module on the controller method. Let's create a new class to describe our query params and see how it affects the swagger UI.

```
export class UpdateCommentHeaders {
      @ApiModelPropertyOptional()
      public testHeaderA: string;

      @ApiModelPropertyOptional()
      public testHeaderB: string;
}
```

In the `UpdateCommentHeaders` class, we have created two properties and used the `@ApiModelPropertyOptional` decorator so the Nest.js swagger module knows to include these properties with their types in the swagger document. We can change our comment `put` controller method to use the new class.

```
@Put('comments/:commentId')
public async update(
      @Body() body: UpdateCommentRequest,
      @Param('commentId') comment: string,
      @Query('testQuery') testQuery: string,
      @Headers() headers: UpdateCommentHeaders
) {
}
```

We have changed the controller so that all request parameters the controller expects are provided to the controller method's `queryParameters` parameter as an object.



The swagger UI has been updated to show the comment `put` API expects two headers: `testHeaderA` and `testHeaderB`. If you will be writing APIs that will use a single parameter in your method controller to hold all of the expected headers, and this should be your preferred method of informing the Nest.js swagger module that you are expecting as query parameters. Using a class as the type for

your expected headers not only informs the Nest.js swagger module of the headers, it also helps in writing your application by providing type checking and code auto-completion.

If, however, you do not wish to make a new class to use as the type for your expected headers, you use an interface, or the headers are used in a Nest.js guard or middleware or in a custom decorator, but not in the controller method. You can still inform the Nest.js swagger module about the query parameters using the `@ApiImplicitHeader` or the `@ApiImplicitHeaders` decorators.

```
@Put('comments/:commentId')
@ApiImplicitHeader({ name: 'testHeader' })
public async update(
      @Body() body: UpdateCommentRequest,
      @Param('commentId') comment: string,
      @Query('testQuery') testQuery: string,
      @Headers() headers: any
) {
}
```

If a header is required to reach the controller method, but the controller method does not use the header specifically. The Nest.js swagger module will not include it in the swagger document unless the controller method is decorated with the `@ApiImplicitHeader` or `@ApiImplicitHeaders` decorators. Use the `@ApiImplicitHeader` decorator once for each header, or the `@ApiImplicitHeaders` decorator once to describe all the headers. This is necessary to reach the controller method but it isn't used in the controller itself.

```
@Put('comments/:commentId')
@ApiImplicitHeader({ name: 'testHeaderA' })
@ApiImplicitHeader({ name: 'testHeaderB' })
public async update(
      @Body() body: UpdateCommentRequest,
      @Param('commentId') comment: string,
      @Query('testQuery') testQuery: string,
) {
}

@Put('comments/:commentId')
@ApiImplicitHeader([
      { name: 'testHeaderA' },
      { name: 'testHeaderB' }
```

```
])
public async update(
      @Body() body: UpdateCommentRequest,
      @Param('commentId') comment: string,
      @Query('testQuery') testQuery: string,
) {
}
```

For example, the above controllers requires two headers: `testHeaderA` and `testHeaderB`. Since the controller does not contain and `@Headers` decorators in the method parameters, `@ApiImplicitHeader`, and `@ApiImplicitHeaders` that is used to describe both headers.

The `@ApiImplicitHeader` and `@ApiImplicitHeaders` decorators can take several additional properties in an object or array of objects, respectively, passed to the decorator that will further define the query parameter in the swagger document.

- **name**: A string containing the name of the header. This decorator property is the only one required.
- **description**: A string that can be used to describe what the header should contain or what it is used for.
- **required**: A boolean indicating if the header is required.

Note: the `@ApiImplicitHeaders` decorator is just a shortcut for using the `@ApiImplicitHeader` decorator multiple times. If you need to describe multiple headers, use `@ApiImplicitHeaders`. Also, you should not use these headers to describe authentication mechanics. There are other decorators for that.

Authentication

It is very likely that you will need to have some form of authentication in your application at some point. The blog example application uses a `username` and `password` combination to authenticate a user and provides a JSON web token to allow the user to access the APIs. However you decide to setup authentication, one thing is for sure: you will require either query parameters or headers to maintain an authentication state and you will most likely use Nest.js middleware or guards to check a user's authentication state. You do this because writing that code in every controller method creates a lot of code duplication and would complicate every controller method.

If your application does require authentication, first, be sure your document settings are properly configured using the `addOAuth2` or the `addBearerAuth` method. Refer back to the **Document Settings** section if you are unsure of what those methods do.

In addition to setting the authentication scheme for the swagger document, you should also use the `ApiBearerAuth` and/or the `ApiOAuth2Auth` decorators on the controller class or controller methods. When used on an entire controller class, these decorators inform the Nest.js swagger module that all controller methods require authentication. If not all controller methods require authentication, you will need to decorate the individual controller methods that do.

```
@Put('comments/:commentId')
@ApiBearerAuth()
public async update(
      @Body() body: UpdateCommentRequest,
      @Param('commentId') comment: string,
      @Query('testQuery') testQuery: string,
      @Headers('testHeader') testHeader: string
) {
}
```

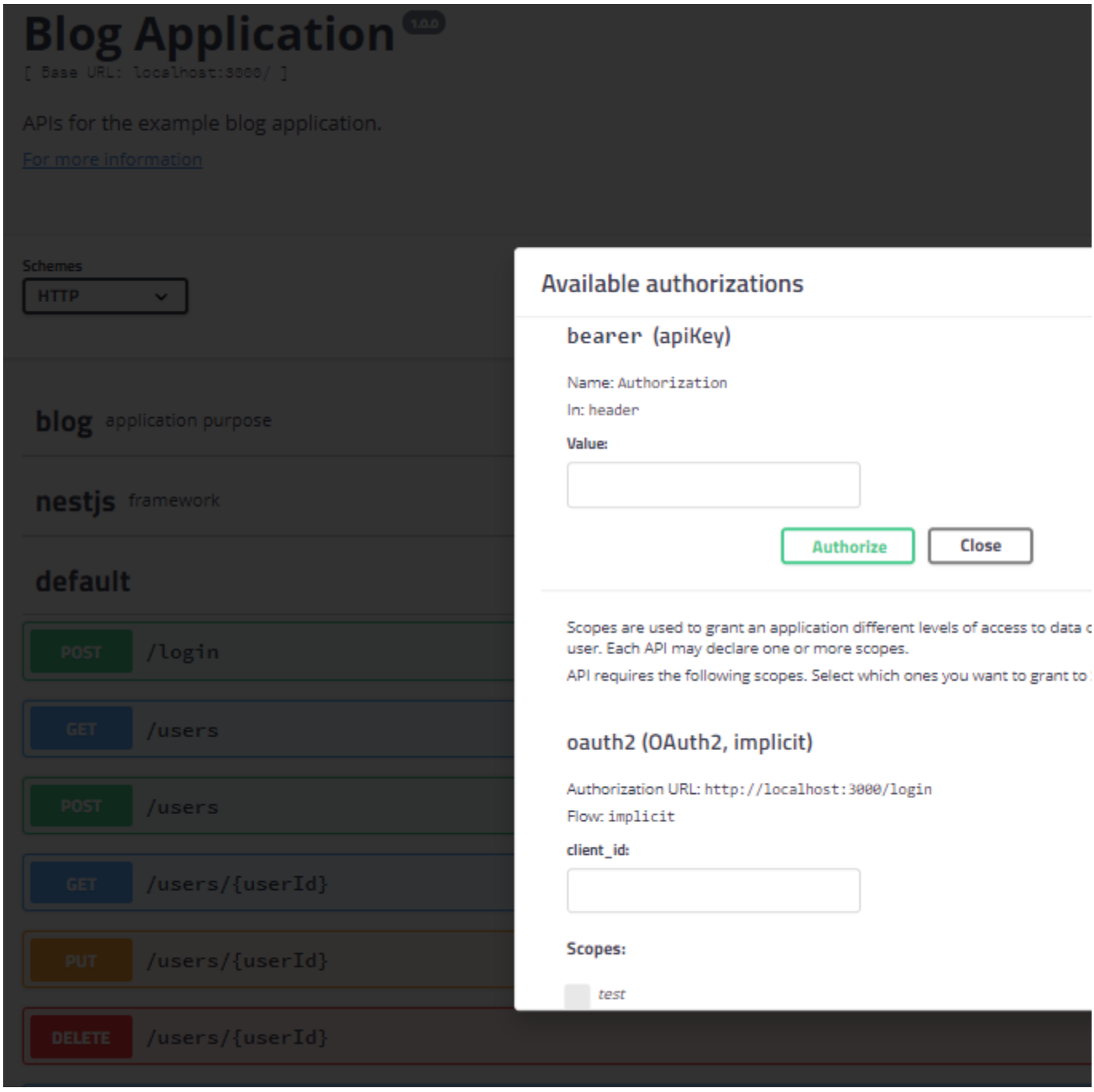
This example depicts a single controller method, API, that requires a bearer token to be able to use the API.

```
@Put('comments/:commentId')
@ApiOAuth2Auth(['test'])
public async update(
      @Body() body: UpdateCommentRequest,
      @Param('commentId') comment: string,
      @Query('testQuery') testQuery: string,
      @Headers('testHeader') testHeader: string
) {
}
```

This example depicts a single controller method, API, that requires a specific set of OAuth2 roles to be able to use the API. The `@ApiOAuth2Auth` decorator takes an array of all the roles the user should have in order to have access to the API.

These decorators are used in conjunction with the `ApiBearerAuth` and `ApiOAuth2Auth` document settings to build a form the user can enter their credentials, either an API key or an Oauth key, and select their roles, if OAuth2 is being used, inside the swagger UI. These values are then

passed in the appropriate places, either as query params or header values, when the user executes a specific API.



Clicking the **Authorize** button at the top of the swagger UI page will open the authorizations form. For a bearer token, log into the application and copy the auth token returned into the space provided in the swagger UI authorization. The token should be in the form of `Bearer <TOKEN VALUE>`. For OAuth2 authentication, enter your credentials and select the roles you are requesting. Clicking the **Authorize**button will save the credentials for use when executing the APIs in the swagger UI.

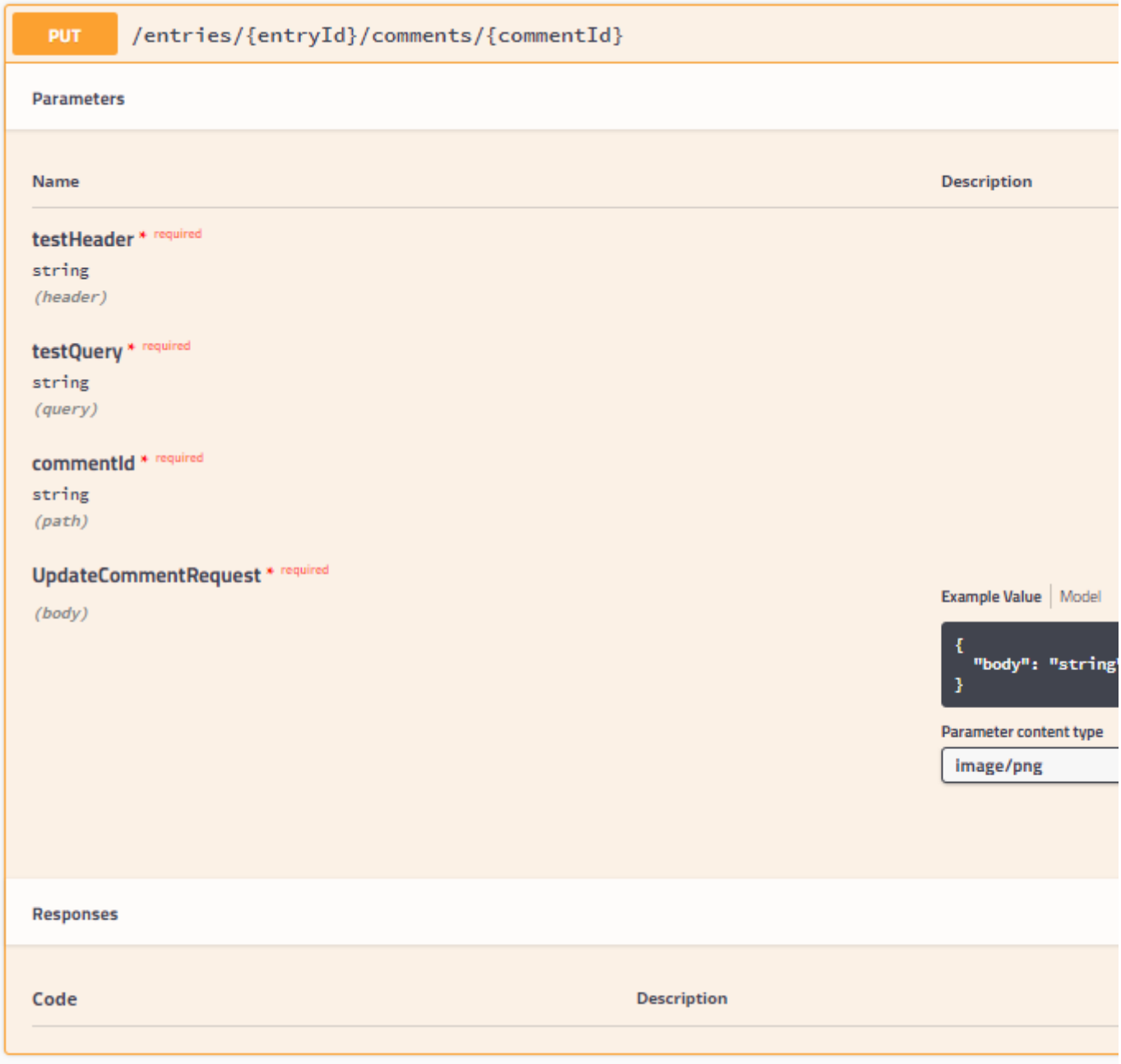
API request and response decorators

So far we have been primarily focused on decorating controllers, so the Nest.js swagger module can build a swagger document containing all of the inputs our APIs expect or could use. The Nest.js swagger module also contains decorators that can be used to describe with what and how APIs respond and the format of the content it expects to receive and send. These decorators help form a complete picture of how a specific API works when looking at the swagger document or when using the swagger UI.

All of the APIs we have covered in our example blog application follow a typical modal of accepting inputs in the form of JSON. However, it is possible that an application may need to take a different input type, often referred to as a MIME type. For example, we could allow users of our example blog application to upload an avatar image. An image cannot easily be represented as JSON so we would need to build an API that takes an input MIME type of `image/png`. We can ensure this information is present in our application's swagger document by using the `@ApiConsumes` decorator.

```
@Put('comments/:commentId')
@ApiConsumes('image/png')
public async update(
      @Body() body: UpdateCommentRequest,
      @Param('commentId') comment: string,
      @Query('testQuery') testQuery: string,
      @Headers('testHeader') testHeader: string
) {
}
```

In this example, we have used the `@ApiConsumes` decorator to inform the Nest.js swagger module that the comment `put` API expects to receive a png image.



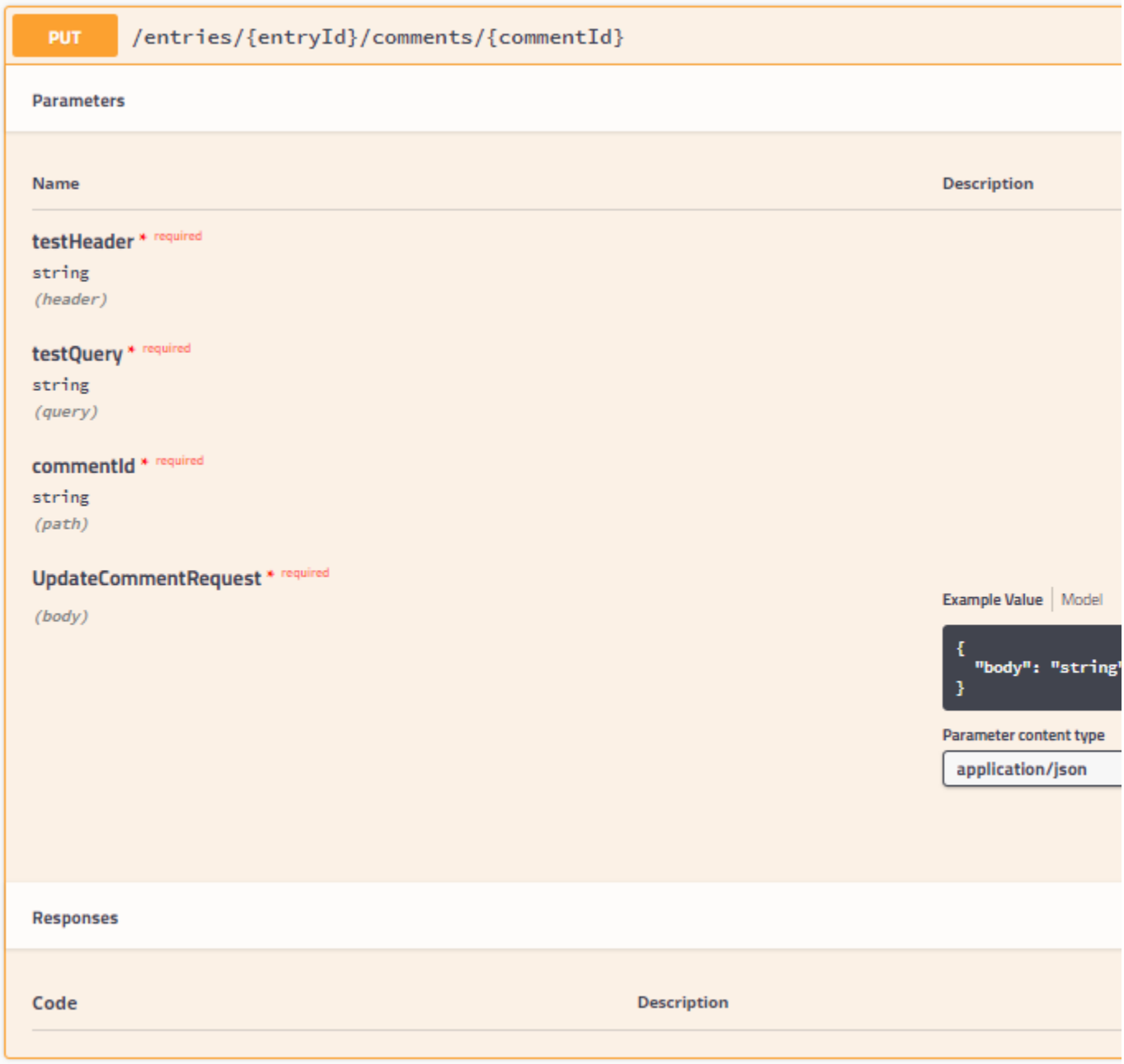
The Swagger UI now shows the **Parameter content type** drop down as `image/png`. The `@ApiConsumes`decorator can take any number of MIME types as parameters. Multiple values in the decorator will result in the **Parameter content type** drop down containing multiple values with the first value always being the default. If a controller is dedicated to handling a specific MIME type, like `application/json`, the `@ApiConsumes` decorator can be placed on the controller class instead of on every single controller method. However, if your APIs will be consuming JSON, the decorator can be left off and the Nest.js swagger module will default the APIs to `application/json`.

In addition to consuming various MIME data types, APIs can also respond with various MIME data types. For example, our fictitious avatar upload API may

store the images in a database or cloud storage provider. Such storage locations may not be directly accessible to users so an API can be created to lookup and return the avatar image for any user. We can use the `@ApiProduces` decorator to let the Nest.js swagger module know the API returns data using the `image/png` MIME type.

```
@Put('comments/:commentId')
@ApiProduces('image/png')
public async update(
      @Body() body: UpdateCommentRequest,
      @Param('commentId') comment: string,
      @Query('testQuery') testQuery: string,
      @Headers('testHeader') testHeader: string
) {
}
```

In this example, we have used the `@ApiProduces` decorator to inform the Nest.js swagger module that the comment `put` API expects to return a png image.



The Swagger UI now shows the **Response content type** drop down as `image/png`. The `@ApiProduces`decorator can take any number of MIME types as parameters. Multiple values in the decorator will result in the **Response content type** drop down containing multiple values with the first value always being the default. If a controller is dedicated to handling a specific MIME type, like `application/json`, the `@ApiConsumes` decorator can be placed on the controller class instead of on every single controller method. However, if your APIs will be consuming JSON, the decorator can be left off and the Nest.js swagger module will default the APIs to `application/json`.

Request and response MIME type information goes a long way to informing the end use of the swagger document, and how to use an API and how an API

works. However, we have not fully documented everything an API can respond with. For example, what data values are contained in the APIs response body and what are the potential HTTP status codes it could return? Such information can be provided using the `@ApiResponse` decorator.

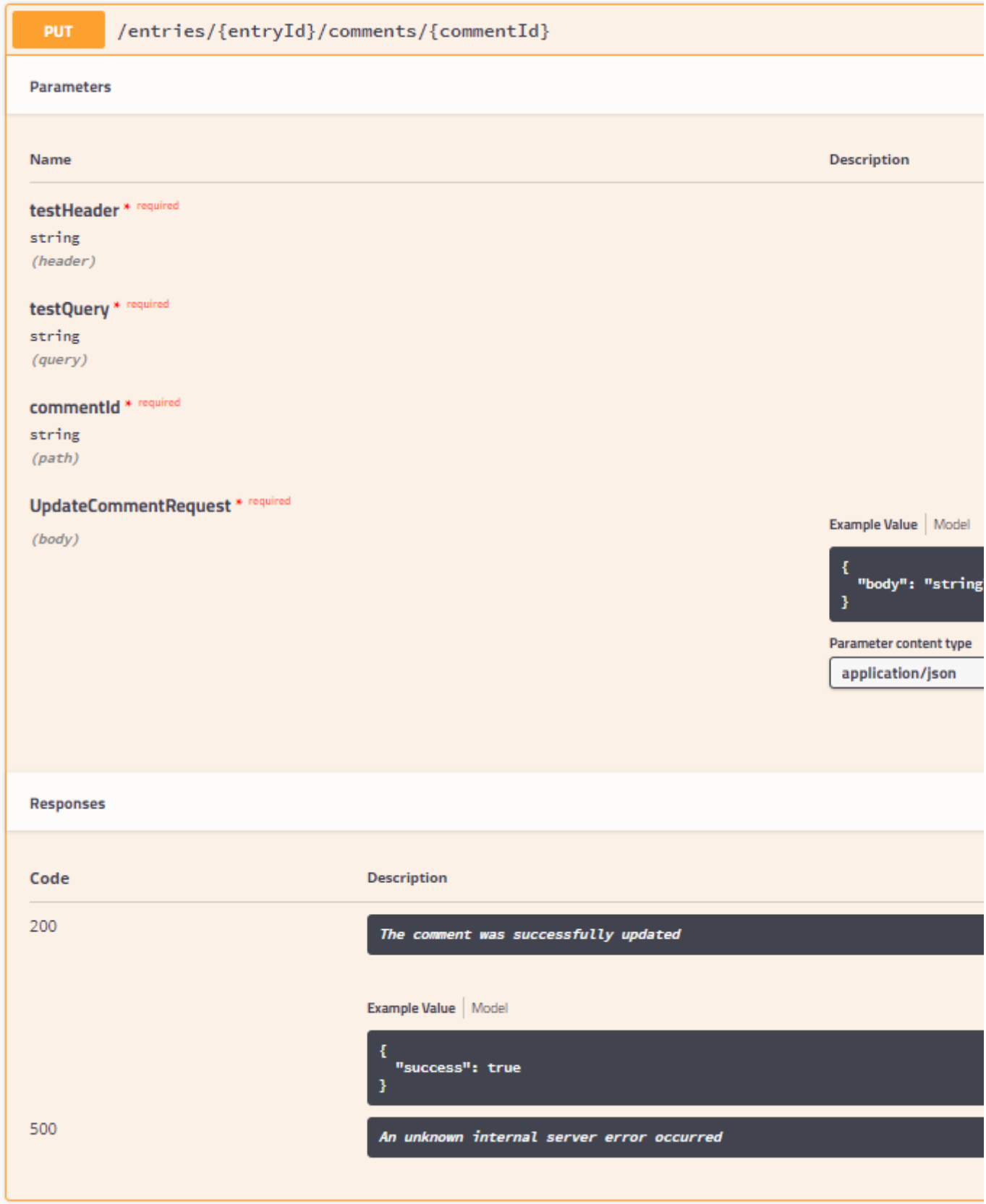
The `@ApiResponse` decorator can be placed on individual controller methods or on the controller class. The Nest.js swagger module will collect the controller class level decorator data and pair it with the controller method decorator data to produce a list of possible responses each individual API could produce.

```
@Controller('entries/:entryId')
@ApiResponse({
      status: 500,
      description: 'An unknown internal server error occurred'
})
export class CommentController {
      @Put('comments/:commentId')
      @ApiResponse({
            status: 200,
            description: 'The comment was successfully updated',
            type: UpdateCommentResponse
      })
      public async update(
            @Body() body: UpdateCommentRequest,
            @Param('commentId') comment: string,
            @Query('testQuery') testQuery: string,
            @Headers('testHeader') testHeader: string
      ) {
      }
}
```

In this example, we decorated the comment controller so that all the APIs will contain a generic response for internal server errors. The update controller method has been decorated so that responses with a status code of `200` indicate the comment was successfully updated. The type is another data model created to provide the Nest.js swagger module with information regarding the individual properties in the response body.

```
export class UpdateCommentResponse {
   @ApiModelPropertyOptional()
   public success?: boolean;
}
```

The `UpdateCommentResponse` data model contains one optional property, `success`, that could be used to further relay to the UI that the comment was updated successfully.



The swagger UI now lists both possible responses in the **Responses** section of the API card. Use the `@ApiResponse` decorator to inform user's of your APIs about the different success and error scenarios they may need to deal with when using the APIs. The `@ApiResponse` decorator can take additional properties in the object passed to it.

- **status**: A number containing the HTTP status code the API will respond with. This decorator property is the only one required.
- **description**: A string that can be used to describe what the response indicates or how the user should react when the response is encountered.
- **type**: Use a data model class any of the data types defined in the swagger specification to inform users of the API what they can expect in the response body. If you use the **isArray** property, it indicated the response will be an array of values with the provided type.
- **isArray**: A boolean indicating if the response body will contain an array of values. If the response body will contain an array of values, be sure to include this value in the decorator or the Nest.js swagger module will not know to represent the response body an array.

API metadata decorators

If you work through any Nest.js project and properly decorate all the controllers and controller methods with the decorators we have covered so far, the swagger document the Nest.js swagger module produces will have every technical detail a user of the APIs would need to understand and use the APIs. The last two decorators we will cover in this chapter simply provide more metadata for the swagger document. The swagger UI will use this metadata to produce a cleaner UI, but functionality will not change.

The first decorator we will cover is `@ApiOperation`. Don't confuse this decorator with the HTTP method decorators like `@Put`. This decorator is used to provide a **title**, **description**, and unique identifier called an **operationId** for individual controller methods.

```
@Put('comments/:commentId')
@ApiOperation({
      title: 'Comment Update',
```

```
        description: 'Updates a specific comment with new content',
        operationId: 'commentUpdate'
    })
    public async update(
        @Body() body: UpdateCommentRequest,
        @Param('commentId') comment: string,
        @Query('testQuery') testQuery: string,
        @Headers('testHeader') testHeader: string
    ) {
    }
```

In this example, we have provided a brief **title** and a much longer **description** of the comment put API. The **title** should be kept short, less than 240 character, and is used to populate the `summary` portion of the swagger specification. While the **description** in the example is short, use verbose descriptions in your own projects. This should convey why a user would use the API or what they accomplish through the use of the API. The **operationId** must be kept unique per the swagger documentation. The value could be used in various swagger codegen projects to reference the specific API.

PUT `/entries/{entryId}/comments/{commentId}` Comment Update

Updates a specific comment with new content

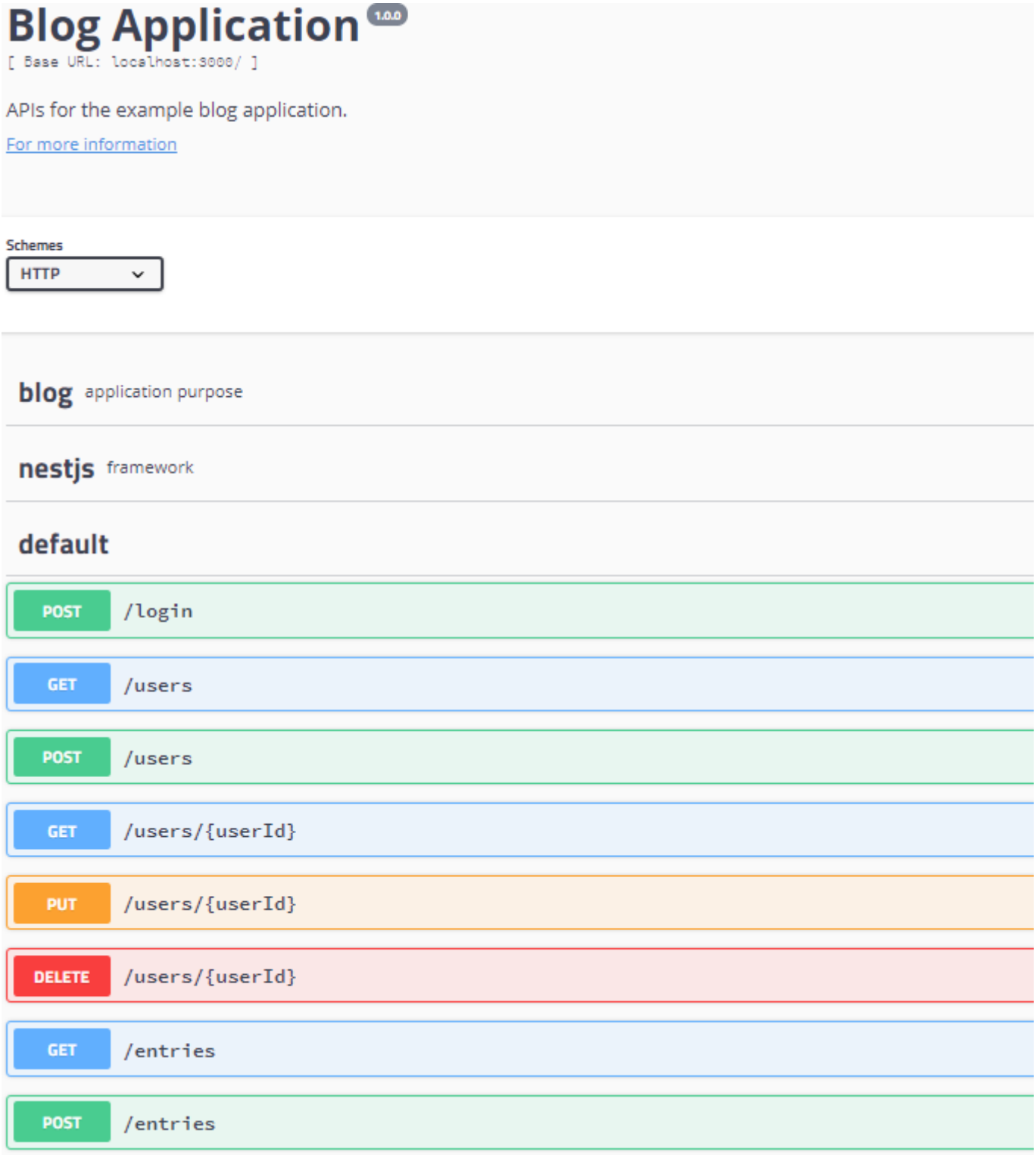
Parameters

Name	Description
testHeader * required string *(header)*	
testQuery * required string *(query)*	
commentId * required string *(path)*	
UpdateCommentRequest * required *(body)*	Example Value | Model `{ "body": "string" }` Parameter content type application/json

Responses

Code	Description
200	*The comment was successfully updated* Example Value | Model `{ "success": true }`
500	*An unknown internal server error occurred*

In the swagger UI, we can see the values we have passed to the `@ApiOperation` decorator, and how they are used to fill in additional details of the API card. The **title** is placed in the header next to the API path.
The **description** is the first bit of information in the API card following the header. We can see how using a long **title** and **description** negatively impacts the API card header, but works very well in the API card body.

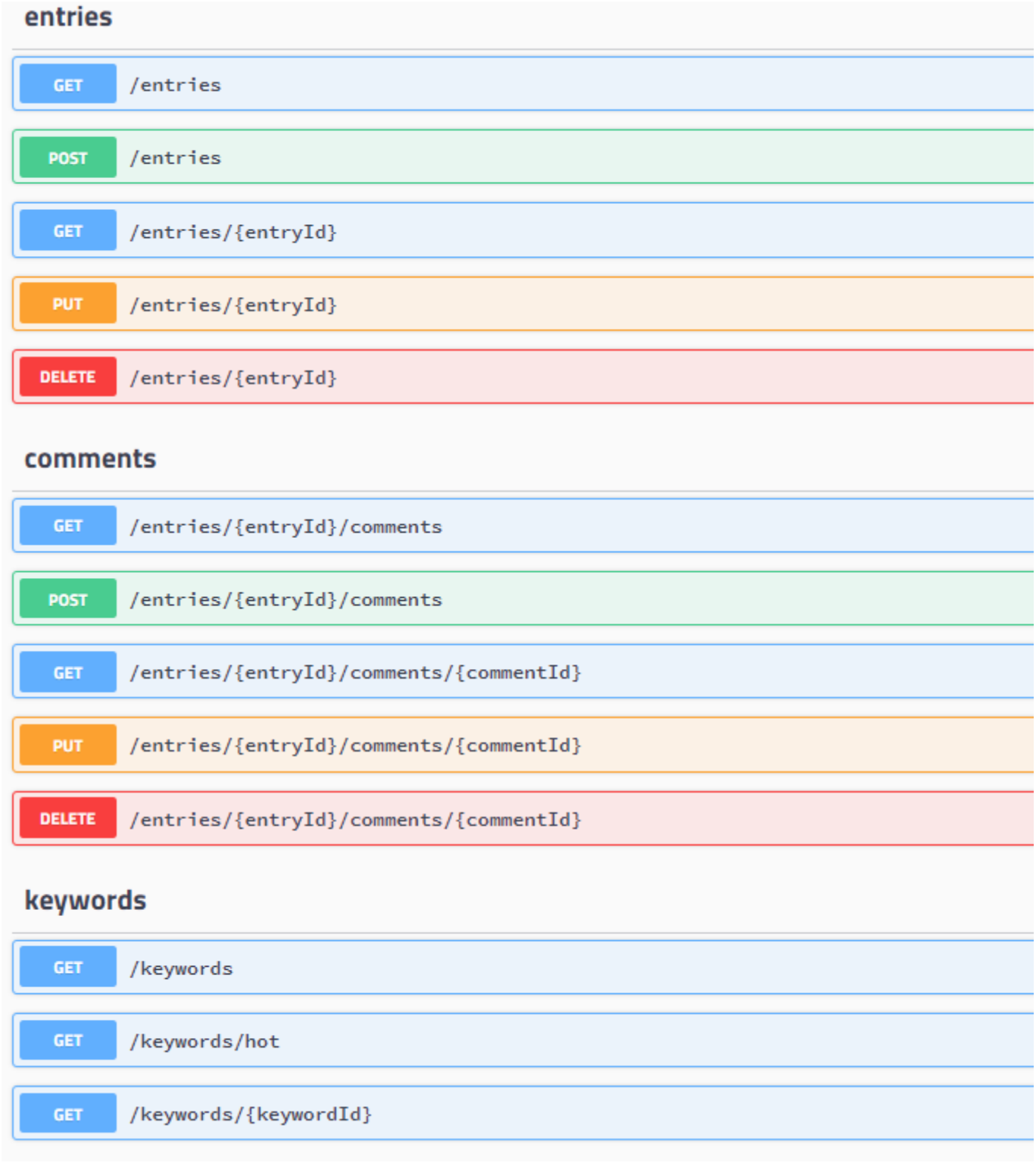


Looking at the overall swagger UI application, we can see that all of the APIs for the example blog application are grouped together. While this works, it would be nicer to group the APIs based on the operations they perform or the resources, comment, entry, or keyword that they act upon. This is what the `@ApiUseTags` decorator is used for.

The `@ApiUseTags` decorator can be placed on a controller class or individual controller methods and can take any number of string parameters. These values will be placed in the swagger document for each individual API.

```
@Controller('entries/:entryId')
@ApiUseTags('comments')
export class CommentController {


}
```

In this example, we decorated the comment controller class so that all of the controller methods will be given the `comments` tag.



The swagger UI now groups the APIs using the tags. This ensures like APIs are grouped and provides a little spacing between each group to produce a nicer UI. The groups are also expandable and collapsible giving users the option of hiding APIs they may not be interested in.

Saving the swagger document

We have covered all of the available decorators in the Nest.js swagger module and the decorators already available in Nest.js to produce a swagger document and expose the swagger UI. This works great when your APIs are primarily used by developers in their own projects or when testing the APIs on a local development server or in a staging environment. For APIs that are primarily used for a specific front-end application, you may not wish to expose the swagger UI for the general public to be able to use. In such a case, you can still produce a swagger document for storage and use it on your own or your teams other projects.

To accomplish this, we will write a new Typescript file that can be executed as part of a build chain. We will use the `fs-extras` NodeJS module to make writing our file to disk much simpler.

```
import * as fs from 'fs-extra';

async function writeDoc() {
      const app = await NestFactory.create(AppModule);
      const document = SwaggerModule.createDocument(app, swaggerOptions);

      fs.ensureDirSync(path.join(process.cwd(), 'dist'));
      fs.writeJsonSync(path.join(process.cwd(), 'dist', 'api-doc.json'),
document, { spaces: 2 });
}

writeDoc();
```

You can place this file in the root of your project or in the source directory and use an NPM script entry to execute it or run it using NodeJS. The example code will use the Nest.js swagger module to build a swagger document and `fs-extras` to write the document to the `dist` directory as a JSON file.

Summary

In this chapter, we covered how the Nest.js swagger module makes use of the existing decorators you use in your application to create a swagger v2 specification document. We also covered all the additional decorators the Nest.js swagger module provides to enhance the information in the swagger

document. We also setup the example blog application to expose the swagger UI.

Use the Nest.js swagger module to not only document your application's controllers, but to also provide UI for testing your application. If you fully document your application, the swagger UI can be an excellent replacement UI or provide an easy testing area that you or your users can use instead of having to watch for network calls in your applications real UI. The swagger UI can also be a great substitute for tools like Postman.

If you don't wish to use the Swagger UI or expose your swagger document with you application in a production environment, remember you can always write the file to disk as a separate build job of your application. This allows you to store and use the document in a number of ways, most notably with Swagger Codegen.

The next chapter will bring you up to speed on Command Query Responsibility Separation (CQRS).

Chapter 12. Command Query Responsibility Separation (CQRS)

Up to this point in this book, we have worked to put together a simple blog application using the CRUD pattern: Create, Retrieve, Update, and Delete. We have done an excellent job of ensuring services are handling our business logic and our controllers are simply gateways into those services. The controllers take care of validating the request and then pass the request to the service for processing. In a small application like this, CRUD works wonderfully.

But what happens when we are dealing with a large scale application that may have unique and complex business logic for saving data? Or maybe we would like to initiate some logic in the background so the UI is able to call APIs without having to wait for all the business logic to finish. These are areas where CQRS makes sense. CQRS can be used to isolate and break apart complex business logic, initiate that business logic synchronously or asynchronously, and compose the isolated pieces to solve new business problems.

Nest.js implements this pattern by providing two separate streams for processing the command aspect of CQRS: a command and an event bus, with some sugar in the form of sagas. In this chapter, we will tackle the problem of adding keyword metadata to our blog entries. We could certainly do this using the CRUD pattern, but having the UI make multiple API calls to store a blog entry and all it's keywords, or even having our blog entry module perform this, would complicate the business logic of the UI and our application.

Instead, we will convert the blog entry module to use CQRS commands, and the command `bus` to perform all data persistance, removing it from the service in the blog entry module. A new entity and module will be created for our keywords. The keyword entity will maintain a last updated timestamp and a reference to all associated entries. Two new APIs will be created: one to provide a list of "hot keywords" and one to provide a list of all entries associated with a keyword.

To ensure the UI does not suffer any performance loss, all keyword entity operations will be done asynchronously. Keywords will be stored on the blog entry entity as a string to provide the UI a quick reference without having to query the keyword table in the database. Before getting started, be sure you ran `npm install @nestjs/cqrs` in your project.

To see a working example, remember you can clone the accompanying Git repository for this book:

```
git clone https://github.com/backstopmedia/nest-book-example.git
```

Entry module commands

To make the business logic around changes to entry models easier to extend, we will first need to extract out the methods in the module's services that update the database as individual commands. Let's start with converting the blog entry `create` method to a command in Nest.js CQRS fashion.

```
export class CreateEntryCommand implements ICommand {
     constructor(
          public readonly title: string,
          public readonly content: string,
          public readonly userId: number
     ) {}
}
```

Our command is a simple object that implemented the `ICommand` interface. The `ICommand` interface is used internally by Nest.js to indicate an object is a command. This file is typically created in a sub-directory of our module with a pattern similar to `commands/impl/`. Now that we have one example done, let's finish up the remaining commands for the comment module.

```
export class UpdateEntryCommand implements ICommand {
     constructor(
          public readonly id: number,
          public readonly title: string,
          public readonly content: string
     ) {}
}

export class DeleteEntryCommand implements ICommand {
     constructor(
          public readonly id: number
     ) {}
}
```

Notice some differences with the update and delete commands? For the update command, we need to know which database model we are updating. Likewise, for the delete command, we only need to know the id of the database model we are deleting. In both cases, having the `userId` does not make sense since a blog entry can never be moved to another user and the `userId` has no influence on the deletion of a blog entry.

Command handlers

Now that we have commands for our database write operations, we need some command handlers. Each command should have an accompanying handler in a one-to-one fashion. The command handler is much like our current blog entry service. It will take care of all the database operations. Typically, the command handlers are placed in a sub-directory of the module similar to `commands/handlers`.

```
@CommandHandler(CreateEntryCommand)
export class CreateEntryCommandHandler implements
ICommandHandler<CreateEntryCommand> {
     constructor(
          @Inject('EntryRepository') private readonly entryRepository:
typeof Entry,
          @Inject('SequelizeInstance') private readonly
sequelizeInstance
     ) { }

     async execute(command: CreateEntryCommand, resolve: () => void) {
     }
}
```

Command handlers are simple classes with a single method, `execute`, that is responsible for handling the command. Implementing the `ICommandHandler<CreateEntryCommand>` interface helps ensure we write our command handler correctly. Nest.js uses the `@CommandHandler` annotation in our example to know this class is meant to handle our new `CreateEntryCommand` command.

Since the command handler is going to be a drop-in replacement for our module's service, the command handler will also need access to our database. This may differ depending on what ORM you are using and how your application is configured. Our command handler doesn't actually do anything at

this point. In fact, using it would break the application since we have not implemented the details of the `execute` method.

```
async execute(command: CreateEntryCommand, resolve: () => void) {
    await this.sequelizeInstance.transaction(async transaction => {
        return await this.entryRepository.create<Entry>(command, {
            returning: true,
            transaction
        });
    });

    resolve();
}
```

If you are following along with the example project, you may notice our `execute` method looks almost like the `create` method of the blog entry service. In fact, almost all of the code for the command handler is a direct copy from the blog entry service. The big difference is that we do not return a value. Instead, the `execute` method of all command handlers takes a callback method as their second argument.

Nest.js allows us to do a couple of different things with the callback it provides to the `execute` method. In our example, we use the ORM to create and persist a new blog entry. Once the transaction resolves, we call the `resolve` callback to let Nest.js know our command is done executing. If this looks familiar, it is because behind the scenes Nest.js is wrapping our `execute` in a Promise and passing in the promise's own `resolve` callback as the second argument to our `execute` method.

Notice that we do not get a `reject` callback passed to our command handler. Nest.js does not perform any type of error handling when invoking command handlers. Since our command handler is invoking our ORM to store data in a database, it is very possible that an exception could be thrown. If this happens with the way our command handler is currently wrote, depending on the version of NodeJS being used, an `UnhandledPromiseRejectionWarning` warning being logged to the console and the UI will be stuck waiting for the API to return until it times out. To prevent this, we should wrap our command handler logic in a `try...catch` block.

```
async execute(command: CreateEntryCommand, resolve: () => void) {
    try {
        await this.sequelizeInstance.transaction(async transaction => {
            return await this.entryRepository.create<Entry>(command, {
```

```
                    returning: true,
                    transaction
                });
            });
        } catch (error) {

        } finally {
            resolve();
        }
    }
```

Notice we invoke the `resolve` callback in the `finally` block. This is done to ensure that, no matter the outcome, the command handler will complete execution and the API will finish processing. But what happens when an exception is thrown from our ORM. The blog entry wasn't saved to the database, but since the API controller did not know an error occurred, it will return a 200 HTTP status to the UI. To prevent this, we can catch the error and pass that as an argument to the `resolve` method. This might break with the CQRS pattern but it is better to let the UI know something went wrong than assume the blog entry was saved.

```
async execute(command: CreateEntryCommand, resolve: (error?: Error) =>
void) {
    let caught: Error;

    try {
        await this.sequelizeInstance.transaction(async transaction => {
            return await this.entryRepository.create<Entry>(command, {
                returning: true,
                transaction
            });
        });
    } catch (error) {
        caught = error
    } finally {
        resolve(caught);
    }
}
```

Note: Nest.js does not provide any stipulation for when the callback method must be invoked. We could invoke the callback at the beginning of the `execute` method. Nest.js would return processing back to the controller so

the UI is immediately updated and process the remaining pieces of the `execute`method afterwards.

Let's finish converting our blog entry module to CQRS by creating commands to handle updating and deleting blog entries from the database.

```
@CommandHandler(UpdateEntryCommand)
export class UpdateEntryCommandHandler implements
ICommandHandler<UpdateEntryCommand> {
     constructor(
          @Inject('EntryRepository') private readonly entryRepository:
typeof Entry,
          @Inject('SequelizeInstance') private readonly
sequelizeInstance: Sequelize,
          private readonly databaseUtilitiesService:
DatabaseUtilitiesService
     ) { }

     async execute(command: UpdateEntryCommand, resolve: (error?: Error)
=> void) {
          let caught: Error;

          try {
               await this.sequelizeInstance.transaction(async transaction
=> {
                    let entry = await
this.entryRepository.findById<Entry>(command.id, { transaction });
                    if (!entry) throw new Error('The blog entry was not
found.');

                    entry = this.databaseUtilitiesService.assign(
                         entry,
                         {
                              ...command,
                              id: undefined
                         }
                    );
                    return await entry.save({
                         returning: true,
                         transaction,
                    });
```

```
            });
        } catch (error) {
            caught = error
        } finally {
            resolve(caught);
        }
    }
}
```

The command handler for our `UpdateEntryCommand` command needs a couple changes from what we have in the blog entry service. Since our command contains all of the data for the blog entry being updated, including the `id`, we need to strip out the `id` and apply the remaining values in the command to the entity before saving it back to the database. Just like our last command handler, we use a `try...catch` to handle errors and pass any thrown exceptions back as an argument to the `resolve`callback.

```
@CommandHandler(DeleteEntryCommand)
export class DeleteEntryCommandHandler implements
ICommandHandler<DeleteEntryCommand> {
    constructor(
        @Inject('EntryRepository') private readonly entryRepository:
typeof Entry,
        @Inject('SequelizeInstance') private readonly
sequelizeInstance: Sequelize
    ) { }

    async execute(command: DeleteEntryCommand, resolve: (error?: Error)
=> void) {
        let caught: Error;

        try {
            await this.sequelizeInstance.transaction(async transaction
=> {
                return await this.entryRepository.destroy({
                    where: { id: command.id },
                    transaction,
                });
            });
        } catch (error) {
            caught = error
```

```
            } finally {
                resolve(caught);
            }

            resolve();
        }
    }
```

The command handler for our `DeleteEntryCommand` is pretty much a copy of the `delete` method in the blog entry service. We now have three new commands and their accompanying handlers. All that's left is to hook them up and begin using them. Before we can do that, we must decide on where we are going to invoke these new commands.

Invoking command handlers

Documentation and the general consensus around separation of concerns within NodeJS applications would probably dictate that we invoke our commands from the blog entry service. Doing so would leave the controller as simple as it is now but would not simplify the service at all. Alternatively, the approach we will be taking is to reduce the complexity of our service so it is used strictly for data retrieval and invoke our commands from the controller. No matter the route taken, the first step in making use of the new commands is to inject the Nest.js `CommandBus`.

Note: Where you plan to use your commands, whether it be the controller or service, makes no difference for the implementation. Feel free to experiment.

```
@Controller()
export class EntryController {
    constructor(
        private readonly entryService: EntryService,
        private readonly commandBus: CommandBus
    ) { }

    @Post('entries')
    public async create(@User() user: IUser, @Body() body: any, @Res()
res) {
        if (!body || (body && Object.keys(body).length === 0)) return
res.status(HttpStatus.BAD_REQUEST).send('Missing some information.');
```

```
            const error = await this.commandBus.execute(new
CreateEntryCommand(
                  body.title,
                  body.content,
                  user.id
            ));

            if (error) {
                  return
res.status(HttpStatus.INTERNAL_SERVER_ERROR).send(result);
            } else {
                  return res.set('location',
`/entries/${result.id}`).status(HttpStatus.CREATED).send();
            }
      }
```

The above example incorporates two key changes. First, we have added `commandBus` to the constructor. Nest.js will take care of injecting an instance of the `commandBus` into this variable for us. The last change is to the `create` controller method. Instead of invoking the `create` method in the blog entry service, we create and execute a new `CreateEntryCommand` using the command bus. The remaining implementation details for the blog entry controller follow almost the same pattern as the `create` method.

```
@Controller()
export class EntryController {
      constructor(
            private readonly entryService: EntryService,
            private readonly commandBus: CommandBus
      ) { }

      @Get('entries')
      public async index(@User() user: IUser, @Res() res) {
            const entries = await this.entryService.findAll();
            return res.status(HttpStatus.OK).json(entries);
      }

      @Post('entries')
      public async create(@User() user: IUser, @Body() body: any, @Res()
res) {
```

```
        if (!body || (body && Object.keys(body).length === 0)) return
res.status(HttpStatus.BAD_REQUEST).send('Missing some information.');

        const error = await this.commandBus.execute(new
CreateEntryCommand(
            body.title,
            body.content,
            user.id
        ));

        if (error) {
            return
res.status(HttpStatus.INTERNAL_SERVER_ERROR).send(result);
        } else {
            return res.set('location',
`/entries/${result.id}`).status(HttpStatus.CREATED).send();
        }
    }

    @Get('entries/:entryId')
    public async show(@User() user: IUser, @Entry() entry: IEntry, @Res()
res) {
        return res.status(HttpStatus.OK).json(entry);
    }

    @Put('entries/:entryId')
    public async update(@User() user: IUser, @Entry() entry: IEntry,
@Param('entryId') entryId: number, @Body() body: any, @Res() res) {
        if (user.id !== entry.userId) return
res.status(HttpStatus.NOT_FOUND).send('Unable to find the entry.');
        const error = await this.commandBus.execute(new
UpdateEntryCommand(
            entryId,
            body.title,
            body.content,
            user.id
        ));

        if (error) {
            return
res.status(HttpStatus.INTERNAL_SERVER_ERROR).send(error);
```

```
            } else {
                return res.status(HttpStatus.OK).send();
            }
        }

        @Delete('entries/:entryId')
        public async delete(@User() user: IUser, @Entry() entry: IEntry,
@Param('entryId') entryId: number, @Res() res) {
            if (user.id !== entry.userId) return
res.status(HttpStatus.NOT_FOUND).send('Unable to find the entry.');
            const error = await this.commandBus.execute(new
DeleteEntryCommand(entryId));

            if (error) {
                return
res.status(HttpStatus.INTERNAL_SERVER_ERROR).send(error);
            } else {
                return res.status(HttpStatus.OK).send();
            }
        }
}
```

You can see from the example that the controller has been updated so the blog entry service is only used for retrievals and all modification methods now dispatch commands on the command bus. The last thing we need to configure is the blog entry module. To make this easier, let's first setup a Typescript barrel to export all our handlers as a single variable.

```
export const entryCommandHandlers = [
    CreateEntryCommandHandler,
    UpdateEntryCommandHandler,
    DeleteEntryCommandHandler
];
```

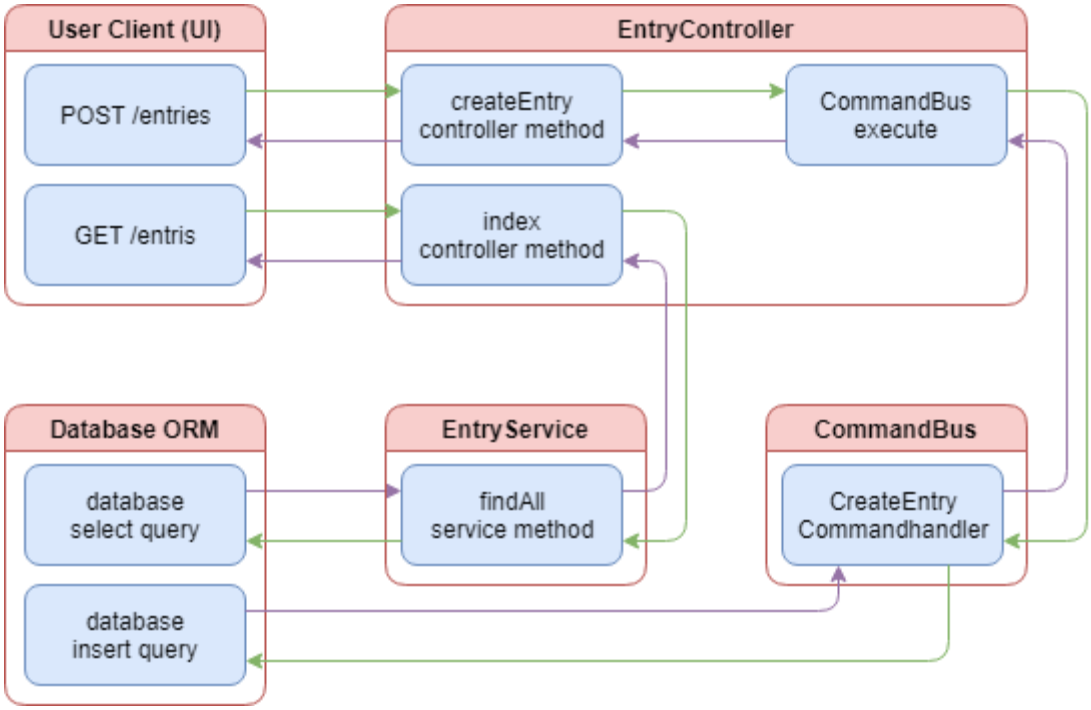
Import the barrel into the blog entry module and hook up the module to the command bus.

```
@Module({
    imports: [CQRSModule, EntryModule],
    controllers: [CommentController],
    components: [commentProvider, CommentService,
...CommentCommandHandlers],
```

```
    exports: [CommentService]
})
export class EntryModule implements NestModule, OnModuleInit {
    public constructor(
        private readonly moduleRef: ModuleRef,
        private readonly commandBus: CommandBus
    ) {}

    public onModuleInit() {
        this.commandBus.setModuleRef(this.moduleRef);
        this.commandBus.register(CommentCommandHandlers);
    }
}
```

To hook up our module to the command bus, we import `CQRSModule` into our module definition and inject the `ModuleRef` and `CommandBus` into the module class constructor. The module class also needs to implement the `OnModuleInit` interface. Finally, the magic happens in the `onModuleInit` lifecycle hook. Nest.js will execute this method immediately after instantiating our module class. Inside the method, we use `setModuleRef` and `register` to register the blog entry command handlers to the command bus that was created for this module.

Note: If you followed along and implemented the invocation of the commands in the controller, you can remove the `create`, `update`, and `delete` methods from the comment service.



The above diagram provides a visual representation of how the command and query aspects of the entry controller have been divided. When a user sends a request to the `create` controller method, processing is executed through the CQRS command bus, but still uses the ORM to update the database. When the users wishes to retrieve all the entries, the entry controller makes use of the `EntryService` that then uses the ORM to query the database. All commands, the `C` in CQRS, are now processed through the command bus while all queries, the `Q` in CQRS, are continue to be processed through the entry service.

Linking keywords with events

Now that we have shown the basics of creating commands and using the command bus in Nest.js CQRS, we need to tackle storing keywords that are associated with a blog entry. Keywords can be added when a blog entry is created and removed later. We could create a new entity for our keywords and have the entry entity maintain a one-to-many relationship with the keyword entity. This would, however, require our database lookups to pull in more data from more tables and the response sent back to the UI would become larger. Instead, let's start off with just storing the keywords as a JSON string on the blog entry entity. To do this, we will need to update the blog entry entity and add a new field.

```
@Table(tableOptions)
export class Entry extends Model<Entry> {
```

```
    @Column({
        type: DataType.TEXT,
        allowNull: true,

    })
    public keywords: string;

}
```

The ORM definition for the new database column will depend on the ORM and database server you are using. Here, we are using the `TEXT` data type. This data type is widely supported in many different database servers and offers a large limit to the amount of data we can store. For example, Microsoft SQL Server limits this field to a maximum of 2^30 - 1 characters, while Postgres does not impose a limit. Since we are using an ORM with migrations, we will also need to create a migration script. If you are unsure of how to do this, reference back to the TypeORM or Sequelize chapters.

```
export async function up(sequelize) {
    // language=PostgreSQL
    await sequelize.query(`
        ALTER TABLE entries ADD COLUMN keywords TEXT;
    `);

    console.log('*keywords column added to entries table*');
}

export async function down(sequelize) {
    // language=PostgreSQL
    await sequelize.query(`
        ALTER TABLE entries DROP COLUMN keywords;
    `);
}
```

If you are following along, your entries database table should now have a keywords column. Testing the `index` API in the blog entries controller should now return objects with a keywords value. We still need to update the blog entry commands, command handlers, and controller to process keywords for new and updated blog entries.

```
@Controller()
```

```
export class EntryController {

     @Post('entries')
     public async create(@User() user: IUser, @Body() body: any, @Res()
res) {
          if (!body || (body && Object.keys(body).length === 0)) return
res.status(HttpStatus.BAD_REQUEST).send('Missing some information.');

          const error = await this.commandBus.execute(new
CreateEntryCommand(
               body.title,
               body.content,
               body.keywords,
               user.id
          ));

          if (error) {
               return
res.status(HttpStatus.INTERNAL_SERVER_ERROR).send(result);
          } else {
               return res.set('location',
`/entries/${result.id}`).status(HttpStatus.CREATED).send();
          }
     }

     @Put('entries/:entryId')
     public async update(@User() user: IUser, @Entry() entry: IEntry,
@Param('entryId') entryId: number, @Body() body: any, @Res() res) {
          if (user.id !== entry.userId) return
res.status(HttpStatus.NOT_FOUND).send('Unable to find the entry.');
          const error = await this.commandBus.execute(new
UpdateEntryCommand(
               entryId,
               body.title,
               body.content,
               body.keywords,
               user.id
          ));

          if (error) {
```

```
                return
res.status(HttpStatus.INTERNAL_SERVER_ERROR).send(error);
            } else {
                return res.status(HttpStatus.OK).send();
            }
        }
}
```

The blog entry controller will accept the keywords as an array of strings. This will help keep the UI simple and prevent the UI from having to perform arbitrary string parsing.

```
export class CreateEntryCommand implements ICommand, IEntry {
    constructor(
        public readonly title: string,
        public readonly content: string,
        public readonly keywords: string[],
        public readonly userId: number
    ) {}
}

export class UpdateEntryCommand implements ICommand, IEntry {
    constructor(
        public readonly id: number,
        public readonly title: string,
        public readonly content: string,
        public readonly keywords: string[],
        public readonly userId: number
    ) {}
}
```

The `CreateEntryCommand` and `UpdateEntryCommand` commands are updated to accept a new property `keywords`. We maintain the string array type so the processing of the commands is offloaded to the command handler.

```
@CommandHandler(CreateEntryCommand)
export class CreateEntryCommandHandler implements
ICommandHandler<CreateEntryCommand> {

    async execute(command: CreateEntryCommand, resolve: (error?: Error)
=> void) {
        let caught: Error;
```

```
            try {
                await this.sequelizeInstance.transaction(async transaction
=> {
                    return await this.EntryRepository.create<Entry>({
                        ...command,
                        keywords: JSON.stringify(command.keywords)
                    }, {
                        returning: true,
                        transaction
                    });
                });
            } catch (error) {
                caught = error;
            } finally {
                resolve(caught);
            }
        }
}

@CommandHandler(UpdateEntryCommand)
export class UpdateEntryCommandHandler implements
ICommandHandler<UpdateEntryCommand> {

    async execute(command: UpdateEntryCommand, resolve: (error?: Error)
=> void) {
        let caught: Error;

        try {
            await this.sequelizeInstance.transaction(async transaction
=> {
                let comment = await
this.EntryRepository.findById<Entry>(command.id, { transaction });
                if (!comment) throw new Error('The comment was not
found.');

                comment = this.databaseUtilitiesService.assign(
                    comment,
                    {
                        ...command,
                        id: undefined,
```

```
                                        keywords: JSON.stringify(command.keywords)
                                    }
                                );
                                return await comment.save({
                                    returning: true,
                                    transaction,
                                });
                            });
                        } catch (error) {
                            caught = error;
                        } finally {
                            resolve(caught);
                        }
                    }
                }
```

Both the `CreateEntryCommandHandler` and `UpdateEntryCommandHandler` command handlers have been updated to convert the keywords string array into a JSON string. Keywords also need to be stored individually in their own table with a list of blog entries they apply to and the last updated date. To do this, we will need to make a new Nest.js module with an entity. We will come back later to add more functionality. First, create the new entity.

```
const tableOptions: IDefineOptions = { timestamp: true, tableName:
'keywords' } as IDefineOptions;

@DefaultScope({
    include: [() => Entry]
})
@Table(tableOptions)
export class Keyword extends Model<Keyword> {
    @PrimaryKey
    @AutoIncrement
    @Column(DataType.BIGINT)
    public id: number;

    @Column({
        type: DataType.STRING,
        allowNull: false,
        validate: {
```

```
                isUnique: async (value: string, next: any): Promise<any>
=> {
                    const isExist = await Keyword.findOne({ where: {
keyword: value } });
                    if (isExist) {
                        const error = new Error('The keyword already
exists.');
                        next(error);
                    }
                    next();
                },
            },
        })
        public keyword: string;

        @CreatedAt
        public createdAt: Date;

        @UpdatedAt
        public updatedAt: Date;

        @DeletedAt
        public deletedAt: Date;

        @BelongsToMany(() => Entry, () => KeywordEntry)
        public entries: Entry[];

        @BeforeValidate
        public static validateData(entry: Entry, options: any) {
            if (!options.transaction) throw new Error('Missing
transaction.');
        }
}
```

The `BelongsToMany` decorator is used to connect keywords to many different blog entries. We will not be placing a `BelongsToMany` column in the blog entry table since we are using a string column to keep lookups fast. The `() => KeywordEntry` parameter tells the ORM that we will be using the `KeywordEntry`entity to store the association. We will need to create the entity as well.

```
const tableOptions: IDefineOptions = { timestamp: true, tableName:
'keywords_entries', deletedAt: false, updatedAt: false } as
IDefineOptions;

@Table(tableOptions)
export class KeywordEntry extends Model<KeywordEntry> {
      @ForeignKey(() => Keyword)
      @Column({
            type: DataType.BIGINT,
            allowNull: false
      })
      public keywordId: number;

      @ForeignKey(() => Entry)
      @Column({
            type: DataType.BIGINT,
            allowNull: false
      })
      public entryId: number;

      @CreatedAt
      public createdAt: Date;
}
```

Our ORM will use the `@ForeignKey` decorators to link entries in this database table to the `keywords` and `entries` tables. We are also adding a `createdAt` column to help us find the latest keywords that have been linked to a blog entry. We will use this to create our list of "hot keywords." Next, create the migration script to add the new tables to the database.

```
export async function up(sequelize) {
      // language=PostgreSQL
      await sequelize.query(`
            CREATE TABLE "keywords" (
                  "id" SERIAL UNIQUE PRIMARY KEY NOT NULL,
                  "keyword" VARCHAR(30) UNIQUE NOT NULL,
                  "createdAt" TIMESTAMP NOT NULL,
                  "updatedAt" TIMESTAMP NOT NULL,
                  "deletedAt" TIMESTAMP
            );
            CREATE TABLE "keywords_entries" (
```

```
                "keywordId" INTEGER NOT NULL
                    CONSTRAINT "keywords_entries_keywordId_fkey"
                    REFERENCES keywords
                    ON UPDATE CASCADE ON DELETE CASCADE,
                "entryId" INTEGER NOT NULL
                    CONSTRAINT "keywords_entries_entryId_fkey"
                    REFERENCES entries
                    ON UPDATE CASCADE ON DELETE CASCADE,
                "createdAt" TIMESTAMP NOT NULL,
                UNIQUE("keywordId", "entryId")
            );
    `);

    console.log('*Table keywords created!*');
}

export async function down(sequelize) {
    // language=PostgreSQL
    await sequelize.query(`DROP TABLE keywords_entries`);
    await sequelize.query(`DROP TABLE keywords`);
}
```

Our migration script includes a unique constraint in the `keywords_entries` table to ensure we do not link the same keyword and blog entry more than once. The `ON DELETE CASCADE` portion of the `entryId`column definition will ensure that when we delete a blog entry, the keyword links will also be deleted. This means we do not have to create any code to handle unlinking keywords when blog entries are deleted. Be sure to add the new database entities to the database provider.

```
export const databaseProvider = {
    provide: 'SequelizeInstance',
    useFactory: async () => {
        let config;
        switch (process.env.NODE_ENV) {
            case 'prod':
            case 'production':
            case 'dev':
            case 'development':
            default:
                config = databaseConfig.development;
```

```
            }

            const sequelize = new Sequelize(config);
            sequelize.addModels([User, Entry, Comment, Keyword,
KeywordEntry]);
            /* await sequelize.sync(); */
            return sequelize;
        },
};
```

Finally, create the keyword provider and module.

```
export const keywordProvider = {
      provide: 'KeywordRepository',
      useValue: Keyword,
};

export const keywordEntryProvider = {
      provide: 'KeywordEntryRepository',
      useValue: KeywordEntry
};

@Module({
      imports: [],
      controllers: [],
      components: [keywordProvider, keywordEntryProvider],
      exports: []
})
export class KeywordModule {}
```

Now that we have a working keyword module, we can begin to think about how we want to construct the the application logic for storing keywords. To stay within the CQRS pattern, we could create new commands and command handlers in the keyword module. However, Nest.js imposes module isolation on all instances of the command bus. This means that the command handlers must be registered within the same module where the commands are executed. For example, if we attempted to execute a keyword command from the blog entry controller, Nest.js would throw an exception indicating there is no handler registered for the command. This is where events within Nest.js CQRS come to the rescue. The event bus is not isolated. In fact, the event bus allows events to be executed from any module, whether there is a handler registered for them or not.

Keyword events

Events can be thought of as commands with a few differences. Outside of not being module scoped, they are also asynchronous, they are typically dispatched by models or entities, and each event can have any number of event handlers. This makes them perfect for handling background updates to the keywords database table when blog entries are created and updated.

Before we start writing code, let's give some thought to how we want our application to work. When a new blog entry is created, the application needs to inform the keyword module that a blog entry has been associated with a keyword. We should leave it up to the keyword module to determine if the keyword is new and needs to be created or already exists and simply needs to be updated. The same logic should apply to updates made to blog entries but we can make our blog entry update process simpler if we do not try to determine which keywords are new and which have been removed. To support both scenarios, we should create a generic event to update all keyword links for the blog entry.

Now that we have a basic understanding of the logic we are trying to accomplish, we can build the event classes. Just like commands, the CQRS events feature requires basic classes for the events. Event files are typically created in a sub-directory of our module with a pattern similar to `events/impl/`.

```
export class UpdateKeywordLinksEvent implements IEvent {
     constructor(
          public readonly entryId: number,
          public readonly keywords: string[]
     ) { }
}
```

The event classes should look pretty similar to the command classes we wrote earlier in this chapter. The difference is the event classes implement the `IEvent` interface to let Nest.js know instances of these classes are CQRS events. We also need to setup handlers for these events. Just like command handlers, our event handlers will take care of all the database operations. Typically, the event handlers are placed in a sub-directory of the module similar to `events/handlers`.

```
@EventsHandler(UpdateKeywordLinksEvent)
export class UpdateKeywordLinksEventHandler implements
IEventHandler<UpdateKeywordLinksEvent> {
     constructor(
```

```
            @Inject('KeywordRepository') private readonly
keywordRepository: typeof Keyword,
            @Inject('SequelizeInstance') private readonly
sequelizeInstance: Sequelize,
      ) { }

      async handle(event: UpdateKeywordLinksEvent) {
      }
}
```

Event handlers are simple classes with a single method, `handle`, that is responsible for handling the event. Implementing the `IEventHandler<UpdateKeywordLinksEvent>` interface helps ensure we write our event handler correctly. Nest.js uses the `@EventsHandler` annotation in our example to know this class is meant to handle our new `UpdateKeywordLinksEvent` event.

One of the key differences in our event handlers compared to command handlers is that the event handler do not get a callback method as a second argument. Nest.js will invoke the `handle` method asynchronously. It will not wait for it to finish, it will not attempt to capture any return values, and it will not catch or handle any errors that may result from invoking our `handle` method. That's not to say we shouldn't still use a `try...catch` to prevent any kind of errors causing issues iss with NodeJS.

For the update links event handler, we should split out the logic into separate methods to make the class a little easier to read and manage. Let's write the `handle` method so it loops through all the keywords and ensures the keyword exists and the blog entry is associated with the keyword. Finally, we should ensure the blog entry is not associated with any keywords that are not in the event `keywords`array.

```
@EventsHandler(UpdateKeywordLinksEvent)
export class UpdateKeywordLinksEventHandler implements
IEventHandler<UpdateKeywordLinksEvent> {
      constructor(
            @Inject('KeywordRepository') private readonly
keywordRepository: typeof Keyword,
            @Inject('SequelizeInstance') private readonly
sequelizeInstance: Sequelize,
      ) { }
```

```
    async handle(event: UpdateKeywordLinksEvent) {
        try {
            await this.sequelizeInstance.transaction(async transaction
=> {
                let newKeywords: string[] = [];
                let removedKeywords: Keyword[] = [];

                const keywordEntities = await
this.keywordRepository.findAll({
                    include: [{ model: Entry, where: { id:
event.entryId }}],
                    transaction
                });

                keywordEntities.forEach(keywordEntity => {
                    if (event.keywords.indexOf(keywordEntity.keyword)
=== -1) {
                        removedKeywords.push(keywordEntity);
                    }
                });

                event.keywords.forEach(keyword => {
                    if (keywordEntities.findIndex(keywordEntity =>
keywordEntity.keyword === keyword) === -1) {
                        newKeywords.push(keyword)
                    }
                });

                await Promise.all(
                    newKeywords.map(
                        keyword =>
this.ensureKeywordLinkExists(transaction, keyword, event.entryId)
                    )
                );
                await Promise.all(
                    removedKeywords.map(
                        keyword => keyword.$remove('entries',
event.entryId, { transaction })
                    )
                );
            });
```

```
            } catch (error) {
                console.log(error);
            }
        }

        async ensureKeywordLinkExists(transaction: Transaction, keyword:
    string, entryId: number) {
            const keywordEntity = await
    this.ensureKeywordExists(transaction, keyword);
            await keywordEntity.$add('entries', entryId, { transaction });
        }

        async ensureKeywordExists(transaction: Transaction, keyword:
    string): Promise<Keyword> {
            const result = await
    this.keywordRepository.findOrCreate<Keyword>({
                where: { keyword },
                transaction
            });
            return result[0];
        }
    }
```

The event handler logic starts with finding all keywords the blog entry is currently linked to. We loop through those and pull out any that are not in the new keywords array. To find all new keywords, we loop trough the keywords array in the event to find those that are not in the `keywordEntities` array. The new keywords are processing through the `ensureKeywordLinkExists` method. The `ensureKeywordLinkExists` uses `ensureKeywordExists` to create or find the keyword in the keywords database table and adds the blog entry to the keywords entries array. The `$add` and `$remove` methods are provided by `sequelize-typescript` and are used to quickly add and remove blog entries without having to query for the blog entry. All processing is done using transactions to ensure any errors will cancel all database updates. If an error does happen, the database will become out of sync, but since we are dealing with metadata, it's not a big deal. We log the error out so application admins will know they need to re-sync the metadata.

Even though we only have one event handler, we should still create a Typescript barrel to export it in an array. This will ensure adding new events later is a simple process.

```
export const keywordEventHandlers = [
      UpdateKeywordLinksEventHandler,
      RemoveKeywordLinksEventHandler
];
```

Import the barrel in the keyword module and connect the event bus.

```
@Module({
      imports: [CQRSModule],
      controllers: [],
      components: [keywordProvider, ...keywordEventHandlers],
      exports: []
})
export class KeywordModule implements OnModuleInit {
      public constructor(
            private readonly moduleRef: ModuleRef,
            private readonly eventBus: EventBus
      ) {}

      public onModuleInit() {
            this.eventBus.setModuleRef(this.moduleRef);
            this.eventBus.register(keywordEventHandlers);
      }
}
```

In the module, import the `CQRSModule` and add `ModuleRef` and `EventBus` as constructor params. Implement the `OnModuleInit` interface and create the `onModuleInit` method. In the `onModuleInit` method, we set the module reference of event bus to the current module using `setModuleRef` and use `register` to register all of the event handlers. Remember to also add the event handlers to the `components` array or Nest.js will not be able to instantiate the event handlers. Now that we have our event and event handler written and linked in our keyword module, we are ready to start invoking the event to store and update keyword links in the database.

Invoking event handlers

Event handlers are invoked from data models. Data models are typically simple classes that represent data stored in a database. The one stipulation Nest.js places on data models is they must extend the `AggregateRoot` abstract class. Depending on which ORM you are using and how it is configured, you may or may not be able to re-use your existing data models for this purpose. Since our

example is using Sequelize, the `sequelize-typescript` package requires our data models extend the `Model` class. In Typescript, classes can only extend one other class. We will need to create a separate data model for invoking our event handlers.

```
export class EntryModel extends AggregateRoot {
  constructor(private readonly id: number) {
    super();
  }

  updateKeywordLinks(keywords: string[]) {
    this.apply(new UpdateKeywordLinksEvent(this.id, keywords));
  }
}
```

We create our data model in the blog entry module since we will be invoking our events when blog entries are created and updated. The data model contains a single method, `updateKeywordLinks`, for refreshing blog entry keyword links when a blog entry is created or updated. If new events are needed, we will add more methods to the model to handle invoking those events.
The `updateKeywordLinks` method instantiates the event we created and call the `apply` method found in the `AggregateRoot` abstract class with the event instance.

With commands, we used the command bus directly to `execute` our commands. With events, we take a less direct approach and use the `EventPublisher` to link our data model to the event bus and then call the method we created in our data model to `apply` an event. Let's update the `CreateEntryCommandHandler` to get a better idea of what's going on.

```
@CommandHandler(CreateEntryCommand)
export class CreateEntryCommandHandler implements
ICommandHandler<CreateEntryCommand> {
    constructor(
        @Inject('EntryRepository') private readonly EntryRepository:
typeof Entry,
        @Inject('SequelizeInstance') private readonly
sequelizeInstance: Sequelize,
        private readonly eventPublisher: EventPublisher
    ) { }
```

```
      async execute(command: CreateEntryCommand, resolve: (error?: Error)
=> void) {
            let caught: Error;

            try {
                  const entry = await
this.sequelizeInstance.transaction(async transaction => {
                        return await this.EntryRepository.create<Entry>({
                              ...command,
                              keywords: JSON.stringify(command.keywords)
                        }, {
                              returning: true,
                              transaction
                        });
                  });

                  const entryModel =
this.eventPublisher.mergeObjectContext(new EntryModel(entry.id));
                  entryModel.updateKeywordLinks(command.keywords);
                  entryModel.commit();
            } catch (error) {
                  caught = error;
            } finally {
                  resolve(caught);
            }
      }
}
```

The command handler constructor has been updated to have an instance of the Nest.js `EventPublisher`injected. The `EventPublisher` has two methods that we care about: `mergeClassContext` and `mergeObjectContext`. Both methods can be used to achieve the same outcome, just in different ways. In our example, we are using `mergeObjectContext` to merge a new instance of our data model with the event bus. This provides the data model instance with a `publish` method that is used inside the abstract `AggregateRoot` class to `publish` new events on the event bus.

Events are never dispatched immediately. When we call `updateKeywordLinks`, the event that is created is placed in a queue. The event queue gets flushed when we call the `commit` method on our data model. If you ever find that your event handlers are not firing, check to make sure you have called the `commit`method on your data model.

We could have accomplished the same functionality using `mergeClassContext` method on the event publisher.

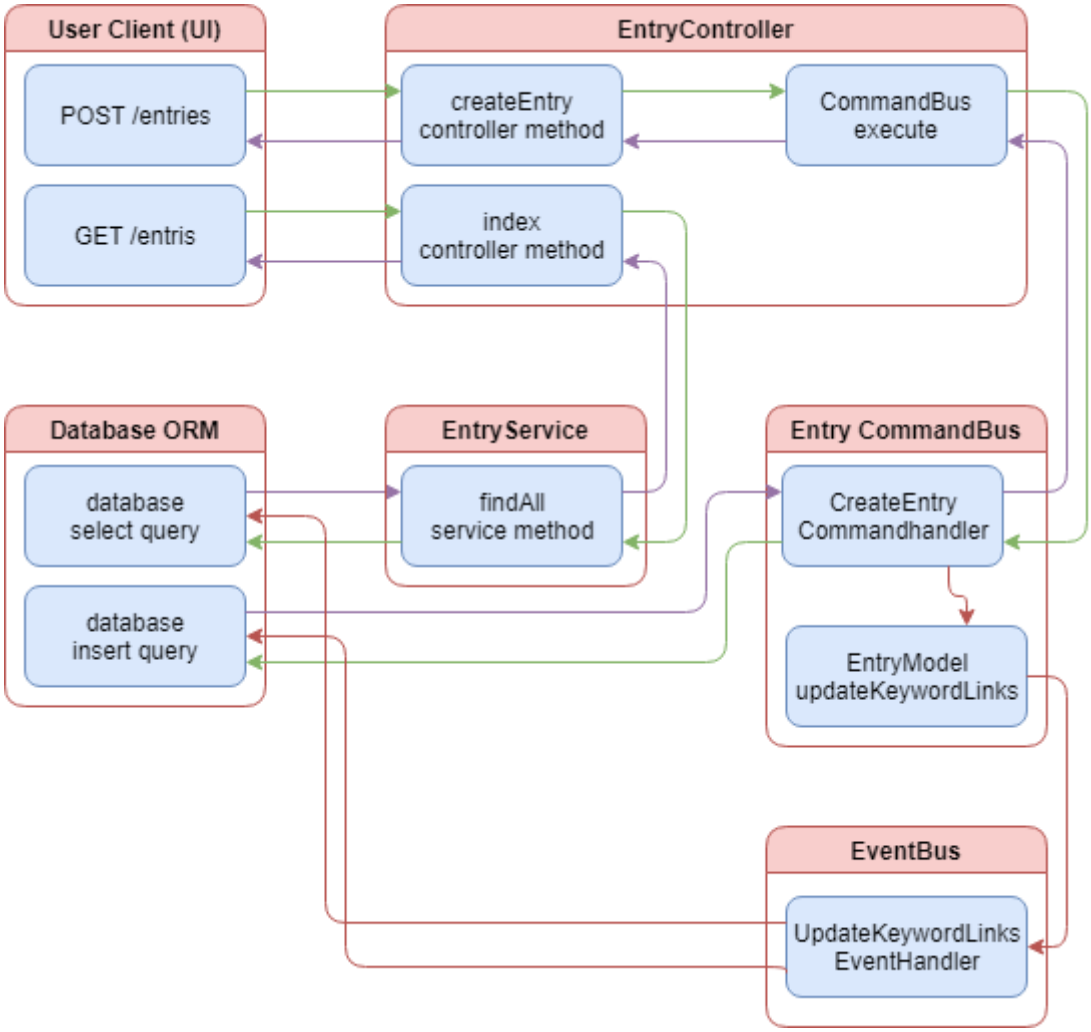
```
const Model = this.eventPublisher.mergeClassContext(EntryModel);
const entryModel = new Model(entry.id);
entryModel.updateKeywordLinks(command.keywords);
entryModel.commit();
```

The same updates need to be made to the `UpdateEntryCommandHandler` command handler so keyword links are updated when blog entries are updated.

```
@CommandHandler(UpdateEntryCommand)
export class UpdateEntryCommandHandler implements
ICommandHandler<UpdateEntryCommand> {
     constructor(
          @Inject('EntryRepository') private readonly EntryRepository:
typeof Entry,
          @Inject('SequelizeInstance') private readonly
sequelizeInstance: Sequelize,
          private readonly databaseUtilitiesService:
DatabaseUtilitiesService,
          private readonly eventPublisher: EventPublisher
     ) { }

     async execute(command: UpdateEntryCommand, resolve: (error?: Error)
=> void) {
          let caught: Error;

          try {
               await this.sequelizeInstance.transaction(async transaction
=> {
                    let entry = await
this.EntryRepository.findById<Entry>(command.id, { transaction });
                    if (!entry) throw new Error('The comment was not
found.');

                    entry = this.databaseUtilitiesService.assign(
                         entry,
                         {
                              ...command,
                              id: undefined,
```

```
                                keywords: JSON.stringify(command.keywords)
                            }
                        );
                        return await entry.save({
                            returning: true,
                            transaction,
                        });
                    });

                    const entryModel =
this.eventPublisher.mergeObjectContext(new EntryModel(command.id));
                    entryModel.updateKeywordLinks(command.keywords);
                    entryModel.commit();
                } catch (error) {
                    caught = error;
                } finally {
                    resolve(caught);
                }
            }
}
```

If you have followed along in your own project, you should now be able to create or update a blog entry with new or existing keywords and see the keyword links being created, updated, and deleted in the database. Of course, we could make these changes easier to view by adding a new API to return all the keywords and blog entries they are linked to.



The above diagram provides a visual representation of how the entry command handlers work to keep the keywords updated. Notice how the flow of control is unidirectional. The command handler invokes the event using the entry model and then forgets about it. This is the asynchronous nature of the event bus in Nest.js CQRS.

Retrieving keywords APIs

We will need to create a new controller and service in the keyword module to support retrieving keywords. We want to allow the UI to list all keywords, get a specific keyword, and get a list of "hot keywords." Let's create the service first.

```
@Component()
export class KeywordService implements IKeywordService {
     constructor(@Inject('KeywordRepository') private readonly
keywordRepository: typeof Keyword,
```

```
                    @Inject('KeywordEntryRepository') private readonly
keywordEntryRepository: typeof KeywordEntry) { }

     public async findAll(search?: string, limit?: number):
Promise<Array<Keyword>> {
          let options: IFindOptions<Keyword> = {};

          if (search) {
               if (!limit || limit < 1 || limit === NaN) {
                    limit = 10;
               }

               options = {
                    where: {
                         keyword: {
                              [Op.like]: `%${search}%`
                         }
                    },
                    limit
               }
          }

          return await this.keywordRepository.findAll<Keyword>(options);
     }

     public async findById(id: number): Promise<Keyword | null> {
          return await this.keywordRepository.findById<Keyword>(id);
     }

     public async findHotLinks(): Promise<Array<Keyword>> {
          // Find the latest 5 keyword links
          const latest5 = await
this.keywordEntryRepository.findAll<KeywordEntry>({
               attributes: {
                    exclude: ['entryId', 'createdAt']
               },
               group: ['keywordId'],
               order: [[fn('max', col('createdAt')), 'DESC']],
               limit: 5
          } as IFindOptions<any>);
```

```
            // Find the 5 keywords with the most links
            const biggest5 = await
this.keywordEntryRepository.findAll<KeywordEntry>({
                  attributes: {
                        exclude: ['entryId', 'createdAt']
                  },
                  group: 'keywordId',
                  order: [[fn('count', 'entryId'), 'DESC']],
                  limit: 5,
                  where: {
                        keywordId: {
                              // Filter out keywords that already exist in
the latest5
                              [Op.notIn]: latest5.map(keywordEntry =>
keywordEntry.keywordId)
                        }
                  }
            } as IFindOptions<any>);

            // Load the keyword table data
            const result = await Promise.all(
                  [...latest5, ...biggest5].map(keywordEntry =>
this.findById(keywordEntry.keywordId))
            );

            return result;
      }
}
```

The `findAll` method takes an optional search string and limit that can be used to filter the keywords. The UI can use this to support keyword search autocomplete. If the limit is not specified when searching, the service will automatically limit the results to 10 items. The `findById` method will support loading all information for a single keyword, including the associated entries. The methods are relatively basic and mimic methods in the services of the other modules. The `findHotLinks` method, however, is a bit more complex.

The `findHotLinks` method is responsible for returning the latest used keywords and the keywords with the most linked blog entries. To do this, we need to incorporate the ORM provider for the joining table, the `KeywordEntry` data model. The joining table contains the actual link between keywords and blog entries as well as the date they we joined. For the `latest5`, we order the list by

the maximum `createdAt`date to get a list of keywords with the newest first. The `biggest5` is ordered by the count of `entryId` to produce a list containing the keywords with the most linked blog entries first. In both lists, we group by the `keywordId` to produce a list of unique keywords and limit the results to the top five. To ensure we do not produce a list with overlaps, the `biggest5` also contains a where clause to not include any keywords that were already included in the `latest5` list.

Once we have the two lists, we reuse the service's `findById` method to load the complete data record for all the found keywords. This list is then returned with the keywords that have the newest links first, ordered newest to oldest, followed by the keywords with the most links, order most to least. All that remains is to create a controller so the UI can take advantage of our new query methods.

Note: Take note of the `as IFindOptions<any>`. This was required to resolve a linting error caused by `sequelize-typescript`. You may or may not need this in your application.

```
@Controller()
export class KeywordController {
     constructor(
          private readonly keywordService: KeywordService
     ) { }

     @Get('keywords')
     public async index(@Query('search') search: string, @Query('limit')
limit: string, @Res() res) {
          const keywords = await this.keywordService.findAll(search,
Number(limit));
          return res.status(HttpStatus.OK).json(keywords);
     }

     @Get('keywords/hot')
     public async hot(@Res() res) {
          const keywords = await this.keywordService.findHotLinks();
          return res.status(HttpStatus.OK).json(keywords);
     }

     @Get('keywords/:keywordId')
     public async show(@Param('keywordId') keywordId: string, @Res() res)
{
```

```
            const keyword = await
this.keywordService.findById(Number(keywordId));
            return res.status(HttpStatus.OK).json(keyword);
        }
}
```

The controller contains three methods that correspond to the three query methods in the service. In all three, we call the appropriate method in the service and return the results as JSON. Take note that the `hot` method is listed before the `show` method. If this order was reversed, calling the `/keywords/hot` API would result in the `show` method executing. Since Nest.js is running on top of ExpressJS, the order in which we declare our controller methods matter. ExpressJS will always execute the first route controller that matches the path requested by the UI.

We now have an application that is using Nest.js CQRS to break apart business logic and implements pieces of it in an asynchronous manor. The application is capable of reacting to blog entry creations and updates to alter the keyword metadata. This is all made possible through the use of events. But there is another way to accomplish the same goal using sagas instead of the event handler we created.

Linking keywords with sagas

A saga can be thought of as a special event handler that returns commands. Sagas do this by leveraging RxJS to receive and react to all events published to the event bus. Using the `UpdateKeywordLinksEvent`event handler, we can logically partition the work into two separate commands: one to create keyword links and one to remove them. Since sagas return commands, the saga and command must be created in the same module. Otherwise, command module scoping will become a problem and Nest.js will throw an exception when our saga attempts to return a command found in a different module. To get started, we will need setup the commands and command handlers that will be replacing our single event handler.

Keyword saga commands

Just because we are using sagas to execute our new commands does not change how we write those commands and command handlers. We will split the `UpdateKeywordLinksEvent` into two separate commands within the keyword module.

```
export class LinkKeywordEntryCommand implements ICommand {
     constructor(
          public readonly keyword: string,
          public readonly entryId: number
     ) { }
}

export class UnlinkKeywordEntryCommand implements ICommand {
     constructor(
          public readonly keyword: string,
          public readonly entryId: number
     ) { }
}
```

The commands take two properties: a `keyword` and an `entryId`. The commands take a simple `keyword`string because the command handler should not assume the keyword already exists in the database. The `entryId` is already known to exist since it is a parameter of the `UpdateKeywordLinksEvent` event.

```
@CommandHandler(LinkKeywordEntryCommand)
export class LinkKeywordEntryCommandHandler implements
ICommandHandler<LinkKeywordEntryCommand> {
     constructor(
          @Inject('KeywordRepository') private readonly
keywordRepository: typeof Keyword,
          @Inject('SequelizeInstance') private readonly
sequelizeInstance: Sequelize
     ) { }

     async execute(command: LinkKeywordEntryCommand, resolve: (error?:
Error) => void) {
          let caught: Error;

          try {
               await this.sequelizeInstance.transaction(async transaction
=> {
                    const keyword = await
this.keywordRepository.findOrCreate({
                         where: {
                              keyword: command.keyword
                         },
```

```
                    transaction
                });

                await keyword[0].$add('entries', command.entryId, {
transaction });
            });
        } catch (error) {
            caught = error;
        } finally {
            resolve(caught);
        }
    }
}
```

The `LinkKeywordEntryCommandHandler` command handler takes care of ensuring the keyword exists in the database and then uses the `$add` method provided by `sequelize-typescript` to link a blog entry to the keyword by it's id.

```
@CommandHandler(UnlinkKeywordEntryCommand)
export class UnlinkKeywordEntryCommandHandler implements
ICommandHandler<UnlinkKeywordEntryCommand> {
    constructor(
        @Inject('KeywordRepository') private readonly
keywordRepository: typeof Keyword,
        @Inject('SequelizeInstance') private readonly
sequelizeInstance: Sequelize
    ) { }

    async execute(command: UnlinkKeywordEntryCommand, resolve: (error?:
Error) => void) {
        let caught: Error;

        try {
            await this.sequelizeInstance.transaction(async transaction
=> {
                const keyword = await
this.keywordRepository.findOrCreate<Keyword>({
                    where: {
                        keyword: command.keyword
                    },
                    transaction
```

```
                    });

                    await keyword[0].$remove('entries', command.entryId, {
transaction });
                });
            } catch (error) {
                caught = error;
            } finally {
                resolve(caught);
            }
        }
}
```

The `UnlinkKeywordEntryCommandHandler` command handler takes care of ensuring the keyword exists in the database and then uses the `$remove` method provided by `sequelize-typescript` to remove the link of a blog entry to the keyword by its id. These commands are substantially simpler than `UpdateKeywordLinksEventHandler` event handler was. They have a single purpose, link or unlink a keyword and blog entry. The heavy lifting of determining which keywords to link and unlink is reserved for the saga. Don't forget to hook up the command handlers in the keyword module.

```
export const keywordCommandHandlers = [
    LinkKeywordEntryCommandHandler,
    UnlinkKeywordEntryCommandHandler
];

@Module({
    imports: [CQRSModule],
    controllers: [KeywordController],
    components: [keywordProvider, keywordEntryProvider,
...keywordEventHandlers, KeywordService, ...keywordCommandHandlers],
    exports: []
})
export class KeywordModule implements OnModuleInit {
    public constructor(
        private readonly moduleRef: ModuleRef,
        private readonly eventBus: EventBus,
        private readonly commandBus: CommandBus
    ) {}
```

```
    public onModuleInit() {
        this.commandBus.setModuleRef(this.moduleRef);
        this.commandBus.register(keywordCommandHandlers);
        this.eventBus.setModuleRef(this.moduleRef);
        this.eventBus.register(keywordEventHandlers);
    }
}
```

Just like the entry module, we created a Typescript barrel to export the command handlers as an array. That gets imported into the module definition and registered to the command bus using the `register`method.

Keyword saga

Sagas are always written as public methods inside component classes to allow for Dependency Injection. Typically, you would create a single saga class for each module you wish to implement sagas in, but multiple classes would make sense when breaking up complex business logic. For the update keyword saga, we will need a single saga method that accepts the `UpdateKeywordLinksEvent` event and outputs multiple `LinkKeywordEntryCommand` and `UnlinkKeywordEntryCommand` commands.

```
@Component()
export class KeywordSagas {
    constructor(
        @Inject('KeywordRepository') private readonly
keywordRepository: typeof Keyword,
        @Inject('SequelizeInstance') private readonly
sequelizeInstance: Sequelize,
    ) { }

    public updateKeywordLinks(events$: EventObservable<any>) {
        return events$.ofType(UpdateKeywordLinksEvent).pipe(
            mergeMap(event =>
                merge( // From the rxjs package
                    this.getUnlinkCommands(event),
                    this.getLinkCommands(event)
                )
            )
        );
    }
```

```
}
```

The `KeywordSagas` class contains a single saga `updateKeywordLinks` and uses Dependency Injection to get a reference to the keyword repository and Sequelize instance. The parameter passed to the `updateKeywordLinks` saga is provided by the Nest.js CQRS event bus. `EventObservable` is a special observable provided by Nest.js CQRS that contains the `ofType` method. We use this method to filter the `events$`observable so our saga will only handle the `UpdateKeywordLinksEvent` event. If you forget to use the `ofType` method, your saga will be fired for every event published in your application.

The remaining pieces to our saga is strictly RxJS functionality. You are free to use any RxJS operator, as long as the saga emits one or more CQRS commands. For our saga, we will be using `mergeMap` to flatten an inner observable stream of commands. Do not use `switchMap` here or commands could be lost if the API is under heavy load due to how `switchMap` is cancelled when the outer observable fires multiple times. The inner observable is a merging of two different observable streams: `this.getUnlinkCommands(event)` is a stream of `UnlinkKeywordEntryCommand` commands and `this.getLinkCommands(event)` is a stream of `LinkKeywordEntryCommand` commands.

```
private getUnlinkCommands(event: UpdateKeywordLinksEvent) {
      return from(this.keywordRepository.findAll({
            include: [{ model: Entry, where: { id: event.entryId }}]
      })).pipe(
            // Filter keywordEntities so only those being removed are left
            map(keywordEntities =>
                  keywordEntities.filter(keywordEntity =>
event.keywords.indexOf(keywordEntity.keyword) === -1)
            ),
            // Create new commands for each keywordEntity
            map(keywordEntities => keywordEntities.map(keywordEntity => new
UnlinkKeywordEntryCommand(keywordEntity.keyword, event.entryId))),
            switchMap(commands => Observable.of(...commands))
      );
}

private getLinkCommands(event: UpdateKeywordLinksEvent) {
      return from(this.keywordRepository.findAll({
            include: [{ model: Entry, where: { id: event.entryId }}]
      })).pipe(
            // Filter keywordEntities so only those being add are left
```

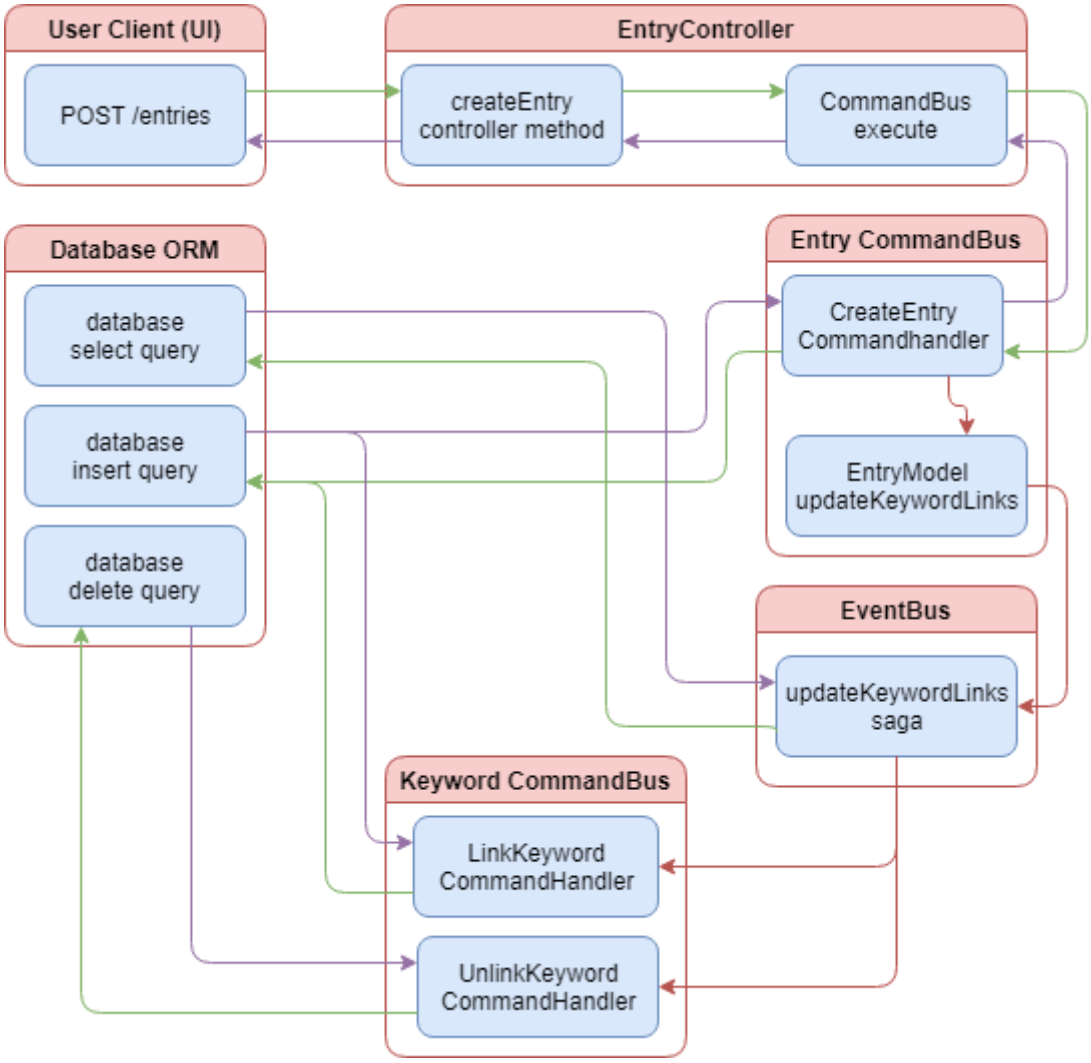
```
            map(keywordEntities =>
                  event.keywords.filter(keyword =>
keywordEntities.findIndex(keywordEntity => keywordEntity.keyword ===
keyword) === -1)
            ),
            // Create new commands for each keyword
            map(keywords => keywords.map(keyword => new
LinkKeywordEntryCommand(keyword, event.entryId))),
            switchMap(commands => Observable.of(...commands))
      );
}
```

The `getUnlinkCommands` and `getLinkCommands` methods start off with getting a list of existing keyword blog entry links. We use `Observable.fromPromise` since we need to return an observable from these methods. The difference between the two commands is how the filtering works. In `getUnlinkCommands`, we need to filter the list of existing keyword blog entry links to find those that do not exist in the keywords array of the event. We reverse the logic in `getLinkCommands` and filter the list of keywords in the event to find those that are not already linked to the blog entry. Finally, we map the arrays to commands and use `switchMap(commands => Observable.of(...commands))` so our observable stream emits all the commands instead of an array of commands. Since the only difference is the filtering, we could clean this up so the database is not queried as much.

```
public updateKeywordLinks(events$: EventObservable<any>) {
      return events$.ofType(UpdateKeywordLinksEvent).pipe(
            mergeMap(event => this.compileKeywordLinkCommands(event))
      );
}

private compileKeywordLinkCommands(event: UpdateKeywordLinksEvent) {
      return from(this.keywordRepository.findAll({
            include: [{ model: Entry, where: { id: event.entryId }}]
      })).pipe(
            switchMap(keywordEntities =>
                  of(
                        ...this.getUnlinkCommands(event, keywordEntities),
                        ...this.getLinkCommands(event, keywordEntities)
                  )
            )
      );
```

```
}

private getUnlinkCommands(event: UpdateKeywordLinksEvent,
keywordEntities: Keyword[]) {
      return keywordEntities
            .filter(keywordEntity =>
event.keywords.indexOf(keywordEntity.keyword) === -1)
            .map(keywordEntity => new
UnlinkKeywordEntryCommand(keywordEntity.keyword, event.entryId));
}

private getLinkCommands(event: UpdateKeywordLinksEvent, keywordEntities:
Keyword[]) {
      return event.keywords
            .filter(keyword => keywordEntities.findIndex(keywordEntity =>
keywordEntity.keyword === keyword) === -1)
            .map(keyword => new LinkKeywordEntryCommand(keyword,
event.entryId));
}
```

Now our saga only queries the database for the existing keyword blog entry links once and the `getUnlinkCommands` and `getLinkCommands` methods have been drastically simplified. These methods now take the event and list of existing keyword blog entry links and returns an array of commands that need to be executed. The heavy lifting of retrieving the existing keyword blog entry links has been offloaded to `compileKeywordLinkCommands` method. This method uses `switchMap` to project the results from the database into `getUnlinkCommands` and `getLinkCommands`. `Observable.of` is still used to take the array of commands and emit them one at a time. Creating and updating blog entries will now process all keyword linking and unlinked through the saga and keyword commands.



The above diagram provides a visual representation of how our new sagas hand off processing of database updates back to the command bus in the keyword module. Once an event to update keyword links is executed, the saga queries the database to determine the keywords to be linked and unlinked and finally returns the appropriate commands. Remember command handlers contain a callback method so it is not explicitly asynchronous. However, since they are called from the event bus, any response is never passed back to the sage or the entry command bus.

Summary

CQRS is not just a Nest.js package. It is a pattern for designing and laying out your application. It requires that you split the command, creation and update of data, from the query, the retrieving of data, and aspects of your application. For small applications, CQRS can add a lot of unnecessary complexity so it’s not

for every application. For medium and large applications, CQRS can help break apart complex business logic into more manageable pieces.

Nest.js provides two means of implementing the CQRS pattern, the command and event bus, and with some sugar in the form of sagas. The command bus isolates command execution to each module meaning a command can only be executed in the same module it is registered. Command handlers are not always asynchronous and limits other parts of the application from reacting to change. For this, Nest.js provides the event bus. The event bus is not isolated to a single module and provides a way for different modules of the same application to react to events published by other modules. In fact, events can have any number of handlers allowing business logic to easily scale without changing the existing code.

Sagas are a different way of reacting to events within a module. Sagas are simple function that listen for events on the event bus and react by returning a command to execute. While seemingly simple, sagas allow you to use the power of RxJS to determine if and how your application should react to events. As we did with our example application, sagas are not limited to returning just one or even one type of command.

The next time you find yourself writing complex code to perform some business logic based on how the user is interacting with your application, consider giving the CQRS pattern a try. The complexity of the pattern may be offset by the complexity or eventual complexity of your applications business logic.

In the next chapter we examine the architecture for two different types of projects: A server application, and an app using `Angular universal` with Nest.js and Angular 6.

Chapter 13. Architecture

As you now know, Nest.js is based on the same principles as Angular, so it is a good idea to have a similar structure as Angular's.

Before going into the file structure, we will see some guidelines about the naming and about how to structure our different directories and files in order to have an easy and more readable project.

We will take a look at the architecture for two different type of projects:

- A server application
- A more complete app using `Angular universal` with Nest.js and Angular 6

By the end of the chapter, you should know how to structure your app either for a server application or a complete app with a client front-end.

Style guide of naming conventions

In this part, we will see the naming conventions that we can use in order to have better maintainability and readability. For each decorator, you should use the name with a hyphen for a composed name, followed by a dot and the name of the decorator or object to which it corresponds.

Controller

The naming of the controller should respect the following principle:

user.controller.ts

```
@Controller()
export class UserController { /* ... */ }
```

Service

The naming of the service should respect the following principle:

user.service.ts

```
@Injectable()
```

```
export class UserService { /* ... */ }
```

Module

The naming of the module should respect the following principle:

user.module.ts

```
@Module()
export class UserModule { /* ... */ }
```

Middleware

The naming of the middleware should respect the following principle:

authentication.middleware.ts

```
@Injectable()
export class AuthenticationMiddleware { /* ... */ }
```

Exception filter

The naming of the exception filter should respect the following principle:

forbidden.exception.ts

```
export class ForbiddenException { /* ... */ }
```

Pipe

The naming of the pipe should respect the following principle:

validation.pipe.ts

```
@Injectable()
export class ValidationPipe { /* ... */ }
```

Guard

The naming of the guard should respect the following principle:

roles.guard.ts

```
@Injectable()
export class RolesGuard { /* ... */ }
```

Interceptor

The naming of the interceptor should respect the following principle:

logging.interceptor.ts

```
@Injectable()
export class LoggingInterceptor { /* ... */ }
```

Custom decorator

The naming of the custom decorator should respect the following principle:

comment.decorator.ts

```
export const Comment: (data?: any, ...pipes: Array<PipeTransform<any>>)
=> {
     ParameterDecorator = createParamDecorator((data, req) => {
          return req.comment;
     }
};
```

Gateway

The naming of the gateway should respect the following principle:

comment.gateway.ts

```
@WebSocketGateway()
export class CommentGateway {
```

Adapter

The naming of the adapter should respect the following principle:

ws.adapter.ts

```
export class WsAdapter {
```

Unit test

The naming of the unit test should respect the following principle:

user.service.spec.ts

E2E test

The naming of the e2e test should respect the following principle:

user.e2e-spec.ts

Now we have overviewed the tools provided by Nest.js and have put in place some naming guidelines. We can now move onto the next part.

Directory structure

It is important to have a project with a well-structured directory file in order for it to be much more readable, understandable, and easy to work with.

So, let's see how we can structure our directory in order for it to be more clear. You will see in the following example the directory file architecture used for the repository, which has been created for this book using the naming convention described in the previous section.

Server architecture

For the server architecture, you will see a proposed architecture used for the repository to have clean directories.

COMPLETE OVERVIEW

See the base file structure without entering into too much detail:

```
.
├── artillery/
├── scripts/
```

```
├── migrations/
├── src/
├── Dockerfile
├── README.md
├── docker-compose.yml
├── migrate.ts
├── nodemon.json
├── package-lock.json
├── package.json
├── tsconfig.json
├── tslint.json
└── yarn.lock
```

We have four folders for this server that contain all of the files that we need for a complete server:

- `artillery` directory, if you need this it can contain all of the scenarios to test some end points of your API.
- `scripts` directory will contain all of the scripts that you need to use in your application. In our case the script to wait for the port used by `RabbitMQ` to open in order that the Nest.js application waits before starting.
- `migrations` A directory exists because we use `sequelize` and we have written some migration files that are stocked in this directory.
- `src` directory, which will contain all of the code for our server application.

In the repository, we also have a `client` directory. In this case, however, this one is only used as a sample of web socket usage.

THE SRC DIRECTORY

The `src` directory will contain all of the application modules, configurations, gateways and more. Let's take a look at this directory:

```
src
```

```
├── app.module.ts
├── main.cluster.ts
├── main.ts
├── gateways
|   ├── comment
|   └── user
├── modules
|   ├── authentication
|   ├── comment
|   ├── database
|   ├── entry
|   ├── keyword
|   └── user
└── shared
    ├── adapters
    ├── config
    ├── decorators
    ├── exceptions
    ├── filters
    ├── guards
    ├── interceptors
    ├── interfaces
    ├── middlewares
    ├── pipes
    └── transports
```

This directory will also have to be well-structured. For this, we have created three sub-directories that correspond to the web socket gateways, which have all been put in the `gateways` directory. The `modules`will contain all of the modules needed for the application. Finally, `shared` will contain all of the shared content

as its name suggests, corresponding with all of the `adapters`, `config` files, and `decorators` for the custom decorators and elements that can be used everywhere without belonging to any module in particular.

Now we will dive into the modules directory.

Modules

The main part of your application will be structured as a module. This module will contain many different files. Let's have a look at how a module can be structured:

```
src/modules
├── authentication
|   ├── authentication.controller.ts
|   ├── authentication.module.ts
|   ├── authentication.service.ts
|   ├── passports
|   |   └── jwt.strategy.ts
|   └── tests
|       ├── e2e
|       |   └── authentication.controller.e2e-spec.ts
|       └── unit
|           └── authentication.service.spec.ts
├── comment
|   ├── comment.controller.ts
|   ├── comment.entity.ts
|   ├── comment.module.ts
|   ├── comment.provider.ts
|   ├── comment.service.ts
|   ├── interfaces
|   |   ├── IComment.ts
|   |   ├── ICommentService.ts
```

```
|   |   └── index.ts
|   └── tests
|       ├── unit
|       |   └── comment.service.spec.ts
|       └── utilities.ts
├── database
|   ├── database-utilities.service.ts
|   ├── database.module.ts
|   └── database.provider.ts
├── entry
|   ├── commands
|   |   ├── handlers
|   |   |   ├── createEntry.handler.ts
|   |   |   ├── deleteEntry.handler.ts
|   |   |   ├── index.ts
|   |   |   └── updateEntry.handler.ts
|   |   └── impl
|   |       ├── createEntry.command.ts
|   |       ├── deleteEntry.command.ts
|   |       └── updateEntry.command.ts
|   ├── entry.controller.ts
|   ├── entry.entity.ts
|   ├── entry.model.ts
|   ├── entry.module.ts
|   ├── entry.provider.ts
|   ├── entry.service.ts
|   ├── interfaces
|   |   ├── IEntry.ts
```

```
|   |   ├── IEntryService.ts
|   |   └── index.ts
|   └── tests
|       ├── unit
|       |   └── entry.service.spec.ts
|       └── utilities.ts
├── keyword
|   ├── commands
|   |   ├── handlers
|   |   |   ├── index.ts
|   |   |   ├── linkKeywordEntry.handler.ts
|   |   |   └── unlinkKeywordEntry.handler.ts
|   |   └── impl
|   |       ├── linkKeywordEntry.command.ts
|   |       └── unlinkKeywordEntry.command.ts
|   ├── events
|   |   ├── handlers
|   |   |   ├── index.ts
|   |   |   └── updateKeywordLinks.handler.ts
|   |   └── impl
|   |       └── updateKeywordLinks.event.ts
|   ├── interfaces
|   |   ├── IKeyword.ts
|   |   ├── IKeywordService.ts
|   |   └── index.ts
|   ├── keyword.controller.ts
|   ├── keyword.entity.ts
|   ├── keyword.module.ts
```

```
|   ├── keyword.provider.ts
|   ├── keyword.sagas.ts
|   ├── keyword.service.ts
|   └── keywordEntry.entity.ts
└── user
    ├── interfaces
    |   ├── IUser.ts
    |   ├── IUserService.ts
    |   └── index.ts
    ├── requests
    |   └── create-user.request.ts
    ├── tests
    |   ├── e2e
    |   |   └── user.controller.e2e-spec.ts
    |   ├── unit
    |   |   └── user.service.spec.ts
    |   └── utilities.ts
    ├── user.controller.ts
    ├── user.entity.ts
    ├── user.module.ts
    ├── user.provider.ts
    └── user.service.ts
```

In our repository, we have many modules. Some of them also implement the `cqrs`, and it is in the same directory as the module, because it concerns the module and is part of it. The `cqrs` parts are separated into the `commands` and `events` directories. A module can also define some interfaces and these are put into a separate `interfaces` directory. Separate directories allow us to have something that is much more readable and clear without having lots of different files mixed together. Of course, all of the tests concerning the modules are also included in their own directory `tests` and separated into the `unit` and `e2e`.

Finally, the main files defining the module itself, including the injectables, the controllers, and the entity, are in the root of the module `directory`.

In this section, we have seen how to structure our server application in a way to keep it clearer and much more readable. You now know where to put all of your modules and how to structure a module, as well as where to put your gateways or your shared files if you used them.

Angular Universal architecture

The Angular Universal part of the repository is a standalone application, using the Nest.js server and Angular 6. It will be composed of just two main directories: `e2e` for the end-to-end tests and the `src`, which contains the server and the client.

Let's start by seeing an overview of this architecture:

```
├── e2e/
├── src/
├── License
├── README.md
├── angular.json
├── package.json
├── tsconfig.json
├── tslint.json
├── udk.container.js
└── yarn.lock
```

THE `src` DIRECTORY

This directory will contain the `app` directory in order to put our client content with the Angular architecture using modules. Also, we will find the `environments`, which define if we are in production mode or not exporting constant. This environment will be replaced by the production environment config for the production mode, and then the `server` and `shared` directories. The shared directory allows us to share some files as an interface, for example, and the server directory will contain all the server applications as we have seen in the previous section.

But in this case, the server has changed a bit and now looks like this:

```
├── main.ts
├── app.module.ts
├── environments
|   ├── environment.common.ts
|   ├── environment.prod.ts
|   └── environment.ts
└── modules
    ├── client
    |   ├── client.constants.ts
    |   ├── client.controller.ts
    |   ├── client.module.ts
    |   ├── client.providers.ts
    |   ├── interfaces
    |   |   └── angular-universal-options.interface.ts
    |   └── utils
    |       └── setup-universal.utils.ts
    └── heroes
        ├── heroes.controller.ts
        ├── heroes.module.ts
        ├── heroes.service.ts
        └── mock-heroes.ts
```

The modules directory will contain all of the Nest.js modules, exactly as we have seen in the previous section. One of the modules is the `client` module and will serve the Universal app and all of the required assets, as well as setting up the initializer to set the engine and provide some Angular configurations.

Regarding the `environments`, this one will contain all of the configuration paths related to the Angular application. This configuration references the project configured into the `angular.json` file seen in the base of the previous section's project.

Summary

This chapter allows you to set up the architecture of your application in a way that is much more understandable, readable and easier to work with. We have seen how to define the architecture's directories for a server application, but also for a complete application using Angular Universal. With these two examples, you should be able to build your own project in a clearer way.

The next chapter shows how to use testing in Nest.js.

Chapter 14. Testing

Automated testing is a critical part of software development. Even though it cannot (and it's not intended) to replace manual testing and other quality assurance methods. Automated testing is a very valuable tool when used properly, to avoid regressions, bugs, or incorrect functionality.

Software development is a tricky discipline: even though many developers try to isolate different parts of software, it's often unavoidable that some pieces of the puzzle have effect on other pieces, be it intended or unintended.

One of the main goals of using automated testing is to detect the kind of errors in which new code might break previously working functionality. These tests are known as *regression tests,* and they make the most sense when triggered as part of the merging or deployment process, as a required step. This means that, if the automated tests fail, the merge or deployment will be interrupted, thus avoiding the introduction of new bugs to the main codebase or to productive environments.

Automated testing also enables a developmental workflow known as *test driven development (TDD.)*When following a TDD approach, automated tests are written beforehand, as very specific cases that reflect the requirements. Once the new tests are written, the developer runs *all tests;* the new ones should fail, since no new code has been written yet. At that point it's when the new code has to be written so that the new tests pass, while not breaking the old ones.

The test driven development approach, if done right, can improve the confidence in code quality and requirement compliance. They can also make refactoring or even full code migrations, less risky.

There are two main types of automated tests we are going to cover in this book: unit tests and end-to-end tests.

Unit testing

As the name implies, each unit test cover one specific functionality. The most important principles when dealing with unit tests are:

- **isolation;** each component has to be tested without any other related components; it cannot be affected by side effects and, likewise, it cannot emit any side effects.
- **predictability;** each test has to yield the same results as long as the input doesn't change.

In many cases, complying with these two principles means *mocking* (i.e. simulating the functionality of) the component dependencies.

Tooling

Unlike Angular, Nest.js doesn't have an "official" toolset for running tests; this means we are free to set up our own tooling for running automated tests when we work in Nest.js projects.

There are multiple tools in the JavaScript ecosystem focused on writing and running automated unit tests. The typical solutions involve using several different packages for one setup, because those packages used to be limited in their scope (one for test running, a second one for assertions, a third one for mocking, maybe even another one for code coverage reporting).

We will, however, be using Jest, an "all-in-one", "zero-configuration" testing solution from Facebook that has greatly decreased the amount of configuration effort needed for running automated tests. It also officially supports TypeScript so it's a great match for Nest.js projects!

Preparation

As you would expect, Jest is distributed as an npm package. Let's go and install it in our project. Run the following command from your command line or terminal:

```
npm install --save-dev jest ts-jest @types/jest
```

We are installing three different npm packages as development dependencies: Jest itself; `ts-jest` that allows us to use Jest with our TypeScript code; and the Jest typings for their valuable contribution to our IDE experience!

Remember how we mentioned that Jest is a "zero-configuration" testing solution? Well, this is what their homepage claims. Unfortunately, it's not entirely true: we still need to define a little amount of configuration before we are able to run our tests. In our case, this is mostly because we are using

TypeScript. On the other hand, the configuration we need to write is really not that much, so we can write it as a plain JSON object.

So, let's create a new JSON file in our project's root folder, which we will name `nest.json`.

/nest.json

```
{
  "moduleFileExtensions": ["js", "ts", "json"],
  "transform": {
    "^.+\\.ts": "<rootDir>/node_modules/ts-jest/preprocessor.js"
  },
  "testRegex": "/src/.*\\.(test|spec).ts",
  "collectCoverageFrom": [
    "src/**/*.ts",
    "!**/node_modules/**",
    "!**/vendor/**"
  ],
  "coverageReporters": ["json", "lcov", "text"]
}
```

This small JSON file is setting up the following configuration:

1. Established files with `.js`, `.ts` and `.json` as modules (i.e. code) of our app. You might think we would not need `.js` files, but the truth is that our code will not run without that extension, because of some of Jest's own dependencies.
2. Tells Jest to process files with a `.ts` extension using the `ts-jest` package (which was installed before from the command line).
3. Specifies that our test files will be inside the `/src` folder, and will have either the `.test.ts` or the `.spec.ts` files extension.
4. Instructs Jest to generate code coverage reports from any `.ts` file on the `/src` folder, while ignoring the `node_modules` and the `vendor` folder contents. Also, to generate coverage it reports in both `JSON`and `LCOV` formats.

Finally, the last step before we can start writing tests will be to add a couple of new scripts to your `package.json` file:

```
{
  ...
```

```
  "scripts": {
    ...
    "test": "jest --config=jest.json",
    "test:watch": "jest --watch --config=jest.json",
    ...
  }
}
```

The three new scripts will, respectively: run the tests once, run the tests in watch mode (they will run after each file save), and run the tests and generate the code coverage report (which will be output in a `coverage` folder).

NOTE: Jest receives its configuration as a `jest` property in the `package.json` file. If you decide to do things that way, you will need to omit the `--config=jest.json` arguments on your npm scripts.

Our testing environment is ready. If we now run `npm test` in our project folder, we will most likely see the following:

```
No tests found
In /nest-book-example
  54 files checked.
  testMatch:  - 54 matches
  testPathIgnorePatterns: /node_modules/ - 54 matches
  testRegex: /src/.*\.(test|spec).ts - 0 matches
Pattern:  - 0 matches
npm ERR! Test failed.  See above for more details.
```

The tests have failed! Well, they actually haven't; we just have not written any tests yet! Let's write some now.

Writing our first test

If you have read some more chapters of the book, you probably remember our blog entries and the code we wrote for them. Let's take a look back to the `EntryController`. Depending on the chapters, the code looked something like the following:

`/src/modules/entry/entry.controller.ts`

```
import { Controller, Get, Post, Param } from '@nestjs/common';

import { EntriesService } from './entry.service';

@Controller('entries')
export class EntriesController {
  constructor(private readonly entriesSrv: EntriesService) {}

  @Get()
  findAll() {
    return this.entriesSrv.findAll();
  }
  ...
}
```

Notice that this controller is a dependency to `EntriesService`. Since we mentioned that each component has to be tested in *isolation*, we will need to mock any dependency it might have; in this case, the `EntriesService`.

Let's write a unit test for the controller's `findAll()` method. We will be using a special Nest.js package called `@nestjs/testing`, which will alow us to wrap our service in a Nest.js module specially for the test.

Also, it's important to follow the convention and name the test file `entry.controller.spec.ts`, and place it next to the `entry.controller.ts` file, so it gets properly detected by Jest when we trigger a test run.

/src/modules/entry/entry.controller.spec.ts

```
import { Test } from '@nestjs/testing';
import { EntriesController } from './entry.controller';
import { EntriesService } from './entry.service';

describe('EntriesController', () => {
  let entriesController: EntriesController;
  let entriesSrv: EntriesService;

  beforeEach(async () => {
    const module = await Test.createTestingModule({
      controllers: [EntriesController],
    })
      .overrideComponent(EntriesService)
```

```
        .useValue({ findAll: () => null })
        .compile();

      entriesSrv = module.get<EntriesService>(EntriesService);
      entriesController =
module.get<EntriesController>(EntriesController);
    });
});
```

Let's now take a close look at what the test code is achieving.

First of all, we are declaring a test suite on `describe('EntriesController', () => {`. We also declare a couple of variables, `entriesController` and `entriesSrv`, to hold both the tested controller itself, as well as the service the controller depends on.

Then, it comes the `beforeEach` method. The code inside that method will be executed right before each of the following tests are run. In that code, we are instantiating a Nest.js module for each test. Note that this is a particular kind of module, since we are using the `.createTestingModule()` method from the `Test` class that comes from the `@nestjs/testing` package. So, let's think about this module as a "mock module," which will serve us for testing purposes only.

Now comes the fun part: we include the `EntriesController` as a controller in the testing module. We then proceed to use:

```
.overrideComponent(EntriesService)
.useValue({ findAll: () => null })
```

This substitutes the original `EntryService`, which is a dependency of our tested controller. This is for a mock version of the service, which is not even a class, since we don't need it to be, but rather an object with a `findAll` method that takes no arguments and returns null.

You can think of the result of the two code lines above as an empty, dumb service that only repeats the methods we will need to use later, without any implementation inside.

Finally, the `.compile()` method is the one that actually instantiates the module, so it gets bound to the `module` constant.

Once the module is properly instantiated, we can bind our previous `entriesController` and `entriesSrv`variables to the instances of the controller and the service inside the module. This is achieved with the `module.get` method call.

Once all this initial setup is done, we are good to start writing some actual tests. Let's implement one that checks whether the `findAll()` method in our controller correctly returns an array of entries, even if we only have one entry:

```
import { Test } from '@nestjs/testing';
import { EntriesController } from './entry.controller';
import { EntriesService } from './entry.service';

describe('EntriesController', () => {
  let entriesController: EntriesController;
  let entriesSrv: EntriesService;

  beforeEach(async () => {
    const module = await Test.createTestingModule({
      controllers: [EntriesController],
    })
      .overrideComponent(EntriesService)
      .useValue({ findAll: () => null })
      .compile();

    entriesSrv = module.get<EntriesService>(EntriesService);
    entriesController =
module.get<EntriesController>(EntriesController);
  });

  describe('findAll', () => {
    it('should return an array of entries', async () => {
      expect(Array.isArray(await entriesController.findAll())).toBe(true);
    });
  });
});
```

The `describe('findAll', () => {` line is the one that starts the actual test suite. We expect the resolved value of `entriesController.findAll()` to be an array. This is basically how we wrote the code in the first place, so it should work, right? Let's run the tests with `npm test` and check the test output.

```
FAIL  src/modules/entry/entry.controller.spec.ts
  EntriesController
    findAll
      ✕ should return an array of entries (4ms)

  ● EntriesController › findAll › should return an array of entries

    expect(received).toBe(expected) // Object.is equality

    Expected value to be:
      true
    Received:
      false

      30 |       ];
      31 |       // jest.spyOn(entriesSrv, 'findAll').mockImplementation(() => result);
    > 32 |       expect(Array.isArray(await entriesController.findAll())).toBe(true);
      33 |     });
      34 |
      35 |     // it('should return the entries retrieved from the service', async () => {

      at src/modules/entry/entry.controller.spec.ts:32:64
      at fulfilled (src/modules/entry/entry.controller.spec.ts:3:50)

Test Suites: 1 failed, 1 total
```

```
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.112s, estimated 2s
Ran all test suites related to changed files.
```

It failed... Well, of course it failed! Remember the `beforeEach()` method?

```
...
.overrideComponent(EntriesService)
.useValue({ findAll: () => null })
.compile();
...
```

We told Nest.js to exchange the original `findAll()` method in the service for another one that returns just `null`. We will need to tell Jest to mock that method with something that returns an array, so to check that when the `EntriesService` returns an array, the controller is in fact returning that result as an array as well.

```
...
describe('findAll', () => {
   it('should return an array of entries', async () => {
      jest.spyOn(entriesSrv, 'findAll').mockImplementationOnce(() => [{}]);
      expect(Array.isArray(await entriesController.findAll())).toBe(true);
   });
});
...
```

In order to mock the `findAll()` method from the service, we are using two Jest methods. `spyOn()` takes an object and a method as arguments, and starts watching the method for its execution (in other words, sets up a *spy*).
And `mockImplementationOnce()`, which as its name implies changes the implementation of the method when it's next called (in this case, we change it to return an array of one empty object.)

Let's try to run the test again with `npm test`:

```
 PASS  src/modules/entry/entry.controller.spec.ts
  EntriesController
```

```
    findAll
      ✓ should return an array of entries (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.134s, estimated 2s
Ran all test suites related to changed files.
```

The test is passing now, so you can be sure that the `findAll()` method on the controller will always behave itself and return an array, so that other code components that depend on this output being an array won't break themselves.

If this test started to fail at some point in the future, it would mean that we had introduced a regression in our codebase. One of the great sides of automated testing is that we will be notified about this regression before it's too late.

Testing for equality

Up until this point, we are sure that `EntriesController.findAll()` returns an array. We can't be sure that it's not an array of empty objects, or an array of booleans, or just an empty array. In other words, we could rewrite the method to something like `findAll() { return []; }` and the test would still pass.

So, let's improve our tests to check that the method really returns the output from the service, without messing things up.

```
import { Test } from '@nestjs/testing';
import { EntriesController } from './entry.controller';
import { EntriesService } from './entry.service';

describe('EntriesController', () => {
  let entriesController: EntriesController;
  let entriesSrv: EntriesService;

  beforeEach(async () => {
    const module = await Test.createTestingModule({
      controllers: [EntriesController],
    })
```

```
        .overrideComponent(EntriesService)
        .useValue({ findAll: () => null })
        .compile();

      entriesSrv = module.get<EntriesService>(EntriesService);
      entriesController =
module.get<EntriesController>(EntriesController);
   });

   describe('findAll', () => {
      it('should return an array of entries', async () => {
         jest.spyOn(entriesSrv, 'findAll').mockImplementationOnce(() => [{}]);
         expect(Array.isArray(await entriesController.findAll())).toBe(true);
      });

      it('should return the entries retrieved from the service', async ()
=> {
         const result = [
            {
               uuid: '1234567abcdefg',
               title: 'Test title',
               body:
                  'This is the test body and will serve to check whether
the controller is properly doing its job or not.',
            },
         ];
         jest.spyOn(entriesSrv, 'findAll').mockImplementationOnce(() =>
result);

         expect(await entriesController.findAll()).toEqual(result);
      });
   });
});
```

We just kept most of the test file as it was before, although we did add a new test, the last one, in which:

- We set an array of one *not-empty* object (the `result` constant).
- We mock the implementation of the service's `findAll()` method once again to return that `result`.

- We check that the controller returns the `result` object exactly as the original when called. Note that we are using the Jest's `.toEqual()` method which, unlike `.toBe()`, performs a deep equality comparison between both objects for all of their properties.

This is what we get when we run `npm test` again:

```
 PASS  src/modules/entry/entry.controller.spec.ts
  EntriesController
    findAll
      ✓ should return an array of entries (2ms)
      ✓ should return the entries retrieved from the
service (1ms)


Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.935s, estimated 2s
Ran all test suites related to changed files.
```

Both our tests pass. We accomplished quite a lot already. Now that we have a solid foundation, extending our tests to cover as many test cases as possible will be an easy task.

Of course, we have only written a test for a controller. But testing services and the rest of the pieces of our Nest.js app works the same way.

Covering our code in tests

One critical aspect in code automation is code coverage reporting. Because, how do you know that your tests actually cover as many test cases as possible? Well, the answer is checking code coverage.

If you want to be really confident in your tests as a regression detection systems, make sure that they cover as much functionality as possible. Let's imagine we have a class with five methods and we only write tests for two of them. We would have roughly two-fifths of the code covered with tests, which

means that we won't have any insights about the other three-fifths, and about whether they still work as our codebase keeps on growing.

Code coverage engines analyze our code and tests together, and check the amount of lines, statements, and branches that are covered by the tests running in our suites, returning a percentage value.

As mentioned in previous sections, Jest already includes code coverage reporting out of the box, you just need to activate it by passing a `--coverage` argument to the `jest` command.

Let's add a script in our `package.json` file that, when executed, will generate the coverage report:

```
{
  ...
  "scripts": {
    ...
    "test:coverage":"jest --config=jest.json --coverage --
coverageDirectory=coverage",
    ...
  }
}
```

When running `npm run test:coverage` on the controller written before, you will see the following output:

```
 PASS  src/modules/entry/entry.controller.spec.ts
  EntriesController
    findAll
      ✓ should return an array of entries (9ms)
      ✓ should return the entries retrieved from the
service (2ms)

----------------------|----------|----------|----------|---
-------|-------------------|
File                  |  % Stmts | % Branch |  % Funcs |  %
Lines | Uncovered Line #s |
```

```
--------------------|----------|----------|----------|---
-------|-------------------|
All files             |      100 |    66.67 |      100 |
100 |                   |
 entry.controller.ts |      100 |    66.67 |      100 |
100 |                 6 |
--------------------|----------|----------|----------|---
-------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        4.62s
Ran all test suites.
```

In order to have a better vision of the console output within this book, we will transform the console output to a proper table.

File	% Stmts	% Branch	% Funcs	% Lines	Ur
All files	100	66.67	100	100	
entry.controller.ts	100	66.67	100	100	6

You can easily see we are covering 100% of our code lines in our tests. This makes sense since we wrote two tests for the only method in our controller.

FAILING TESTS FOR LOW COVERAGE

Let's imagine now that we work in a complex project with several developers working on the same base at the same time. Let's imagine also that our workflow includes a Continuous Integration/Continuous Delivery pipeline, running on something like Travis CI, CircleCI, or even Jenkins. Our pipeline would likely include a step that runs our automated tests before merging or deploying, so that the pipeline will be interrupted if the tests fail.

All the imaginary developers working in this imaginary project will add (as well as refactor and delete, but those cases don't really apply to this example) new functionality (i.e. *new code*) all the time, but they might forget about properly

testing that code. What would happen then? The coverage percentage value of the project would go down.

In order to still be sure that we can rely on our tests as a regression detection mechanism, we need to be sure that the coverage never goes *too low*. What is too low? That really depends on multiple factors: the project and the stack it uses, the team, etc. However, it' normally a good rule of thumb not letting the coverage value go down on each coding process iteration.

Anyway, Jest allows you to specify a coverage threshold for tests: if the value goes below that threshold, the tests will return failed *even if they all passed*. This way, our CI/CD pipeline will refuse to merge or deploy our code.

The coverage threshold has to be included in the Jest configuration object; in our case, it lives in the `jest.json` file in our project's root folder.

```
{
  ...
  "coverageThreshold": {
    "global": {
      "branches": 80,
      "functions": 80,
      "lines": 80,
      "statements": 80
    }
  }
}
```

Each number passed to each property of the object is a percentage value; below it, the tests will fail.

To demonstrate it, let's run our controller tests with the coverage threshold set as above. `npm run test:coverage` returns this:

```
 PASS  src/modules/entry/entry.controller.spec.ts
  EntriesController
    findAll
      ✓ should return an array of entries (9ms)
      ✓ should return the entries retrieved from the
service (1ms)
```

```
----------------------|----------|----------|----------|---
-------|-------------------|
File                  |  % Stmts | % Branch |  % Funcs |  %
Lines | Uncovered Line #s |
----------------------|----------|----------|----------|---
-------|-------------------|
All files             |      100 |    66.67 |      100 |
100 |                   |
 entry.controller.ts |      100 |    66.67 |      100 |
100 |                 6 |
----------------------|----------|----------|----------|---
-------|-------------------|
Jest: "global" coverage threshold for branches (80%) not
met: 66.67%

Test Suites: 1 passed, 1 total

Tests:       2 passed, 2 total

Snapshots:   0 total

Time:        2.282s, estimated 4s

Ran all test suites.

npm ERR! code ELIFECYCLE

npm ERR! errno 1

npm ERR! nest-book-example@1.0.0 test:coverage: `jest --
config=jest.json --coverage --coverageDirectory=coverage`

npm ERR! Exit status 1

npm ERR!

npm ERR! Failed at the nest-book-example@1.0.0
test:coverage script.

npm ERR! This is probably not a problem with npm. There is
likely additional logging output above.
```

As you can see, the tests passed, yet the process failed with status 1 and returned an error. Also, Jest reported `"global" coverage threshold for branches (80%) not met: 66.67%`. We have successfully kept non-acceptable code coverage away from our main branch or productive environments.

The following step could be now to implement a few end-to-end tests, along with our unit tests, to improve our system.

E2E testing

While unit tests are isolated and independent by definition, end-to-end (or E2E) tests serve for, in a way, the opposite function: they intend to check the health of the system as a whole, and try to include as many components of the solution as possible. For this reason, in E2E tests we will focus on testing complete modules, rather than isolated components or controllers.

Preparation

Fortunately, we can use Jest for E2E testing just like we did for unit testing. We will only need to install the `supertest` npm package to perform API requests and assert their result. Let's install it by running `npm install --save-dev supertest` in your console.

Also, we will create a folder called `e2e` in our project's root folder. This folder will hold all of our E2E test files, as well as the configuration file for them.

This brings us to the next step: create a new `jest-e2e.json` file inside the `e2e` folder with the following contents:

```
{
  "moduleFileExtensions": ["js", "ts", "json"],
  "transform": {
    "^.+\\.tsx?$": "<rootDir>/node_modules/ts-jest/preprocessor.js"
  },
  "testRegex": "/e2e/.*\\.(e2e-test|e2e-spec).ts|tsx|js)$",
  "coverageReporters": ["json", "lcov", "text"]
}
```

As you can see, the new E2E configuration object is very similar to the one for unit tests; the main difference is the `testRegex` property, which now points to files in the `/e2e/` folder that have a `.e2e-test` or `e2e.spec` file extension.

The final step of the preparation will be to include an npm script in our `package.json` file to run the end-to-end tests:

```
{
  ...
  "scripts": {
    ...
    "e2e": "jest --config=e2e/jest-e2e.json --forceExit"
  }
  ...
}
```

Writing end-to-end tests

The way of writing E2E tests with Jest and Nest.js is also very similar to the one we used for unit tests: we create a testing module using the `@nestjs/testing` package, we override the implementation for the `EntriesService` to avoid the need for a database, and then we are ready to run our tests.

Let's write the code for the test. Create a new folder called `entries` inside the `e2e` folder, and then create a new file there called `entries.e2e-spec.ts` with the following content:

```
import { INestApplication } from '@nestjs/common';
import { Test } from '@nestjs/testing';
import * as request from 'supertest';

import { EntriesModule } from '../../src/modules/entry/entry.module';
import { EntriesService } from '../../src/modules/entry/entry.service';

describe('Entries', () => {
  let app: INestApplication;
  const mockEntriesService = { findAll: () => ['test'] };

  beforeAll(async () => {
    const module = await Test.createTestingModule({
      imports: [EntriesModule],
    })
      .overrideComponent(EntriesService)
      .useValue(mockEntriesService)
```

```
      .compile();

    app = module.createNestApplication();
    await app.init();
  });

  it(`/GET entries`, () => {
    return request(app.getHttpServer())
      .get('/entries')
      .expect(200)
      .expect({
        data: mockEntriesService.findAll(),
      });
  });

  afterAll(async () => {
    await app.close();
  });
});
```

Let's review what the code does:

1. The `beforeAll` method creates a new testing module, imports the `EntriesModule` in it (the one we are going to test), and overrides the `EntriesService` implementation with the very simple `mockEntriesService` constant. Once that's done, it uses the `.createNestApplication()` method to create an actual running app to make requests to, and then waits for it to be initialized.
2. The `'/GET entries'` test uses a supertest to perform a GET request to the `/entries` endpoint, and then asserts whether the status code of the response from that request was a `200` and the received body of the response matches the `mockEntriesService` constant value. If the test passes, it will mean that our API is correctly reacting to requests received.
3. The `afterAll` method ends the Nest.js app we created when all of the tests have run. This is important to avoid side effects when we run the tests the next time.

Summary

In this chapter we have explored the importance of adding automated tests to our projects and what kind of benefits it brings.

Also, we got started with the Jest testing framework, and we learned how to configure it in order to use it seamlessly with TypeScript and Nest.js

Lastly, we reviewed how to use the testing utilities that Nest.js provides for us, and learned how to write tests, both unit tests as well as end-to-end ones, and how to check the percentage of the code our tests are covering.

In the next and last chapter we cover server-side rendering with Angular Universal.

Chapter 15. Server-side Rendering with Angular Universal

If you are not familiar with the Angular platform for client-side application development it is worth taking a look into. Nest.js has a unique symbiotic relationship with Angular because they are both written in TypeScript. This allows for some interesting code sharing between your Nest.js server and Angular app, because Angular and Nest.js both use TypeScript, and classes can be created in a package that is shared between the Angular app and the Nest.js app. These classes can then be included in either app and help keep the objects that are sent and received over HTTP requests between client and server consistent. This relationship is taken to another level when we introduce Angular Universal. Angular Universal is a technology that allows your Angular app to be pre-rendered on your server. This has a number of benefits such as:

1. Facilitate web crawlers for SEO purposes.
2. Improve the load performance of your site.
3. Improve performance of the site on low-powered devices and mobile.

This technique is called server-side rendering and can be extremely helpful, but, requires some restructuring of the project as the Nest.js server and Angular app are built in sequence and the Nest.js server will actually run the Angular app itself when a request is made to get a webpage. This essentially emulates the Angular app inside a browser complete with the API calls and loading any dynamic elements. This page built on the server is now served as a static webpage to the client and the dynamic Angular app is loaded quietly in the background.

If you are just hopping into this book now and want to follow along with the example repository it can be cloned with:

```
git clone https://github.com/backstopmedia/nest-book-example
```

Angular is another topic that can, and has, have an entire book written about it. We will be using an Angular 6 app that has been adapted for use in this book by one of the authors. The original repository can be found here.

```
https://github.com/patrickhousley/nest-angular-universal.git
```

This repository is using Nest 5 and Angular 6 so there have been some changes made, because this book is based on Nest 4. Not to worry, though, we have included an Angular Universal project inside the main repository shown at the start of this chapter. It can be found inside the `universal` folder at the root of the project. This is a self-contained Nest + Angular project, rather than adapting the main repository for this book to serve an Angular app, we isolated it to provide a clear and concise example.

Serving the Angular Universal App with Nest.js

Now that we are going to be serving the Angular app with our Nest.js server, we are going to have to compile them together so that when our Nest.js server is run, it knows where to look for the Universal app. In our `server/src/main.ts` file there are a couple of key things we need to have in there. Here we create a function `bootstrap()` and then call it from below.

```
async function bootstrap() {
  if (environment.production) {
    enableProdMode();
  }

  const app = await NestFactory.create(ApplicationModule.moduleFactory());

  if (module.hot) {
    module.hot.accept();
    module.hot.dispose(() => app.close());
  }

  await app.listen(environment.port);
}

bootstrap()
  .then(() => console.log(`Server started on port ${environment.port}`))
  .catch(err => console.error(`Server startup failed`, err));
```

Let's step through this function line by line.

```
if (environment.production) {
    enableProdMode();
  }
```

This tells the application to enable production mode for the application. There are many differences between production and development modes when writing web servers, but this is required if you want to run a web server in a production context.

```
const app = await NestFactory.create(ApplicationModule.moduleFactory());
```

This will create the Nest app variable of type `INestApplication` and will be run using `ApplicationModule` in the `app.module.ts` file as the entry point. `app` will be the instance of the Nest app that is running on port `environment.port`, which can be found in `src/server/environment/environment.ts`. There are three different environment files here:

1. `environment.common.ts`-As its name implies, this file is common between both production and development builds. It provides information and paths on where to find the packaged build files for the server and client applications.
2. `environment.ts`-This is the default environment used during development, and it includes the settings from the `environment.common.ts` file as well as sets `production: false` and the port mentioned above to 3000.
3. `environment.prod.ts`-This file mirrors #2 except that it sets `production: true` and does not define a port, instead defaulting to default, normally 8888.

If we are developing locally and want to have hot reloading, where the server restarts if we change a file, then we need to have the following included in our `main.ts` file.

```
if (module.hot) {
   module.hot.accept();
   module.hot.dispose(() => app.close());
}
```

This is set within the `webpack.server.config.ts` file based on our `NODE_ENV` environment variable.

Finally, to actually start the server, call the `.listen()` function on our `INestApplication` variable and pass it a port to run on.

```
await app.listen(environment.port);
```

We then call `bootstrap()`, which will run the function described above. At this stage we now have our Nest server running and able to serve the Angular App and listen to serve API requests.

In the `bootstrap()` function above when creating the `INestApplication` object we supplied it with `ApplicationModule`. This is the entry point for the app and handles both the Nest and Angular Universal Apps. In `app.module.ts` we have:

```
@Module({
  imports: [
    HeroesModule,
    ClientModule.forRoot()
  ],
})
export class ApplicationModule {}
```

Here we are importing two Nest modules, the `HeroesModule`, which will supply the API endpoints for the *Tour of Heroes* application, and the `ClientModule` that is the module that is handling the Universal stuff. The `ClientModule` has a lot going on, but we will touch on the main things that are handling setting up Universal, here is the code for this module.

```
@Module({
  controllers: [ClientController],
  components: [...clientProviders],
})
export class ClientModule implements NestModule {
  constructor(
    @Inject(ANGULAR_UNIVERSAL_OPTIONS)
    private readonly ngOptions: AngularUniversalOptions,
    @Inject(HTTP_SERVER_REF) private readonly app: NestApplication
  ) {}

  static forRoot(): DynamicModule {
    const requireFn = typeof __webpack_require__ === "function" ?
__non_webpack_require__ : require;
    const options: AngularUniversalOptions = {
      viewsPath: environment.clientPaths.app,
      bundle: requireFn(join(environment.clientPaths.server, 'main.js'))
    };

    return {
```

```
        module: ClientModule,
        components: [
          {
            provide: ANGULAR_UNIVERSAL_OPTIONS,
            useValue: options,
          }
        ]
      };
  }

  configure(consumer: MiddlewareConsumer): void {
    this.app.useStaticAssets(this.ngOptions.viewsPath);
  }
}
```

We will start with the `@Module` decorator at the top of the file. As with regular Nest.js modules (And Angular, remember how Nest.js is inspired by Angular?), there are `controllers` (for the endpoints) property and a `components` (for services, providers and other components we want to be part of this module) property. Here we are including the `ClientController` in the `controllers` array and `...clientProviders` in `components`. Here the triple dot (`...`) essentially means "insert each of the array elements into this array." Let's disect each of these a bit more.

`ClientController`

```
@Controller()
export class ClientController {
  constructor(
    @Inject(ANGULAR_UNIVERSAL_OPTIONS) private readonly ngOptions:
AngularUniversalOptions,
  ) { }

  @Get('*')
  render(@Res() res: Response, @Req() req: Request) {
    res.render(join(this.ngOptions.viewsPath, 'index.html'), { req });
  }
}
```

This is like any other controller that we have learned about, but with one small difference. At the URL path `/*` instead of supplying an API endpoint, the

Nest.js server will render an HTML page, namely `index.html`, from that same `viewsPath` we have seen before in the environment files.

As for the `clientProoviders` array:

```
export const clientProviders = [
  {
    provide: 'UNIVERSAL_INITIALIZER',
    useFactory: async (
      app: NestApplication,
      options: AngularUniversalOptions
    ) => await setupUniversal(app, options),
    inject: [HTTP_SERVER_REF, ANGULAR_UNIVERSAL_OPTIONS]
  }
];
```

This is similar to how we define our own provider inside the return statement of `ClientModule`, but instead of `useValue` we use `useFactory`, this passes in the Nest app and the `AngularUniversalOptions` we defined earlier to a function `setupUniversal(app, options)`. It has taken us a while, but this is where the Angular Universal server is actually created.

`setupUniversal(app, options)`

```
export function setupUniversal(
  app: NestApplication,
  ngOptions: AngularUniversalOptions
) {
  const { AppServerModuleNgFactory, LAZY_MODULE_MAP } =
ngOptions.bundle;

  app.setViewEngine('html');
  app.setBaseViewsDir(ngOptions.viewsPath);
  app.engine(
    'html',
    ngExpressEngine({
      bootstrap: AppServerModuleNgFactory,
      providers: [
        provideModuleMap(LAZY_MODULE_MAP),
        {
          provide: APP_BASE_HREF,
          useValue: `http://localhost:${environment.port}`
```

```
            }
          ]
        })
    );
}
```

There are three main functions being called here: `app.setViewEngine()`, `app.setBaseViewDir()`, and an `app.engine`. The first `.setViewEngine()` is setting the view engine to HTML so that the engine rendering views knows we are dealing with HTML. The second `.setBaseViewDir()` is telling Nest.js where to find the HTML views, which again was defined in the `environment.common.ts` file from earlier. The last is very important, `.engine()` defines the HTML engine to use, in this case, because we are using Angular, it is `ngExpressEngine`, which is the Angular Universal engine. Read more about the Universal express-engine here: https://github.com/angular/universal/tree/master/modules/express-engine. This sets `bootstrap` to the `AppServerModuleNgFactory` object, which is discussed in the next section.

In the `ClientModule` we can see the `.forRoot()` function that was called when we imported the `ClientModule` in the `AppliationModule` (server entry point). Essentially, `forRoot()` is defining a module to return in place of the originally imported `ClientModule`, also called `ClientModule`. This module being returned has a single component that provides `ANGULAR_UNIVERSAL_OPTIONS`, which is an interface that defines what kind of object will be passed into the `useValue` property of the component.

The structure of `ANGULAR_UNIVERSAL_OPTIONS` is:

```
export interface AngularUniversalOptions {
  viewsPath: string;
  bundle: {
    AppServerModuleNgFactory: any,
    LAZY_MODULE_MAP: any
  };
}
```

It follows that the value of `useValue` is the contents of `options` defined at the top of `forRoot()`.

```
const options: AngularUniversalOptions = {
  viewsPath: environment.clientPaths.app,
  bundle: requireFn(join(environment.clientPaths.server, 'main.js'))
```

```
};
```

The value of `environment.clientPaths.app` can be found in the `environment.common.ts` file we discussed earlier. As a reminder, it points to where to find the compiled client code to be served. You may be wondering why the value of `bundle` is a require statement when the interface clearly says it should be of the structure:

```
bundle: {
    AppServerModuleNgFactory: any,
    LAZY_MODULE_MAP: any
};
```

Well, if you trace that require statement back (`..` means go up one directory) then you will see we are setting the `bundle` property equal to another module `AppServerModule`. This will be discussed in a bit, but the Angular App will end up being served.

The last piece in the `ClientModule` is in the `configure()` function that will tell the server where to find static assets.

```
configure(consumer: MiddlewareConsumer): void {
    this.app.useStaticAssets(this.ngOptions.viewsPath);
}
```

Building and running the Universal App

Now that you have the Nest.js and Angular files setup, it is almost time to run the project. There are a number of configuration files that need your attention and are found in the example project: https://github.com/backstopmedia/nest-book-example. Up until now we have been running the project with `nodemon` so that our changes are reflected whenever the project is saved, but, now that we are packaging it up to be serving an Angular App we need to build the server using a different package. For this, we have chosen `udk`, which is a `webpack` extension. It can both build our production bundles as well as start a development server, much like `nodemon` did for our plain Nest.js app. It would be a good idea to familiarize yourself with the following configuration files:

1. `angular.json`-Our Angular config file that handles things like, what environment file to use, commands that can be used with `ng`, and the Angular CLI command.

2. `package.json`-The project global dependency and command file. This file defines what dependencies are needed for production and development as well as what commands are available for your command line tool, ie. `yarn` or `npm`.
3. `tsconfig.server.json`-An extension on the global `tsconfig.json` file, this provides some Angular Compiler options like where to find the Universal entry point.

Summary

And that is it! We have a working Angular Universal project to play around with. Angular is a great client side framework that has been gaining a lot of ground lately. There is much more that can be done here as this chapter only scratched the surface, especially in terms of Angular itself.

And, this is the last chapter in this book. We hope you are excited to use Nest.js to create all sorts of apps.