Practical Nest.js

Develop clean MVC web applications

Daniel Correa - Greg Lim

Practical Nest.js

Develop clean MVC web applications

Daniel Correa – Greg Lim

Practical Books

Practical Nest.js

by Daniel Correa and Greg Lim

First Edition: January 2022.

Contributors

About the author

Daniel Correa has been a researcher and a software developer for several years. Daniel has a Ph.D. in Computer Science; currently, he is a professor at Universidad EAFIT in Colombia. He is interested in software architectures, frameworks (such as Laravel, Django, Nest, Express, Vue, React, Angular, and many more), web development, and clean code.

Daniel is very active on Twitter; he shares tips about software development and reviews software engineering books. Contact Daniel on Twitter at @danielgarax.

About the co-author

Greg Lim is a technologist and author of several programming books. Greg has many years in teaching programming in tertiary institutions and he places special emphasis on learning by doing. Contact Greg at support@i-ducate.com or on Twitter at @greglim81.

Table of Contents

Preface

Nest (NestJS) is a framework for building efficient, scalable Node.js server-side applications. We will use Nest to develop an Online Store application that uses several Nest features. The Online Store application will be the means to understand straightforward and complex Nest concepts and how Nest features and third-party libraries can be used to implement real-world applications.

The main difference between this book and other similar books, is that this book is not just about Nest. Instead, this book is about a “clean” design and implementation of web applications using Nest. By ‘clean’, we refer to an understandable, maintainable, usable, and well-divided application.

The authors have developed several applications over many frameworks, including Laravel, Django, Nest, Express, Flask, Spring, Vue, Angular, and React. And we are going to use that knowledge to create a clean design and clean code strategies which can be applied not just to Nest, but to the design and implementation of most web applications using frameworks such as Django, Laravel, Flask, Express, and more.

This book is written with brief explanations direct to the point. It includes tips, short discussions, and useful phrases found in other books that we have read to provide you with a practical approach that will improve your coding skills.

This is a short book divided into 31 chapters, with six pages on average per chapter. It was designed not to overwhelm you. With this division, you will feel like you're making fast progress. We won’t cover all Nest features, but some of the most important to develop MVC web applications.

We hope you enjoy this journey as we did when we wrote this book.

Who is this book for?

This book is for web developers or programmers who want to learn Nest and improve their coding skills. No previous knowledge of Nest is required. However, basic programming knowledge is required. This book is also suitable for experienced Nest developers. They can revise previous concepts and learn new clean code strategies.

Download the example code files

You can download the example code files for this book from the GitHub repository https://github.com/PracticalBooks/Practical-Nest. In it, you will find the code of each chapter. You can replicate this book’s code or download the code directly from GitHub. If there's an update to the code, it will be updated on the existing GitHub repository.

Questions and discussions

If you have questions about any aspect of this book or want to discuss something, we recommend you to use the discussion zone of the GitHub repository (see Fig. P-1). In that way, you can learn from other questions, and we can learn from you. Besides, others in the community can answer your questions.

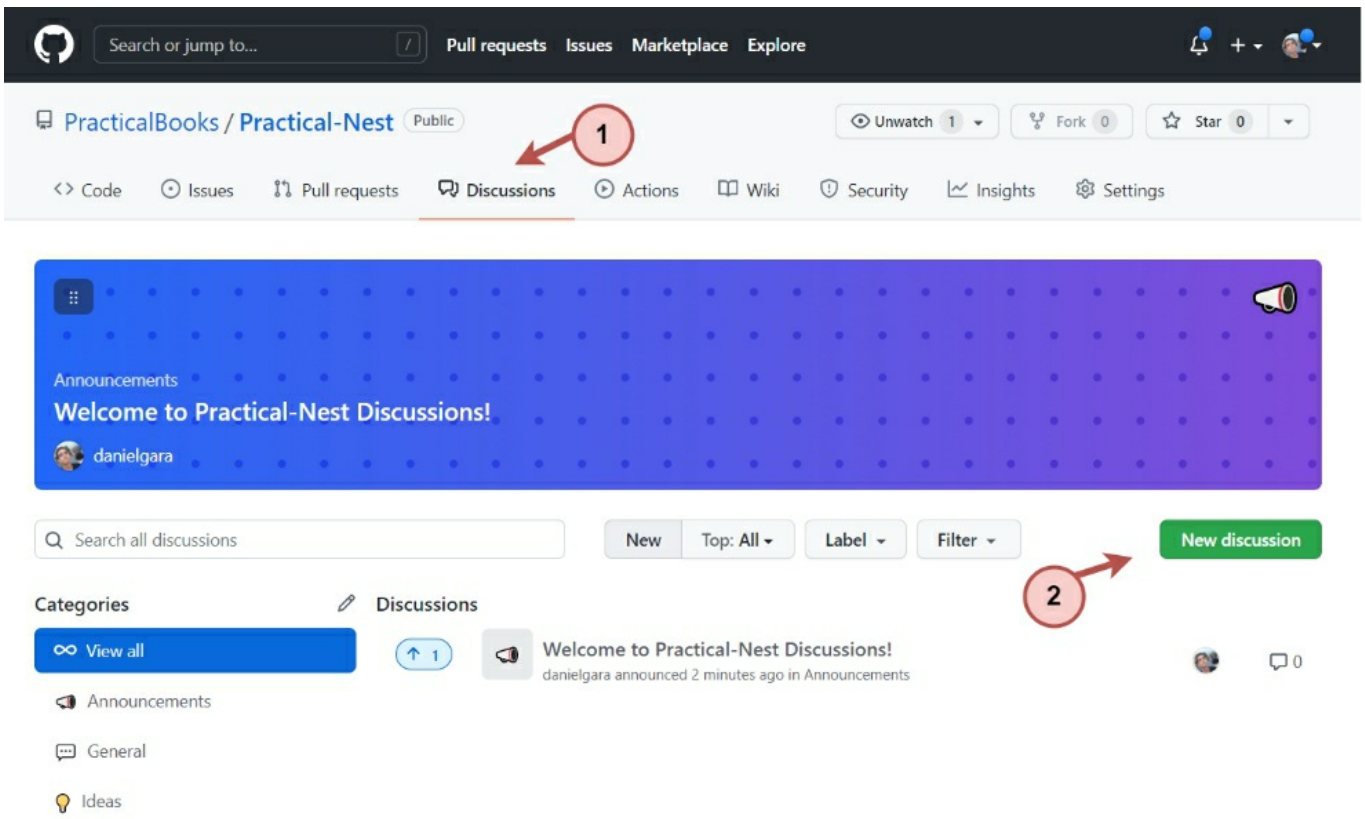

Figure P-1. Discussion zone of the GitHub repository.

Additionally, you can email your questions to practicalbooksco@gmail.com. Please mention the book title in the subject of your message.

Download colored images

We also provide a PDF file with colored images of the figures/diagrams used in this book. You can download it here: https://github.com/PracticalBooks/Practical-Nest/tree/main/BookImages.

Getting book updates

If you want to receive book updates, please email us at practicalbooksco@gmail.com. We will also subscribe you to our mailing list.

Chapter 01 – Introduction

We will begin our journey to understand and apply many Nest concepts and features to develop MVC web applications.

The book is divided into the following chapters. We will highlight the Nest concepts we will learn and third-party libraries and tools we will use across the chapters.

- **Chapter 01 – Introduction.**
- **Chapter 02 – Online Store running example.**
- **Chapter 03 – Introduction to Nest and Installation:** Nest, Node.js, and npm.
- **Chapter 04 – Handlebars:** Handlebars and views.
- **Chapter 05 – Introduction to MVC applications.**
- **Chapter 06 – Layout View:** Bootstrap and Handlebars Partials.
- **Chapter 07 – Index and About Pages:** Nest Controllers.
- **Chapter 08 – Refactoring Index and About Pages.**
- **Chapter 09 – Use of a Coding Standard:** ESLint.
- **Chapter 10 – List Products with Dummy Data:** Nest Modules.
- **Chapter 11 – Configuration of MySQL Database:** XAMPP, MySQL, and phpMyAdmin.
- **Chapter 12 – Configuration of TypeORM:** Nest Databases and TypeORM.
- **Chapter 13 – Product Entity:** TypeORM Entities.
- **Chapter 14 – List Products with Database Data:** Nest Providers and Dependency Injection.
- **Chapter 15 – Refactoring List Products.**
- **Chapter 16 – Admin Panel.**
- **Chapter 17 – List Products in Admin Panel.**
- **Chapter 18 – Create Products:** forms.
- **Chapter 19 – Create Products with Images:** multer and Interceptors.
- **Chapter 20 – Edit and Delete Products.**
- **Chapter 21 – Create Users:** bcrypt.
- **Chapter 22 – Login System:** Session and Middleware.
- **Chapter 23 – Validations:** validator.
- **Chapter 24 – Authorizations.**
- **Chapter 25 – Shopping Cart.**
- **Chapter 26 – Orders and Items:** TypeORM relations.
- **Chapter 27 – Product Purchase.**
- **Chapter 28 – Orders Page**.
- **Chapter 29 – Deploying to the Cloud – Clever-Cloud – MySQL Database:** Clever-Cloud.
- **Chapter 30 – Deploying to the Cloud – Heroku – Nest Application:** Heroku.
- **Chapter 31 – Continue your Nest Journey.**

In this book, we will develop an Online Store. This Online Store will serve us to understand some of the more important Nest concepts. Figures 1-1, 1-2, 1-3, and 1-4 show the kind of application we will develop.

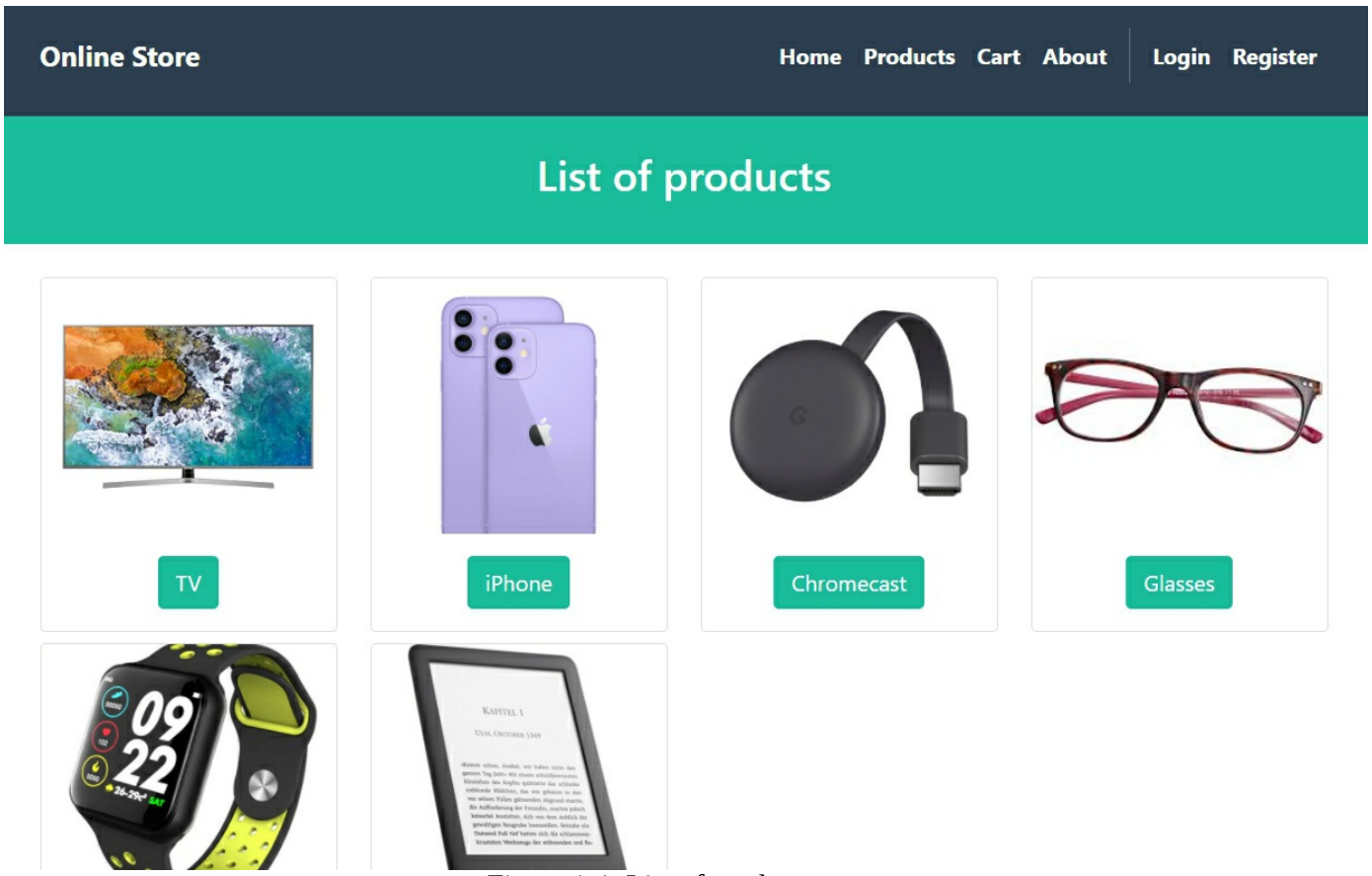

Figure 1-1. List of products page.

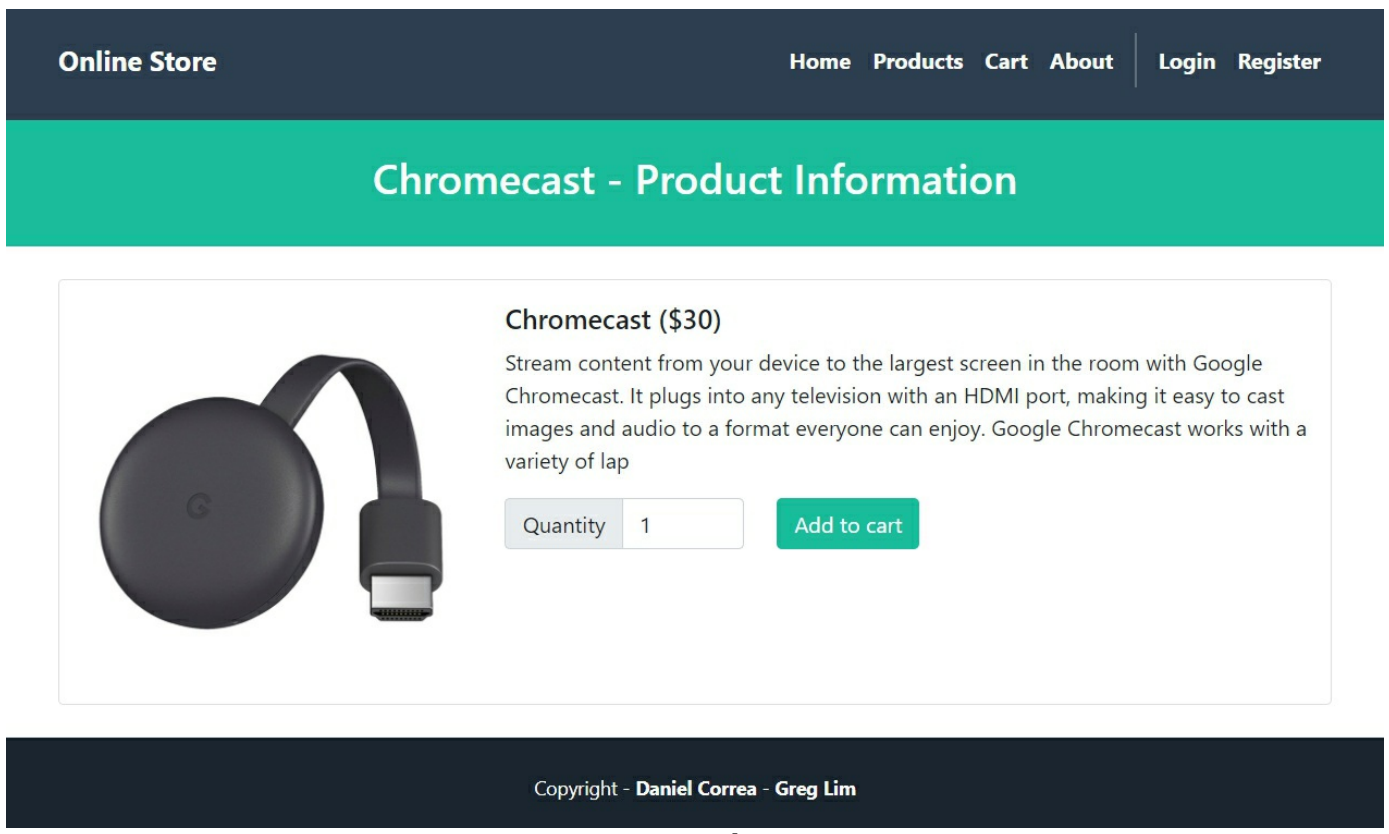

Figure 1-2. Product page.

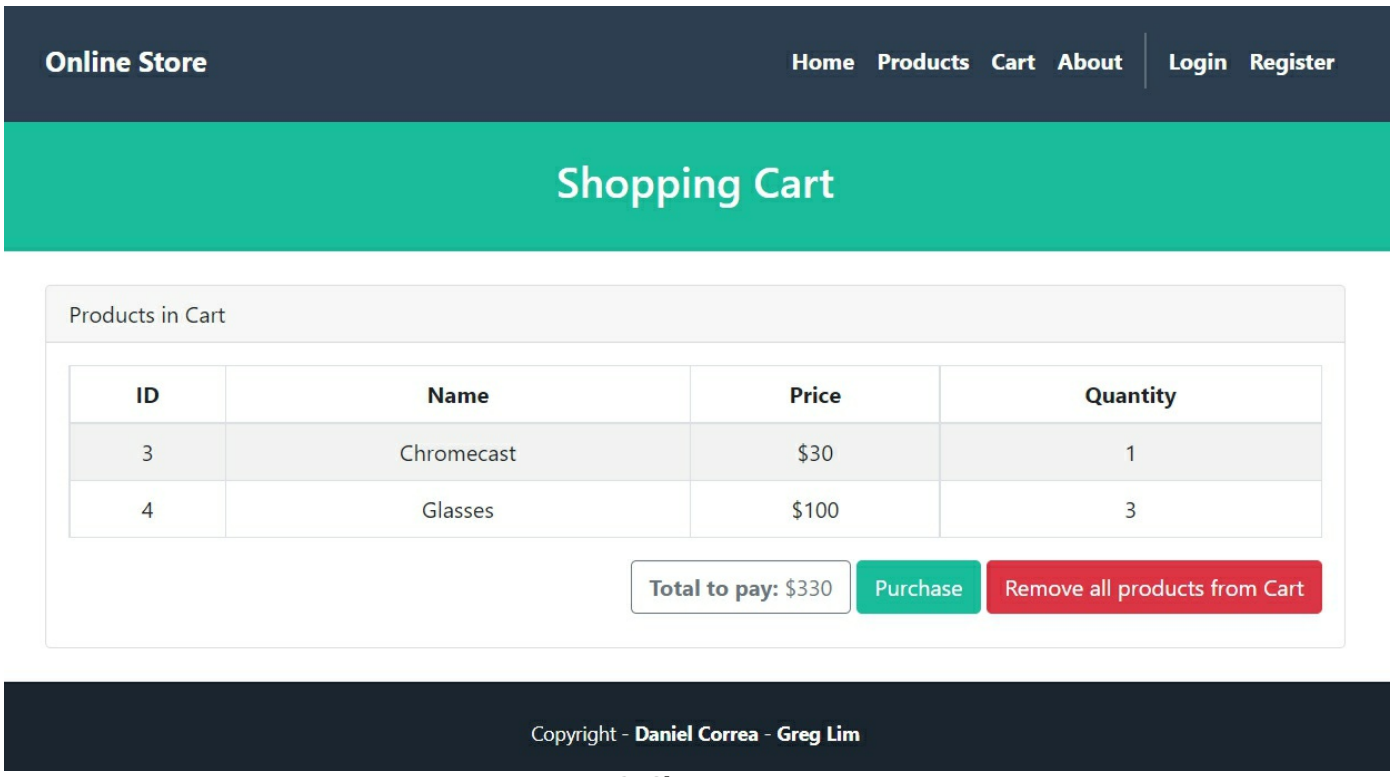

Figure 1-3. Shopping cart page.

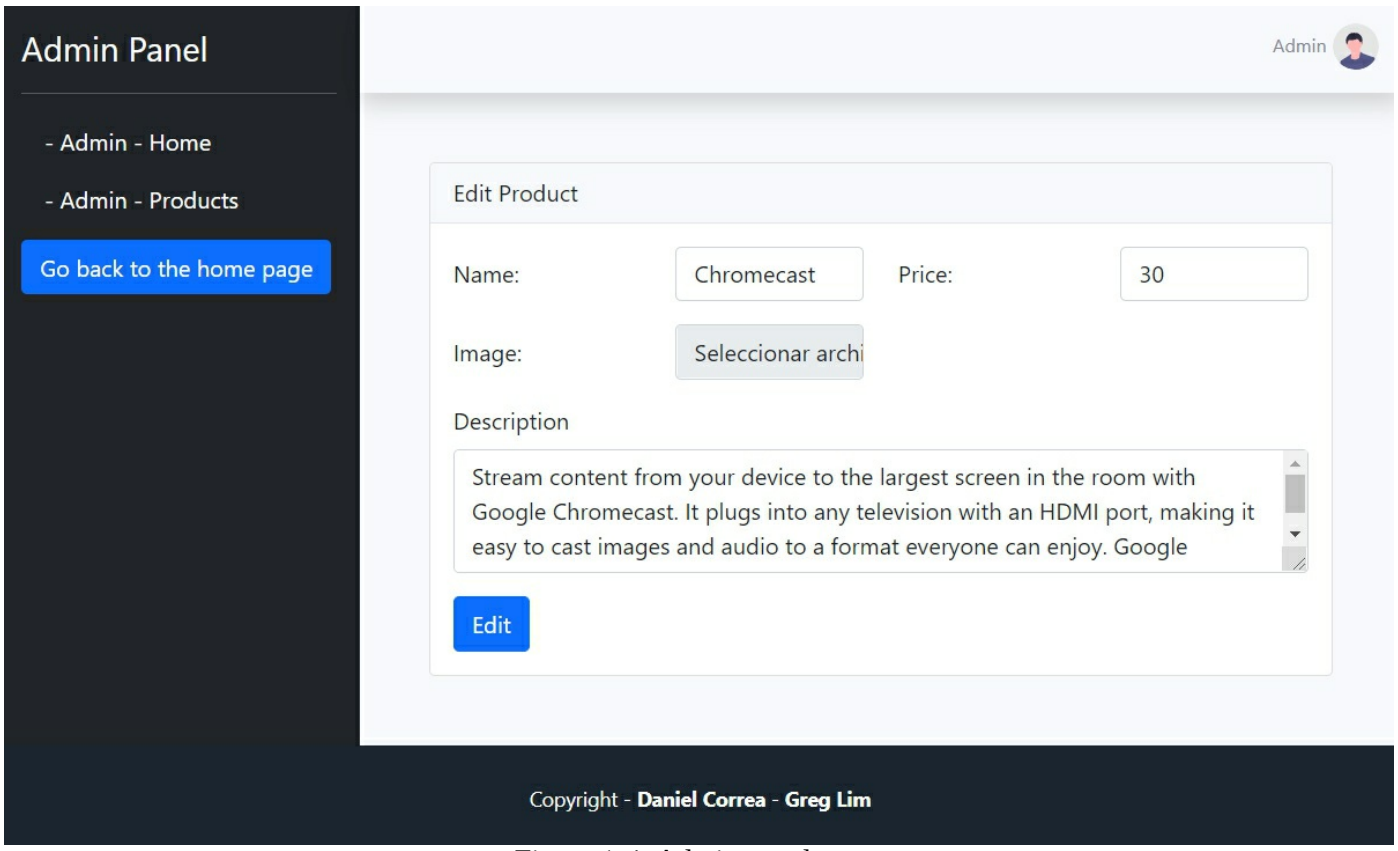

Figure 1-4. Admin panel page.

Let’s start our journey!

Chapter 02 – Online Store Running Example

Using a running example is a common strategy in programming books. A running example is an example where we visit repeatedly throughout the book. It provides a practical way to illustrate the concepts of a methodology, process, tool, or technique. In this case, we define an Online Store running example.

Online Store is a web application where users place orders to buy products.

Let's define the application scope for the app.

- **Home page** will display a welcome message and some images.
- **About page** will display information about the online store and developers.
- **Products page** will display the available products information. You can click on a specific product and see its information.
- **Cart page** will display the products added to the cart and the total price to be paid. A user can remove products from the cart and make purchases.
- **Login page** will display a form to allow users to log in to the application.
- **Register page** will display a form to allow users to sign up for accounts.
- **My orders page** will display the orders placed by the logged in user.
- **Admin panel** will contain sections to manage the store's products (create, update, delete, and list them).

The online store will be implemented with Nest (Node.js), MySQL database, Bootstrap (a CSS framework), and Handlebars (HBS; a templating system). We will learn about these elements in the upcoming chapters.

Below is a class diagram to illustrate the application scope and design (see Fig. 2-1). We have a *User* class with its data (id, name, email, password, etc.) which can place *Orders*. Each *Order* is composed of one or more *Items* which are related to a single *Product*. Each *Product* will have its corresponding data (id, name, description, image, etc.).

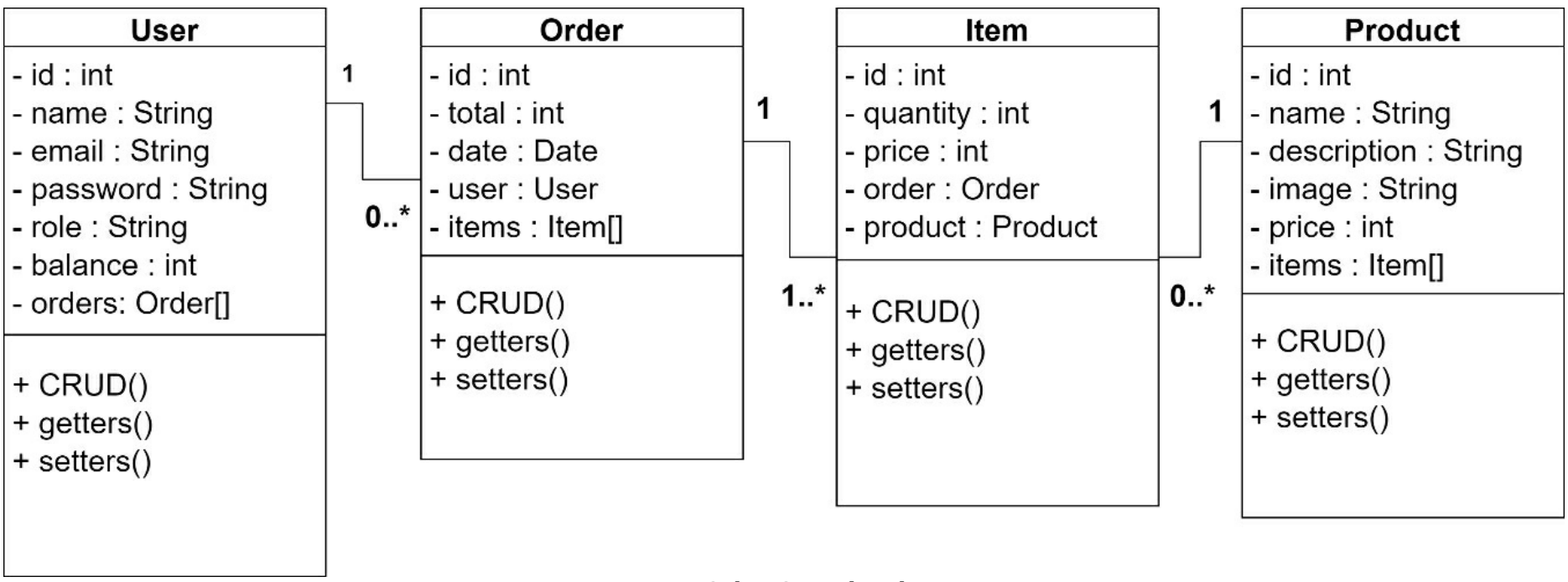

Figure 2-1. Online Store class diagram.

This book is not about class diagrams, so we won't explain other details in the class diagram. You will see a relationship between the code and this diagram as you advance through the book. This diagram serves as a blueprint for the construction of our application.

TIP: Designing a class diagram before starting to code helps us understand the application's scope and identify important data. It also helps us know how the application elements are related. You can share a diagram like this with your team or colleagues, obtain quick feedback, and make adjustments as needed. Since it is a diagram, changes can be made quickly. Else, when the project has been coded, the replacement cost will be higher to move data from one class to another. Let's check this phrase from (*2015 – Newman, S. - Building microservices*) book. *"I tend to do much of my thinking in the place where the cost of change and the cost of mistakes is as low as it can be: the whiteboard."*

Now that we have considered the kind of application we want to build, let's next understand what Nest is and how to install it.

Chapter 03 – Introduction to Nest and Installation

Introduction to Nest

Nest (NestJS) is a framework for building efficient, scalable Node.js server-side applications (https://docs.nestjs.com/). It uses JavaScript, is built with and fully supports TypeScript. It combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming). Under the hood, Nest makes use of robust HTTP Server frameworks like Express (https://expressjs.com/).

Nest vs Angular, React, and Vue

The main difference between Nest and Angular, React, and Vue is that Nest is used to develop server-side applications, while Angular, React, and Vue are used to develop front-end applications.

Nest vs Express

Both Nest and Express are used to develop server-side applications. The main difference is that Express is a minimalist framework which does not provide a predefined architecture. Instead, Nest provides an out-of-the-box application architecture which allows developers and teams to create highly testable, scalable, loosely coupled, and easily maintainable applications. The Nest architecture is heavily inspired by Angular.

Nest v8

When writing this book, the latest version is Nest 8 which we will use to build our Online Store application.

Note: A new Nest version might be available at the time you are reading this book. We recommend you continue using Nest 8 for this project. Once you complete this book, you can upgrade to the latest Nest version. In this way, most of the code will remain reusable. Some others might require minor adjustments.

Requirements (Node.js and npm)

To create Nest applications, we need to install Node.js and npm.

Node.js

Node.js is a free, open-source server environment. Node.js uses JavaScript on the server. Node.js represents a "JavaScript everywhere" paradigm, unifying web application development around a single programming language, rather than different languages for server-side and client-side scripts.

If you don’t have Node.js installed, go to https://nodejs.org/en/download/. Download and install it.

Once Node.js is installed, go to the Terminal, and execute the following command.

Execute in Terminal

```
node -v
```

If the installation was successful, you would see a result as presented in Fig 3-1.

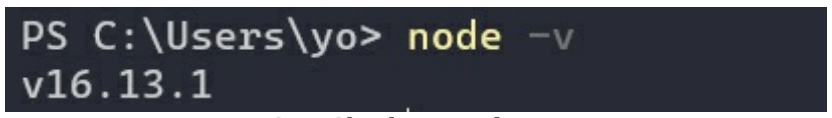

Figure 3-1. Checking Node.js version.

npm

npm is a package manager for Node.js. It consists of a command line client, called npm, and an online database of public and paid-for private packages, called the npm registry. With npm, you can declare the libraries your project depends on, and it will manage (install/update) them for you.

The Node.js installer already includes the npm package manager. So, we already have npm. Go to the Terminal and execute the following command.

Execute in Terminal

```
npm -v
```

You would see a result as presented in Fig 3-2.

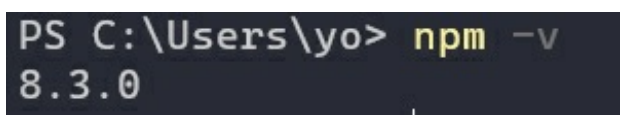

Figure 3-2. Checking composer version.

Create a new Nest project

There are a couple of ways of creating Nest projects. We will use the Nest CLI in this book. The **Nest CLI** is a command-line interface tool that helps you initialize, develop, and maintain your Nest applications. To install the Nest CLI, go to the Terminal, and execute the following command.

Execute in Terminal

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

To create a Nest project, open your Terminal, and in a location of your choice, execute the following command.

Execute in Terminal

```
nest new online-store
```

You will be asked “Which package manager would you like to use?” -> Select “npm” and press “Enter”.

The previous command creates a new Nest project inside the *online-store* folder.

Next, in your Terminal, move to the *online-store* folder, and run the application with the following commands.

Execute in Terminal

```
cd online-store
npm run start
```

The *npm run start* command executes a script command defined in the *package.json* file. This script executes *nest start* which launches the server (see Fig. 3-3).

```
PS C:\xampp\htdocs\online-store> npm run start

> online-store@0.0.1 start
> nest start

[Nest] 21428  - 12/12/2021, 5:45:49 p. m.     LOG [NestFactory] Starting Nest application...
[Nest] 21428  - 12/12/2021, 5:45:49 p. m.     LOG [InstanceLoader] AppModule dependencies initialized +29ms
[Nest] 21428  - 12/12/2021, 5:45:49 p. m.     LOG [RoutesResolver] AppController {/}: +6ms
[Nest] 21428  - 12/12/2021, 5:45:49 p. m.     LOG [RouterExplorer] Mapped {/, GET} route +3ms
[Nest] 21428  - 12/12/2021, 5:45:49 p. m.     LOG [NestApplication] Nest application successfully started +2ms
```

Figure 3-3. Running Nest project.

If the installation and setup were successful, open your browser at http://localhost:3000/ and you should see a hello-world message as shown in Fig. 3-4.

Figure 3-4. Nest default main page.

Note: you can stop the server with *Ctrl + C* (on Windows), or *Cmd + C* (on Mac) .

Nest project structure

Fig. 3-5 shows the Nest project structure. We will not explain all the folders and files since we want to start developing our web applications quickly. We will explain some of the more important ones. The others will be covered in upcoming chapters.

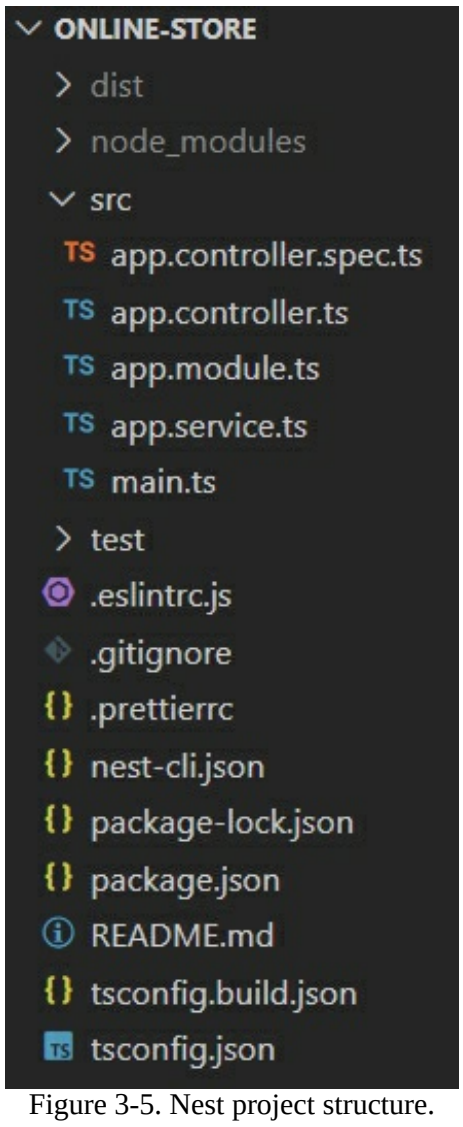

Figure 3-5. Nest project structure.

- **dist/*:** The */dist* folder stands for distributable. The */dist* folder contains the compiled version of the source code. It is the code used in production. We will see how to generate the application's production code later.
- **node_modules/*:** The */node_modules* folder contains all libraries downloaded from npm. The libraries/dependencies are listed in the *package.json* file. When we create a Nest project, internally, Nest uses npm to download and install all required libraries in the */node_modules* folder.
- **src/*:** Nest projects contain a *src/* folder populated with several core files.
- **src/main.ts:** This is the entry file of the application. It uses the core function NestFactory to create a Nest application instance. The *main.ts* includes an async function, which bootstraps our application.
- **src/app.controller.spec.ts:** Stores the unit tests for the app controller. We won't design unit tests in this book since it is out of the book's scope.
- **src/app.module.ts:** The root module of the application.
- **src/app.controller.ts:** A basic app controller with a single route.
- **src/app.service.ts:** A basic app service with a single method.
- **test/*:** The /test folder is used to define the application's end-to-end tests. Again, we won't use it since it is out of the book's scope.
- **package.json:** It holds metadata relevant to the project and is used for managing the project's dependencies, scripts, version and many more.
- **tsconfig.json:** It contains the TypeScript compiler configuration for our app.

As you can see, several files with different functionalities are used in the construction of Nest applications. For now, we will remove and modify some of those files to make our learning process more manageable. Later, we will re-create some of those files again.

Simplified Hello-World

Let's make some changes to simplify our initial project to make it easier to understand.

src/app.controller.spec.ts and src/app.service.ts

We won't use Nest services yet, so we delete the *app.service.ts* file. We will also remove the unit tests for the app controller by deleting the *app.controller.spec.ts* file.

src/app.controller.ts

In *src/app.controller.ts* , remove the *AppService* import, remove the *constructor* , and replace the *getHello return* with the following in **bold**.

Modify Bold Code

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return '<b>Hello World!</b>';
  }
}
```

We removed everything related to the app service (since we removed the *app.service.ts* file). Instead of using the *AppService* , we returned a simple “Hello World!” text. We wrapped this text around a “b” tag. This is an HTML tag which displays our text in bold. As you can see, we have some Nest annotations (i.e., *@Get* and *@Controller*), and some methods. We will explain them in upcoming chapters.

For now, know that the main app route (http://localhost:3000/) will invoke the *getHello* method which returns the “Hello World!” text.

src/app.module.ts

In *src/app.module.ts* , remove the *AppService* import and also the *providers* parameter.

Modify Bold Code

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

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

We will explain Nest Modules in upcoming chapters.

Running the app

In the Terminal, stop the server with *Ctrl + C* (on Windows), or *Cmd + C* (on Mac), and execute the following command.

Execute in Terminal

```
npm run start:dev
```

The *npm run start:dev* launches the server and watches your files. It rebuilds the app as you make changes to those files. This is the command that we will use from now. Otherwise, we will need to stop and start the server each time we have a code change.

Now, you can check the “Hello World!” message in bold (see Fig. 3-6).

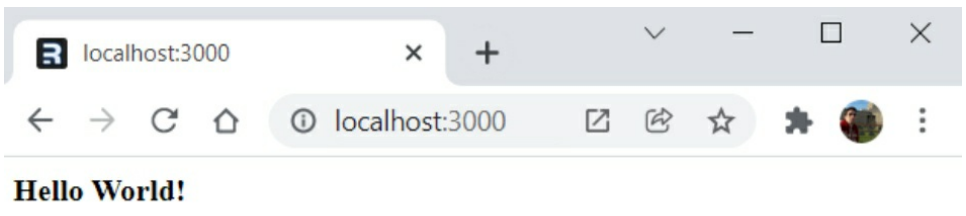

Figure 3-6. Nest main page.

In the next chapter, we will include the view layer to our application to avoid defining our HTML code inside our controllers, i.e.:

Analyze Code

```
getHello(): string {
  return '<b>Hello World!</b>';
}
```

Chapter 04 – Handlebars

In the previous chapter, we defined HTML code inside the controllers' methods. This strategy does not scale well since we are mixing logic code with view code. Nest official documentation suggests using a template engine to render our HTML views, and the recommended template engine is called Handlebars.

Introducing Handlebars

Handlebars (https://handlebarsjs.com/) is a simple templating language where you pass in input objects to output HTML. Handlebars templates look like regular text with embedded Handlebars expressions.

For example, the following code shows an excerpt of a simple view in Nest using Handlebars.

Analyze Code

```
<p>{{firstname}} {{lastname}}</p>
```

A handlebars expression is a *{{*, some contents, followed by a *}}*. When the template is executed, these expressions are replaced with values from an input object. For example, we can pass *firstname* and *lastname* objects from a Nest controller to a Handlebars view; the template is then executed and *firstname* and *lastname* will be replaced with their object values. We will see more examples in upcoming chapters.

Installing Handlebars

Let's first install Handlebars in our Online Store project. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm install --save hbs
```

This installs Handlebars in our project. It modifies our *package.json* file to include the *hbs* dependency and includes the *hbs* library inside the *node_modules* folder.

Configuring Handlebars

Once the installation is complete, we need to configure the Nest express instance to use Handlebars. In *src/main.ts*, make the following changes in **bold**.

Modify Bold Code

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { NestExpressApplication } from '@nestjs/platform-express';
import { join } from 'path';


async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const app = await NestFactory.create<NestExpressApplication>(
    AppModule,
  );

  app.useStaticAssets(join(__dirname, '..', 'public'));
  app.setBaseViewsDir(join(__dirname, '..', 'views'));
  app.setViewEngine('hbs');


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

Let's analyze the code by parts.

Analyze Code

```
import { NestExpressApplication } from '@nestjs/platform-express';
import { join } from 'path';
```

We import the *NestExpressApplication* interface which provides a set of Express methods. These methods will be available through the *app* object later. We also import the *join* function from the Node.js *path* library. The *path* library provides utilities for working with file and directory paths.

Analyze Code

```
  const app = await NestFactory.create<NestExpressApplication>(
    AppModule,
  );
```

```
app.useStaticAssets(join(__dirname, '..', 'public'));
app.setBaseViewsDir(join(__dirname, '..', 'views'));
app.setViewEngine('hbs');
```

We create a new *app* object which has access to a set of *NestExpressApplication* methods. We then invoke three methods. We tell Express that the *public* folder will be used for storing static assets(*useStaticAssets* method). *views* folder will contain templates(*setBaseViewsDir* method), and we specify that the *hbs* template engine will be used to render HTML output(*setViewEngine* method).

Using Handlebars

Finally, let's refactor our code to avoid using HTML inside controller's methods and use Handlebars instead.

Index view

In the project root directory, create a folder called *views* . This is the folder where we store our views. In *views/* create a new file called *index.hbs* and fill it with the following code.

Add Entire Code

```
<b>Hello World!</b>
```

App.controller.ts

Delete all the existing code in *src/app.controller.ts* and fill it with the following.

Replace Entire Code

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


@Controller()
export class AppController {
  @Get("/")
  @Render('index')
  index() {}
}
```

Let's analyze some important changes in *src/app.controller.ts* . First, we import the *Render* decorator used to render Handlebars views. Then, we add the "/" route to the *@Get* decorator. This is not necessary since the default route is already "/"; however, it improves the code readability. We use the *@Render* decorator for the *index* method (check that we change the *getHello* method name to *index*). In this case, we specify that the *index* method will render the *index* view.

In summary, once a user visits the "/" route, Nest will execute the *AppController index* method, which will render the *index* view.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

If you go to http://localhost:3000/ you will see the "Hello World!" message in bold (see Fig. 4-1).

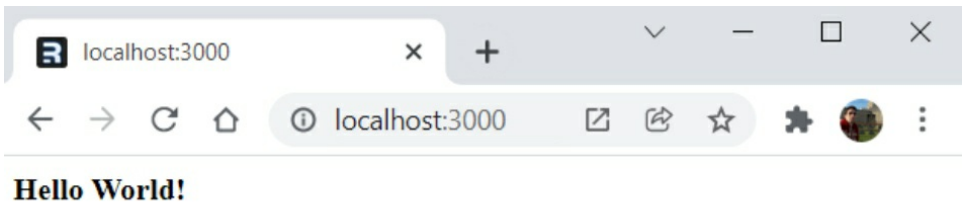

Figure 4-1. Online Store index page with Handlebars.

Chapter 05 – Introduction to MVC applications

There are different ways of designing and implementing web applications. For example, you can create an entire web application by placing all your code in a single file. However, finding an error in such a file (which contains thousands of lines of code) is not an easy task. Other approaches split the code over different files and folders. You will even find approaches that split your application over different small applications distributed over several servers (the distribution of these servers is not an easy task).

As you can see, structuring your code is not an easy task. That's the reason why developers and computer scientists have developed what is called software architectural patterns. **Software architectural patterns** are structural layouts used to solve commonly faced software design problems. With these patterns startups or novice developers do not have to "reinvent the wheel" each time they start a new project. There are many architectural patterns, such as model-view-controller, layers, service-oriented, and micro-services. Each one has its advantages and disadvantages. Many are widely adopted. Still, one of the most used is the model-view-controller pattern.

Model-view-controller (MVC) is a software architectural pattern commonly used to develop web applications containing user interfaces. This pattern divides the application into three interconnected elements.

- **Model** contains the business logic of the application. For example, the online store application product data and its functions.
- **View** contains the user interface of the application. For example, a view to register products or users.
- **Controller** acts as an interface between model and view elements. For example, a product controller collects information from a "create product" view and passes it to the product model to be stored in the database.

Nest provides support for the MVC pattern thanks to the integration of the Handlebars templating engine. Other similar frameworks provide support to this popular pattern too. We will see this pattern in action (with actual code) later.

The MVC pattern provides some advantages: better code separation, multiple team members can work and collaborate simultaneously, finding an error is easier, and maintainability is improved. Fig. 5-1 shows the Online Store software architecture we will implement in this book. It can be a little overwhelming now, but you will understand the elements of this architecture when you finish this book. We will review the architecture in the final chapters.

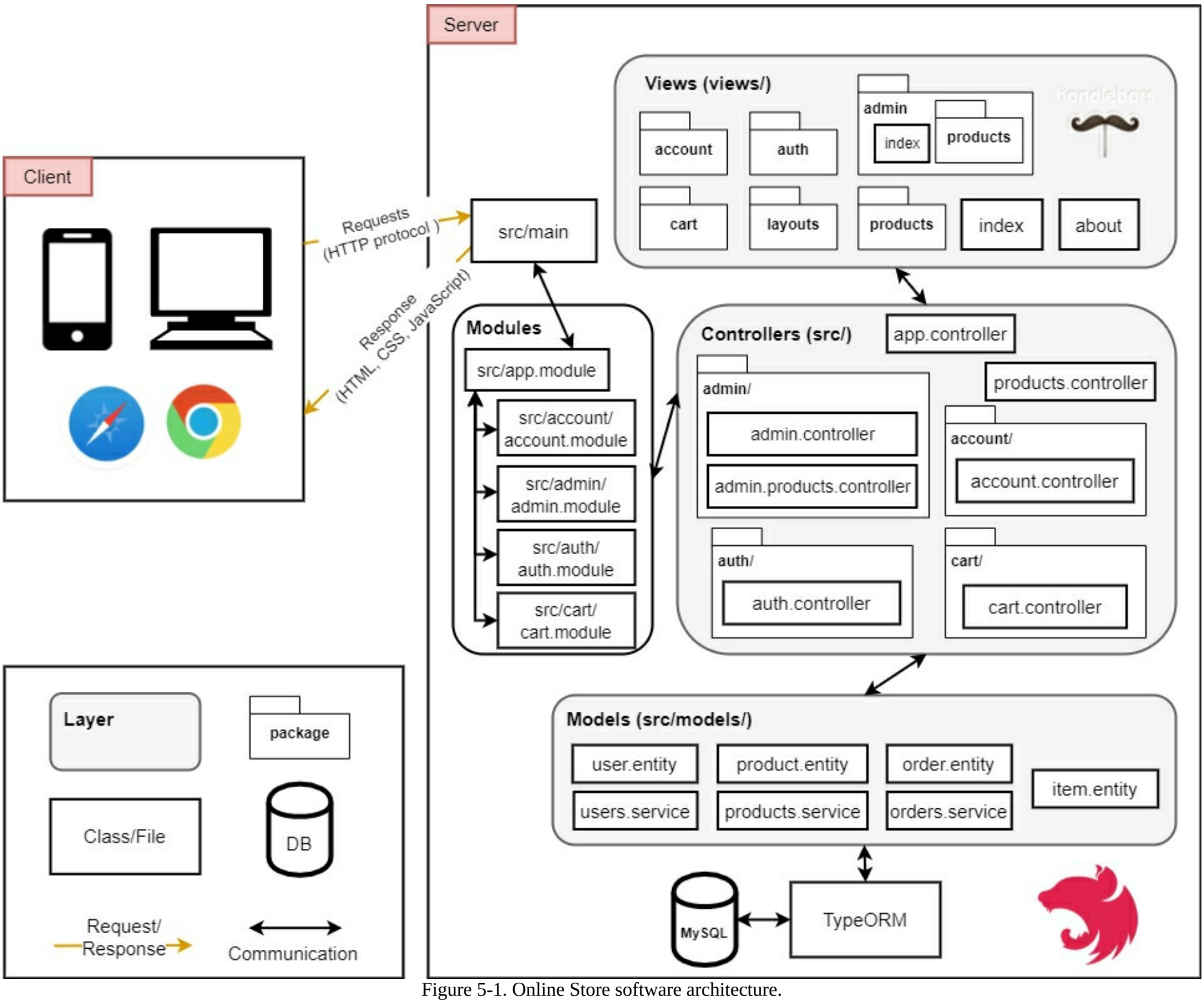

Figure 5-1. Online Store software architecture.

Let’s have a quick analysis of this architecture:

- On the left, we have clients (users of our application e.g., browsers in mobile/desktop devices). Clients connect to the application through the Hypertext Transfer Protocol (HTTP). HTTP gives users a way to interact with our web application.
- On the right, we have the server where we place our application code.
- Clients’ interactions are connected to the main application file (main.ts). This file loads the root module (AppModule), which internally will load other modules (described in Chapter 10), and those modules pass the clients’ interactions to the controllers.
- Controllers (described in Chapter 7) execute the clients’ interactions based on the specific requested routes, i.e. (“/products”) or (“/cart”) routes. Controllers communicate with models (described in Chapter 13), and pass information to the views (described in Chapter 4), which are finally delivered to the clients as HTML, CSS, and JavaScript code.

We highlight the Model, View and Controller layers in grey. We have four models (entities) corresponding to the classes defined in our class diagram (in Fig. 2-1). As mentioned, there are different approaches to implement web applications with Nest. There are even different versions of MVC used in a Nest application. In the following chapters, we will see advantages of adopting the MVC architecture defined in Fig. 5-1.

Chapter 06 – Layout View

Introducing Bootstrap

Bootstrap is the most popular CSS framework for developing responsive and mobile-first websites (see Fig. 6-1). For Nest projects, a developer can design the user interface from scratch. But because this book is not about user interfaces, we will take advantage of CSS frameworks (such as Bootstrap) and use a starter template to still create something that looks professional. You can find out more about Bootstrap at: https://getbootstrap.com/.

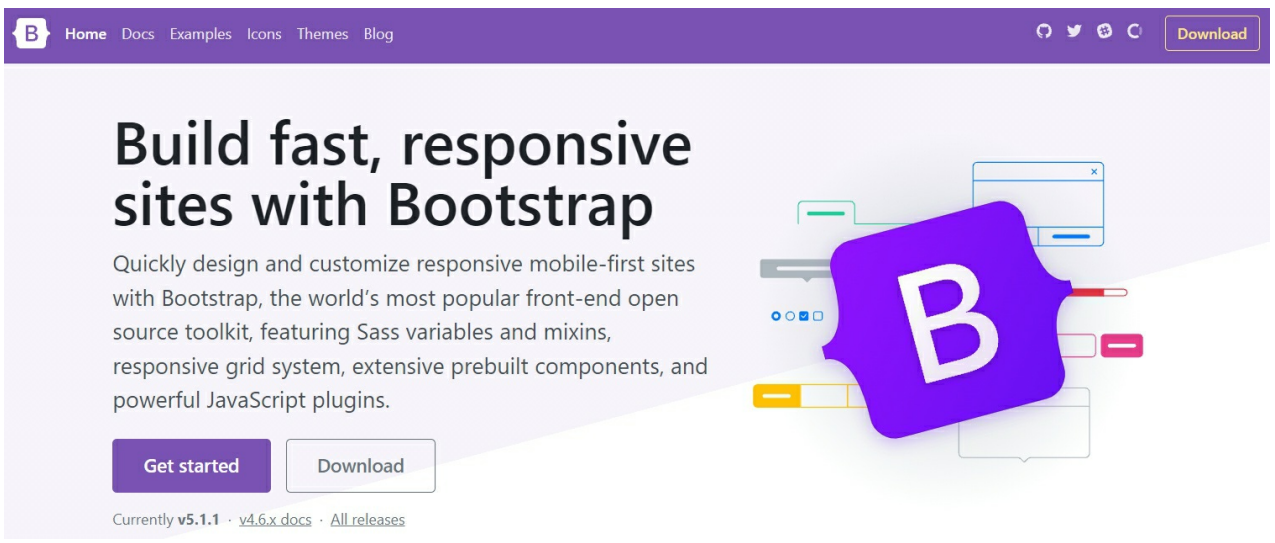

Figure 6-1. Bootstrap website.

Introducing Handlebars Partials

Most web applications maintain the same general layout across various pages (common header, navigation bar, footer). However, it would be incredibly cumbersome to maintain our application if we had to repeat the entire header, navbar, footer HTML in every view. Fortunately, we can define this layout as a single hbs file and use it throughout our application.

Configuring Handlebars Partials

To create this common layout, we need to configure Handlebars Partials. **Handlebars Partials** are normal Handlebars views that may be called by other views. We will configure Handlebars Partials and include a library to watch for changes in those Partials.

Installing hbs-utils

Hbs-utils provides a set of functions that are useful when using hbs. One such function allows us to watch changes in Partials. Let's install hbs-utils in our Online Store project. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm install hbs-utils
```

This installs *hbs-utils* in our project. It modifies our *package.json* file to include the *hbs-utils* dependency and includes the *hbs-utils* library inside the *node_modules* folder.

Modifying src/main.ts

In *src/main.ts* , make the following changes in **bold**.

Modify Bold Code

```
…
import { AppModule } from './app.module';
import * as hbs from 'hbs';
import * as hbsUtils from 'hbs-utils';


async function bootstrap() {
  const app = await NestFactory.create<NestExpressApplication>(
    AppModule,
  );


  app.useStaticAssets(join(__dirname, '..', 'public'));
  app.setBaseViewsDir(join(__dirname, '..', 'views'));
  hbs.registerPartials(join(__dirname, '..', 'views/layouts'));
  hbsUtils(hbs).registerWatchedPartials(join(__dirname, '..', 'views/layouts'));
  app.setViewEngine('hbs');
  …
```

We import the *hbs* library as a *hbs* object and the *hbs-utils* library as a *hbsUtils* object. We instruct *hbs* that the *views/layouts* folder will be used for storing Handlebars Partials using the *registerPartials* method. We then tell *hbsUtils* to watch the *views/layouts* folder for changes using the *registerWatchedPartials* method.

Creating app.hbs

To get started with Bootstrap and the hbs layout, we first create a folder called *layouts* under the *views* folder. We then use the Bootstrap starter template to create our layout (see Fig. 6-2). The Bootstrap starter template can be found here: https://getbootstrap.com/docs/5.1/getting-started/introduction/.

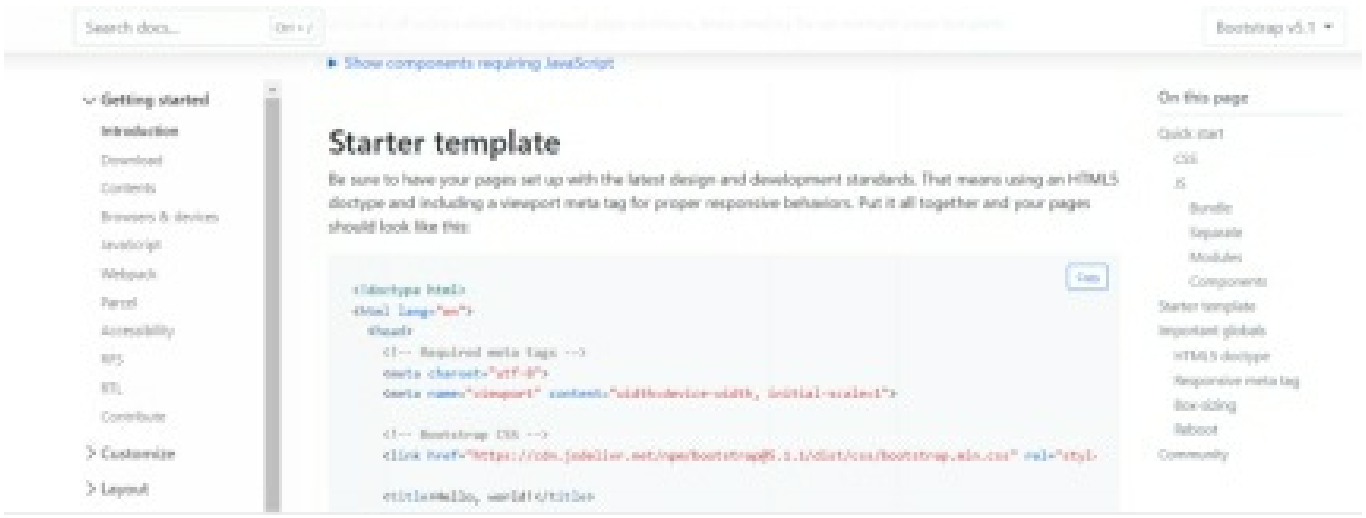

Figure 6-2. Bootstrap starter template.

In *views/layouts* , create a new file called *app.hbs* and fill it with the following code.

Add Entire Code

```
<!doctype html>
<html lang="en">
<head>
 <meta charset="utf-8" />
 <meta name="viewport" content="width=device-width, initial-scale=1" />
 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css" rel="stylesheet" crossorigin="anonymous" />
 <title>{{#if title}}{{ title }}{{else}}Online Store{{/if}}</title>
</head>
<body>
 <!-- header -->
 <nav class="navbar navbar-expand-lg navbar-dark bg-secondary py-4">
  <div class="container">
   <a class="navbar-brand" href="#">Online Store</a>
   <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNavAltMarkup"
    aria-controls="navbarNavAltMarkup" aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
   </button>
   <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
    <div class="navbar-nav ms-auto">
     <a class="nav-link active" href="#">Home</a>
     <a class="nav-link active" href="#">About</a>
    </div>
   </div>
  </div>
 </nav>


 <header class="masthead bg-primary text-white text-center py-4">
  <div class="container d-flex align-items-center flex-column">
   <h2>{{#if subtitle}}{{ subtitle }}{{else}}A Nest Online Store{{/if}}</h2>
  </div>
 </header>
 <!-- header -->


 <div class="container my-4">
  {{> content}}
 </div>


 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js" crossorigin="anonymous">
 </script>
</body>
</html>
```

The above code is based on the Bootstrap starter template code and the Bootstrap Navbar (https://getbootstrap.com/docs/5.1/components/navbar/). We modified the base code including links in the header (*Home* and *About*). The stater template includes a Bootstrap CSS file (*bootstrap.min.css*), and a Bootstrap JS file (*bootstrap.bundle.min.js*). We included three objects *title* , *subtitle* , and *content* .

The *title* and *subtitle* objects are wrapped by an if condition. The *if* hbs helper can be used to conditionally render a block. For example, if *title* object is defined, it will render the *title* value. Otherwise, it renders the content of the *else* block (in this case "Online Store" text).

Finally, *{{> content}}* defines a partial call. In that specific location, we will inject other views code.

Modifying index.hbs

Delete all the existing code in *views/index.hbs* and fill it with the following code.

Replace Entire Code

```
{{#> app}}
  {{#*inline "content"}}
    <div class="text-center">
      Welcome to the application
    </div>
  {{/inline}}
{{/app}}
```

The *index* view invokes and renders the *app* layout with the use of the *app* block *{{#> app}}…{{/app}}* . We define an inline partial called *content* , which is rendered inside the *app* layout in the *{{> content}}* line. In summary, the *index* view injects an HTML div with a welcome message inside the *{{> content}}* line of the *app* layout.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Open the browser to http://localhost:3000/ and you will see the application with the new layout (see Fig. 6-3). If you reduce the browser window width, you will see a responsive navbar (thanks to the bootstrap starter template, navbar, and the inclusion of the bootstrap files, see Fig. 6-4). **Note:** sometimes, you need to stop and start the server to see the changes due to the inclusion of new libraries in the project.

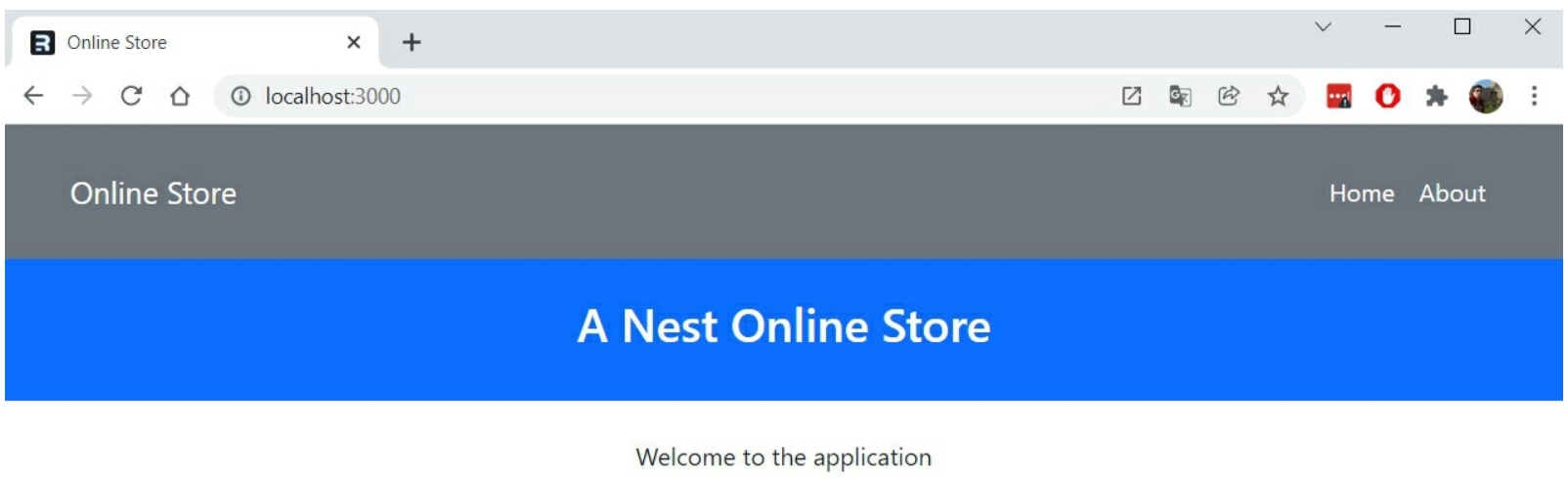

Figure 6-3. Application home page with layout.

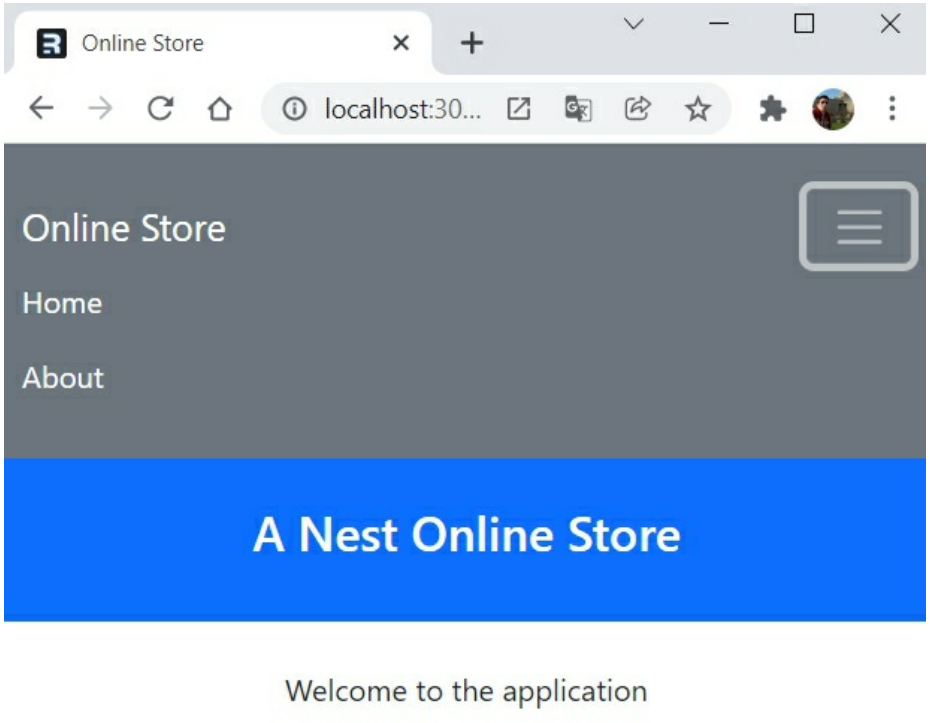

Figure 6-4. Application home page with reduced browser window width.

Adding custom CSS styles and a Footer

Let’s make our app interface more professional. We will include a custom CSS file and a footer in our layout.

Custom style (app.css)

Create a folder called *public* under the root project folder. Then, create a subfolder called *css* under the *public/* folder. Then, in *public/css* create a new file called *app.css* and fill it with the following:

Add Entire Code

```
.bg-secondary {
  background-color: #2c3e50 !important;
}

.copyright {
  background-color: #1a252f;
}

.bg-primary {
  background-color: #1abc9c !important;
}

nav {
  font-weight: 700;
}

.img-card {
  height: 18vw;
  object-fit: cover;
}
```

We have some custom CSS styles in the previous file. We override some Bootstrap elements with our own values and colors.

This CSS file is available for public access. If you have the server running you can access it from: http://localhost:3000/css/app.css. This is because we told Express that the *public* folder will be used for storing static assets. So, all files inside the *public* folder can be accessed from our server link.

Modifying app.hbs

Finally, in *views/layouts/app.hbs* , make the following changes in **bold** to include the previous CSS file and create the footer section.

Modify Bold Code

```
<!doctype html>
<html lang="en">
<head>
   ...
   <link href="/css/app.css" rel="stylesheet" />
   <title>{{#if title}}{{ title }}{{else}}Online Store{{/if}}</title>
</head>
<body>
   …
   <!-- footer -->
   <div class="copyright py-4 text-center text-white">
      <div class="container">
         <small>
            Copyright - <a class="text-reset fw-bold text-decoration-none" target="_blank"
               href="https://twitter.com/danielgarax">
               Daniel Correa
            </a> - <a class="text-reset fw-bold text-decoration-none" target="_blank"
               href="https://twitter.com/greglim81">
               Greg Lim
            </a>
         </small>
      </div>
   </div>
   <!-- footer -->
 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js" crossorigin="anonymous">
 </script>
</body>
</html>
```

We include our custom CSS file (*css/app.css*), and created a footer section with this book’s author names and links to their Twitter accounts.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

You will see our index page, with the refined layout (see Fig. 6-5). It looks more professional (at least for us). This design was inspired by a free Bootstrap template called Freelancer. You can find more information about this template here: https://startbootstrap.com/theme/freelancer.

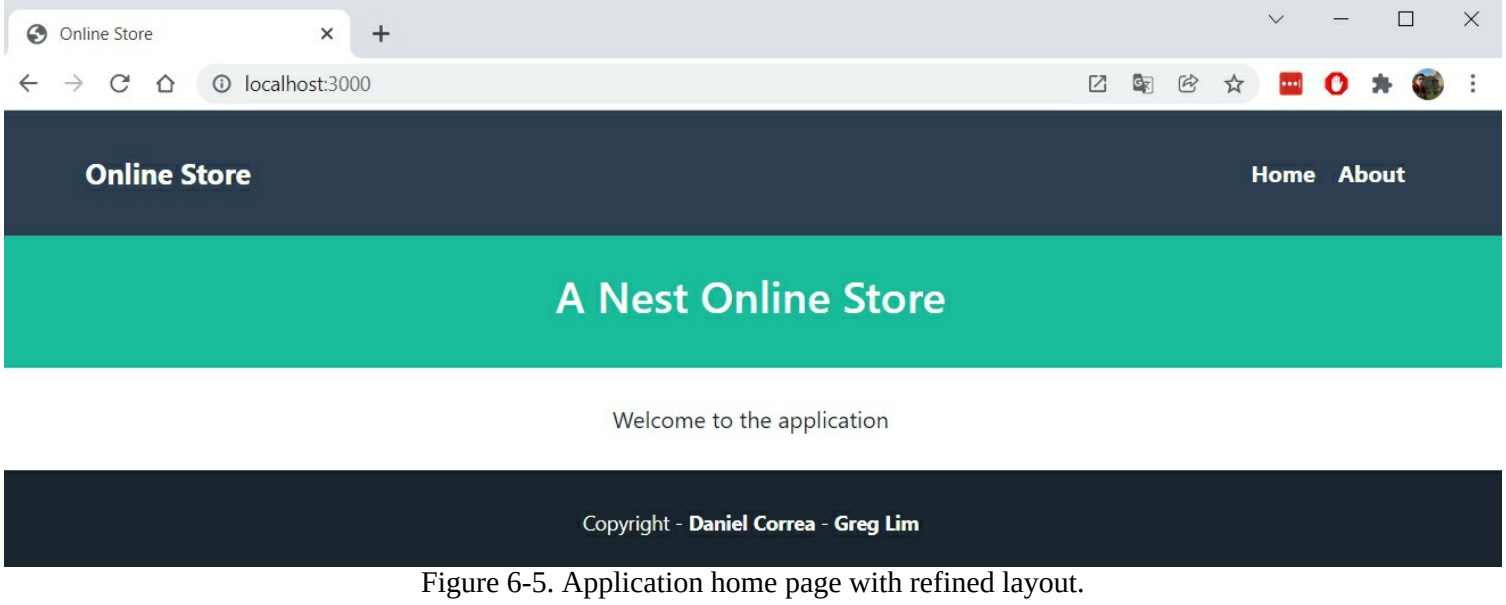

Figure 6-5. Application home page with refined layout.

We will refine the index page and create an about page in the next chapter.

Chapter 07 – Index and About Pages

Index view

The home page currently just displays a welcome message. Let's modify the *index* view to improve the home page design. In *views/index.hbs* , make the following changes in **bold**.

Modify Bold Code

```
{{#> app}}
  {{#*inline "content"}}
    <div class="text-center">
      Welcome to the application
    </div>
    <div class="row">
      <div class="col-md-6 col-lg-4 mb-2">
        <img src="/img/game.png" class="img-fluid rounded">
      </div>
      <div class="col-md-6 col-lg-4 mb-2">
        <img src="/img/safe.png" class="img-fluid rounded">
      </div>
      <div class="col-md-6 col-lg-4 mb-2">
        <img src="/img/submarine.png" class="img-fluid rounded">
      </div>
    </div>
  {{/inline}}
{{/app}}
```

We define some divisions to display some images. We need to download these images and store them in our *public* folder. First, in *public* folder, create a subfolder called *img* . Then, download the following three images from this link https://github.com/PracticalBooks/Practical-Nest/tree/main/Chapter07/online-store/public/img and store them inside the *public/img* folder (see Fig. 7-1).

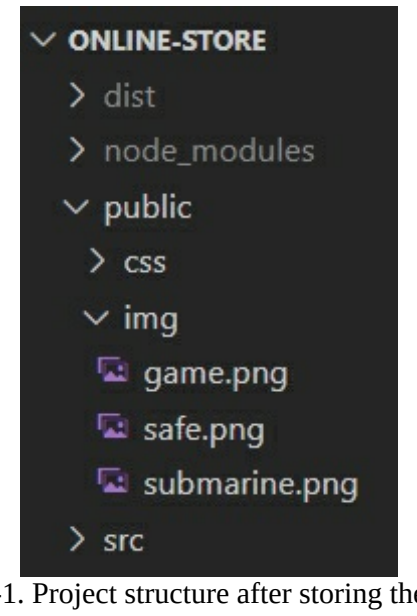

Figure 7-1. Project structure after storing the images.

About view

Let's create the *About* view. In *views* , create a new file called *about.hbs* , and fill it with the following code.

Add Entire Code

```
{{#> app}}
  {{#*inline "content"}}
    <div class="container">
      <div class="row">
        <div class="col-lg-4 ms-auto">
          <p class="lead">{{ viewData.description }}</p>
        </div>
        <div class="col-lg-4 me-auto">
          <p class="lead">{{ viewData.author }}</p>
        </div>
      </div>
    </div>
  {{/inline}}
{{/app}}
```

We have a simple view that displays a description of the application and some information about the author. We will pass the *viewData.description* and *viewData.author* data from a controller to this view later. Remember that hbs allows curly braces to display data passed to the view.

Introducing Nest Controllers

Controllers are responsible for handling incoming requests and returning responses to the client. Controllers can group related request handling logic into a single class. For example, a *UserController* class might handle all incoming requests related to users, including showing, creating, updating, and deleting users.

Nest Controllers are classes with decorators. Decorators associate these classes with required metadata and enable Nest to create a routing map (tie requests to the corresponding controllers). More information about Nest Controllers can be found here: https://docs.nestjs.com/controllers.

Linking the 'About' Page

Let's modify our *AppController* to define a route for our 'about' page. In *src/app.controller.ts*, make the following changes in **bold**.

Modify Bold Code

```
…
@Controller()
export class AppController {
  @Get("/")
  @Render('index')
  index() {
    return {
      title: "Home Page - Online Store"
    };
  }

  @Get("/about")
  @Render('about')
  about() {
    let viewData = [];
    viewData["description"] = "This is an about page ...";
    viewData["author"] = "Developed by: Your Name";
    let data1 = 'About us - Online Store';
   return {
      title: data1,
      subtitle: "About us",
      viewData: viewData
    };

  }
}
```

Let's analyze the code by parts.

Analyze Code

```
  @Get("/")
  @Render('index')
  index() {
    return {
      title: "Home Page - Online Store"
    };
  }
```

We included a return section in the *index* method. *return* passes a JavaScript object (which defines a *title* data that contains a text) to the *index* view and *app* layout. This enables the application to show the "Home Page - Online Store" text over the browser tab.

Analyze Code

```
  @Get("/about")
  @Render('about')
  about() {
    let viewData = [];
    viewData["description"] = "This is an about page ...";
    viewData["author"] = "Developed by: Your Name";
    let data1 = 'About us - Online Store';
   return {
      title: data1,
      subtitle: "About us",
      viewData: viewData
    };


  }
```

We have the *about* method connected to the ("/about") route. This connection is established by the *@Get* decorator. *about* method defines an associative array called *viewData*. It also defines a variable called *data1* which contains the title of the page. Then, we define the values for the *description* and *author* keys of the

viewData array. We pass the *title* , *subtitle* , and *viewData* to the *about* view and *app* layout. The *about* view will show the *viewData.description* and *viewData.author* values, and the *app* layout will show the *title* and *subtitle* values.

Recap, when a user goes to the "/" route, the *index* view will be displayed (rendered in the *AppController index* method). Likewise, when a user goes to the "/about" route, the *about* view will be displayed (rendered in the *AppController about* method).

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Go to the "/" route, and you will see the new home page (see Fig. 7-2). Then, go to the "/about" route, and you will see the about page (see Fig. 7-3).

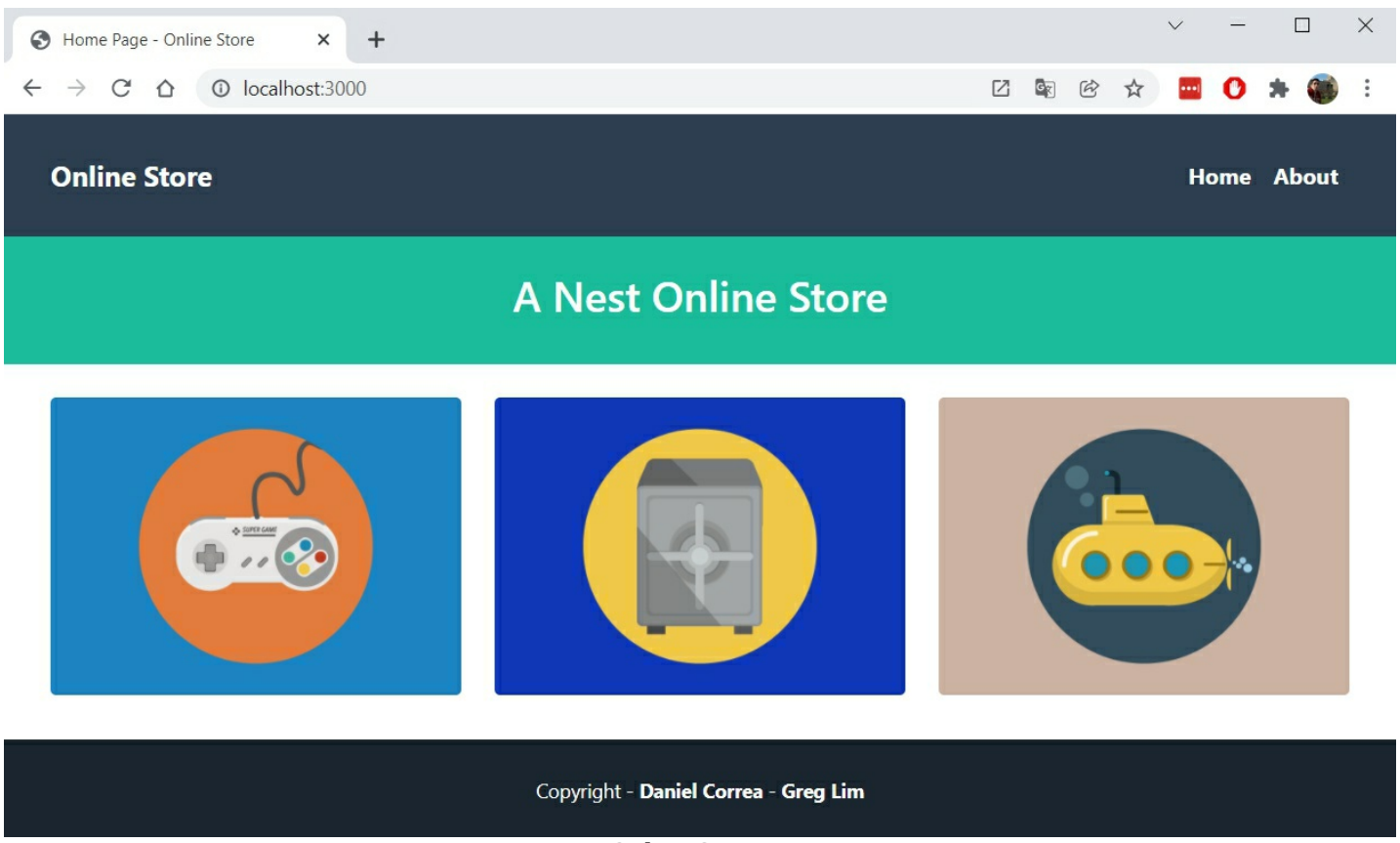

Figure 7-2. Online Store – Home page.

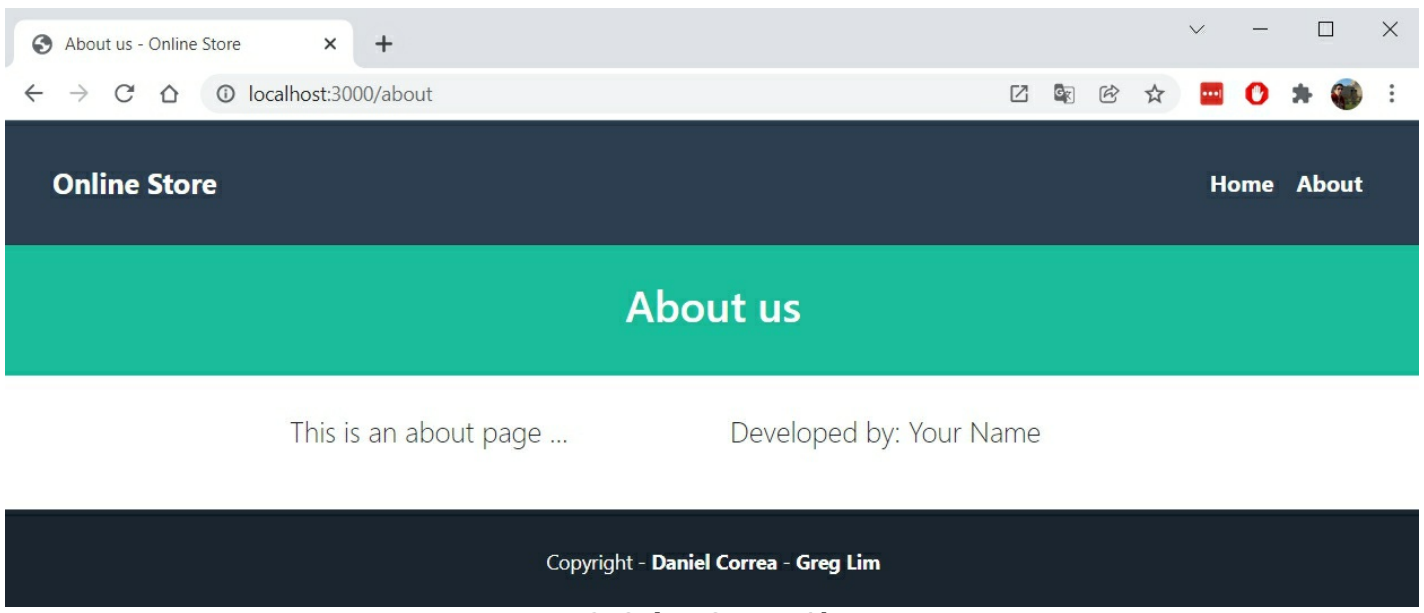

Figure 7-3. Online Store – About page.

We created a couple of Nest views, a Nest Controller, and defined a couple of routes. However, we have not linked the navbar menu to the new routes. We have also purposely introduced some mistakes when defining the previous elements. Why? This is because we want to illustrate the concept of clean code. In the next chapter, we will refactor these elements and apply some good strategies about clean code in controllers and views.

Chapter 08 – Refactoring Index and About Pages

The code in the previous chapter can be further cleaned and improved. This is because, we have not defined rules for coding any of the elements. We will see how to refactor the code to make it cleaner and more maintainable. We will also provide general tips for handling controllers and views. Keep in mind that most of the principles presented in this chapter apply not only to a Nest project but can be applied in other MVC frameworks (such as Django, Spring, Laravel, Express, and more).

Refactoring controllers

Previous controller

The previous chapter showed a controller with two methods. Let's analyze the *about* method.

Analyze Code

```
@Get("/about")
@Render('about')
about() {
    let viewData = [];
    viewData["description"] = "This is an about page ...";
    viewData["author"] = "Developed by: Your Name";
    let data1 = 'About us - Online Store';
  return {
       title: data1,
       subtitle: "About us",
       viewData: viewData
    };


}
```

Here we have three problems.

- Variable naming is a mess. Using names such as *data1* is horrible; it does not say anything. Instead of that, we can use *title* .
- We have three elements defined in the return JavaScript object (*title* , *subtitle* , and *viewData*). Imagine if we have 20 variables to pass to the view. We do not have consistency. We send some variables to the view one by one, and we also send some variables grouped in an array. We will use the *viewData* variable to pass all the information to the view within a single array variable. This will be our preferred strategy as we will see it next.
- Finally, we have a blank line before the ending of the curly brackets, and we define some texts with single quotes (check *data1*) and others with double quotes (check *subtitle*). We need to define a consistent coding style guide. This one will be solved in the next chapter.

Quick discussion: Let's see the importance of variable naming with two quotes from the (*2019 - Thomas, D., & Hunt, A. - The Pragmatic Programmer: your journey to mastery*) book. *"The beginning of wisdom is the ability to call things by their right names. - Confucius."* - *"Why is naming important? Because good names make code easier to read, and you have to read it to change it."*

Next, let's refactor the controller.

New controller

Let's refactor our controller. In *src/app.controller.ts* , make the following changes in **bold** (replace the content of the *index* and *about* methods).

Modify Bold Code

```
…
@Controller()
export class AppController {
   @Get("/")
   @Render('index')
   index() {
      let viewData = [];
      viewData['title'] = 'Home Page - Online Store';
    return {
         viewData: viewData
      };
   }


   @Get("/about")
   @Render('about')
```

```
  about() {
    const viewData = [];
    viewData['title'] = 'About us - Online Store';
    viewData['subtitle'] = "About us";
    viewData['description'] = "This is an about page ...";
    viewData['author'] = 'Developed by: Your Name';
    return {
      viewData: viewData,
    };

  }
}
```

Now that we use the single variable strategy both methods are consistent. The return sends only *viewData* . With this approach, it does not matter if we pass one variable to the view or dozens. In both cases, we pass the associative array.

Check that we have some differences in the coding style (spacing, variable definition, and use of single quotes vs double quotes). Those mistakes were introduced on purpose, but we will fix in the next chapter.

Refactoring views

The previous chapter showed two views that display data a little differently. The *views/about.hbs* view will not be changed since it displays the data using the *viewData* strategy. However, we will need to modify the *views/layouts/app.hbs* view to match the single variable strategy previously defined.

Let's refactor our *app* view. In *views/layouts/app.hbs* , make the following changes in **bold**.

Modify Bold Code

```
...
<head>
  ...
  <link href="/css/app.css" rel="stylesheet" />
  <title>{{#if viewData.title}}{{ viewData.title }}{{else}}Online Store{{/if}}</title>
</head>
…
  <header class="masthead bg-primary text-white text-center py-4">
    <div class="container d-flex align-items-center flex-column">
      <h2>{{#if viewData.subtitle}}{{ viewData.subtitle }}{{else}}A Nest Online Store{{/if}}</h2>
    </div>
  </header>
…
```

As you can see, we now access the data through the single *viewData* associative array. We will use this strategy across the entire application, making our views more consistent.

TIP: As a software developer, a good strategy is to create a document with architectural rules and share that document with your team (if you have one). You can make that document in the project repository wiki (if you have one). Encourage all the members to read that document. A first rule that you can include in that document could be: "controllers should only pass an associative array called *viewData* to the views". These simple rules will save you a lot of time and a lot of headaches; believe us, Daniel always creates a document like that for all his projects, and he encourages his students to do it in their projects.

Quick discussion: Some of the previous data (such as the *description* and *author*) can also be placed directly over the *about* view. We mean, you do not need to define some of those texts as variables in the controller and send them to the view. Instead, you can place the text directly in the views. We did it that way to illustrate and explain some Nest elements. There is even a better option that is out of this book's scope. That option is called Internationalization or i18n. In i18n, you move away those texts from controllers and views and place them in the *src/i18n* folder. I18n allows you even to retrieve strings in various languages to create a multi-language application. It is not difficult to implement; you can use this library to get more info about it https://www.npmjs.com/package/nestjs-i18n, search in Google, or let us know if you need a good example (use the discussion zone of the book repository).

Updating links in Header

Now that we have the proper controller and views, let's include the links in the header. In *views/layouts/app.hbs* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
...
<body>
<!-- header -->
   <nav class="navbar navbar-expand-lg navbar-dark bg-secondary py-4">
      <div class="container">
         <a class="navbar-brand" href="/">Online Store</a>
     <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNavAltMarkup"
            aria-controls="navbarNavAltMarkup" aria-expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
         </button>
         <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
            <div class="navbar-nav ms-auto">
               <a class="nav-link active" href="/">Home</a>
               <a class="nav-link active" href="/about">About</a>
            </div>
         </div>
      </div>
   </nav>
...
```

We declared the corresponding routes for each link.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now, you can navigate between the *Home* page and the *About* page by using the links in the navigation bar (see Fig. 8-1).

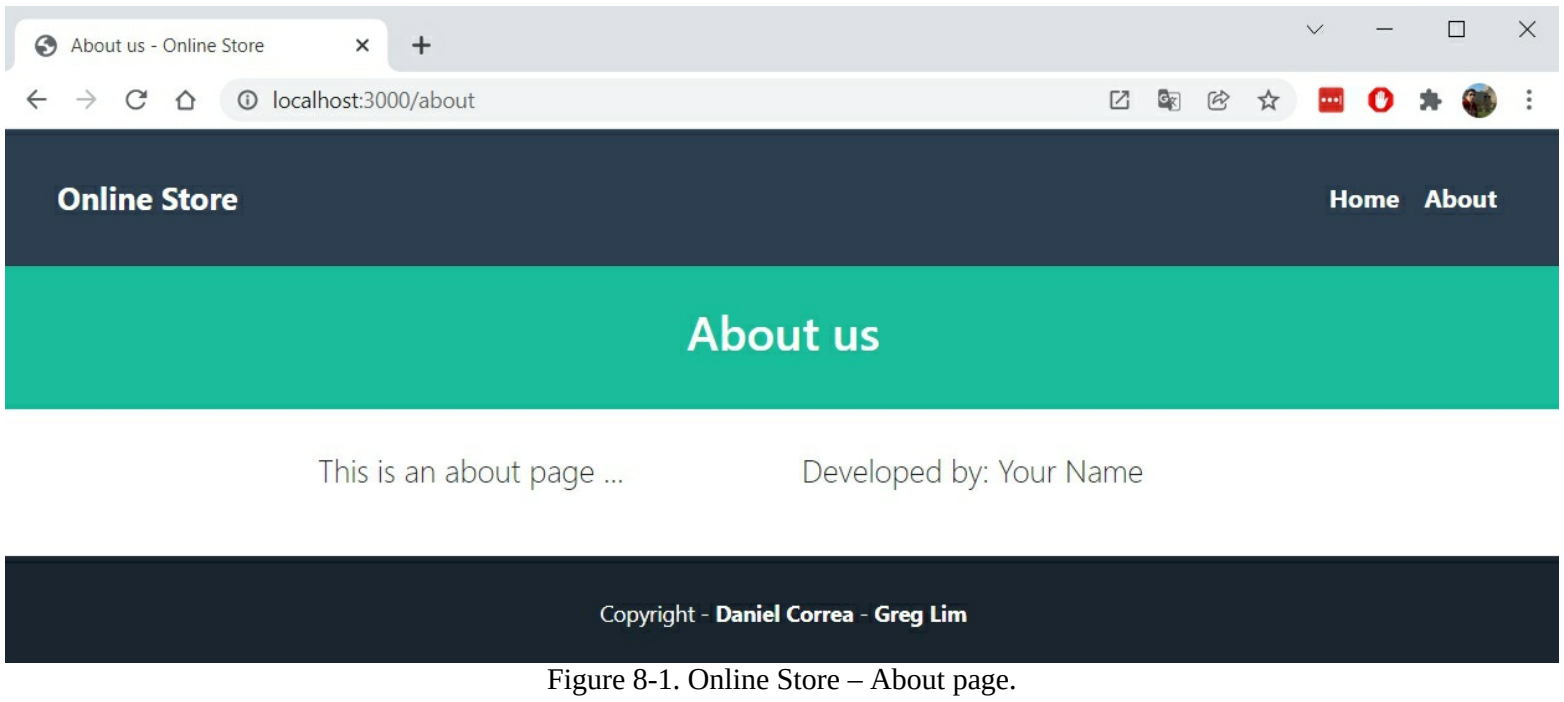

Figure 8-1. Online Store – About page.

Note: You can find the application code at the GitHub repository in https://github.com/PracticalBooks/Practical-Nest.

Chapter 09 – Use of a Coding Standard

A **coding standard** is a set of rules and agreements used when writing source code in a particular programming language. Using a coding standard provides some advantages, such as:

- It gives a uniform appearance to the codes written by different programmers.
- It improves the readability and maintainability of the code and reduces complexity.
- It helps in code reuse and helps to detect errors easily.
- It promotes good programming practices and increases the efficiency of the programmers.

Many frameworks come with tools which automate code checking and fixing against a set of predefined rules. For example, Nest comes with ESLint and Prettier out of the box. In this book, we will use ESLint, so let's learn more about it.

Introducing ESLint

ESLint is a JavaScript linter that enables you to enforce a set of style, formatting, and coding standards for your codebase. It looks at your code and tells you when you're not following the standard that you set in place (https://eslint.org/).

By default, ESLint does not support TypeScript (the language we use to create our controllers and other files). However, Nest includes *typescript-eslint/eslint-plugin* and *typescript-eslint/parser* dependencies which enables ESLint to support TypeScript.

Nest defines the ESLint configuration and rules in the *.eslintrc.js* file. Therefore, Nest defines a script to execute ESLint. That script (*lint*) can be found in the scripts section of the *package.json* file. Let's analyze it.

Analyze Code

```
"lint": "eslint \  "{src,apps,libs,test}/**/*.ts \ " --fix",
```

When we run *npm run lint* , the above script is executed. This script executes ESLint and checks and fixes all the TypeScript files inside the *src* , *apps* , *libs* , and *test* folders.

Running ESLint

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run lint
```

This executes ESLint based on the custom configuration defined in *.eslintrc.js* (see Fig. 9-1). You will see changes in some of your TypeScript files (such as the *src/app.controller.ts*). If you open that file, you will see that the strange spaces have disappeared. All texts are defined with single quotes, and variables are defined with "cons" instead of "let" when it applies.

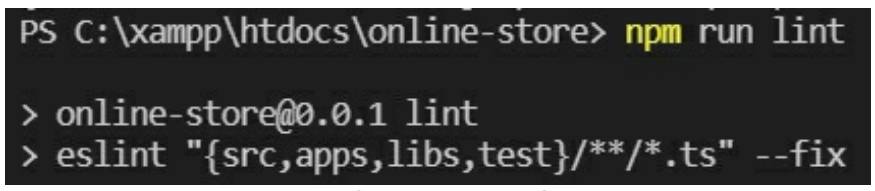

Figure 9-1. Running ESLint.

Final remark

The current ESLint configuration only checks and fixes TypeScript files. So, if you want to format Handlebars files, you can use a third-party library (such as *eslint-plugin-hbs*) or use a Visual Studio Code formatter such as "HTML Language Features".

TIP: There is a good story about "The Broken Window Theory" described in the (*2019 - Thomas, D., & Hunt, A. - The Pragmatic Programmer: your journey to mastery*) book. You can search it in Google. From that story, we want to highlight the next tip. *Don't leave "broken windows" (e.g., bad designs, wrong decisions, or poor code) unrepaired. Fix each one as soon as it is discovered.* The previous three chapters showed many broken windows which fortunately were fixed.

TIP: Always use a coding standard tool, formatter, static code analysis tool, or even a combination of them in your projects. It will save you a lot of time and improve the code quality. In addition, you will find linters available for most programming languages. Besides, include a

rule in your architectural rules document mentioning that all code changes should be previously verified using these tools. You can even automate this process (with a pipeline or CI/CD strategy). However, this is out of the scope of this book.

Chapter 10 – List Products with Dummy Data

In this chapter, we will implement functionality to list products and to be able to click those products and display their data in a separate section.

Listing products

Let’s start by creating a new controller to implement the list products functionality.

Products controller

In *src/* , create a new file called *products.controller.ts* , and fill it with the following code.

Add Entire Code

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


@Controller('/products')
export class ProductsController {
  static products = [
    {
      id: '1',
      name: 'TV',
      description: 'Best tv',
      image: 'game.png',
      price: '1000',
    },
    {
      id: '2',
      name: 'iPhone',
      description: 'Best iPhone',
      image: 'safe.png',
      price: '999',
    },
    {
      id: '3',
      name: 'Chromecast',
      description: 'Best Chromecast',
      image: 'submarine.png',
      price: '30',
    },
    {
      id: '4',
      name: 'Glasses',
      description: 'Best Glasses',
      image: 'game.png',
      price: '100',
    },
];

@Get('/')
@Render('products/index')
index() {
    const viewData = [];
    viewData['title'] = 'Products - Online Store';
    viewData['subtitle'] = 'List of products';
    viewData['products'] = ProductsController.products;
    return {
      viewData: viewData,
    };
  }
}
```

Let’s analyze the code by parts.

Analyze Code

```
@Controller('/products')
```

We use the *@Controller* decorator and pass in a path prefix (“/products”). This allows us to group a set of related routes and minimize repetitive code. All methods defined in this controller will automatically include the “/products” path prefix.

Analyze Code

```
  static products = [
    {
      id: '1',
      name: 'TV',
      description: 'Best tv',
```

```
      image: 'game.png',
      price: '1000',
    },
    …
```

products is an array that contains a set of products with its data. In the array index 0, we have the product with id=1 (the "TV"). We have four dummy products. Currently, they are stored as a static attribute of the *ProductsController* class. We will later retrieve product data from a MySQL database in upcoming chapters.

Analyze Code

```
@Get('/')
@Render('products/index')
index() {
  const viewData = [];
  viewData['title'] = 'Products - Online Store';
  viewData['subtitle'] = 'List of products';
  viewData['products'] = ProductsController.products;
  return {
    viewData: viewData,
  };
}
```

The *index* method gets the array of products and sends them to the *products/index* view to be displayed.

Products index view

In *views/*, create a subfolder called *products*. Then, in *views/products*, create a new file called *index.hbs*, and fill it with the following code.

Add Entire Code

```
{{#> app}}
  {{#*inline "content"}}
    <div class="row">
      {{#each viewData.products}}
      <div class="col-md-4 col-lg-3 mb-2">
        <div class="card">
          <img src="/img/{{image}}" class="card-img-top img-card">
          <div class="card-body text-center">
                  <a href="/products/{{id}}" class="btn bg-primary text-white">{{name}}</a>
          </div>
        </div>
      </div>
      {{/each}}
    </div>
  {{/inline}}
{{/app}}
```

The important part of the previous code is the *{{#each viewData.products}}*. *each* is a hbs built-in helper which allows us to iterate over a list. We iterate through each product and display the product image and name. More information about hbs built-in helpers can be found here: https://handlebarsjs.com/guide/builtin-helpers.html.

Analyze Code

```
<a href="/products/{{id}}" class="btn bg-primary text-white">{{name}}</a>
```

Finally, we put a link to the product name. The link will route to a new *ProductsController* method (which will be defined later) and it requires a parameter to be sent. In this case, we send the product id of the current iterated product.

Showing a specific product

Products controller

In *src/products.controller.ts*, make the following changes in **bold**.

Modify Bold Code

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


@Controller('/products')
export class ProductsController {
    …
  @Get('/')
  @Render('products/index')
  index() {
    …
  }


  @Get('/:id')
```

```
  @Render('products/show')
  show(@Param() params) {
    const product = ProductsController.products[params.id - 1];
    const viewData = [];
    viewData['title'] = product.name + ' - Online Store';
    viewData['subtitle'] = product.name + ' - Product Information';
    viewData['product'] = product;
    return {
      viewData: viewData,
    };
  }
}
```

First, we import the *Param* decorator. This decorator will be used to collect route parameters later.

We link the *show* method to the ("/:id") route. Because we include the *ProductsController* path prefix, the complete route is ("products/:id"). "/products/:id" takes a parameter called *id* . This parameter is the product *id* to identify which product data to show. For example, if we access "/products/1" URL, the application will display the data of the product with id=1.

The *show* method contains a *params* argument decorated with *@Param()* . *@Param()* makes the route parameters available as properties inside the method. We can access the *id* parameter by referencing *params.id* . Then, we extract the corresponding product (from the *ProductsController.products* array) based on the *params.id* value. We subtract one unit since we stored the product with id=1 in the *ProductsController.products* array index 0, the product with id=2 in the *ProductsController.products* array index 1, and so on. We then send the product data to the *products/show* view.

Product show view

In *views/products* , create a new file called *show.hbs* , and fill it with the following code.

Add Entire Code

```
{{#> app}}
  {{#*inline "content"}}
    {{#with viewData.product}}
    <div class="card mb-3">
      <div class="row g-0">
        <div class="col-md-4">
          <img src="/img/{{image}}" class="img-fluid rounded-start">
        </div>
        <div class="col-md-8">
          <div class="card-body">
                <h5 class="card-title">
                  {{name}} (${{price}})
                </h5>
                <p class="card-text">{{description}}</p>
                <p class="card-text"><small class="text-muted">Add to Cart</small></p>
          </div>
        </div>
      </div>
    </div>
    {{/with}}
  {{/inline}}
{{/app}}
```

The *with* hbs helper dives into the *viewData.product* object and gives us direct access to the *product* properties(*name* , *description* , *image* , and *price*). We show the *product image* , *name* , *price* , and *description* in the markup. Remember, we are using dummy data. This will change in upcoming chapters.

TIP: In the last examples, we have defined a structure to store our controllers, controllers' methods, routes, and views. For example, the *products/* route, is linked to the *ProductsController index* method, which displays the *products/index* view. Try to use this strategy across the entire project as it facilitates finding the views of the corresponding controllers' methods and vice versa.

Introducing Nest Modules

Modules are classes annotated with a *@Module()* decorator. The *@Module()* decorator provides metadata that Nest uses to organize the application structure. Each application has at least one module, a root module. The root module is the starting point Nest uses to build the application. In our case, *src/app.module.ts* is our root module.

The *@Module()* decorator allows defining a set of properties:

- **controllers:** the set of controllers in this module to be instantiated.

- **providers:** the providers that will be instantiated by the Nest injector and shared at least across this module (we will use providers in Chapter 14).
- **imports:** the list of imported modules required in this module (we will use imports in Chapter 12).
- **exports:** the subset of providers provided by this module and available to other modules (we will use exports in Chapter 17).

You can design small Nest applications which consist of only one module. However, for complex applications, it is recommended to employ multiple modules. We will split our Online Store application into four modules (detailed later).

Registering ProductsController in AppModule

Let's register the *ProductsController* in our *AppModule* . In *src/app.module.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { ProductsController } from './products.controller';

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

We import our newly created *ProductsController* , and register it in the *AppModule controllers* property. This way, all the *ProductsController* routes will be available for access from the browser.

Updating links in Header

Now, let's include the products links in the header. In *views/layouts/app.hbs* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
...
            <div class="navbar-nav ms-auto">
              <a class="nav-link active" href="/">Home</a>
              <a class="nav-link active" href="/products">Products</a>
              <a class="nav-link active" href="/about">About</a>
            </div>
...
```

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now, you can go to the *Products* page (see Fig. 10-1) and navigate to a specific product (see Fig. 10-2).

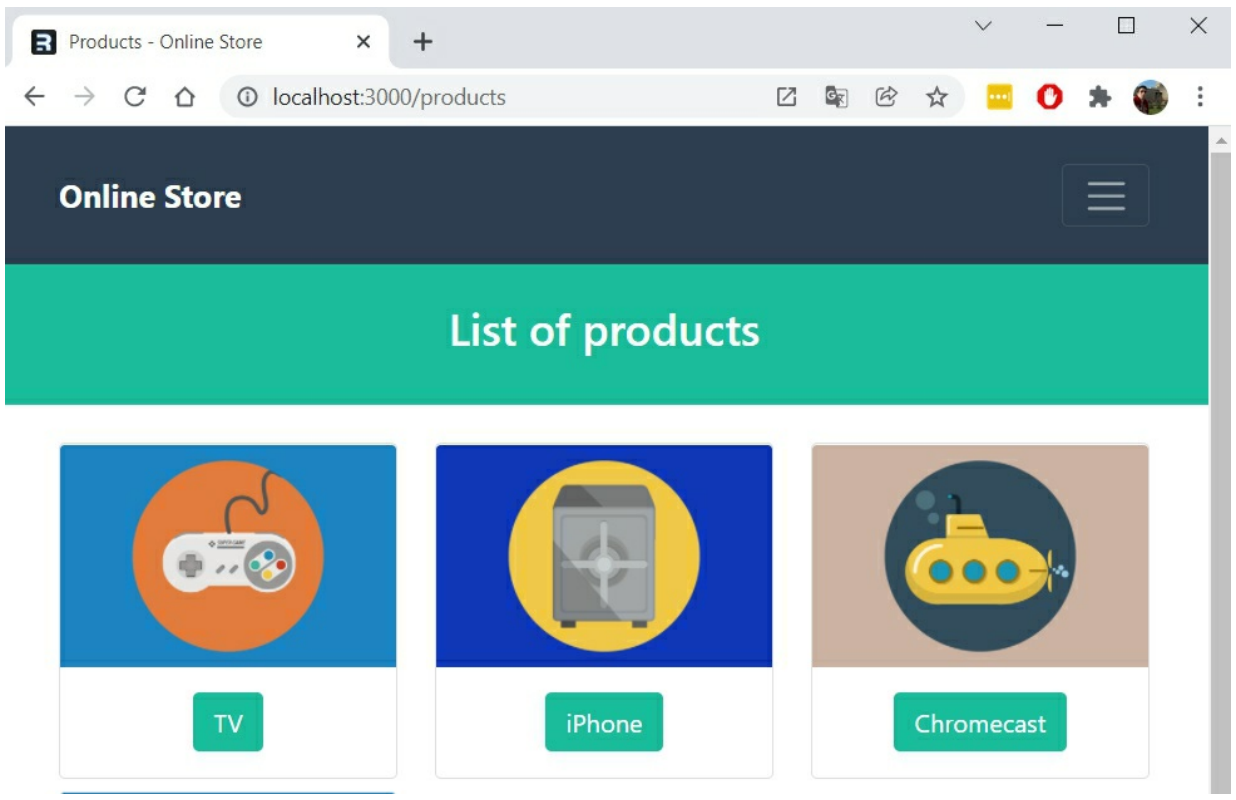

Figure 10-1. Online Store – Products page.

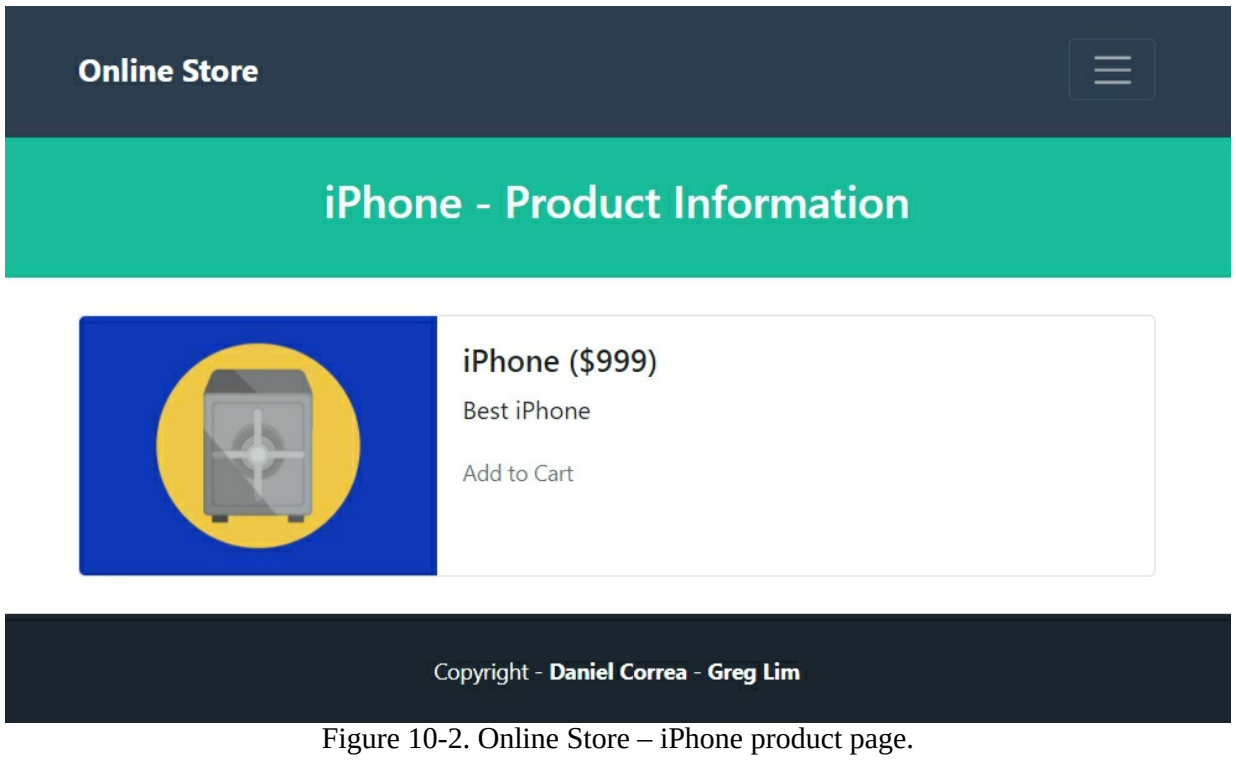

Figure 10-2. Online Store – iPhone product page.

Chapter 11 – Configuration of MySQL Database

Introduction to MySQL

MySQL is the most popular Open-Source SQL database management system developed, distributed, and supported by Oracle.

- **MySQL is a database management system**. A database is a structured collection of data. It may be anything from a simple shopping list to a picture gallery or the vast amounts of information in a corporate network. To add, access, and process data stored in a computer database, you need a database management system such as **MySQL**.
- **MySQL databases are relational.** A relational database stores data in separate tables rather than putting all the data in one big storeroom. The database structures are organized into physical files optimized for speed. MySQL provides a logical model, with objects such as databases, tables, views, rows, and columns to offer a flexible programming environment.

MySQL tables

A table is used to organize data in the form of rows and columns. It is used for both storing and displaying records in a structured format. It is like worksheets in a spreadsheet application. The **columns** specify the data type, whereas the **rows** contain the actual data. Below is how you could imagine a MySQL table (see Fig. 11-1).

id	name	description	image	price
1	TV	Best tv	game.png	1000
2	iPhone	Best iPhone	safe.png	999
3	Chromecast	Best Chromecast	submarine.png	30
4	Glasses	Best Glasses	game.png	100

Figure 11-1. Product table.

MySQL installation

There are several different ways of installing MySQL. In this book, we will install MySQL and a MySQL administration tool called phpMyAdmin. Both tools can be found in a development environment called XAMPP. So, let's install XAMPP.

XAMPP

XAMPP is a popular PHP development environment. XAMPP is a free, easy to install Apache distribution containing MySQL, PHP, and Perl. XAMPP also includes phpMyAdmin. If you don't have XAMPP installed, go to https://www.apachefriends.org/download.html, download and install it.

Configuring our database

Execute XAMPP, *start* the Apache Module, then *start* MySQL module, and click the *MySQL Admin* button (of the MySQL module). This takes us to the phpMyAdmin application (see Fig. 11-2).

Figure 11-2. Starting MySQL module in XAMPP.

Note: If you are using WAMP or another similar application, the phpMyAdmin application can be commonly accessed through the following route: http://localhost/phpmyadmin/.

In the phpMyAdmin application enter your username and password. Default values are "root" (for the username),

and an empty password (see Fig. 11-3).

Figure 11-3. XAMPP phpMyAdmin application.

Once logged in to phpMyAdmin, click the *Databases* tab. Enter the database name "online_store", and click *Create* (see Fig. 11-4).

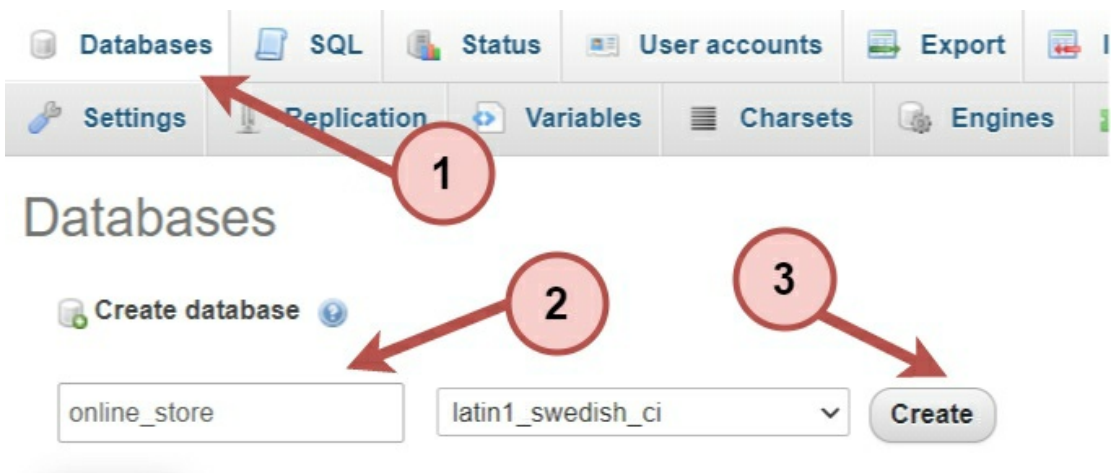

Figure 11-4. Database creation.

We will use the username, password, and database name in the following chapter.

Chapter 12 – Configuration of TypeORM

Nest Databases

Nest is database agnostic, allowing you to integrate with any SQL or NoSQL database easily. Nest provides tight integration with TypeORM, Sequelize, and Mongoose through the @nestjs/typeorm, @nestjs/sequelize, and @nestjs/mongoose packages respectively (https://docs.nestjs.com/techniques/database).

Nest suggests using TypeORM because it's the most mature Object Relational Mapper (ORM) available for TypeScript. Since it's written in TypeScript, it integrates well with the Nest framework.

Introduction to TypeORM

TypeORM is an ORM that can run in NodeJS, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo, and Electron platforms and can be used with TypeScript and JavaScript. Its goal is to help you develop any application that uses databases, from small applications with a few tables to large scale enterprise applications with multiple databases.

TypeORM provides support for many relational databases, such as PostgreSQL, MySQL, Oracle, Microsoft SQL Server, SQLite, and even NoSQL databases like MongoDB.

Nest TypeORM integration

TypeORM installation

To connect our Online Store application with our MySQL database, we need to install and integrate TypeORM. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm install --save @nestjs/typeorm typeorm mysql2
```

This installs TypeORM and other required libraries to work with MySQL.

TypeORM config file

Now, let’s define the configuration file (where we will place our database credentials). In the project root directory, create a new file called *ormconfig.json* , and fill it with the following code.

Add Entire Code

```
{
   "type": "mysql",
   "host": "localhost",
   "port": 3306,
   "username": "root",
   "password": "",
   "database": "online_store",
   "entities": ["dist/**/*.entity{.ts,.js}"],
   "synchronize": true
}
```

We used the username, password, and database name from the previous chapter.

TypeORM import

We import the TypeORM in our *AppModule* . In *src/app.module.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { ProductsController } from './products.controller';
import { TypeOrmModule } from '@nestjs/typeorm';


@Module({
   imports: [
      TypeOrmModule.forRoot(),
   ],
   controllers: [AppController, ProductsController],
})
export class AppModule {}
```

We imported the *TypeOrmModule* and register it in the *AppModule imports* property by invoking the *TypeOrmModule.forRoot()* method. Remember that the *@Module imports* property is used to register other

modules required by the current module. In this case, our root module (*AppModule*) requires *TypeOrmModule* .

The *TypeOrmModule.forRoot()* method configures the TypeORM library. It receives an optional parameter with the configuration values (i.e., database credentials). If no parameter is passed, the Nest application will try to load *ormconfig.json* file and pass the information to the TypeORM library.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

You should see the application running normally (see Fig. 12-1). This means that the integration was successful, and the application is now connected to the database. If you get an error like “ERROR [TypeOrmModule] Unable to connect to the database”, please check that MySQL is running and the database credentials are correct.

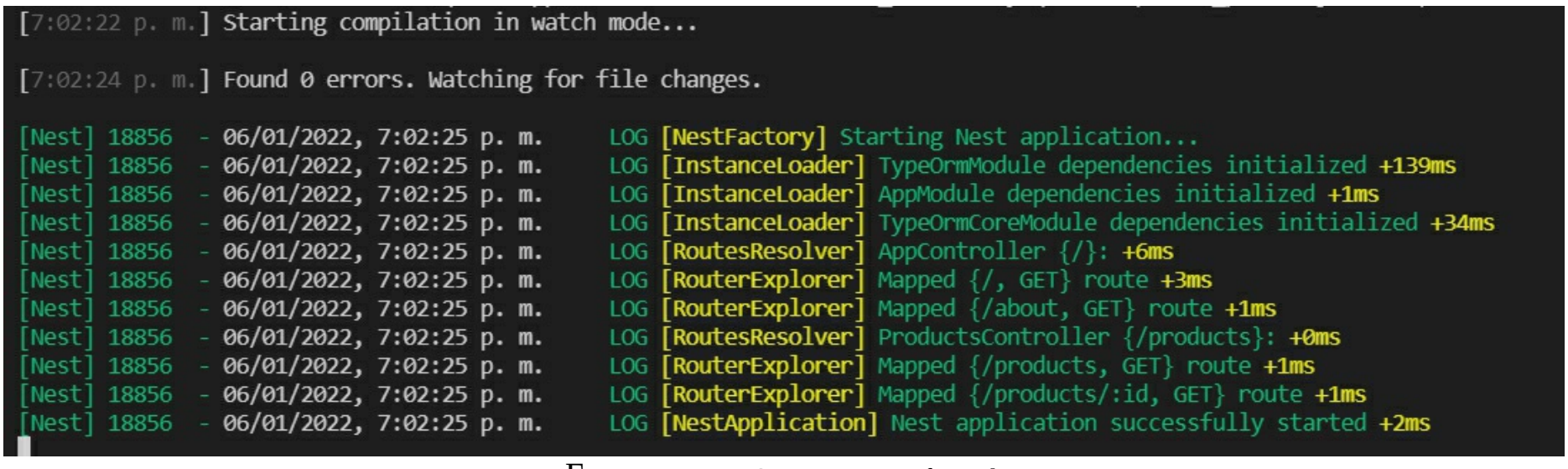

Figure 12-1. TypeORM proper configured.

Chapter 13 – Product Entity

Introduction to TypeORM Entities

An **entity** is a class that maps to a database 'table' (or 'collection' in MongoDB). You can create an entity by defining a new class and mark it with *@Entity()* decorator. More information about TypeORM entities can be found here: https://typeorm.io/#/entities. Let's create an Entity to understand it in a practical approach.

Creating Product Entity

First, in *src* folder, create a subfolder called *models* . In *src/models* , create a new file *product.entity.ts* , and fill it with the following code.

Add Entire Code

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


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

    @Column()
    name: string;

    @Column()
    description: string;

    @Column()
    image: string;

    @Column()
    price: number;
}
```

TypeORM entities consist of columns and relations. In the above code, we define a *Product* entity (which will become into a *product* table) with five attributes (*id* , *name* , *description* , *image* , and *price*). By specifying the *@Column* decorator, all these attributes will become columns of the product table.

TypeORM requires that each entity have a primary column (or ObjectId column if using MongoDB). In our case, the Product entity primary column is *id* . The *@PrimaryGeneratedColumn()* creates a primary column that automatically generates an auto-increment value.

Quick discussion: We are storing our entities inside the *src/models* folder. We named that folder as *models* to keep it consistent with our MVC architecture. However, not all Nest applications store their models (or entities) inside a *models* folder. The official documentation recommends storing the models near their domain, in the corresponding 'module' directory. Currently, we only have one module (the *AppModule*). Later, we will create four modules. Most of those modules will use most of the entities that we will define. Thus, we decided not to place the entities in specific a 'module' directory but under a "global" *src/models* folder.

Synchronizing the database

We have created a *Product* entity. Our TypeORM config file (*ormconfig.json*) contains an option called *synchronize* with a value of *true* . It also contains an option called *entities* that specifies the directories where TypeORM should search for entities. The value is *dist/**/*.entity{.ts,.js}* . The value is a regular expression which searches for all files (in the *dist* folder) that contain *.entity* in their filenames. Remember that the *dist* folder contains the compiled production version of the source code.

Why are the *synchronize* and *entities* options important? Because, once we start the server, Nest generates the *dist* folder, runs the code, and executes TypeORM. Since *synchronize* option is *true* , TypeORM searches for new entities or changes in those entities and automatically creates the corresponding tables based on those entities. Be careful to note that setting *synchronize* to *true* shouldn't be used in a production environment, otherwise you can mistakenly lose production data.

Let's synchronize our database. **First, stop the server.** Then, In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now, you can see the new *product* table in your *online_store* database (see Fig. 13-1).

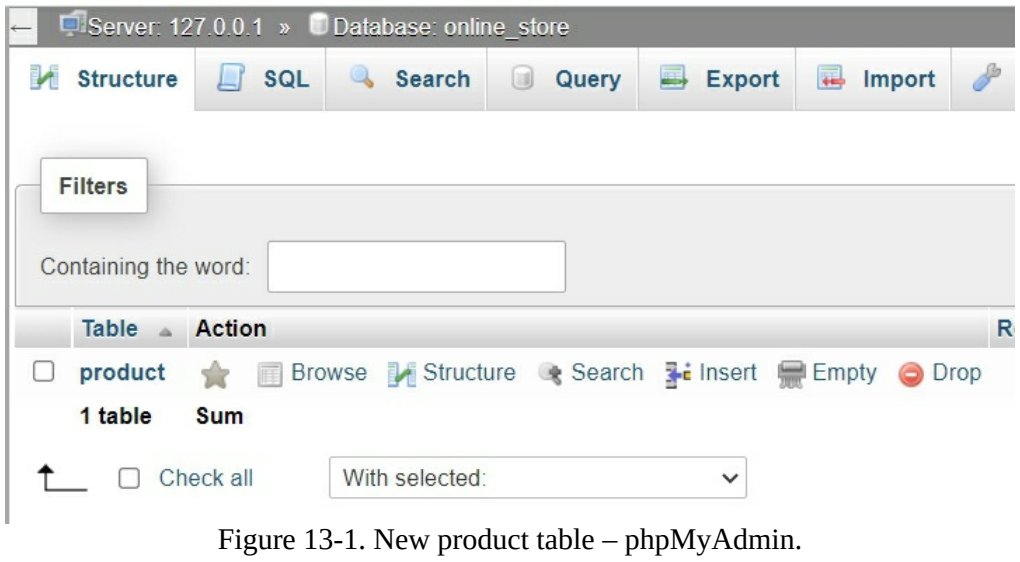

Figure 13-1. New product table – phpMyAdmin.

Note: if you create a new entity or modify an existing one, you should repeat the previous process to apply the changes over the database (stop and start the server).

Inserting products

Let’s insert four products into our database. For now, we will insert it manually (through SQL queries). Later, we will insert products through a form in an upcoming chapter.

In phpMyAdmin, click the *online_store* database, click the *SQL* tab, paste the following SQL queries, and click *go* (see Fig. 13-2).

Execute in Database

```
INSERT INTO product (name, description, image, price) VALUES ('TV', 'Best TV', 'game.png', '1000');
INSERT INTO product (name, description, image, price) VALUES ('iPhone', 'Best iPhone', 'safe.png', '999');
INSERT INTO product (name, description, image, price) VALUES ('Chromecast', 'Best Chromecast', 'submarine.png', '30');
INSERT INTO product (name, description, image, price) VALUES ('Glasses', 'Best Glasses', 'game.png', '100');
```

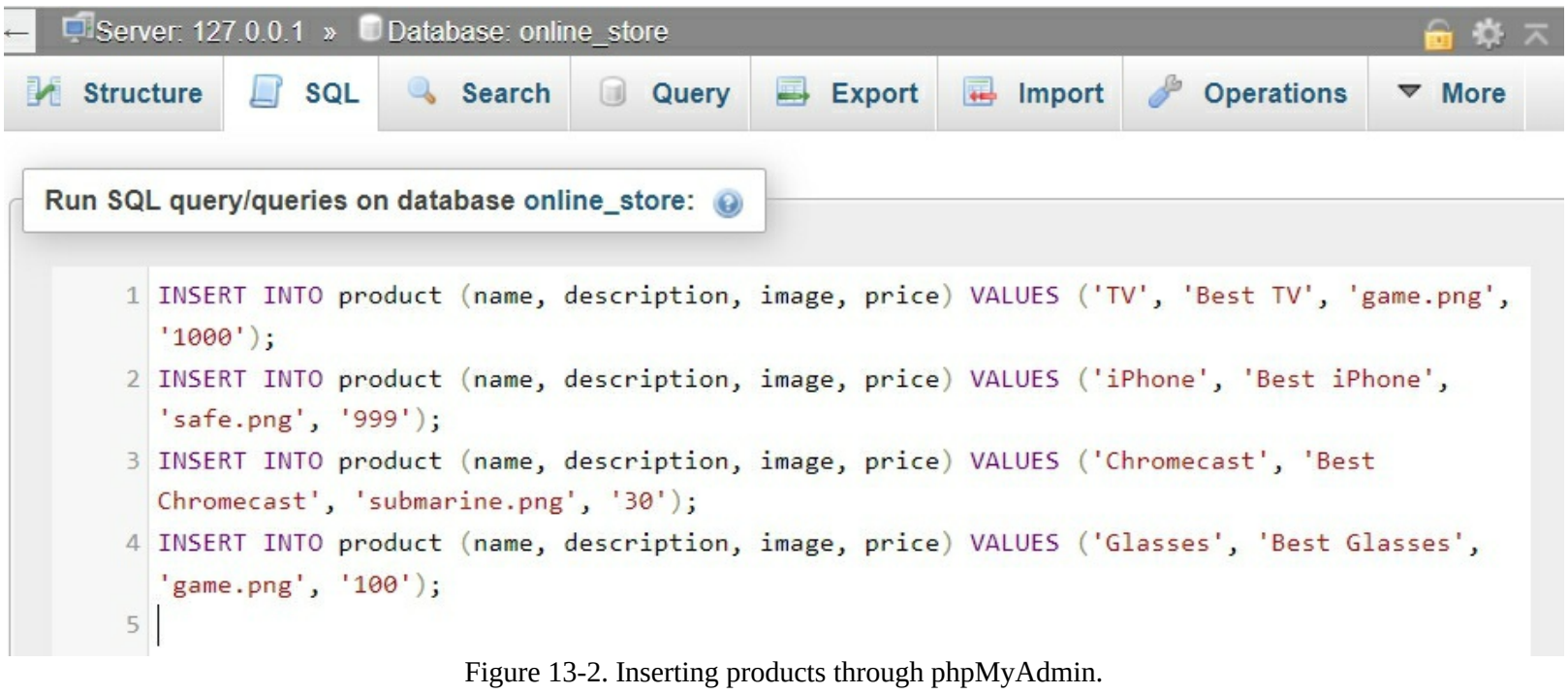

Figure 13-2. Inserting products through phpMyAdmin.

Let’s check that the products were successfully inserted. In phpMyAdmin, click the *online_store* database, and click the *product* table. Hopefully, you will see the four products inserted (see Fig. 13-3).

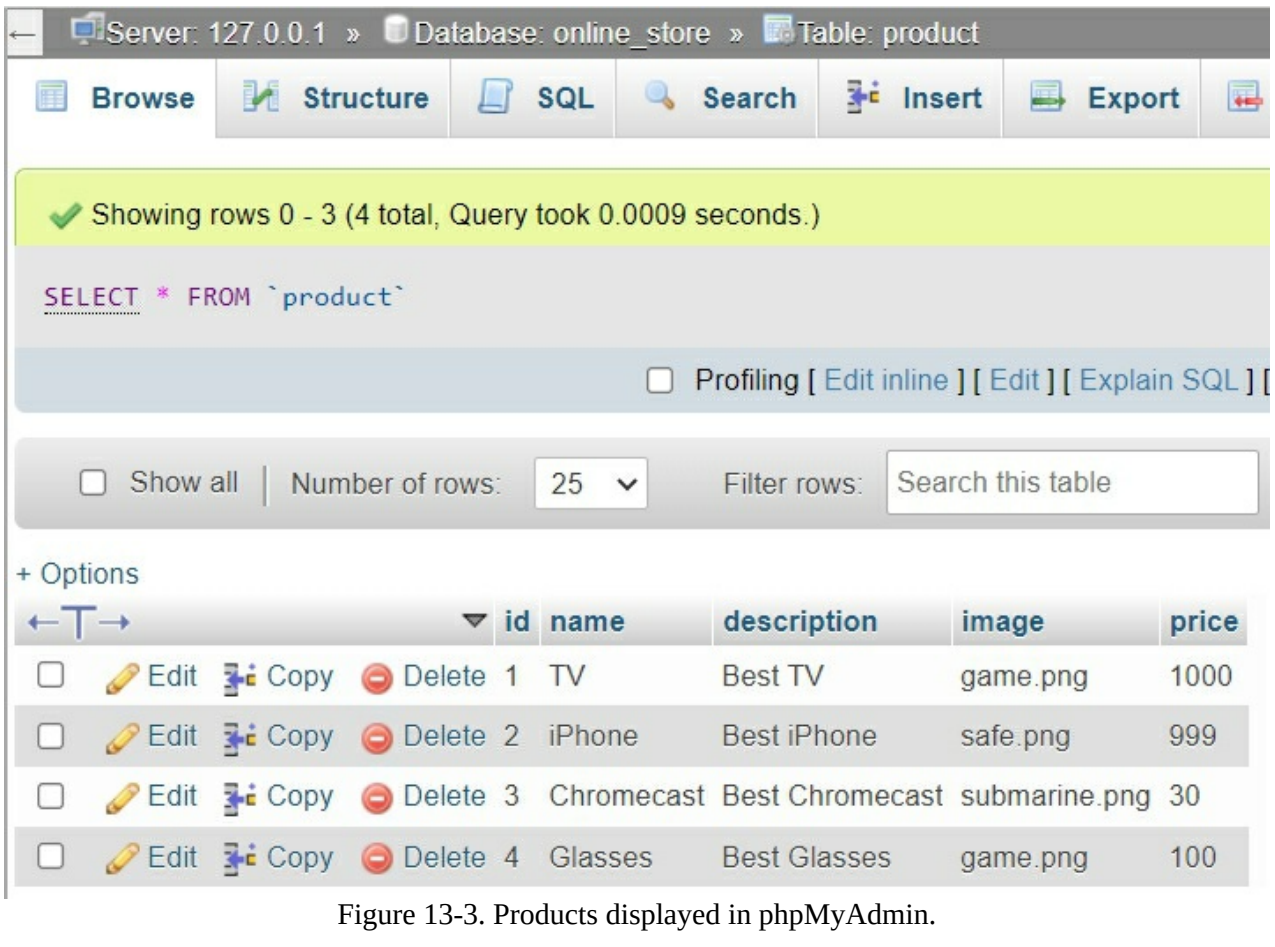

Figure 13-3. Products displayed in phpMyAdmin.

Chapter 14 – List Products with Database Data

To extract data from the MySQL database, we will create a *ProductsService* (a Nest provider).

Introducing Nest Providers

Until now, we have introduced Nest Controllers and Modules. The last essential part of Nest applications is Nest Providers.

A **provider** is a class or component that provides specific functionality to an application and can be **injected** as a dependency. Many basic Nest classes such as services, repositories, factories, and helpers may be treated as providers. With providers, you delegate the instantiation of dependencies to the NestJS runtime system, instead of doing it in your code imperatively. We will see how Nest instantiates and injects those dependencies later in this chapter.

Nest suggests that controllers should handle HTTP requests, and delegate more complex tasks to providers. For example, storing a product into the database should be a provider's task, not a controller's task. More information about Nest Providers can be found at: https://docs.nestjs.com/providers.

To define a provider, we need to create a class and attach the *@Injectable()* decorator. Let's see a provider in action.

Creating ProductsService

In *src/models* , create a new file called *products.service.ts* , and fill it with the following code.

Add Entire Code

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Product } from './product.entity';


@Injectable()
export class ProductsService {
   constructor(
      @InjectRepository(Product)
      private productsRepository: Repository<Product>,
   ) {}


   findAll(): Promise<Product[]> {
      return this.productsRepository.find();
   }


   findOne(id: string): Promise<Product> {
      return this.productsRepository.findOne(id);
   }
}
```

Let's analyze the code by parts.

Analyze Code

```
@Injectable()
export class ProductsService {
```

The *@Injectable()* decorator declares that *ProductsService* can be injected and instantiated by Nest as a provider.

Analyze Code

```
  constructor(
     @InjectRepository(Product)
     private productsRepository: Repository<Product>,
  ) {}
```

In the *constructor* , the *@InjectRepository()* decorator injects the products repository. The products repository must be registered to be injected in our *AppModule* later. We will explain how those Nest injections work in detail later in this chapter.

The *productsRepository* attribute is an instance of the *Repository<Product>* class. **TypeORM *Repository*** is a class that works with our entity objects (in this case our *Product* entity). The TypeORM *Repository* class provides a set of useful methods (such as find, create, remove, insert, and update) to manage a specific entity.

Analyze Code

```
  findAll(): Promise<Product[]> {
```

```
    return this.productsRepository.find();
  }
```

The *findAll* method returns a Promise consisting of an array of products. *findAll* uses the *productsRepository* attribute, which invokes the *find* method (inherited from the TypeORM *Repository* class) and returns all products from the database.

Analyze Code

```
  findOne(id: string): Promise<Product> {
    return this.productsRepository.findOne(id);
  }
```

The *findOne* method receives an *id* and returns a Promise consisting of one product. This method uses the *productsRepository* attribute, invokes its *findOne* method (inherited from the TypeORM *Repository* class), and returns the specific product from the database based on *id* .

Note: more information about the available TypeORM *Repository* methods can be found here: https://typeorm.delightful.studio/classes/_repository_repository_.repository.html.

Modifying AppModule

In *src/app.module.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { ProductsController } from './products.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { ProductsService } from './models/products.service';
import { Product } from './models/product.entity';


@Module({
  imports: [
    TypeOrmModule.forRoot(),
    TypeOrmModule.forFeature([Product]),
  ],
  controllers: [AppController, ProductsController],
  providers: [ProductsService],
})
export class AppModule {}
```

We import *ProductsService* and the *Product* entity. We register *ProductsService* in the module *providers* property. This means, that *ProductsService* will be available to be injected and used across *AppModule* (for example, inside *AppController* and *ProductsController*). We also register the *TypeOrmModule.forFeature([Product])* in the module *imports* property. The *forFeature()* method is used to define which TypeORM repositories are registered in the current scope. With that in place, Nest injects the products repository into the *ProductsService* constructor using the *@InjectRepository()* decorator.

Modifying ProductsController

In *src/products.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Controller, Get, Render, Param, Res } from '@nestjs/common';
import { ProductsService } from './models/products.service';


@Controller('/products')
export class ProductsController {
  constructor(private readonly productsService: ProductsService) {}


  static products = [
    {
      id: '1',
      name: 'TV',
      description: 'Best tv',
      image: 'game.png',
      price: '1000',
    },
    {
      id: '2',
      name: 'iPhone',
      description: 'Best iPhone',
      image: 'safe.png',
      price: '999',
    },
```

```
    {
      id: '3',
      name: 'Chromecast',
      description: 'Best Chromecast',
      image: 'submarine.png',
      price: '30',
    },
    {
      id: '4',
      name: 'Glasses',
      description: 'Best Glasses',
      image: 'game.png',
      price: '100',
    },
];

@Get('/')
@Render('products/index')
async index() {
    const viewData = [];
    viewData['title'] = 'Products - Online Store';
    viewData['subtitle'] = 'List of products';
    viewData['products'] = ProductsController.products;
    viewData['products'] = await this.productsService.findAll();
    return {
      viewData: viewData,
    };
  }

  @Get('/:id')
  @Render('products/show')
  async show(@Param() params, @Res() response) {
    const product = ProductsController.products[params.id - 1];
    const product = await this.productsService.findOne(params.id);
    if (product === undefined) {
      return response.redirect('/products');
    }
    const viewData = [];
    viewData['title'] = product.name + ' - Online Store';
    viewData['subtitle'] = product.name + ' - Product Information';
    viewData['product'] = product;
    return {
      viewData: viewData,
    };
    return response.render('products/show', { viewData: viewData });
  }
}
```

Let’s analyze the code by parts.

Analyze Code

```
import { ProductsService } from './models/products.service';

@Controller('/products')
export class ProductsController {
  constructor(private readonly productsService: ProductsService) {}
```

We import the *ProductsService* (remember this is a provider marked with the *@Injectable()* decorator). Then, we define a constructor for our *ProductsController* class. In the constructor, Nest will resolve the *productsService* private attribute by creating and returning an instance of *ProductsService* (this is called “constructor injection”). Nest will inject this instance since we registered the *ProductsService* provider in the *AppModule* . Remember that the *AppModule* registered the *AppController* and *ProductsController* as controllers. So, all the providers defined in the *AppModule* will be available to be injected in the *AppController* and *ProductsController* when they are needed. **That is how Nest dependency injection works**. It can be challenging to understand, so you can check out a more detailed explanation here: https://docs.nestjs.com/fundamentals/custom-providers.

We will use the *ProductsService* to connect to the *product* table.

Note: Providers and dependency injection are fundamental concepts in Nest. You will find many examples and tutorials based on those concepts. There is also a great article that the Nest official documentation recommends reading: https://angular.io/guide/dependency-injection.

Analyze Code

```
  static products = [
  {
    id: '1',
    name: 'TV',
```

```
      description: 'Best tv',
      image: 'game.png',
      price: '1000',
    },
    ...
```

We remove the *products* dummy attribute since we don't need it anymore. We will instead retrieve the products data from the database.

Analyze Code

```
@Get('/')
@Render('products/index')
async index() {
  const viewData = [];
  viewData['title'] = 'Products - Online Store';
  viewData['subtitle'] = 'List of products';
  viewData['products'] = ProductsController.products;
  viewData['products'] = await this.productsService.findAll();
  return {
    viewData: viewData,
  };
}
```

Note that we are not using the *products* static attribute anymore. Instead, we invoke the *this.productsService.findAll()* method which returns a Promise with all the products collected from the database. Since that method returns a Promise, we use the *await* keyword (which makes the application wait until that promise completes and returns its result). Once the products are returned from the database, the code continues its execution, passes the *viewData* to the view, and renders the *products/index* view. Finally, since we use the *await* keyword, we must define the index method as *async* . More information about async/await can be found here: https://javascript.info/async-await.

Analyze Code

```
@Get('/:id')
@Render('products/show')
async show(@Param() params, @Res() response) {
  const product = ProductsController.products[params.id - 1];
  const product = await this.productsService.findOne(params.id);
  if (product === undefined) {
    return response.redirect('/products');
  }
  const viewData = [];
  viewData['title'] = product.name + ' - Online Store';
  viewData['subtitle'] = product.name + ' - Product Information';
  viewData['product'] = product;
  return {
    viewData: viewData,
  };
  return response.render('products/show', { viewData: viewData });
}
```

We remove the *@Render* decorator because we will use redirections inside the *show* method (and there is a conflict when using the *@Render* decorator and redirection at the same time). We import and use the *@Res* decorator in the *response* method argument. This decorator allows us to access the express response object. Then, we use the *productsService.findOne* method to retrieve the specific product from the database. If the product is not found, we use the *response* object to redirect the user to the ("/products") route. Otherwise, we set the *viewData* and use the *response* object to render the original view with its data (*products/show*).

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Navigate to the ("/products") route, and you will see the products retrieved from our MySQL database. Try to visit a specific product with an *id* that does not exists (i.e., "/products/21"), the application will be redirected to the ("/products") route.

Chapter 15 – Refactoring List Products

Many things can be improved in the previous code. For example, we will refactor our *Product* entity, *ProductsController* , and product *views* . These changes will make our code more maintainable, understandable, and clean. Again, many of these changes apply not only to a Nest project but are also general tips that can be replicated in most MVC frameworks.

A project without getters and setters

Let’s analyze our current code. It is an excerpt of our *products.controller.ts* file.

Analyze Code

```
@Get('/:id')
@Render('products/show')
async show(@Param() params) {
   …
   const viewData = [];
   viewData['title'] = product.name + ' - Online Store';
   viewData['subtitle'] = product.name + ' - Product Information';
   viewData['product'] = product;
   return {
      viewData: viewData,
   };
}
```

We are accessing the product entity attributes directly (*product.name*). Now, let’s analyze the *views/products/show.hbs* view.

Analyze Code

```
{{#with viewData.product}}
        …
                <h5 class="card-title">
                    {{name}} ($ {{price}})
                </h5>
                <p class="card-text">{{description}}</p>
                <p class="card-text"><small class="text-muted">Add to Cart</small></p>
        …
{{/with}}
```

Again, we are accessing the product’s name directly (*name*).

What is the problem with accessing the entity data this way? Imagine that your boss tells you, “We need to display all products’ names in uppercase throughout the entire application”. That’s a big issue as we extract products’ names over several different views and controllers. This simple requirement will require us modifying several views and controllers. For now, let’s see what have to do to achieve that requirement.

We should modify the *products.controller.ts* controller this way (**do not implement this change, it is used just to exemplify this scenario**):

Analyze Code

```
@Get('/:id')
@Render('products/show')
async show(@Param() params) {
   …
   const viewData = [];
   viewData['title'] = product.name . toUpperCase() + ' - Online Store';
   viewData['subtitle'] = product.name.toUpperCase() + ' - Product Information';
   viewData['product'] = product;
   return {
      viewData: viewData,
   };
}
```

And the *views/products/show.hbs* view this way (**do not implement this change, it is used just to exemplify this scenario**):

Analyze Code

```
{{#with viewData.product}}
        …
                <h5 class="card-title">
                    {{loud name}} ($ {{price}})
                </h5>
                <p class="card-text">{{description}}</p>
                <p class="card-text"><small class="text-muted">Add to Cart</small></p>
        …
```

```
    {{/with}}
```

Finally, we should create and register a hbs helper (called *loud*). Let's look at that helper.

Analyze Code

```
Handlebars.registerHelper('loud', function (aString) {
    return aString.toUpperCase()
})
```

There are two significant issues with this strategy. (i) We have many duplicate codes throughout the application (the *toUpperCase* function is used over several places). (ii) We must check all controllers, all views, and maybe dozens or hundreds of files to check where we need to apply the *toUpperCase* function or the *loud* hbs custom helper. This is not maintainable. So, let's refactor our code to implement a better strategy.

Project with getters and setters

Refactoring the Product Entity

First, let's refactor our *Product* entity. In *src/models/product.entity.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
@Entity()
export class Product {
  ...

  @Column()
  price: number;

  getId(): number {
    return this.id;
  }

  setId(id: number) {
    this.id = id;
  }

  getName(): string {
    return this.name;
  }

  setName(name: string) {
    this.name = name;
  }

  getDescription(): string {
    return this.description;
  }

  setDescription(description: string) {
    this.description = description;
  }


  getImage(): string {
    return this.image;
  }

  setImage(image: string) {
    this.image = image;
  }

  getPrice(): number {
    return this.price;
  }

  setPrice(price: number) {
    this.price = price;
  }
}
```

For each *Product* attribute, we define its corresponding getter and setter. We will use getters and setters to access and modify our entity/model attributes. It has a lot of advantages which we will talk about later.

Refactoring the ProductsController

In *src/products.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
  @Get('/:id')
  async show(@Param() params, @Res() response) {
    const product = await this.productsService.findOne(params.id);
    if (product === undefined) {
      return response.redirect('/products');
    }
    const viewData = [];
    viewData['title'] = product.getName() + ' - Online Store';
    viewData['subtitle'] = product.getName() + ' - Product Information';
    viewData['product'] = product;
    return response.render('products/show', { viewData: viewData });
  }
}
```

Now, we access the *product* attributes through the corresponding getters.

Refactoring the products/index view

In *views/products/index.hbs* , make the following changes in **bold**.

Modify Bold Code

```
...
        {{#each viewData.products}}
        <div class="col-md-4 col-lg-3 mb-2">
          <div class="card">
            <img src="/img/{{getImage}}" class="card-img-top img-card">
            <div class="card-body text-center">
                    <a href="/products/{{getId}}" class="btn bg-primary text-white">{{getName}}</a>
            </div>
          </div>
        </div>
        {{/each}}
...
```

Like the previous controller, we access the product attributes through the corresponding getters. Notice that hbs don't allow parentheses '()' so getters should be called without them.

Refactoring the products/show view

In *views/products/show.hbs* , make the following changes in **bold**.

Modify Bold Code

```
{{#> app}}
  {{#*inline "content"}}
    {{#with viewData.product}}
    <div class="card mb-3">
      <div class="row g-0">
        <div class="col-md-4">
          <img src="/img/{{getImage}}" class="img-fluid rounded-start">
        </div>
        <div class="col-md-8">
          <div class="card-body">
                  <h5 class="card-title">
                    {{getName}} (${{getPrice}})
                  </h5>
                  <p class="card-text">{{getDescription}}</p>
                  <p class="card-text"><small class="text-muted">Add to Cart</small></p>
        </div>
      </div>
    </div>
  </div>
  {{/with}}
  {{/inline}}
{{/app}}
```

We access the *product* attributes through getters.

Analyzing getters and setters

For now, the application looks the same. We only modified the way we access entity attributes. So, what is the advantage? Let's revisit the boss requirement where we need to display all products' names in uppercase over the entire application.

In this case, we only need to modify the *Product* entity file. Specifically, the *getName* method. Let's see the

modification (**you can apply the following change or leave it as it is**).

Analyze Code

```
...
   getName(): string {
      return this.name.toUpperCase();
   }
...
```

If you run the application, you will see that all products' names appear in uppercase. We only required one single change in one specific location. That is the power of the use of getters or setters. **The definition and use of getters and setters guarantee a unique access point to the entity/model attributes**. That is part of what some people call encapsulation, one of the three pillars of object-oriented programming.

TIP: Always try to access your entity/model attributes through getters and setters. **It will make it easier to add functionalities in the future**. You can even include a new rule saying that entities/models' attributes must be accessed through their corresponding getters and setters (in your architectural rules document).

For the rest of the application, we will use the getters and setters' strategy. We hope you understand their importance now.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

The application should be properly working. If you applied the modification in the *getName* method, you would see all products' names in uppercase (see Fig. 15-1).

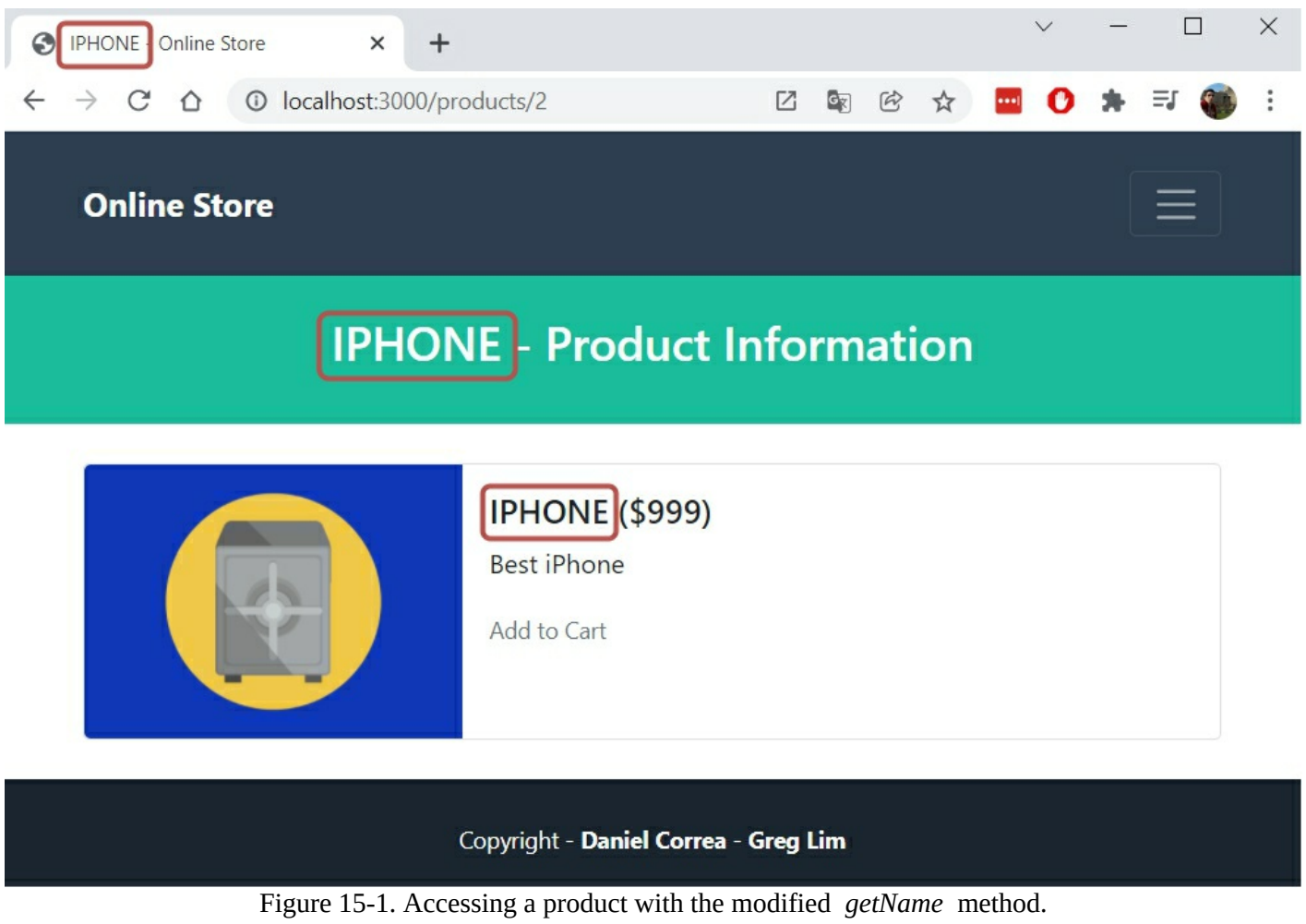

Figure 15-1. Accessing a product with the modified *getName* method.

Chapter 16 – Admin Panel

Many web applications have an admin section that administrators or moderators can access to manage the application data such as registering products, managing sales, generating reports, and managing users. Commonly, this section is called an **administration panel**. Due to its nature, this section is secured by a login or authentication system. For now, we will create a public administration panel, which any visitor can access. We will later implement a login system to secure and verify that only allowed users (admins) can access this section.

To create the admin panel, we need to include a layout, a controller, a view, some new files, and a new module. So, let's begin.

Admin Layout

Commonly, administration panels look different from the main pages. They are quite minimal, and many of them display information like a spreadsheet. Let's start our admin panel construction by defining a new layout. This layout will be used across the admin panel pages.

In *views/layouts* , create a new file called *admin.hbs* , and fill it with the following code.

Add Entire Code

```
<!doctype html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css" rel="stylesheet"
    crossorigin="anonymous" />
  <link href="/css/admin.css" rel="stylesheet" />
  <title>{{#if viewData.title}}{{ viewData.title }}{{else}}Admin - Online Store{{/if}}</title>
</head>

<body>
  <div class="row g-0">
    <!-- sidebar -->
    <div class="p-3 col fixed text-white bg-dark">
      <a href="/admin" class="text-white text-decoration-none">
        <span class="fs-4">Admin Panel</span>
      </a>
      <hr />
      <ul class="nav flex-column">
        <li><a href="/admin" class="nav-link text-white">- Admin - Home</a></li>
        <li><a href="#" class="nav-link text-white">- Admin - Products</a></li>
        <li>
          <a href="/" class="mt-2 btn bg-primary text-white">Go back to the home page</a>
        </li>
      </ul>
    </div>
    <!-- sidebar -->
    <div class="col content-grey">
      <nav class="p-3 shadow text-end">
        <span class="profile-font">Admin</span>
        <img class="img-profile rounded-circle" src="/img/undraw_profile.svg">
      </nav>

      <div class="g-0 m-5">
        {{> content}}
      </div>
  </div>
</div>

<!-- footer -->
<div class="copyright py-4 text-center text-white">
  <div class="container">
    <small>
      Copyright - <a class="text-reset fw-bold text-decoration-none" target="_blank"
        href="https://twitter.com/danielgarax">
        Daniel Correa
      </a> - <a class="text-reset fw-bold text-decoration-none" target="_blank"
        href="https://twitter.com/greglim81">
        Greg Lim
      </a>
    </small>
  </div>
</div>
```

```
  <!-- footer -->


 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js" crossorigin="anonymous">
 </script>
</body>


</html>
```

Note: Remember you can find the application code at the GitHub repository in https://github.com/PracticalBooks/Practical-Nest.

We have a new layout. This layout presents a different structure and contains a sidebar with three links.

- **"- Admin - Home"** links to the admin panel home page.
- **"- Admin - Products"** links to the admin panel product management page (this page will be implemented in the next chapter).
- **"Go back to the home page"** links back to the online store home page.

The *admin.hbs* layout imports an *admin.css* file and a new image. Let's add these elements to our project.

Creating the admin.css file

In *public/css* create a new file called *admin.css* and fill it with the following code.

Add Entire Code

```
.copyright {
   background-color: #1a252f;
}

.fixed {
   -ms-flex: 0 0 250px;
   flex: 0 0 250px;
}

.content-grey {
   background-color: #f8f9fc;
}

hr {
   margin-top: 0.8em;
   margin-bottom: 0.8em;
}

.img-profile {
   height: 2rem;
   width: 2rem;
}

.profile-font {
   color: #858796 !important;
   font-size: 80%;
   font-weight: 400;
}

.card-header {
   background-color: #f8f9fc;
   border-bottom: 1px solid #e3e6f0;
}
```

We have some custom CSS classes. Most of them are used in the *admin* layout.

Adding the undraw profile image

Download *undraw_profile.svg* file from this link https://github.com/PracticalBooks/Practical-Nest/blob/main/Chapter16/online-store/public/img/undraw_profile.svg and store it inside the *public/img* folder. Or store your own admin profile image.

AdminController

In *src/* , create a subfolder called *admin* . Then, in *src/admin* create a new file *admin.controller.ts* and fill it with the following code.

Add Entire Code

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

```
@Controller('/admin')
export class AdminController {
   @Get('/')
   @Render('admin/index')
   index() {
      const viewData = [];
      viewData['title'] = 'Admin Page - Admin - Online Store';
      return {
         viewData: viewData,
      };
   }
}
```

We have a simple *index* method, linked to the ("/admin/") route that displays the *views/admin/index.hbs* view.

Admin index view

Now, let's create the admin index view. In *views/* , create a subfolder called *admin* . Then, in *views/admin* create a new file called *index.hbs* and fill it with the following code.

Add Entire Code

```
{{#> admin}}
   {{#*inline "content"}}
      <div class="card">
         <div class="card-header">
            Admin Panel - Home Page
         </div>
         <div class="card-body">
            Welcome to the Admin Panel, use the sidebar to navigate between the different options.
         </div>
      </div>
   {{/inline}}
{{/admin}}
```

We have a simple view that displays a "welcome to the admin panel" message. Note that this view uses the new *admin* layout (not the *app* layout).

Admin Module

In *src/admin* , create a new file called *admin.module.ts* and fill it with the following code.

Add Entire Code

```
import { Module } from '@nestjs/common';
import { AdminController } from './admin.controller';


@Module({
   controllers: [AdminController],
})
export class AdminModule {}
```

We created the *AdminModule* and register *AdminController* .

Registering AdminModule in AppModule

Let's register the *AdminModule* in the application root module (*AppModule*). In *src/app.module.ts* , make the following changes in **bold**.

Modify Bold Code

```
…
import { ProductsService } from './models/products.service';
import { Product } from './models/product.entity';
import { AdminModule } from './admin/admin.module';


@Module({
   imports: [
      TypeOrmModule.forRoot(),
      TypeOrmModule.forFeature([Product]),
      AdminModule,
   ],
   controllers: [AppController, ProductsController],
   providers: [ProductsService],
})
export class AppModule {}
```

We import *AdminModule* and register it in the module *imports* property. This makes the route defined in *AdminController* accessible.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now go to the (“/admin”) route, and you will see the new Admin Panel (see Fig. 16-1).

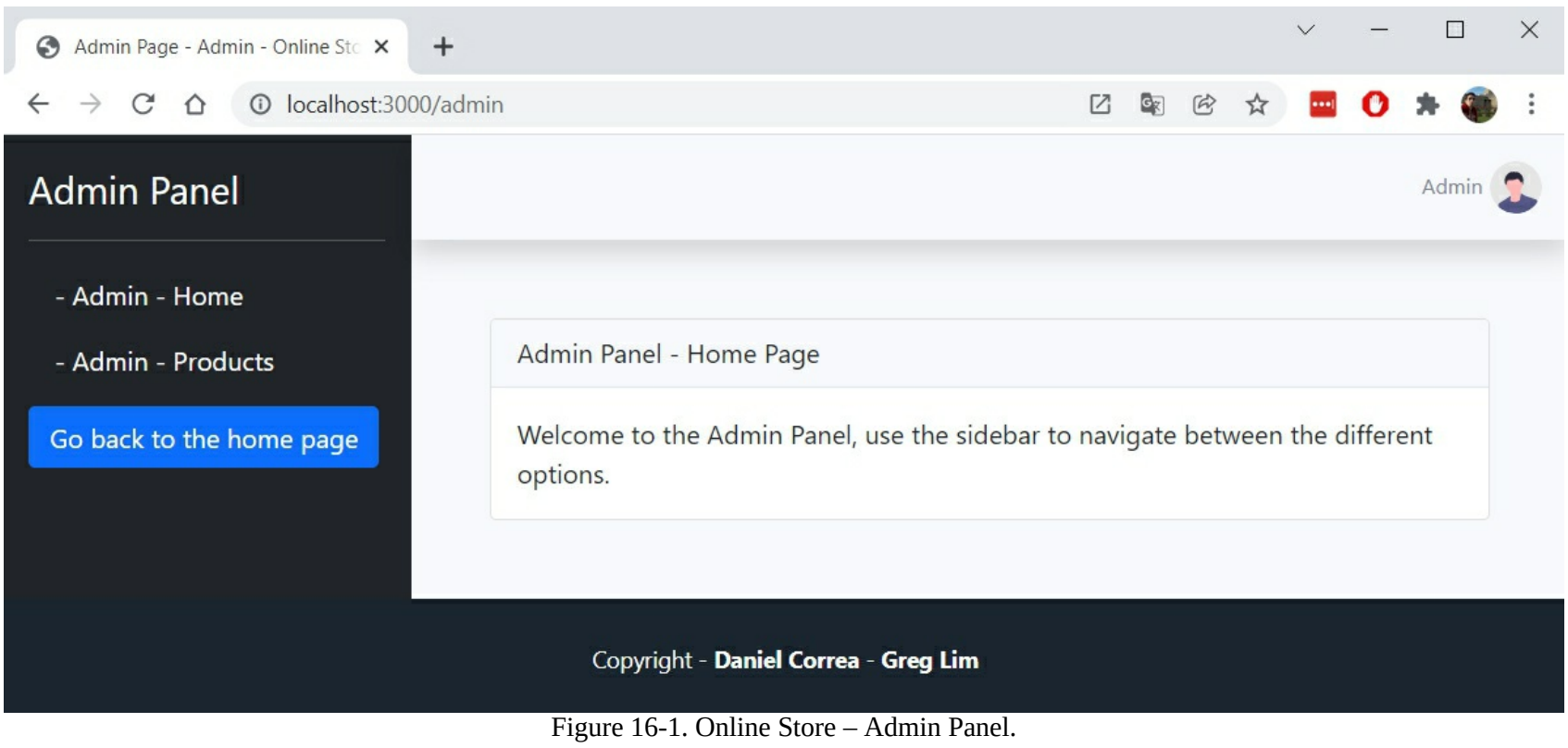

Figure 16-1. Online Store – Admin Panel.

Chapter 17 – List Products in Admin Panel

Let’s create the admin product management section. For now, we will create an index page that lists all products.

AdminProductsController

In *src/admin* , create a new file called *admin.products.controller.ts* and fill it with the following code.

Add Entire Code

```
import { Controller, Get, Render } from '@nestjs/common';
import { ProductsService } from '../models/products.service';


@Controller('/admin/products')
export class AdminProductsController {
   constructor(private readonly productsService: ProductsService) {}


   @Get('/')
   @Render('admin/products/index')
   async index() {
      const viewData = [];
      viewData['title'] = 'Admin Page - Admin - Online Store';
      viewData['products'] = await this.productsService.findAll();
      return {
         viewData: viewData,
      };
   }
}
```

We create the *AdminProductsController* class. We imported the *ProductsService* which will be injected in the *productsService* attribute. Then, we define the *index* method that collects the products data (though the *productsService.findAll* method) and renders the *admin.products.index* view.

Admin products index view

In *views/admin* , create a subfolder called *products* . In *views/admin/products* create a new file *index.hbs* and fill it with the following code.

Add Entire Code

```
{{#> admin}}
   {{#*inline "content"}}
      <div class="card">
         <div class="card-header">
            Manage Products
         </div>
         <div class="card-body">
            <table class="table table-bordered table-striped">
               <thead>
                     <tr>
                        <th scope="col">ID</th>
                        <th scope="col">Name</th>
                        <th scope="col">Edit</th>
                        <th scope="col">Delete</th>
                     </tr>
               </thead>
               <tbody>
                     {{#each viewData.products}}
                     <tr>
                        <td>{{getId}}</td>
                        <td>{{getName}}</td>
                        <td>Edit</td>
                        <td>Delete</td>
                     </tr>
                     {{/each}}
               </tbody>
            </table>
         </div>
      </div>
   {{/inline}}
{{/admin}}
```

We have a table which displays the products’ ids and names. This table was designed based on Bootstrap Tables (https://getbootstrap.com/docs/5.1/content/tables/). Later, we will include a link to edit and delete specific products.

Registering AdminProductsController

In *src/admin/admin.module.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Module } from '@nestjs/common';
import { AdminController } from './admin.controller';
import { AdminProductsController } from './admin.products.controller';


@Module({
  controllers: [AdminController, AdminProductsController],
})
export class AdminModule {}
```

Running the app

Let’s try to run the app. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

You will get an error saying that “Nest can't resolve dependencies of the AdminProductsController”. Basically, there is a problem trying to inject the *ProductsService* in the *AdminProductsController* constructor. It is because we have not registered the *ProductsService* as a provider in *AdminModule* . Because we registered it only in *AppModule* , *ProductsService* is available to be injected only under the *AppModule* scope, not the *AdminModule* scope.

Here we have two options (i) import the *ProductsService* and register it as a provider in the *AdminModule* file. But we will also need to import and register the *TypeOrmModule* (because remember that the *ProductsService* injects a TypeORM repository). Or (ii) make the *AppModule* a global module (to avoid importing or registering the same providers over multiple modules).

Let’s implement the second option.

Global AppModule

In *src/app.module.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Module, Global } from '@nestjs/common';
…


@Global()
@Module({
  imports: [
    TypeOrmModule.forRoot(),
    TypeOrmModule.forFeature([Product]),
    AdminModule,
  ],
  controllers: [AppController, ProductsController],
  providers: [ProductsService],
  exports: [ProductsService],
})
export class AppModule {}
```

We imported the *Global* decorator and marked *AppModule* with it. The *@Global* decorator makes the *AppModule* global-scoped, making the *AppModule* providers available everywhere (not only attached to the *AppModule* scope). We use the *exports* property to specify which providers will be available in other modules. In this case, the *ProductsService* provider will be available in all modules since we used the *@Global* decorator.

Now, Nest can inject the *ProductsService* in the *AdminProductsController* constructor (without requiring us to import or register it in the *AdminModule*).

Updating links in Admin layout

Now that we have the proper admin products route, let’s include it in the *admin* layout. In *views/layouts/admin.hbs* , make the following changes in **bold**.

Modify Bold Code

```
...
    <!-- sidebar -->
      ...
      <ul class="nav flex-column">
        <li><a href="/admin" class="nav-link text-white">- Admin - Home</a></li>
        <li><a href="/admin/products" class="nav-link text-white">- Admin - Products</a></li>
        <li>
```

```
            <a href="/" class="mt-2 btn bg-primary text-white">Go back to the home page</a>
          </li>
        </ul>
      </div>
      <!-- sidebar -->
...
```

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now go to the ("/admin/products") route, and you will see the new Admin Product page (see Fig. 17-1). Remember that you will need to have MySQL running; otherwise, you will get a database connection error.

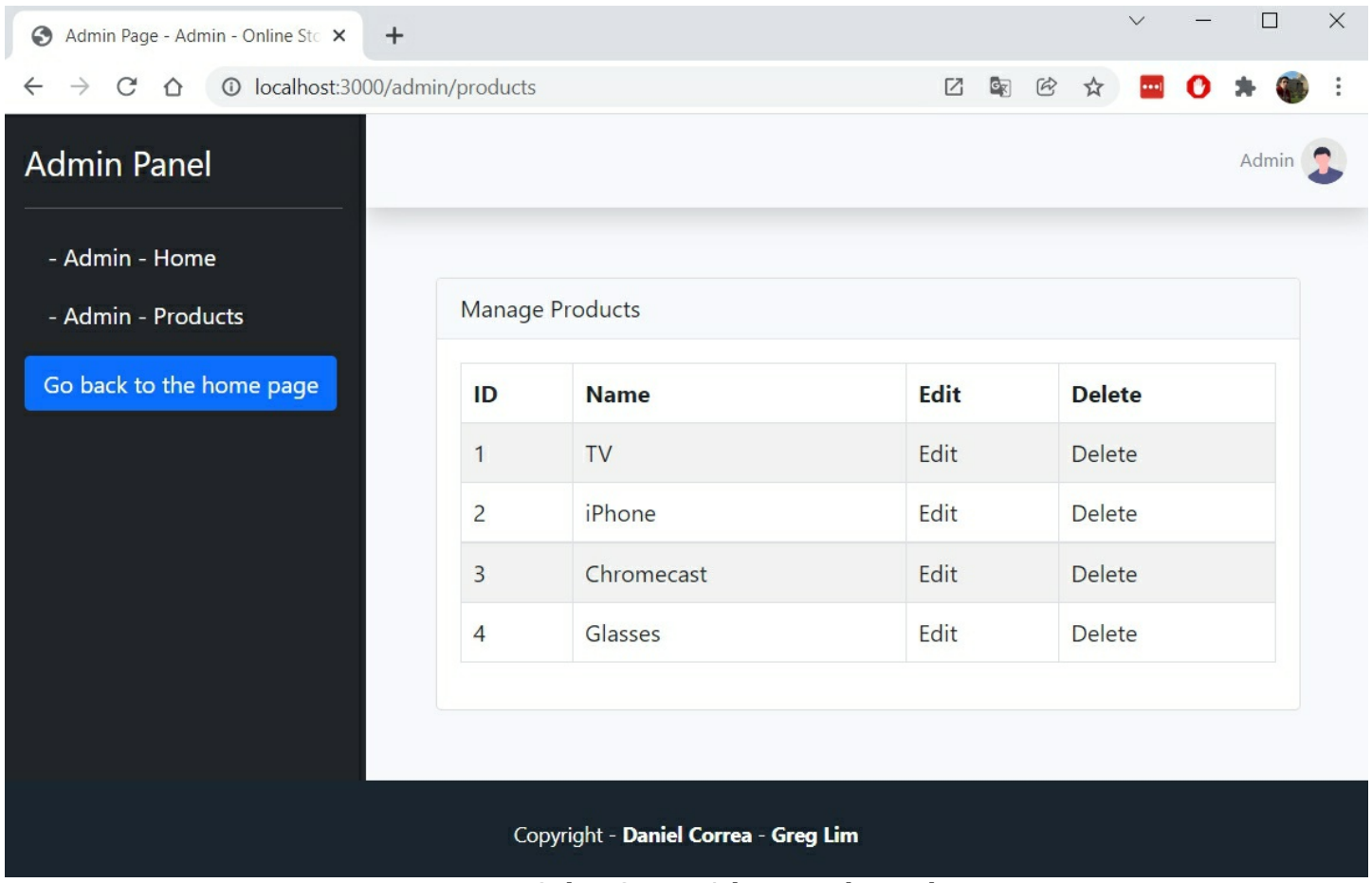

Figure 17-1. Online Store – Admin Panel – Products.

Chapter 18 – Create Products

Now, we will focus on the admin panel system to create products.

Modifying admin/products/index.hbs view

In *views/admin/products/index.hbs* , make the following changes in **bold**.

Modify Bold Code

```
{{#> admin}}
  {{#*inline "content"}}
    <div class="card mb-4">
      <div class="card-header">
        Create Products
      </div>
      <div class="card-body">
        <form method="POST" action="/admin/products/store">
          <div class="row">
                <div class="col">
                    <div class="mb-3 row">
                        <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Name:</label>
                        <div class="col-lg-10 col-md-6 col-sm-12">
                            <input name="name" type="text" class="form-control" required>
                        </div>
                    </div>
                </div>
                <div class="col">
                    <div class="mb-3 row">
                        <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Price:</label>
                        <div class="col-lg-10 col-md-6 col-sm-12">
                            <input name="price" type="number" class="form-control" required>
                        </div>
                    </div>
                </div>
          </div>
          <div class="mb-3">
                <label class="form-label">Description</label>
                <textarea class="form-control" name="description" rows="3" required></textarea>
          </div>
          <button type="submit" class="btn btn-primary">Submit</button>
        </form>
      </div>
    </div>
    <div class="card">
      <div class="card-header">
        Manage Products
...
```

Let’s analyze the code by parts.

Analyze Code

```
<form method="POST" action="/admin/products/store">
    …
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

We have an HTML *form* . This *form* specifies a POST method and links the form with the (“/admin/products/store”) route. This route will be attached to a controller method later. The POST method is used to send data to the application server. Finally, we have the *submit* button to submit the form.

Analyze Code

```
<div class="mb-3 row">
  <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Name:</label>
  <div class="col-lg-10 col-md-6 col-sm-12">
        <input name="name" type="text" class="form-control" required>
  </div>
</div>
```

The rest of the code shows form inputs, buttons, and a textarea. This form and elements were designed based on Bootstrap Forms (https://getbootstrap.com/docs/5.1/forms/overview/). They show an input to enter the product name.

Note: we are not yet collecting the product image. We will do so in the next chapter.

WARNING: Our forms are not protected against Cross-site request forgery (CSRF) attacks. To protect against CSRF, we should use a library to generate CSRF tokens and use those tokens in our

forms. Then, the application can validate that the form request comes from our trusted site. Nest suggests using the *csurf* library. However, the implementation is out of the scope of this book. More information can be found here: https://docs.nestjs.com/security/csrf.

Modifying AdminProductsController

Next, we create the *store* method in *AdminProductsController* . In *src/admin/admin.products.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Controller, Get, Render, Post, Body, Redirect } from '@nestjs/common';
import { ProductsService } from '../models/products.service';
import { Product } from '../models/product.entity';


@Controller('/admin/products')
export class AdminProductsController {
  ...


  @Post('/store')
  @Redirect('/admin/products')
  async store(@Body() body) {
    const newProduct = new Product();
    newProduct.setName(body.name);
    newProduct.setDescription(body.description);
    newProduct.setPrice(body.price);
    newProduct.setImage('game.png');
    await this.productsService.createOrUpdate(newProduct);
  }
}
```

Let’s analyze the code by parts.

Analyze Code

```
  @Post('/store')
  @Redirect('/admin/products')
  async store(@Body() body) {
```

We use the *@Post* decorator in our *store* method to receive form data. This controller method won’t render a view. Instead, it redirects to the (“/admin/products”) route using the *@Redirect* decorator. Finally, the *store* method receives the data from the *admin/products/index.hbs* form. To extract this information, we use the *@Body* decorator. The *@Body* decorator maps the inputs received from the form into the *body* variable.

Analyze Code

```
    const newProduct = new Product();
    newProduct.setName(body.name);
    newProduct.setDescription(body.description);
    newProduct.setPrice(body.price);
    newProduct.setImage('game.png');
    await this.productsService.createOrUpdate(newProduct);
```

We create a new *Product* instance. Then, we set the product *name* , *description* , and *price* based on the values collected from the form’s *body* variable. We currently set a default product image (*game.png*). Finally, we invoke the *productsService.createOrUpdate* method and pass in *newProduct* to create the new product in the database (or update it if it exists). We will implement the *createOrUpdate* method in the next section.

Modifying ProductsService

In *src/models/products.service.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
@Injectable()
export class ProductsService {
  ...


  findOne(id: string): Promise<Product> {
    return this.productsRepository.findOne(id);
  }


  createOrUpdate(product: Product): Promise<Product> {
    return this.productsRepository.save(product);
  }
}
```

The *createOrUpdate* method receives a *Product* instance and returns a Promise consisting of a product. This

method uses the *productsRepository* attribute which invokes its *save* method (inherited from the TypeORM *Repository* class). This saves the product in the database (creates a new one if it doesn't exist or updates an existing one) and returns a new object based on the saved product.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now go to the ("/admin/products") route, and you will see the new form. When you complete and submit the form (see Fig. 18-1), you will see the new product listed in the admin panel (see Fig. 18-2).

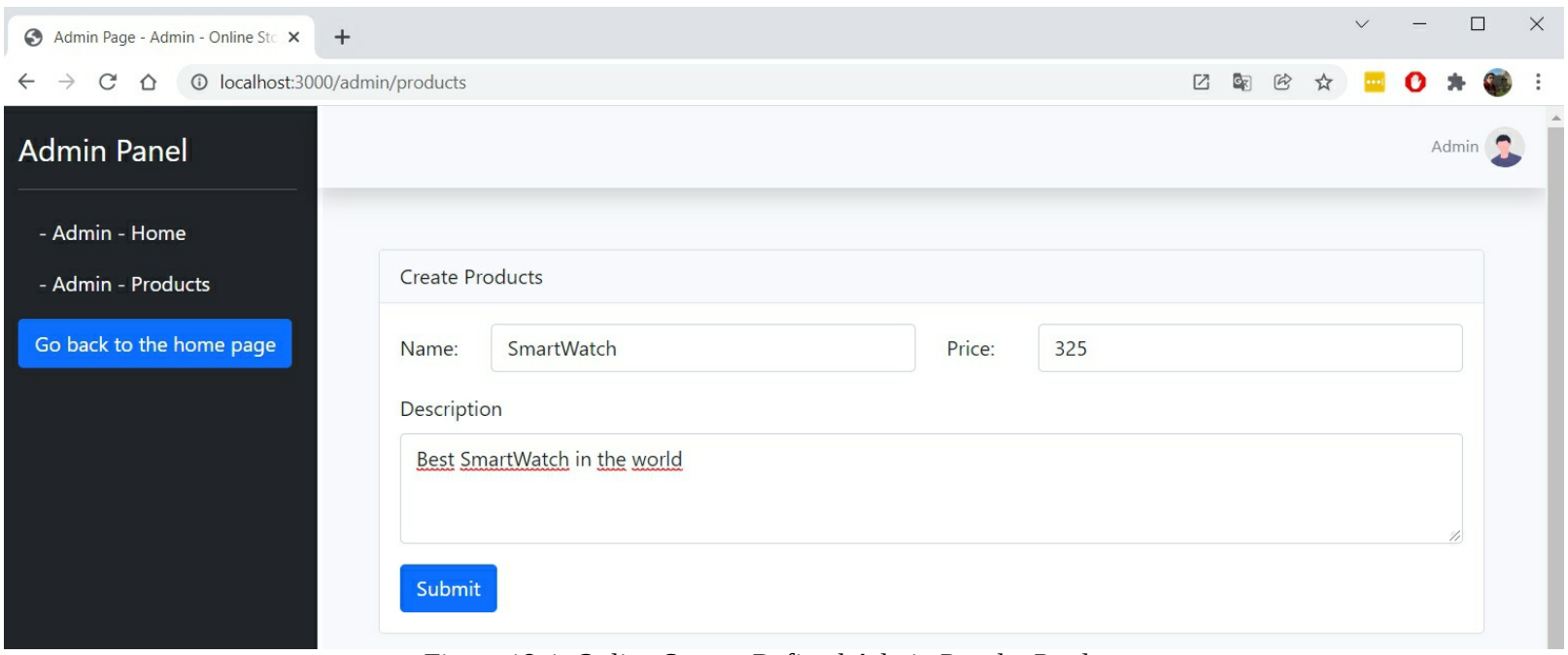

Figure 18-1. Online Store – Refined Admin Panel – Products.

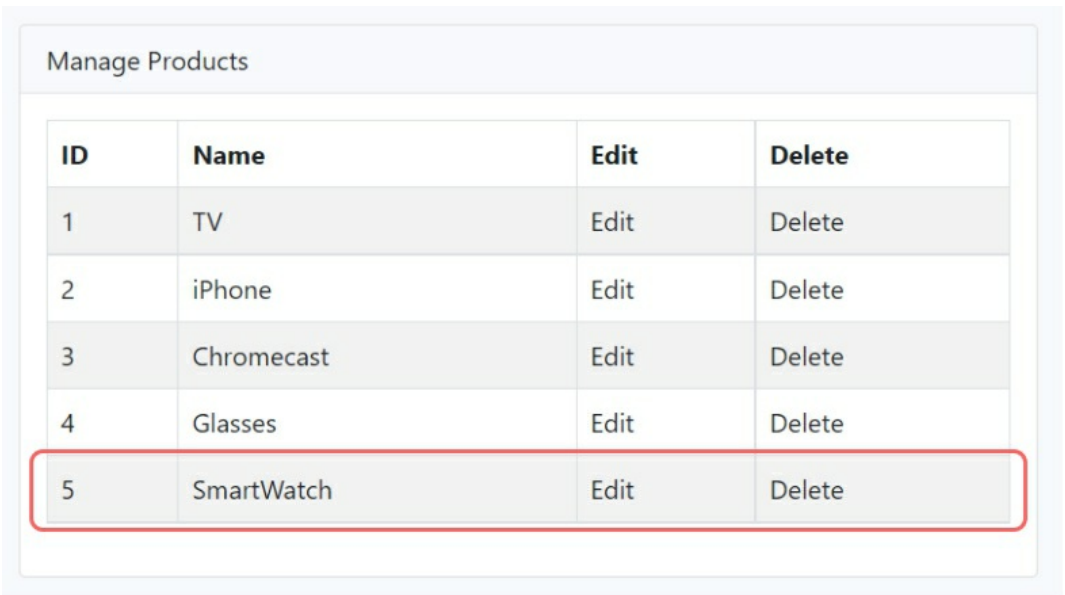

Figure 18-2. New product listed.

We have a couple of problems with the current implementation. We are not collecting the product images from the form, and we are not validating the product data. For example, we can create products with negative prices (i.e., -180). We will implement the product image in the next chapter. However, we will need to wait some chapters to introduce the concepts of sessions and apply the proper validations.

Chapter 19 – Create Products with Images

In the previous chapter, we created all our products with a default *game.png* image. Let's fix this and enable uploading images.

Installing Image Upload Support Library

To handle file uploading, Nest provides a built-in module based on the multer middleware package for Express. Multer handles data posted in the multipart/form-data format primarily used for uploading files via an HTTP POST request. For better type safety, Nest suggests installing a Multer typings package. Let's install it. In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm install -D @types/multer
```

Modifying admin/products/index view

In *views/admin/products/index.hbs* , make the following changes in **bold**.

Modify Bold Code

```
@extends('layouts.admin')
@section('title', $viewData["title"])
@section('content')
...
    <form method="POST" action="/admin/products/store" enctype="multipart/form-data">
      <div class="row">
      ...
      </div>
      <div class="row">
        <div class="col">
          <div class="mb-3 row">
                <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Image:</label>
                <div class="col-lg-10 col-md-6 col-sm-12">
                    <input class="form-control" type="file" name="image">
                </div>
          </div>
        </div>
        <div class="col">
           
        </div>
      </div>
      <div class="mb-3">
        <label class="form-label">Description</label>
...
```

We include an *enctype="multipart/form-data"* attribute in our form. This attribute is used in form elements that have a file upload. Then, we add a file upload input (called *image*) where the admin user will select the product image.

Modifying AdminProductsController

In *src/admin/admin.products.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Controller, Get, Render, Post, Body, Redirect,
  UseInterceptors, UploadedFile } from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';
import { ProductsService } from '../models/products.service';
import { Product } from '../models/product.entity';


@Controller('/admin/products')
export class AdminProductsController {
  …


  @Post('/store')
  @UseInterceptors(FileInterceptor('image', { dest: './public/uploads' }))
  @Redirect('/admin/products')
  async store(@Body() body, @UploadedFile() file: Express.Multer.File) {
    const newProduct = new Product();
    newProduct.setName(body.name);
    newProduct.setDescription(body.description);
    newProduct.setPrice(body.price);
    newProduct.setImage(file.filename);
    await this.productsService.createOrUpdate(newProduct);
  }
```

```
}
```

We import the *UseInterceptors* , *UploadedFile* , and *FileInterceptor* decorators. To upload our image file, we need to use the *@UseInterceptors* decorator tied to a specific route (in this case "/admin/products/store"). Then, we place the *FileInterceptor* inside the *@UseInterceptors* decorator. This *FileInterceptor* will be tied to the ("/admin/products/store") route, extract the file from the request, upload the file to the *public/uploads* folder, and place the uploaded file information in the *file* variable using the *@UploadedFile* decorator.

The *FileInterceptor()* decorator takes two arguments:

- **fieldName:** name of the field from the HTML form that holds a file. In our case, remember we create an input file with the *image* name.
- **options:** optional object of type MulterOptions. In our case, we pass the destination where we want to store the image (the *public/uploads* folder). More information about multer options can be found here: https://github.com/expressjs/multer#multeropts.

Finally, we modify the *newProduct.setImage* argument. We remove the previous *game.png* default filename and pass the filename of the file uploaded by the multer library.

WARNING: The current code allows us to upload any kind of file type. This could lead to hackers injecting malicious code. We will refactor this code later to implement proper file type validations.

Modifying products/index and products/show views

In *views/products/index.hbs* , make the following changes in **bold**.

Modify Bold Code

```
...
        {{#each viewData.products}}
        …
            <img src="/uploads/{{getImage}}" class="card-img-top img-card">
        …
        {{/each}}
...
```

In *views/products/show.hbs* , make the following changes in **bold**.

Modify Bold Code

```
...
            <div class="col-md-4">
               <img src="/uploads/{{getImage}}" class="img-fluid rounded-start">
            </div>
...
```

Now, we are accessing the product images through the *uploads* folder path.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now, go to the ("/admin/products") route, and you will see the new form (see Fig. 19-1). Complete the form, upload an image, and create a new product.

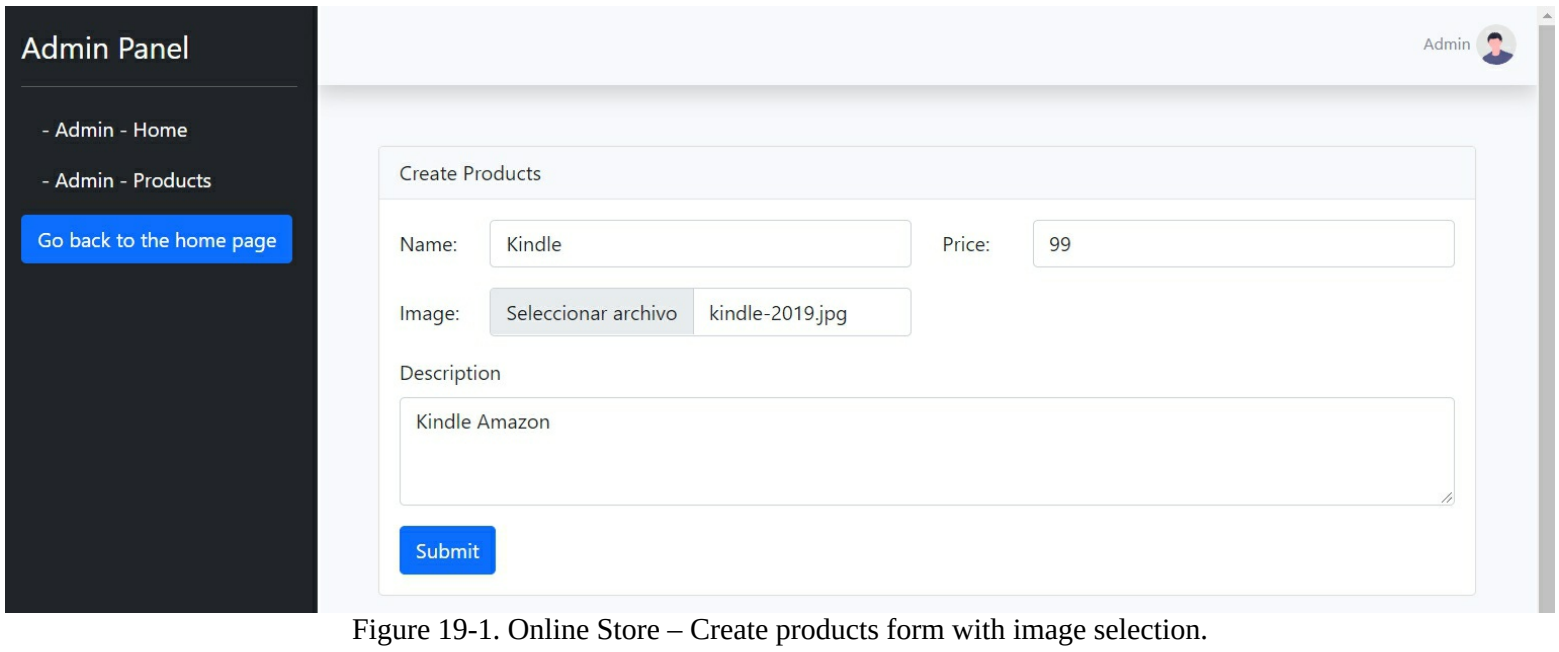

Figure 19-1. Online Store – Create products form with image selection.

Now, go to the (“/products”) route. You will see the new product image loaded. However, the old product images are not loading (see Fig. 19-2). This is because we changed the path from where we are loading the images. Don’t worry. In the next chapter, we will implement a way to edit our products and upload proper images for the rest of our products.

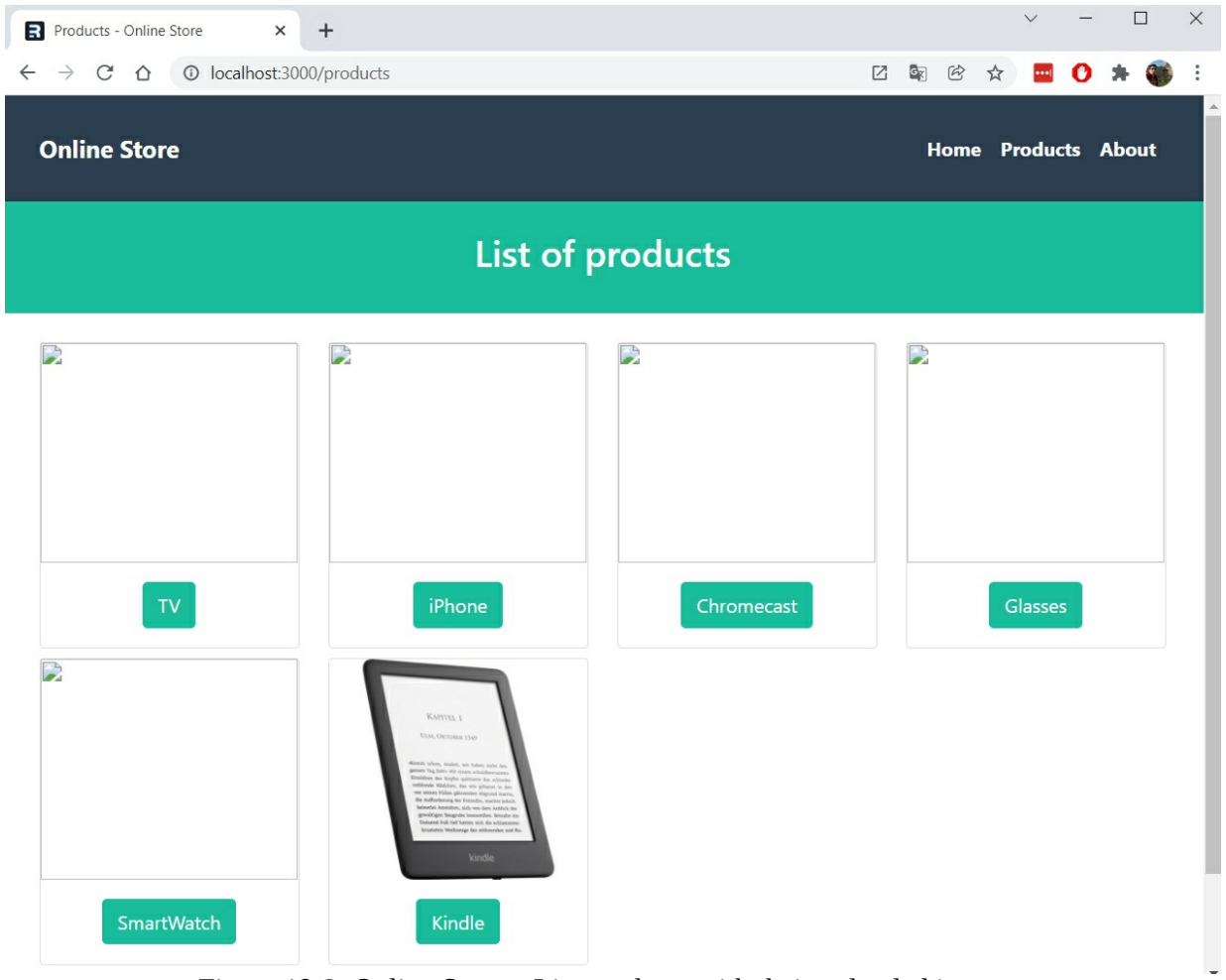

Figure 19-2. Online Store – List products with their uploaded images.

Quick discussion: There is a tremendous advantage of using the multer library for uploading product images. Before, we manually downloaded and added the products’ images to the *public/img* folder. This is a manual task which does not scale well. It requires access to the application code (but not all admins will have it). Now, we automate this process with the product registration form and the use of the multer library. Multer interacts with our server filesystem and places the images in their proper folder location.

Note: if you try to create a new product without selecting an image, you will get a Nest 500 error. We will implement product form validation and a proper way to display the errors in upcoming chapters.

Chapter 20 – Edit and Delete Products

Let’s start with deleting products since it is relatively simple. And then, we will implement the edit products functionality.

Deleting products

To delete a product, we will need to modify a set of files.

Modifying ProductsService

In *src/models/products.service.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
@Injectable()
export class ProductsService {
  ...

  async remove(id: string): Promise<void> {
    await this.productsRepository.delete(id);
  }
}
```

The *remove* method receives the *id* of a product and returns a *void* Promise. *remove* uses the *productsRepository* attribute, invokes its *delete* method (inherited from the TypeORM *Repository* class), and deletes the corresponding product from the database.

Modifying AdminProductsController

In *src/admin/admin.products.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Controller, Get, Render, Post, Body, Redirect,
  UseInterceptors, UploadedFile, Param } from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';
...
@Controller('/admin/products')
export class AdminProductsController {
  ...

  @Post('/:id')
  @Redirect('/admin/products')
  remove(@Param('id') id: string) {
    return this.productsService.remove(id);
  }
}
```

Like the *store* method, the *remove* method won’t render a view. Instead, it redirects to the (“/admin/products”) route using the *@Redirect* decorator. The *remove* method contains an *id* argument. We use the *@Param('id')* to decorate the *id* argument, collect an *id* value from the request route and assign it to the *id* argument. Finally, we invoke *productsService.remove* with *id* to delete the product.

Modifying view/layout/admin

In *views/layout/admin.hbs* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
<html lang="en">


<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.min.css" rel="stylesheet"
    crossorigin="anonymous" />
 <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.5.0/font/bootstrap-icons.css">
...
```

We add a link to a CSS file (Bootstrap icons). This CSS file allows us to use specific icons and fonts inside our views. For example, we will have icons to edit and delete our products. More information about Bootstrap icons can be found at: https://icons.getbootstrap.com/.

Modifying admin/products/index view

In *views/admin/products/index.hbs* , make the following changes in **bold**.

Modify Bold Code

```
...
        {{#each viewData.products}}
        <tr>
          <td>{{getId}}</td>
          <td>{{getName}}</td>
          <td>~~**Edit**~~
                **<button class="btn btn-primary">**
                  **<i class="bi-pencil"></i>**
                **</button>**
          </td>
      <td>~~**Delete**~~
                **<form action="/admin/products/{{getId}}" method="POST">**
                  **<button class="btn btn-danger">**
                    **<i class="bi-trash"></i>**
                  **</button>**
                **</form>**
          </td>
        </tr>
        {{/each}}
...
```

We removed the “Edit” and “Delete” text and replaced them with a couple of buttons. Inside those buttons, we use Bootstrap icons (a *pencil* icon to edit a product and a *trash* icon to delete a product). We completed the delete functionality by wrapping a form around the delete icon. This form will invoke the (“/admin/products/:id”) route and pass the current product *id* .

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now go to the (“/admin/products”) route, and you will see the new form (see Fig. 20-1). Now, try to create a dummy product and delete it.

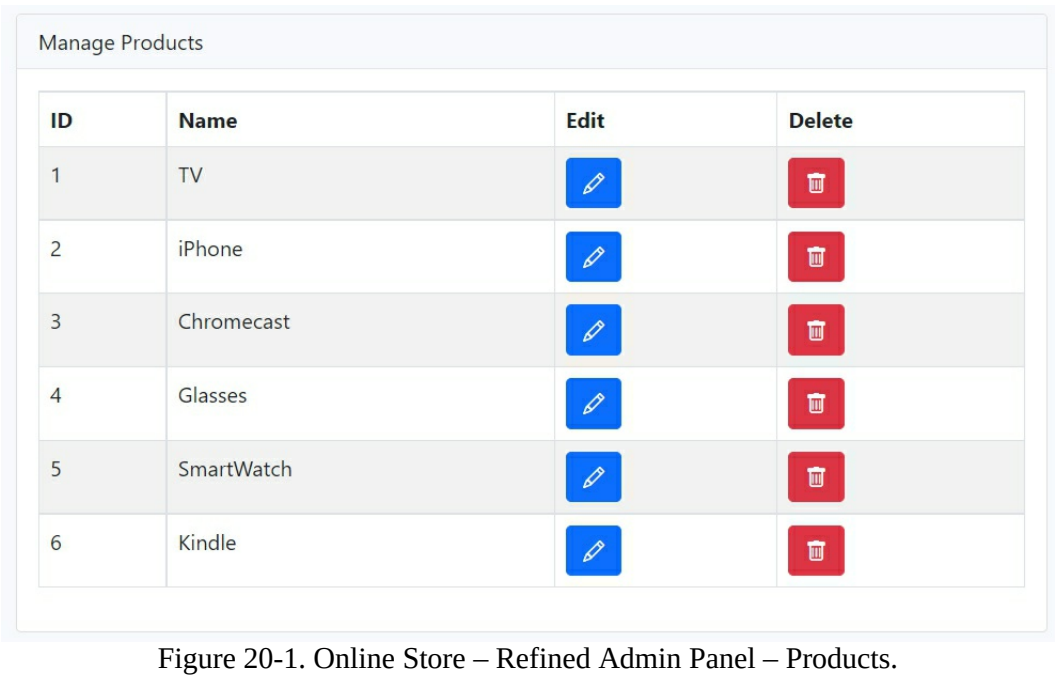

Figure 20-1. Online Store – Refined Admin Panel – Products.

Editing products

Like the previous delete functionality, we will need to modify a set of files.

Modifying AdminProductsController

In *src/admin/admin.products.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
@Controller('/admin/products')
export class AdminProductsController {
  ...

  **@Get('/:id')**
  **@Render('admin/products/edit')**
```

```
async edit(@Param('id') id: string) {
   const viewData = [];
   viewData['title'] = 'Admin Page - Edit Product - Online Store';
   viewData['product'] = await this.productsService.findOne(id);
   return {
      viewData: viewData,
   };
}

@Post('/:id/update')
@UseInterceptors(FileInterceptor('image', { dest: './public/uploads' }))
@Redirect('/admin/products')
async update(
  @Body() body,
   @UploadedFile() file: Express.Multer.File,
   @Param('id') id: string
) {
   const product = await this.productsService.findOne(id);
   product.setName(body.name);
   product.setDescription(body.description);
   product.setPrice(body.price);
   if (file) {
      product.setImage(file.filename);
   }
   await this.productsService.createOrUpdate(product);
}
```

Let’s analyze the code by parts.

Analyze Code

```
@Get('/:id')
@Render('admin/products/edit')
async edit(@Param('id') id: string) {
   const viewData = [];
   viewData['title'] = 'Admin Page - Edit Product - Online Store';
   viewData['product'] = await this.productsService.findOne(id);
   return {
      viewData: viewData,
   };
}
```

We have an *edit* method that searches for a product based on its *id* (received from the route) and sends it to the *admin/products/edit* view. This is the product we are going to edit.

Analyze Code

```
@Post('/:id/update')
@UseInterceptors(FileInterceptor('image', { dest: './public/uploads' }))
@Redirect('/admin/products')
async update(
  @Body() body,
   @UploadedFile() file: Express.Multer.File,
   @Param('id') id: string
) {
   const product = await this.productsService.findOne(id);
   product.setName(body.name);
   product.setDescription(body.description);
   product.setPrice(body.price);
   if (file) {
      product.setImage(file.filename);
   }
   await this.productsService.createOrUpdate(product);
}
```

Then, we have the *update* method. It is like the *store* method.

1. We collect the *id* from the route parameter, which indicates the product to be updated.
2. We search for a product based on that *id* , and to that product we set the new *name* , *price* , and *description* . That data is collected in a form that we will show later.
3. We set the new product *image* value if a new *image* was uploaded.
4. Finally, we invoke the *productsService.createOrUpdate* method with the updated product to update it in the database.

Modifying admin/products/index view

In *views/admin/products/index.hbs* , make the following changes in **bold**.

Modify Bold Code

```
...
        {{#each viewData.products}}
        <tr>
          <td>{{getId}}</td>
          <td>{{getName}}</td>
          <td>
                  <a class="btn btn-primary" href="/admin/products/{{getId}}">
                  <button class="btn btn-primary">
                    <i class="bi-pencil"></i>
                  </button>
                  </a>
          </td>
      …
        </tr>
        {{/each}}
...
```

Now, we link the pencil icon with the product edit route and pass in the current product *id* .

Creating admin/products/edit view

In *views/admin/products/* create a new file called *edit.hbs* and fill it with the following code.

Add Entire Code

```
{{#> admin}}
  {{#*inline "content"}}
    <div class="card mb-4">
      <div class="card-header">
        Edit Product
      </div>
      <div class="card-body">
        {{#with viewData.product}}
        <form method="POST" action="/admin/products/{{getId}}/update" enctype="multipart/form-data">
          <div class="row">
                <div class="col">
                  <div class="mb-3 row">
                    <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Name:</label>
                    <div class="col-lg-10 col-md-6 col-sm-12">
                      <input name="name" value="{{getName}}" type="text" class="form-control" required>
                    </div>
                  </div>
                </div>
                <div class="col">
                  <div class="mb-3 row">
                    <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Price:</label>
                    <div class="col-lg-10 col-md-6 col-sm-12">
                      <input name="price" value="{{getPrice}}" type="number" class="form-control" required>
                    </div>
                  </div>
                </div>
          </div>
          <div class="row">
                <div class="col">
                  <div class="mb-3 row">
                    <label class="col-lg-2 col-md-6 col-sm-12 col-form-label">Image:</label>
                    <div class="col-lg-10 col-md-6 col-sm-12">
                      <input class="form-control" type="file" name="image">
                    </div>
                  </div>
                </div>
                <div class="col">
                   
                </div>
          </div>
          <div class="mb-3">
                <label class="form-label">Description</label>
       <textarea class="form-control" name="description" rows="3" required>{{getDescription}}</textarea>
          </div>
          <button type="submit" class="btn btn-primary">Edit</button>
        </form>
        {{/with}}
      </div>
    </div>
  {{/inline}}
{{/admin}}
```

The previous view is like the form created in the *admin/products/index* view but with minor differences. We pass the product *id* to the route and populate the input values with the product attributes.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now go to the ("/admin/products") route, click an edit button of a specific product, and you will see the edit form (see Fig. 20-2). Now, you can edit products. We suggest editing all products to replace the missing images (see Fig. 20-3).

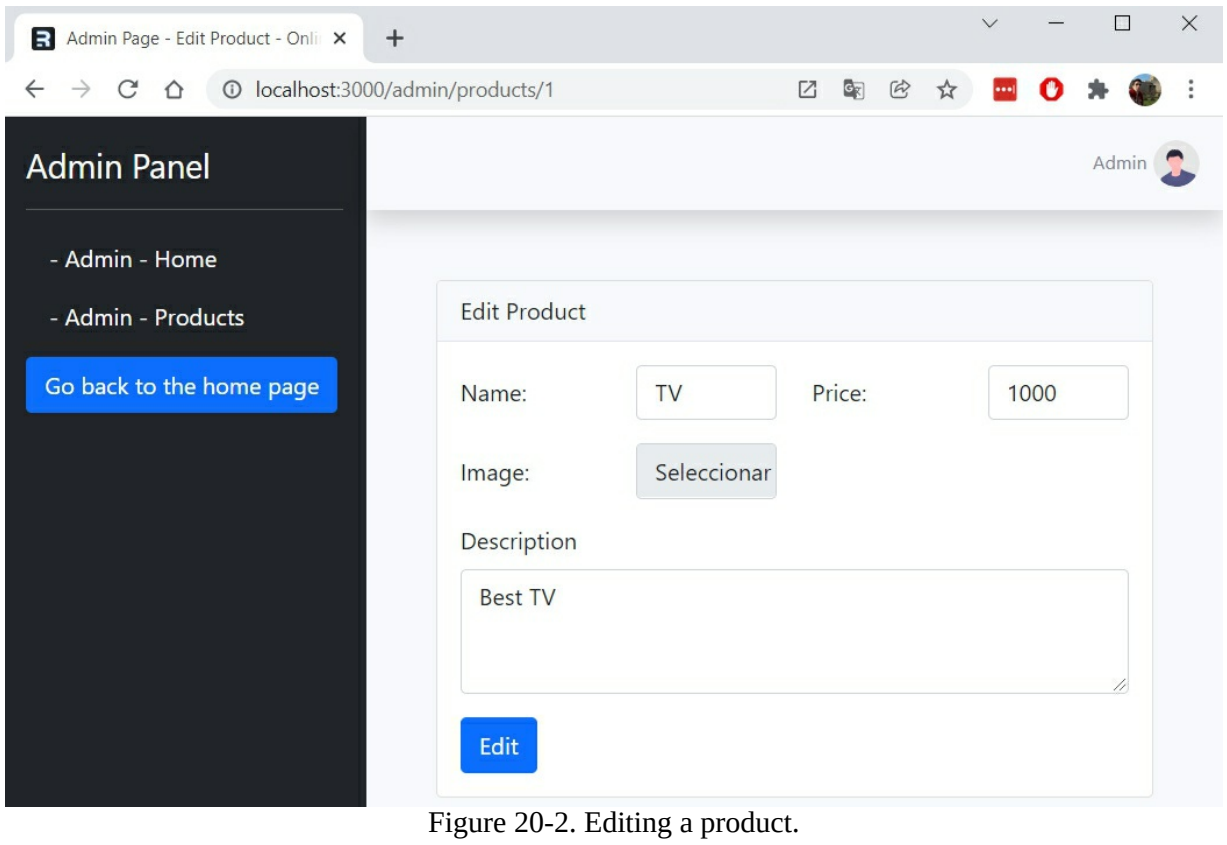

Figure 20-2. Editing a product.

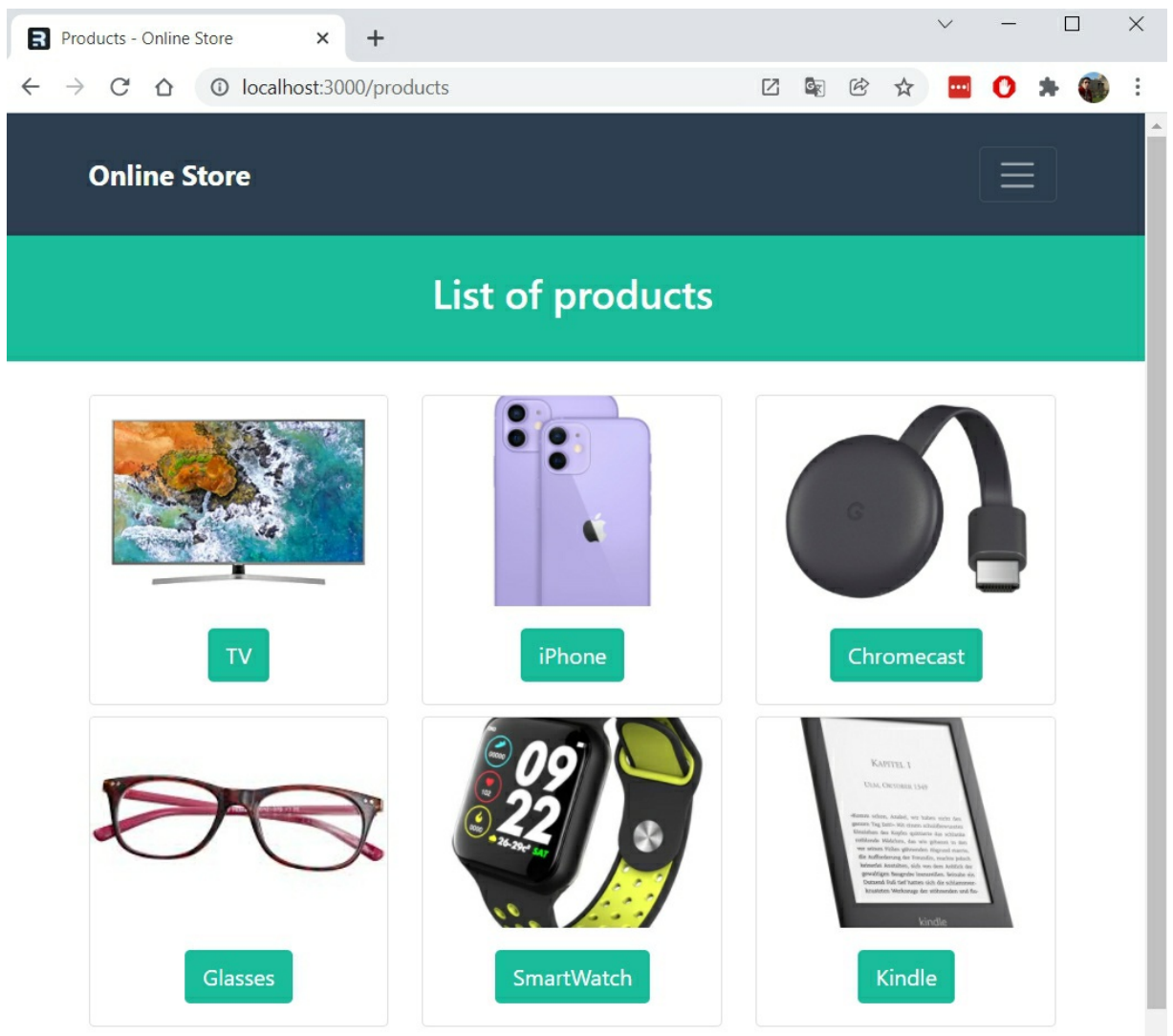

Figure 20-3. List products with proper images.

Chapter 21 – Create Users

In this chapter, we will create users. We will split it into three parts: (i) User Entity, (ii) Users Service, and (iii) Auth Module.

Let’s start by implementing the User entity.

User Entity

Creating User Entity

In *src/models* , create a new file called *user.entity.ts* , and fill it with the following code.

Add Entire Code

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


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


@Column()
name: string;


@Column({ unique: true })
email: string;


@Column()
password: string;


@Column()
role: string;


@Column()
balance: number;


getId(): number {
   return this.id;
}


setId(id: number) {
   this.id = id;
}


getName(): string {
   return this.name;
}


setName(name: string) {
   this.name = name;
}


getEmail(): string {
   return this.email;
}


setEmail(email: string) {
   this.email = email;
}


getPassword(): string {
   return this.password;
}


setPassword(password: string) {
   this.password = password;
}


getRole(): string {
   return this.role;
}
```

```
  setRole(role: string) {
    this.role = role;
  }

  getBalance(): number {
    return this.balance;
  }

  setBalance(balance: number) {
    this.balance = balance;
  }
}
```

We have defined a *User* entity (which becomes a *user* table) with six attributes (*id* , *name* , *email* , *password* , *role* , and *balance*). For *email* attribute, we applied the *@Column* decorator with the *unique* option set to *true* because user emails must be unique. We have two different roles *client* and *admin* (we will manage roles in an upcoming chapter). The *balance* attribute represents the amount of money that the user has in the application (we will work with this later). Finally, we have the attributes getters and setters.

Synchronizing the database

Let's synchronize our database. **First, stop the server.** Then, In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now, you can see the *user* table in your *online_store* database (see Fig. 21-1).

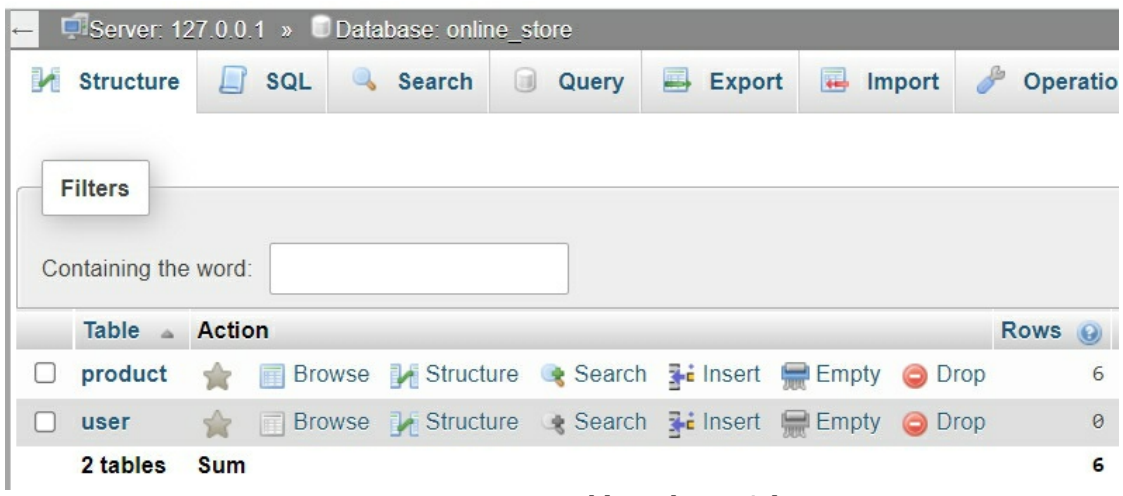

Figure 21-1. New user table – phpMyAdmin.

Users Service

Installing bcrypt

Users will have passwords. We can't store the passwords as plain text in our database as that is a critical risk. Hackers could steal our database information and have full access to users' passwords. In our case, we will use a library called bcrypt which allows us to hash our passwords. More information about bcrypt can be found here: https://www.npmjs.com/package/bcrypt.

Hashing is the process of converting a given key into another value. A hash function is used to generate the new value according to a mathematical algorithm. Once hashing has been applied to a password, it should be reverse-engineer the hashed value to the original text.

Let's install bcrypt in our Online Store project. Go to the project directory, and in the Terminal, execute the following commands.

Execute in Terminal

```
npm install bcrypt
npm install -D @types/bcrypt
```

Creating UsersService

In *src/models* , create a new file called *users.service.ts* , and fill it with the following code.

Add Entire Code

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './user.entity';
import * as bcrypt from 'bcrypt';
```

```
@Injectable()
export class UsersService {
   constructor(
      @InjectRepository(User)
      private usersRepository: Repository<User>,
   ) {}


   async createOrUpdate(user: User): Promise<User> {
      const hash = await bcrypt.hash(user.getPassword(), 10);
      user.setPassword(hash);
      return this.usersRepository.save(user);
   }
}
```

We define the *UsersService* class, which injects a TypeORM *User Repository* instance in the *usersRepository* attribute to access the database. Then, we define the *createOrUpdate* method. This method receives a *User* instance, collects the user *password* , and uses *bcrypt.hash* to generate a hash (based on the user *password* and *10* salt rounds). Then we set the new password (the hashed version), and invoke *this.usersRepository.save* to create the user in the database.

Auth Module

The *AuthModule* will manage all the actions related to authentication, such as login, registration, and logout. In this chapter, we implement user registration which requires us to define a new module, a controller, and a view. Let's start this process.

Creating AuthController

In *src/* , create a subfolder called *auth* . Then, in *src/auth* create a new file called *auth.controller.ts* and fill it with the following code.

Add Entire Code

```
import { Controller, Get, Render, Post, Redirect, Body } from '@nestjs/common';
import { User } from 'src/models/user.entity';
import { UsersService } from '../models/users.service';


@Controller('/auth')
export class AuthController {
   constructor(private readonly usersService: UsersService) {}


   @Get('/register')
   @Render('auth/register')
   register() {
      const viewData = [];
      viewData['title'] = 'User Register - Online Store';
      viewData['subtitle'] = 'User Register';
      return {
         viewData: viewData,
      };
   }


   @Post('/store')
   @Redirect('/')
   async store(@Body() body) {
      const newUser = new User();
      newUser.setName(body.name);
      newUser.setPassword(body.password);
      newUser.setEmail(body.email);
      newUser.setRole('client');
      newUser.setBalance(1000);
      await this.usersService.createOrUpdate(newUser);
   }
}
```

We define the *AuthController* class with two methods.

We have a simple *register* method linked to the ("/auth/register") route that displays the *views/auth/register.hbs* view. This view will contain a form to create new users.

We have a *store* method linked to the ("/auth/store") route which redirects to the ("/") route. This method will be invoked once someone completes the form defined in the *views/auth/register.hbs* view. We create a new *User* instance, set the user *name* , *email* , and *password* based on the values collected from the form and assigned it to the *body* variable. We also set a default user role (*client*), and a default user balance (*1000*). This means that the

user will have 1000 dollars to spend in our application. Finally, we invoke the *usersService.createOrUpdate* method with *newUser* to create the new user in the database.

Auth register view

In *views/* , create a subfolder called *auth* . Then, in *views/auth* , create a new file called *register.hbs* , and fill it with the following code.

Add Entire Code

```
{{#> app}}
  {{#*inline "content"}}
    <div class="container">
      <div class="row justify-content-center">
        <div class="col-md-8">
          <div class="card">
                <div class="card-header">Register</div>

                <div class="card-body">
                    <form method="POST" action="/auth/store">
                        <div class="form-group row">
                            <label for="name" class="col-md-4 col-form-label text-md-right">Name</label>
                            <div class="col-md-6 pb-2">
                                <input id="name" type="text" class="form-control" name="name" required />
                            </div>
                        </div>

                        <div class="form-group row">
                            <label for="email" class="col-md-4 col-form-label text-md-right">Email</label>
                            <div class="col-md-6 pb-2">
                                <input id="email" type="email" class="form-control" name="email" required />
                            </div>
                        </div>

                        <div class="form-group row">
                            <label for="password" class="col-md-4 col-form-label text-md-right">Password</label>
                            <div class="col-md-6 pb-2">
          <input id="password" type="password" class="form-control" name="password" required />
                            </div>
                        </div>

                        <div class="form-group row mb-0">
                            <div class="col-md-6 offset-md-4">
                                <button type="submit" class="btn bg-primary text-white">Register</button>
                            </div>
                        </div>
                    </form>
                </div>
          </div>
        </div>
      </div>
    </div>
  {{/inline}}
{{/app}}
```

We define a form to collect the user *name* , *email* , and *password* . Once submitted, this form invokes the (“auth/store”) route.

Updating link in app layout

Now that we have the proper register route, let’s include it in the *app* layout. In *views/layouts/app.hbs* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
…
            <div class=”navbar-nav ms-auto”>
                <a class=”nav-link active” href=”/”>Home</a>
                <a class=”nav-link active” href=”/products”>Products</a>
                <a class="nav-link active" href="/about">About</a>
                <div class="vr bg-white mx-2 d-none d-lg-block"></div>
                <a class="nav-link active" href="/auth/register">Register</a>
            </div>
...
```

Creating Auth Module

In *src/auth* create a new file *auth.module.ts* and fill it with the following code.

Add Entire Code

```
import { Module } from '@nestjs/common';
import { AuthController } from './auth.controller';


@Module({
  controllers: [AuthController],
})
export class AuthModule {}
```

We created *AuthModule* and registered *AuthController* .

Registering AuthModule, User, and UsersService in AppModule

Let's register the *AuthModule* , *User* , and *UsersService* in the application root module (*AppModule*). In *src/app.module.ts* , make the following changes in **bold**.

Modify Bold Code

```
…
import { ProductsService } from './models/products.service';
import { UsersService } from './models/users.service';
import { Product } from './models/product.entity';
import { User } from './models/user.entity';
import { AdminModule } from './admin/admin.module';
import { AuthModule } from './auth/auth.module';


@Module({
  imports: [
    TypeOrmModule.forRoot(),
    TypeOrmModule.forFeature([Product, User]),
    AdminModule,
    AuthModule,
  ],
  controllers: [AppController, ProductsController],
  providers: [ProductsService, UsersService],
  exports: [ProductsService, UsersService],
})
export class AppModule {}
```

We import *AuthModule* and register it in the module *imports* property. This import gives access to the routes defined in *AuthController* . We also imported and configured the *UsersService* and the *User* entity. It makes *UsersService* and the TypeORM *User Repository* available to be injected across all modules.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now go to the ("/auth/register") route, and you will see the registration form (see Fig. 21-2). Then, create a new user. You will be redirected to the home page, and if you check your database, you will find the new record (see Fig. 21-3).

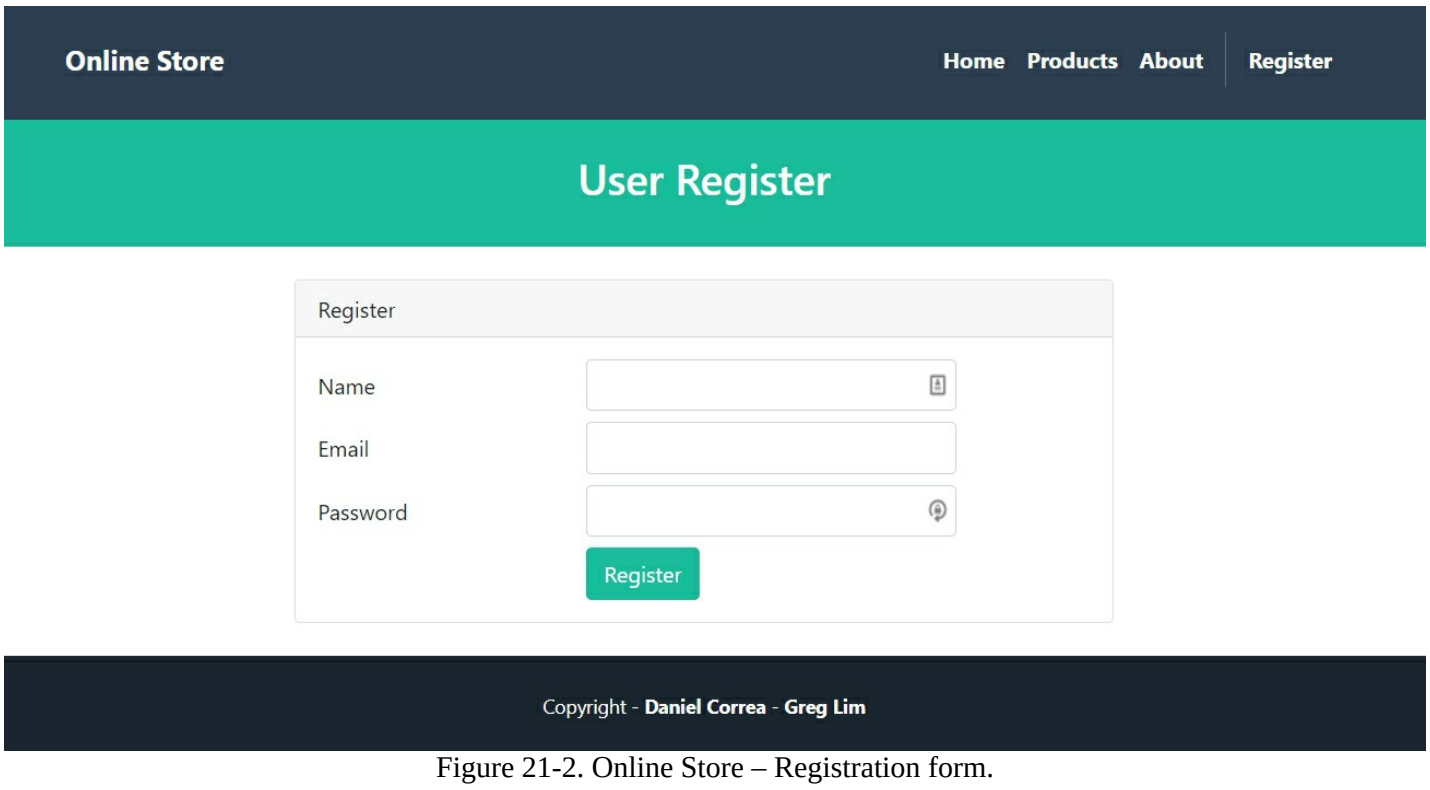

Figure 21-2. Online Store – Registration form.

id	name	email	password	role	balance
1	Daniel	books@danielgara.com	$2b$10$voCE1cYClzpLIMFHB.FckOZZ5oyv0U5FUlMd4am2Ges...	client	1000

Figure 21-3. New user row – phpMyAdmin.

Chapter 22 – Login System

Now that users can create accounts, we need to implement a login system. This process requires a series of steps and the use of some new libraries. So, let's begin.

Introduction to express-session library

For now, we have relied on the HTTP protocol to communicate with our Online Store. For example, if we want to get products' information, we access http://localhost:3000/products. To log in, we access http://localhost:3000/auth/login. Each of our requests uses the HTTP protocol as a means of communication.

However, the HTTP protocol has some limitations. HTTP is a **stateless** protocol, meaning that the server does not keep any data (state) between two requests. In HTTP, every request creates a new connection, and each new request knows nothing about any prior requests. That is, it holds no state data.

There are multiple ways of holding state data. Nest suggests using the express-session library (https://github.com/expressjs/session). This library provides a way to store information about the user across multiple requests, which is particularly useful for MVC applications.

Installing express-session

Let's install express-session in our Online Store project. Go to the project directory, and in the Terminal, execute the following commands.

Execute in Terminal

```
npm install express-session
npm install -D @types/express-session
```

Configuring express-session

We need to configure the Nest application to use express-session. In *src/main.ts* , make the following changes in **bold**.

Modify Bold Code

```
…
import * as hbs from 'hbs';
import * as hbsUtils from 'hbs-utils';
import * as session from 'express-session';

async function bootstrap() {
  ...
  app.setViewEngine('hbs');
  app.use(
    session({
      secret: 'nest-book',
      resave: false,
      saveUninitialized: false,
    }),
  );
  app.use(function (req, res, next) {
    res.locals.session = req.session;
    next();
  });

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

Let's analyze the code by parts.

Analyze Code

```
  app.use(
    session({
      secret: 'nest-book',
      resave: false,
      saveUninitialized: false,
    }),
  );
```

We import the express-session library, and configure our Nest Express application to use sessions. This configuration requires defining a *secret* . The *secret* is used to sign a session ID cookie stored in the user's browser and serves to identify the user accessing the application. The *secret* is just a string. You can define your own *secret* . Enabling the *resave* option forces the session to be saved back to the session store, even if the session

was never modified during the request. Likewise, enabling the *saveUninitialized* option forces a session that is "uninitialized" to be saved to the store. A session is uninitialized when it is new but not modified. We place both options on *false* since it will help to reduce server storage usage.

WARNING: The express-session library uses a lightweight memory store mechanism by default. It stores the session information in a *MemoryStore* instance. *MemoryStore* is purposely not designed for a production environment. It will leak memory under most conditions, does not scale past a single process, and is meant for debugging and developing. **Thus, session data is not persisted across server restarts**. If you run your application, and you modify code and save it, the session data will be lost. You can find session stores alternatives in the following link: https://github.com/expressjs/session#compatible-session-stores.

Analyze Code

```
app.use(function (req, res, next) {
   res.locals.session = req.session;
   next();
});
```

The previous code defines an express middleware. A **middleware** is a function that has access to the request object (*req*), the response object (*res*), and the next middleware function in the application's request-response cycle. Commonly, middleware functions are used to execute code and make changes to the request and the response objects. Let's see what we are doing with the previous middleware.

res.locals is an object that contains response local variables scoped to the request, and therefore available only to the view(s) rendered during that request/response cycle. In this case, we are extending the *res.locals* attribute by including a new attribute called *res.locals.session* , which will use the sessions values of the current request (*req.session*). With the implementation of this middleware, we can access the sessions' values from all Handlebars views (we will use it later). We then invoke the next middleware (*next()*).

Login implementation

Now that we have express-session library installed and configured, let's implement the login system.

Modifying UsersService

In *src/models/users.service.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
@Injectable()
export class UsersService {
   ...

   async login(email: string, password: string): Promise<User> {
      const user = await this.usersRepository.findOne({ email: email });
      if (user) {
         const isMatch = await bcrypt.compare(password, user.getPassword());
         if (isMatch) {
            return user;
         }
      }
      return null;
   }
}
```

We include a *login* method. This method receives an email and a password and returns a Promise consisting of a *User* . First, we search for a user using *this.usersRepository.findOne* based on the received *email* . If a user is found, we use the *bcrypt.compare* function to validate that the received *password* matches the password stored in the database. If they match, we return the *user* instance. Otherwise, return *null* .

Modifying AuthController

In *src/auth/auth.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Controller, Get, Render, Post, Redirect, Body,
   Req, Res } from '@nestjs/common';
…
@Controller('/auth')
export class AuthController {
   …
```

```
@Get('/login')
@Render('auth/login')
login() {
   const viewData = [];
   viewData['title'] = 'User Login - Online Store';
   viewData['subtitle'] = 'User Login';
   return {
      viewData: viewData,
   };
}

@Post('/connect')
async connect(@Body() body, @Req() request, @Res() response) {
   const email = body.email;
   const pass = body.password;
   const user = await this.usersService.login(email, pass);
   if (user) {
      request.session.user = {
         id: user.getId(),
         name: user.getName(),
         role: user.getRole(),
      };
      return response.redirect('/');
   } else {
      return response.redirect('/auth/login');
   }
  }

  @Get('/logout')
  @Redirect('/')
  logout(@Req() request) {
     request.session.user = null;
  }
}
```

Let’s analyze the code by parts.

Analyze Code

```
@Get('/login')
@Render('auth/login')
login() {
   const viewData = [];
   viewData['title'] = 'User Login - Online Store';
   viewData['subtitle'] = 'User Login';
   return {
      viewData: viewData,
   };
}
```

We define a *login* method linked to the (“auth/login”) route which renders the *auth/login* view. This view contains a form where the user enters the email and password information to log in.

Analyze Code

```
@Post('/connect')
async connect(@Body() body, @Req() request, @Res() response) {
   const email = body.email;
   const pass = body.password;
   const user = await this.usersService.login(email, pass);
   if (user) {
      request.session.user = {
         id: user.getId(),
         name: user.getName(),
         role: user.getRole(),
      };
      return response.redirect('/');
   } else {
      return response.redirect('/auth/login');
   }
}
```

The *connect* method uses the imported *Req* and *Res* decorators to access the express *request* and *response* objects respectively. The *connect* method is invoked once the user accesses the (“auth/login”) route and completes the login form. In this method, we collect the *email* and *password* information (sent by the application user) and pass that information to the *this.usersService.login* method (which returns a *user* instance if the login information is correct). If *user* is *null* , we redirect the application user to the (“/auth/login”) route to log in again. Otherwise, login is successful and we populate the *request.session* with a new data (*user*). We store the user *id* , *name* , and *role* in the session. This information will be critical to know if the user is logged or not. Then, we redirect to the home page.

Analyze Code

```
@Get('/logout')
@Redirect('/')
logout(@Req() request) {
   request.session.user = null;
}
```

We then have the *logout* method which destroys the user information stored in the session (in the *request.session.user* variable) and redirects to the home page.

Creating auth/login view

In *views/auth* , create a new file called *login.hbs* and fill it with the following code.

Add Entire Code

```
{{#> app}}
  {{#*inline "content"}}
  <div class="container">
    <div class="row justify-content-center">
      <div class="col-md-8">
        <div class="card">
          <div class="card-header">Login</div>
          <div class="card-body">
                  <form method="POST" action="/auth/connect">
                    <div class="form-group row pb-2">
                      <label for="email" class="col-md-4 col-form-label text-md-right">Email</label>
                      <div class="col-md-6">
                        <input id="email" type="email" class="form-control" name="email" required />
                      </div>
                    </div>

                    <div class="form-group row pb-2">
                      <label for="password" class="col-md-4 col-form-label text-md-right">Password</label>
                      <div class="col-md-6">
                        <input id="password" type="password" class="form-control" name="password" required />
                      </div>
                    </div>

                    <div class="form-group row mb-0">
                      <div class="col-md-8 offset-md-4">
                        <button type="submit" class="btn bg-primary text-white">
                          Login
                        </button>
                      </div>
                    </div>
                  </form>
          </div>
        </div>
      </div>
    </div>
  </div>
  {{/inline}}
{{/app}}
```

We define a login view that collects the user's *email* and *password* . Once submitted, this form invokes the ("auth/connect") route.

Updating links in app layout

Now that we have the proper login and logout routes, let's include them in the *app* layout. In *views/layouts/app.hbs* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
…
          <div class=”navbar-nav ms-auto”>
            <a class=”nav-link active” href=”/”>Home</a>
            <a class=”nav-link active” href=”/products”>Products</a>
            <a class="nav-link active" href="/about">About</a>
            <div class="vr bg-white mx-2 d-none d-lg-block"></div>
            {{#if session.user}}
            <a class="nav-link active" href="/auth/logout">Logout ({{session.user.name}})</a>
            {{else}}
            <a class="nav-link active" href="/auth/login">Login</a>
            <a class="nav-link active" href="/auth/register">Register</a>
            {{/if}}
          </div>
...
```

We use the *if* helper in hbs to verify if we have user data in the session. If that is true, we display a *Logout* link with the name of the logged in user. Otherwise, we display the *Login* and *Register* links. If the user clicks the *Logout* link, the session data will be destroyed and thus is 'logged out'.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now go to the ("/auth/login") route, and you will see the login form (see Fig. 22-1). Login with an existing user account. You will be redirected to the home page and see the *Logout* link with your name (see Fig. 22-2).

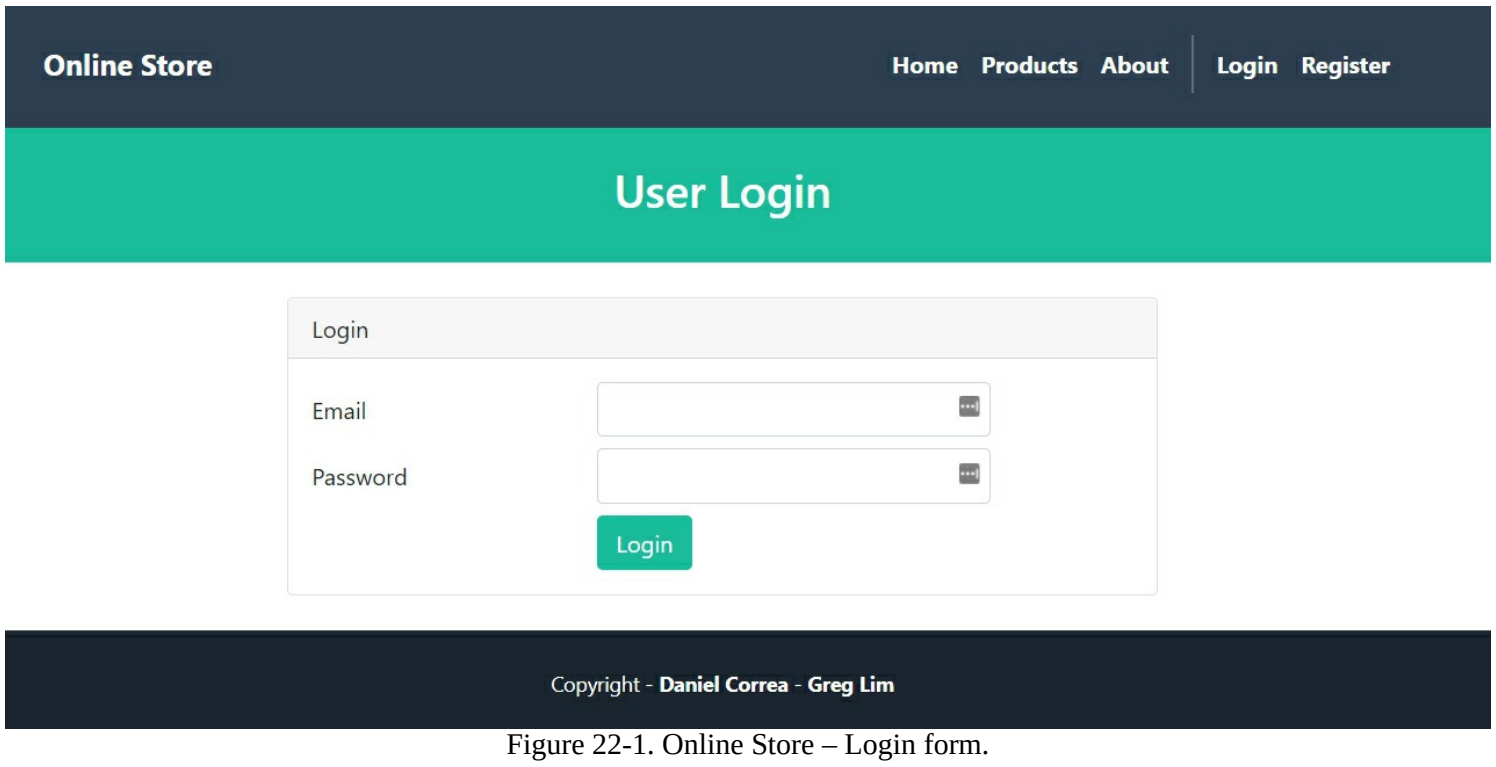

Figure 22-1. Online Store – Login form.

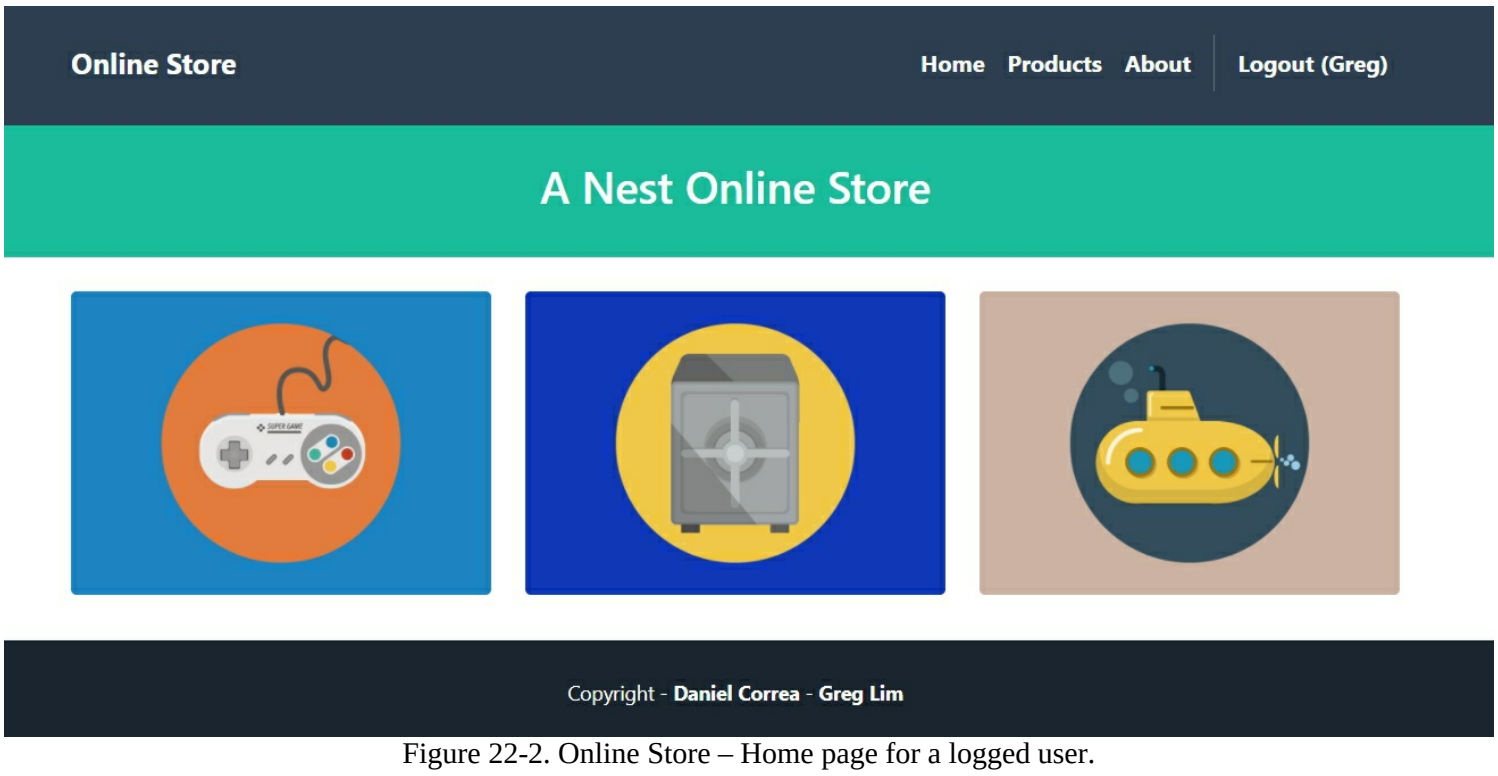

Figure 22-2. Online Store – Home page for a logged user.

Now that we have learned to use the express-session library, it's time to implement application validations.

Chapter 23 – Validations

Currently, we are only validating our application in the client-side (using some HTML elements, such as *required* or *input type email*). Those elements can be easily accessed and hacked. So, we will implement proper validations in our back-end code.

Introduction to Validator library

We can create our own validations, such as a function to validate that an email field contains an @ and a domain. However, we are not going to re-invent the wheel. We will use a JavaScript library called *validator* .

Validator is a library of string validators and sanitizers (https://github.com/validatorjs/validator.js). It provides functions to check data received from different forms (such as *isEmail* , *isEmpty* , and *isInt*).

Quick discussion: Nest official documentation suggests using *ValidationPipe* , *class-validator* , and *class-transformer* for validations (https://docs.nestjs.com/techniques/validation). The problem is that most Nest documentation is oriented to developing of applications that use service-oriented architectures (specifically APIs). But we are developing applications that use MVC architectures. The Nest validators don't work well with MVC applications. That is why we use the validator library.

Installing Validator

Let's install the validator library in our Online Store project. Go to the project directory, and in the Terminal, execute the following commands.

Execute in Terminal

```
npm install validator
```

Product Validator

Let's use the validator library to create our product validator. First, in *src* folder, create a subfolder called *validators* . Then, in *src/validators* , create a new file called *product.validator.ts* , and fill it with the following code.

Add Entire Code

```
import validator from 'validator';


export class ProductValidator {
   static imageWhiteList: string[] = [
      'image/png',
      'image/jpeg',
      'image/jpg',
      'image/webp',
   ];


   static validate(body, file: Express.Multer.File, toValidate: string[]) {
const errors: string[] = [];

if (toValidate.includes('name') && validator.isEmpty(body.name)) {
   errors.push('Product name cannot be empty');
}


if (
   toValidate.includes('description') &&
   validator.isEmpty(body.description)
) {
   errors.push('Product description cannot be empty');
}


if (
   toValidate.includes('price') &&
   !validator.isInt(body.price, { min: 0 })
) {
   errors.push('Product price must be not negative');
}


if (toValidate.includes('imageCreate')) {
   if (file === undefined) {
      errors.push('You must upload a product image');
   } else if (!ProductValidator.imageWhiteList.includes(file.mimetype)) {
```

```
            errors.push('Invalid image format');
        }
    }

    if (toValidate.includes('imageUpdate')) {
        if (
            file !== undefined &&
            !ProductValidator.imageWhiteList.includes(file.mimetype)
        ) {
            errors.push('Invalid image format');
        }
    }

    return errors;
  }
}
```

Let’s analyze the code by parts.

Analyze Code

```
import validator from 'validator';

export class ProductValidator {
  static imageWhiteList: string[] = [
    'image/png',
    'image/jpeg',
    'image/jpg',
    'image/webp',
  ];
```

We import the *validator* library and create a *ProductValidator* class. Then, we define an *imageWhiteList* static attribute, which contains the list of valid MIME Types for our image files. A MIME type is a two-part identifier for file formats and format contents transmitted on the Internet.

Analyze Code

```
  static validate(body, file: Express.Multer.File, toValidate: string[]) {
    const errors: string[] = [];

    if (toValidate.includes('name') && validator.isEmpty(body.name)) {
        errors.push('Product name cannot be empty');
    }

    if (
        toValidate.includes('description') &&
        validator.isEmpty(body.description)
    ) {
        errors.push('Product description cannot be empty');
    }
```

We define a *validate* static method which is the heart of the product validations. This method contains three arguments, *body* contains the information collected from the forms. *file* contains the uploaded image, and *toValidate* is an array that contains a set of labels to determine which validations should be applied which makes our *validation* method more generic.

The previous code shows two validations. The first validation is executed if *toValidate* contains the *name* label. Here, we use the *validator.isEmpty* method to verify if the *body.name* is empty. If that is true, we include an error in the *errors* variable. We validate *description* similarly.

Analyze Code

```
    if (
        toValidate.includes('price') &&
        !validator.isInt(body.price, { min: 0 })
    ) {
        errors.push('Product price must be not negative');
    }
```

For *price* , we validate if *body.price* is *int* , and has a minimum value of *10* .

Analyze Code

```
    if (toValidate.includes('imageCreate')) {
        if (file === undefined) {
            errors.push('You must upload a product image');
        } else if (!ProductValidator.imageWhiteList.includes(file.mimetype)) {
            errors.push('Invalid image format');
        }
    }
```

```
if (toValidate.includes('imageUpdate')) {
  if (
    file !== undefined &&
    !ProductValidator.imageWhiteList.includes(file.mimetype)
  ) {
    errors.push('Invalid image format');
  }
}

return errors;
```

For image , we have two validations. First, if *toValidate* includes the *imageCreate* label, we check if the *file* is *undefined* , we include an error. Then, we check if *file.mimetype* is included in the *imageWhiteList* attribute. If it is not, we include an error. In the second case, if *toValidate* includes the *imageUpdate* label, we check that *file* is not *undefined* and if *file.mimetype* is not included in *imageWhiteList* , we include an error.

In the end, we return the *errors* variable. If its length is greater than zero, it means that some validations failed.

Modifying AdminProductsController

In *src/admin/admin.products.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Controller, Get, Render, Post, Body, Redirect,
  UseInterceptors, UploadedFile, Param, Req } from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';
import { ProductsService } from '../models/products.service';
import { Product } from '../models/product.entity';
import { ProductValidator } from '../validators/product.validator';
import * as fs from 'fs';

@Controller('/admin/products')
export class AdminProductsController {
  ...

  @Post('/store')
  @UseInterceptors(FileInterceptor('image', { dest: './public/uploads' }))
  @Redirect('/admin/products')
  async store(@Body() body, @UploadedFile() file: Express.Multer.File,
    @Req() request,
  ) {
    const toValidate: string[] = ['name', 'description', 'price', 'imageCreate'];
    const errors: string[] = ProductValidator.validate(body, file, toValidate);
    if (errors.length > 0) {
      if (file) {
        fs.unlinkSync(file.path);
      }
      request.session.flashErrors = errors;
    } else {
      const newProduct = new Product();
      newProduct.setName(body.name);
      newProduct.setDescription(body.description);
      newProduct.setPrice(body.price);
      newProduct.setImage(file.filename);
      await this.productsService.createOrUpdate(newProduct);
    }
  }

  …
```

We import the *ProductValidator* class and the *fs* library. The *fs* library enables interacting with the file system. We then modify the *store* method. We include *request* in the method arguments and create a *toValidate* array to define the validations we want to apply. Then, we execute the *ProductValidator.validate* method. If *errors* are found, we check if the image was uploaded, and in that case, we remove the image from the filesystem (using the *fs.unlinkSync*). Besides, we set *request.session.flashErrors* to contain the current *errors* . Finally, if no *errors* are found, we create and save the new *product* into the database.

Modifying admin/products/index view

In *views/admin/products/index.hbs* , make the following changes in **bold**.

Modify Bold Code

```
{{#> admin}}
  {{#*inline "content"}}
    <div class="card mb-4">
```

```
        <div class="card-header">
          Create Products
        </div>
        <div class="card-body">
          {{#if flashErrors}}
            {{#each flashErrors}}
                    <p class="alert alert-danger">{{this}}</p>
            {{/each}}
          {{/if}}
          <form method="POST" action="/admin/products/store"enctype="multipart/form-data">
          …
```

We include some paragraphs to show errors. We iterate through the *flashErrors* object and show its content.

Modifying main.ts

In *src/main.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
app.use(function (req, res, next) {
  res.locals.session = req.session;
  const flashErrors: string[] = req.session.flashErrors;
  if (flashErrors) {
    res.locals.flashErrors = flashErrors;
    req.session.flashErrors = null;
  }
  next();
});

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

We modify the middleware used to enable access to the session values from our hbs views. In this case, we create a *flashErrors* array which contains the *req.session.flashErrors* . If there are *flashErrors* , we extend the *res.locals* property to contain the *flashErrors* in the *res.locals.flashErrors* property. Then, we destroy the *req.session.flashErrors* content.

Our flash errors system work as flash messages. A **flash message** is a type of message that is created in a page, display it once on another page (commonly, in a page that is called after a redirection), and then it disappears. Let's see them in action.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now go to the ("/admin/products") route. Create a product with invalid data (i.e., price -21), and you will see the proper error messages (see Fig. 23-1). If you reload (or revisit) the page, you will see the messages disappear.

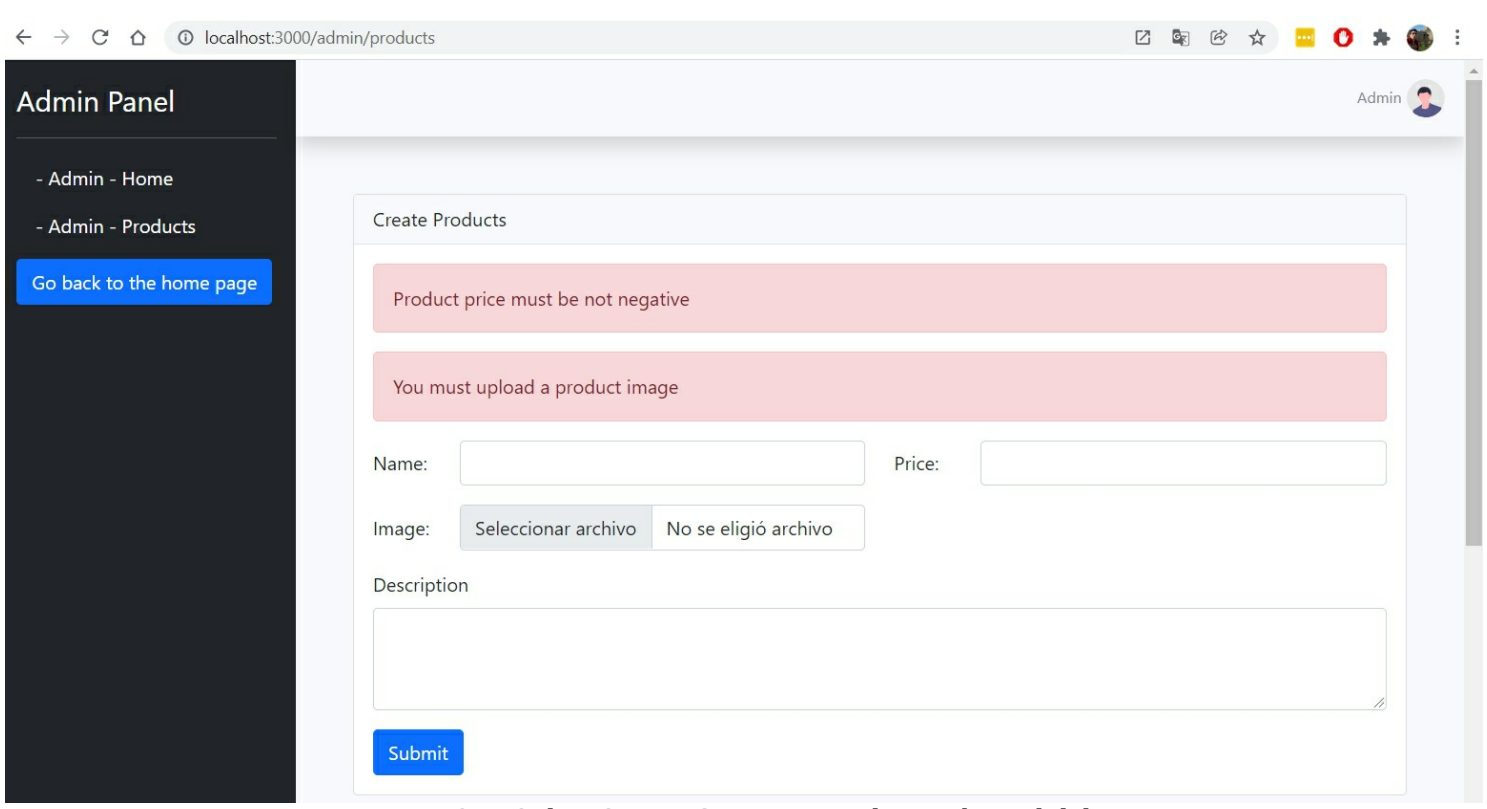

Figure 23-1. Online Store – Creating a product with invalid data.

Other Validations

Let’s complete other pending validations. Explanations will be short since they are similar to the previous validation.

Modifying AdminProductsController

In *src/admin/admin.products.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Controller, Get, Render, Post, Body, Redirect,
   UseInterceptors, UploadedFile, Param, Req, Res } from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';
…


@Controller('/admin/products')
export class AdminProductsController {
   ...


   @Post('/:id/update')
   @UseInterceptors(FileInterceptor('image', { dest: './public/uploads' }))
   @Redirect('/admin/products')
   async update(
   @Body() body,
   @UploadedFile() file: Express.Multer.File,
   @Param('id') id: string,
   @Req() request,
   @Res() response,
) {
   const toValidate: string[] = ['name', 'description', 'price', 'imageUpdate'];
   const errors: string[] = ProductValidator.validate(body, file, toValidate);
   if (errors.length > 0) {
      if (file) {
         fs.unlinkSync(file.path);
      }
      request.session.flashErrors = errors;
      return response.redirect('/admin/products/'+id);
   } else {
      const product = await this.productsService.findOne(id);
      product.setName(body.name);
      product.setDescription(body.description);
      product.setPrice(body.price);
      if (file) {
         product.setImage(file.filename);
      }
      await this.productsService.createOrUpdate(product);
      return response.redirect('/admin/products/');
   }
}
```

We modify the *update* method. We remove the *@Redirect* decorator since we will have two different redirections. We execute the product validations. In this case, we use the *imageUpdate* label. If there are *errors* , we redirect to the (“/admin/products/:id”) route. If there are no errors, we update the product and redirect to the (“/admin/products/”) route.

Modifying admin/products/edit view

In *views/admin/products/edit.hbs* , make the following changes in **bold**.

Modify Bold Code

```
{{#> admin}}
   {{#*inline "content"}}
      <div class="card mb-4">
         <div class="card-header">
            Edit Product
         </div>
         <div class="card-body">
            {{#if flashErrors}}
               {{#each flashErrors}}
                        <p class="alert alert-danger">{{this}}</p>
               {{/each}}
            {{/if}}
            {{#with viewData.product}}
            …
```

We include the *flashErrors* object and iterate through it to show the errors.

Creating user validator

In *src/validators* , create a new file *user.validator.ts* , and fill it with the following code.

Add Entire Code

```
import validator from 'validator';

export class UserValidator {
   static validate(body, toValidate: string[]) {
      const errors: string[] = [];

      if (toValidate.includes('name') && validator.isEmpty(body.name)) {
         errors.push('Name cannot be empty');
      }

      if (toValidate.includes('email') && !validator.isEmail(body.email)) {
         errors.push('Invalid Email format');
      }

      if (
       toValidate.includes('password') &&
         validator.isEmpty(body.password)
      ) {
         errors.push('Password cannot be empty');
      }

      return errors;
   }
}
```

We create a validate method with proper validations for *name* , *password* , and *email* . Note that for *email* , we use the *validator.isEmail* .

Modifying AuthController

In *src/auth/auth.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
…
import { UsersService } from '../models/users.service';
import { UserValidator } from '../validators/user.validator';



@Controller('/auth')
export class AuthController {
   …

   @Post('/store')
   @Redirect('/')
   async store(@Body() body, @Res() response, @Req() request) {
      const toValidate: string[] = ['name', 'email', 'password'];
      const errors: string[] = UserValidator.validate(body, toValidate);
      if (errors.length > 0) {
         request.session.flashErrors = errors;
         return response.redirect('/auth/register');
      } else {
         const newUser = new User();
         newUser.setName(body.name);
         newUser.setPassword(body.password);
         newUser.setEmail(body.email);
         newUser.setRole('client');
         newUser.setBalance(1000);
         await this.usersService.createOrUpdate(newUser);
         return response.redirect('/auth/login');
      }
   }
```

We modify the *store* method to execute the user validations. If there are *errors* , we direct to the (“/auth/register”) route again. If there are none, we redirect the user to the (“/auth/login”) route.

Modifying auth/register view

In *views/auth/register.hbs* , make the following changes in **bold**.

Modify Bold Code

```
{{#> app}}
```

```
{{#*inline "content"}}
    <div class="container">
        <div class="row justify-content-center">
            <div class="col-md-8">
                <div class="card">
                        <div class="card-header">Register</div>

                        <div class="card-body">
                            {{#if flashErrors}}
                                {{#each flashErrors}}
                                    <p class="alert alert-danger">{{this}}</p>
                                {{/each}}
                            {{/if}}
                            <form method="POST" action="/auth/store">
```

We include *flashErrors* as usual.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now you can create and edit products (“/admin/products”) and the validations will be applied. Also, if you try to register a user account (“/auth/register”), the proper validations will be applied and displayed.

Chapter 24 – Authorization

Currently, anyone can access the admin panel and create, edit, and delete products. In a real-world setting, this obviously shouldn't be the way. So, let's implement an authorization system.

Defining admins

If you check your *user* table, you will see that all users are 'clients'. We don't have admins yet. So, let's define an admin. Go to phpMyAdmin, in the *online_store* database, click the *user* table, browse a specific user, and change the user's *role* from *client* to *admin* (Fig. 24-1).

id	name	email	password	role	balance
1	Daniel	books@danielgara.com	$2b$10$bWnAu8QiQLUGSkP0KEjl/eg.qesLvTeX6w2qEF6JtVJ...	admin	1000
2	Greg	support@i-ducate.com	$2b$10$TCA47PMClXECm.HOcrSbF.E..jpXzWVAGeBY1Bko1f7...	client	1000

Figure 24-1. Editing a user's role.

Nest Authorization

Nest provides an authorization mechanism called guards (https://docs.nestjs.com/guards). A **guard** is a class annotated with the *@Injectable()* decorator which implements the *CanActivate* interface. Guards have a single responsibility. They determine whether a given request will be handled by the route handler or not depending on certain conditions (like permissions, roles, etc.) present at run-time. This is often referred to as **authorization**.

The problem with guards (and other Nest mechanisms) is that they are designed for API applications. If we implement Nest Guards, we will see code executions which result in pages showing JSON messages, such as the following.

Analyze Code

```
{
  "statusCode": 403,
  "message": "Forbidden resource",
  "error": "Forbidden"
}
```

These kind of messages and executions are not designed for MVC applications.

Quick discussion: At this point, we have the same problem we faced with Nest validations. We cannot use many of those libraries because they are not designed for MVC applications. Let's hope that in the future, Nest documentation evolves and creators can support a little more in the design of MVC applications. Who knows, maybe this book can help a little in that process.

Due to the previous issues, we will take another path and use express middleware instead to implement a solution.

Modifying main.ts

In *src/main.ts* , make the following changes in **bold**.

Modify Bold Code

```
  ...
  });
  app.use('/admin*', function (req, res, next) {
    if (req.session.user && req.session.user.role == 'admin') {
      next();
    } else {
      res.redirect('/');
    }
  });

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

We create a new middleware for routes that begins with the ("/admin") string (all our admin routes). We check if there is *a req.session.user* property, and that it contains a *role* property with *admin* value. If so, it means that the current user is logged in as an *admin* and can access the current route. We then call the *next()* method to continue to the next middleware. Otherwise, we redirect the user to the home page.

Quick discussion: We implemented a simple authorization system. If you have a better solution,

feel free to use the discussion zone of this book's repository to share it with us. We will learn from these discussions.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

If you access the ("/admin") route with a client user, you will be redirected to the home page. Now, log in to the application with the credentials of an admin user. And if you access the ("/admin") route, you will see the admin panel this time.

Chapter 25 – Shopping Cart

Now that we have used express-sessions a couple of times, let’s implement the shopping cart system.

Cart Module

The *CartModule* will manage all the actions related to the shopping cart, such as adding, removing, and checking products in the shopping cart. It requires defining a new module, controller, and view. Let’s begin.

Creating CartController

In *src/* , create a subfolder called *cart* . Then, in *src/cart* create a new file called *cart.controller.ts* and fill it with the following code.

Add Entire Code

```
import { Controller, Get, Render, Req, Redirect, Param,
   Body, Post } from '@nestjs/common';
import { ProductsService } from '../models/products.service';
import { Product } from '../models/product.entity';


@Controller('/cart')
export class CartController {
  constructor(private readonly productsService: ProductsService) {}

  @Get('/')
  @Render('cart/index')
  async index(@Req() request) {
    let total = 0;
    let productsInCart: Product[] = null;
    const productsInSession = request.session.products;
    if (productsInSession) {
      productsInCart = await this.productsService.findByIds(
        Object.keys(productsInSession),
      );
      total = Product.sumPricesByQuantities(productsInCart, productsInSession);
    }

    const viewData = [];
    viewData['title'] = 'Cart - Online Store';
    viewData['subtitle'] = 'Shopping Cart';
    viewData['total'] = total;
    viewData['productsInCart'] = productsInCart;
    return {
      viewData: viewData,
    };
  }

  @Post('/add/:id')
  @Redirect('/cart')
  add(@Param('id') id: number, @Body() body, @Req() request) {
    let productsInSession = request.session.products;
    if (!productsInSession) {
      productsInSession = {};
    }
    productsInSession[id] = body.quantity;
    request.session.products = productsInSession;
  }

  @Get('/delete')
  @Redirect('/cart/')
  delete(@Req() request) {
    request.session.products = null;
  }
}
```

Let’s analyze the code by parts.

Analyze Code

```
  @Post('/add/:id')
  @Redirect('/cart')
  add(@Param('id') id: number, @Body() body, @Req() request) {
    let productsInSession = request.session.products;
    if (!productsInSession) {
      productsInSession = {};
    }
    productsInSession[id] = body.quantity;
```

```
    request.session.products = productsInSession;
  }
```

The *add* method receives the inputs (quantity of product) collected from the form in the *body* argument, and the *id* of the product to be added in the cart. Then, we get the products stored in the session through the *request.session.products* property. The first time, *request.session.products* won't exist, so we assign it to an empty object. Next, we include in *productsInSession* variable the collected product *id* (as a key) with its *quantity* (as the corresponding value). We then update the products stored in the session. Finally, the *@Redirect* decorator redirects the user to the ("/cart") route.

Analyze Code

```
  @Get('/delete')
  @Redirect('/cart/')
  delete(@Req() request) {
    request.session.products = null;
  }
```

The *delete* method receives the *request* , and removes the products stored in the session associated to that request. Then, we redirect to the ("/cart") route.

Analyze Code

```
  @Get('/')
  @Render('cart/index')
  async index(@Req() request) {
    let total = 0;
    let productsInCart: Product[] = null;
    const productsInSession = request.session.products;
    if (productsInSession) {
      productsInCart = await this.productsService.findByIds(
        Object.keys(productsInSession),
      );
      total = Product.sumPricesByQuantities(productsInCart, productsInSession);
    }

    const viewData = [];
    viewData['title'] = 'Cart - Online Store';
    viewData['subtitle'] = 'Shopping Cart';
    viewData['total'] = total;
    viewData['productsInCart'] = productsInCart;
    return {
      viewData: viewData,
    };
  }
```

The *index* method defines a *total* variable with a zero value and an empty *productsInCart* array. First, we check if the current *request* has products stored in session. If there are *productsInSession* , we extract the related products from the database. In this case, we use the model *findByIds* method, which receives an array with primary keys and returns an array of products (this method will be implemented later). We send *Object.keys(productsInSession)* to this method (remember we store the products id as keys and the quantities of the products as values). Then, we update the *total* value by invoking the *Product.sumPricesByQuantities* method (which will be implemented next). Finally, we send the *total* and *productsInCart* to the *cart/index* view.

Modifying Product Entity

In *src/models/product.entity.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
  setPrice(price: number) {
    this.price = price;
  }

  static sumPricesByQuantities(products: Product[], productsInSession): number {
    let total = 0;
    for (let i = 0; i < products.length; i++) {
      total =
        total + products[i].getPrice() * productsInSession[products[i].getId()];
    }
    return total;
  }
}
```

We includ a new static method called *sumPricesByQuantities* . *sumPricesByQuantities* receives an array of products and the products stored in session. It iterates over the products and calculates the *total* to be paid (based on the *price* of each product and its corresponding *quantity*). It then returns the *total* to be paid.

TIP: We have designed an architecture where models are separated into two parts: entities and services. We use entities to store our classes information and operations related to those classes (that's why we include *sumPricesByQuantities* method in our *Product* entity file). And we use services to group all the database operations. Having this in mind, it will be easier for you to organize your code.

Modifying Products Service

In *src/models/products.service.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
@Injectable()
export class ProductsService {
  ...

  findOne(id: string): Promise<Product> {
    return this.productsRepository.findOne(id);
  }

  findByIds(ids: string[]): Promise<Product[]> {
    return this.productsRepository.findByIds(ids);
  }

  ...
```

The *findByIds* method receives an array of strings (*ids*) and returns a Promise consisting of an array of products. This method uses the *productsRepository* attribute, which invokes the *findByIds* method (inherited from the TypeORM *Repository* class). It takes the array of *ids* , and returns the corresponding products based on those *ids* .

Creating Cart Module

In *src/cart* create a new file called *cart.module.ts* and fill it with the following code.

Add Entire Code

```
import { Module } from '@nestjs/common';
import { CartController } from './cart.controller';

@Module({
  controllers: [CartController],
})
export class CartModule {}
```

We created *CartModule* and registered *CartController* .

Modifying AppModule

Let's register *CartModule* in the application root module (*AppModule*). In *src/app.module.ts* , make the following changes in **bold**.

Modify Bold Code

```
…
import { AuthModule } from './auth/auth.module';
import { CartModule } from './cart/cart.module';

@Global()
@Module({
  imports: [
    TypeOrmModule.forRoot(),
    TypeOrmModule.forFeature([Product, User]),
    AdminModule,
    AuthModule,
    CartModule,
  ],
  …
```

Cart index view

In *views/* , create a subfolder called *cart* . In *views/cart* create a new file *index.hbs* and fill it with the following code.

Add Entire Code

```
{{#> app}}
  {{#*inline "content"}}
    <div class="card">
```

```
<div class="card-header">
   Products in Cart
</div>
<div class="card-body">
   <table class="table table-bordered table-striped text-center">
      <thead>
               <tr>
                  <th scope="col">ID</th>
                  <th scope="col">Name</th>
                  <th scope="col">Price</th>
                  <th scope="col">Quantity</th>
               </tr>
      </thead>
      <tbody>
               {{#each viewData.productsInCart}}
               <tr>
                  <td>{{getId}}</td>
                  <td>{{getName}}</td>
                  <td>${{getPrice}}</td>
                  <td>{{lookup ../session.products id}}</td>
               </tr>
               {{/each}}
      </tbody>
   </table>
   <div class="row">
      <div class="text-end">
               <a class="btn btn-outline-secondary mb-2"><b>Total to pay:</b> ${{viewData.total}}</a>
               <a class="btn bg-primary text-white mb-2">Purchase</a>
               <a href="/cart/delete" class="btn btn-danger mb-2">
                  Remove all products from Cart
               </a>
         </div>
      </div>
   </div>
</div>
  {{/inline}}
{{/app}}
```

This view is like the *admin/products/index* view. We iterate and display the products added in session. We also use the *session* hbs object to access the products' quantities. Since the *session* object is outside the *each* scope, we must use the "../". In theory we can access the products quantities through this form *../session.products.[getID]* . But handlebars is very restrictive. So, we need to use the *lookup* helper which allows for dynamic parameter resolution. This helper doesn't allow us to use *getId* , so we use the *id* variable to access product *id* and extract the quantity. Finally, we display the *total* to be paid, a *purchase* button (which doesn't do anything yet), and a button to remove all products from the cart linking the ("/cart/delete") route.

Modifying the products/show view

In *views/products/show.hbs* , make the following changes in **bold**.

Modify Bold Code

```
...
         <div class="card-body">
               <h5 class="card-title">
                  {{getName}} (${{getPrice}})
               </h5>
               <p class="card-text">{{getDescription}}</p>
               <p class="card-text"><small class="text-muted">Add to Cart</small></p>
               <p class="card-text">
                  <form method="POST" action="/cart/add/{{getId}}">
                     <div class="row">
                        <div class="col-auto">
                           <div class="input-group col-auto">
                              <div class="input-group-text">Quantity</div>
      <input type="number" min="1" max="10" class="form-control quantity-input" name="quantity" value="1">
                           </div>
                        </div>
                        <div class="col-auto">
                           <button class="btn bg-primary text-white" type="submit">Add to cart</button>
                        </div>
                     </div>
                  </form>
               </p>
         </div>
...
```

We add a new form where the user enters the product's quantity to the cart. This form is linked to the ("/cart/add/:id") route.

Updating link in app layout

Now that we have the proper cart route, let's include it in the *app* layout. In *views/layouts/app.hbs* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
…
            <div class=”navbar-nav ms-auto”>
               <a class=”nav-link active” href=”/”>Home</a>
               <a class=”nav-link active” href=”/products”>Products</a>
               <a class="nav-link active" href="/cart">Cart</a>
               <a class="nav-link active" href="/about">About</a>
               <div class="vr bg-white mx-2 d-none d-lg-block"></div>
               ...
```

We added a new link to the cart page.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now go to the (“/products”) route, click a specific product, and you will see the new *products/show* view (see Fig. 25-1). Next, you can add some products to the cart and visit the (“/cart”) route. It will show you the total to be paid and a button to remove all products from the cart (see Fig. 25-2).

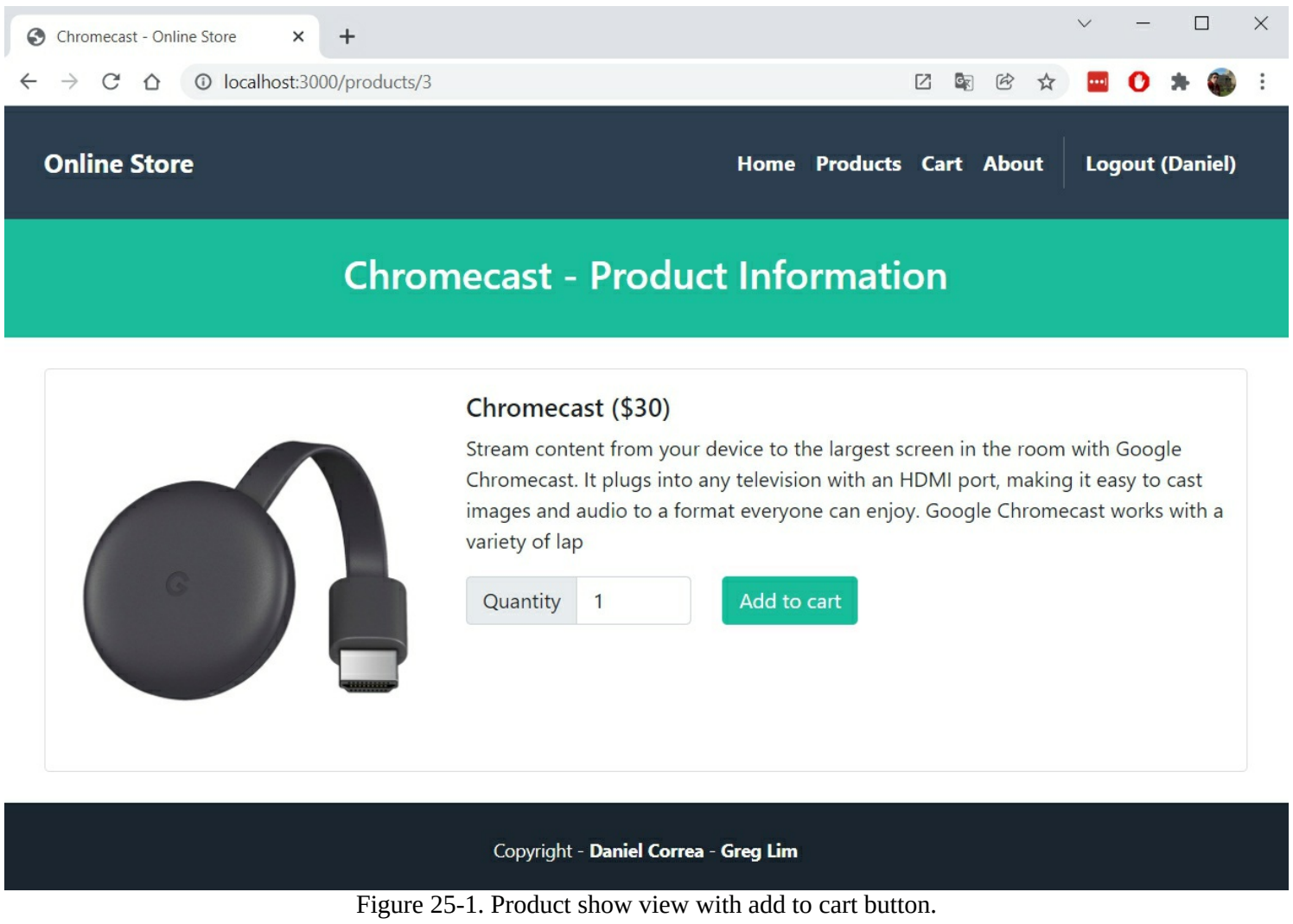

Figure 25-1. Product show view with add to cart button.

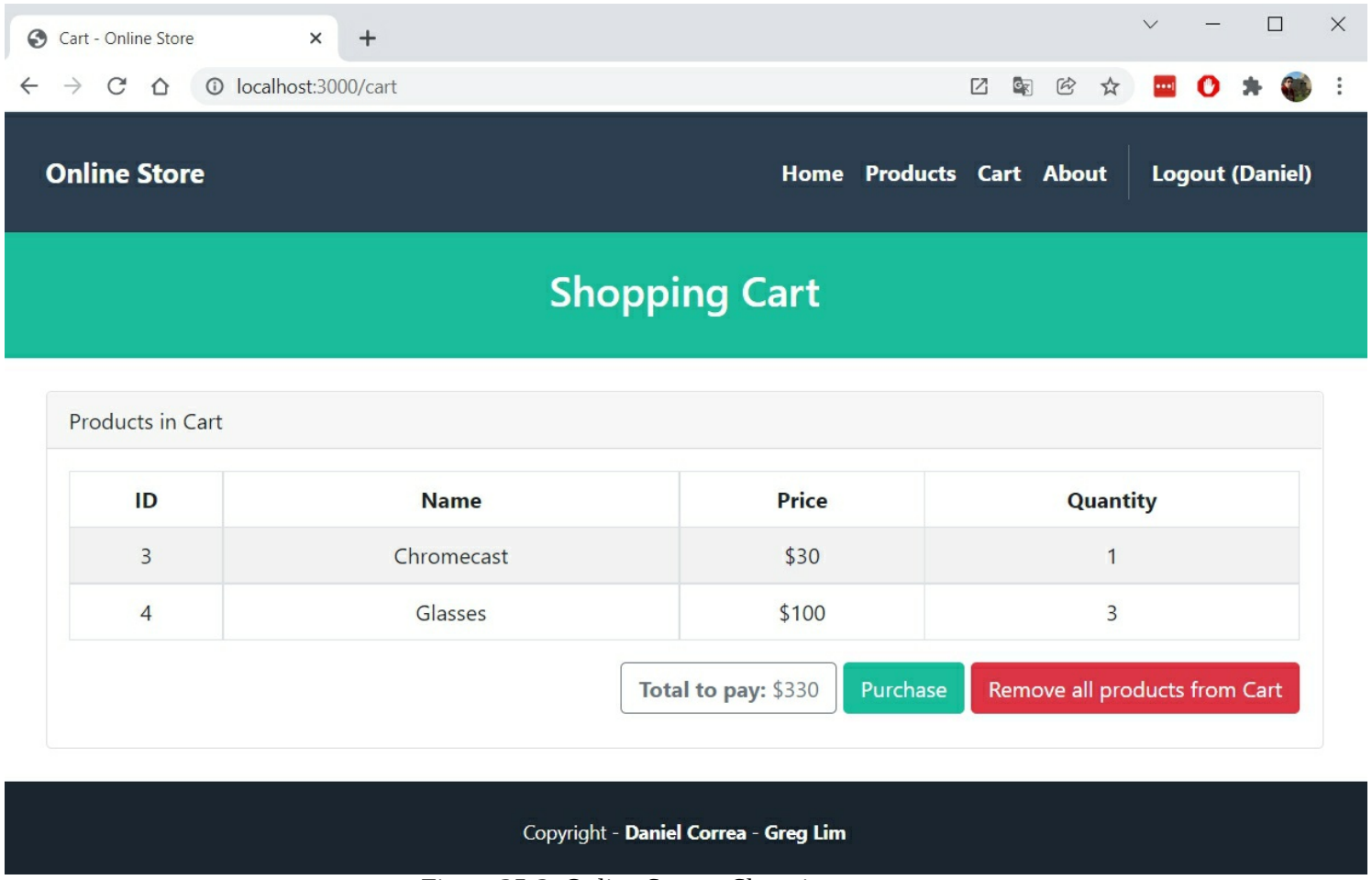

Figure 25-2. Online Store – Shopping cart page.

Chapter 26 – Orders and Items

To complete our shopping cart, we must be able to make purchases. However, we need to set in place some things before adding the purchase functionality.

Orders and Items

Let’s take Fig. 26-1 as a base. Though a simplified invoice, it is helpful to show the data we need to store.

Order #1

Date: 2021-10-14

Total: $1060

User id - name: 1 - Daniel

Qty	Product id - name	Price
1	1 - TV	1000
2	3 - Chromecast	30

Figure 26-1. Example of an invoice.

For the **Order**, we need to store:

- **Id:** in the example, it is *#1* .
- **Date:** in the example, it is *2021-10-14* .
- **Total:** in the example, it is *$1060* .
- **User id:** in the example, it is *1* .
- **User name:** in the example, it is *Daniel* . Username won’t be stored since we can retrieve it with user *id* .

Orders are composed of items (see the internal table in Fig. 26-1). So, for each **Item**, we need to store:

- **Quantity:** in the example, it is *1* for the first item. It means that the user is buying one TV.
- **Product id:** in the example, it is *1* for the first item. It means that the user is buying the product with *id 1* .
- **Product name:** in the example, it is *TV* for the first item. Product name won’t be stored since we can retrieve it based on product *id* .
- **Price:** in the example, it is *1000* for the first item. We will store this price since it is common for products to change their price. Also, storing the price in the item table will allow us to know at which price the user bought each product.
- **Id:** we will include the item *id* to trace each item.

Now that we understand how these simplified invoices work, let’s create the proper entities.

Order Entity

In *src/models* create a new file called *order.entity.ts* and fill it with the following code.

Add Entire Code

```
import { Entity, Column, PrimaryGeneratedColumn, ManyToOne,
   OneToMany, CreateDateColumn } from 'typeorm';
import { User } from './user.entity';
import { Item } from './item.entity';


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


   @Column()
   total: number;


   @CreateDateColumn()
   date: Date;
```

```
@ManyToOne(() => User, (user) => user.orders)
user: User;

@OneToMany(() => Item, (item) => item.order)
items: Item[];

getId(): number {
   return this.id;
}

setId(id: number) {
   this.id = id;
}

getTotal(): number {
   return this.total;
}

setTotal(total: number) {
   this.total = total;
}

   getDate(): Date {
      return this.date;
   }

   setDate(date: Date) {
      this.date = date;
   }

   getUser(): User {
      return this.user;
   }

   setUser(user: User) {
      this.user = user;
   }

   getItems(): Item[] {
      return this.items;
   }

   setItems(items: Item[]) {
      this.items = items;
   }
}
```

Let’s analyze the code by parts.

The *Order* entity contains an *id* , a *total* , a *date* , and two attributes (*user* and *items*) representing relations to other entities. Let’s analyze the *date* , *user* and *items* in detail.

Analyze Code

```
@CreateDateColumn()
date: Date;
```

TypeORM provides a special column(*CreateDateColumn*) that sets to the *date* attribute the entity’s insertion time. Thus, we won’t need to set this attribute as it will be automatically set.

Analyze Code

```
@ManyToOne(() => User, (user) => user.orders)
user: User;
```

The *user* attribute is decorated with the *@ManyToOne* . *@ManyToOne* indicates a relation where *User* contains multiple instances of *Orders* , but *Order* contains only one *User* instance. In our application, a user can create multiple orders, but an order is only assigned to a specific user. The use of these decorators will create some foreign keys in our database tables and allow us to navigate across our entities. For example, we will be able to extract the name of the user who completed a specific order with a piece of code like: *order->getUser()->getName()* (we will see an example later). We will add *orders* attribute to *user* later.

Analyze Code

```
@OneToMany(() => Item, (item) => item.order)
items: Item[];
```

Here we add *@OneToMany* to the *items* attribute (the inverse of *@ManyToOne* decorator). An *Item* contains only one instance of *Order* , but an *Order* contains multiple instances of *Items* . This is consistent with Fig. 26-1. More information about TypeORM relations can be found here: https://typeorm.io/#/relations.

Analyze Code

```
getUser(): User {
   return this.user;
}

setUser(user: User) {
   this.user = user;
}

getItems(): Item[] {
   return this.items;
}

setItems(items: Item[]) {
   this.items = items;
}
```

Finally, we have proper getters and setters to manage these attributes.

Modifying User Entity

In *src/models/user.entity.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Entity, Column, PrimaryGeneratedColumn, OneToMany } from 'typeorm';
import { Order } from './order.entity';

@Entity()
export class User {
   ...

   @Column()
   balance: number;

   @OneToMany(() => Order, (order) => order.user)
   orders: Order[];

   ...

   getOrders(): Order[] {
      return this.orders;
   }

   setOrders(orders: Order[]) {
      this.orders = orders;
   }
}
```

We import the *Order* entity and use it to define the *orders* attribute. Note that we use the *@OneToMany* decorator since a *User* contains multiple *Orders* . Then, we define the proper getter and setter for the new attribute.

Item Entity

In *src/models* create a new file called *item.entity.ts* and fill it with the following code.

Add Entire Code

```
import { Entity, Column, PrimaryGeneratedColumn, ManyToOne } from 'typeorm';
import { Order } from './order.entity';
import { Product } from './product.entity';

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

   @Column()
   quantity: number;
```

```
@Column()
price: number;

@ManyToOne(() => Order, (order) => order.items)
order: Order;

@ManyToOne(() => Product, (product) => product.items)
product: Product;

getId(): number {
   return this.id;
}

setId(id: number) {
   this.id = id;
}

getQuantity(): number {
   return this.quantity;
}

setQuantity(quantity: number) {
   this.quantity = quantity;
}

getPrice(): number {
   return this.price;
}

setPrice(price: number) {
   this.price = price;
}

getOrder(): Order {
   return this.order;
}

setOrder(order: Order) {
   this.order = order;
}

getProduct(): Product {
   return this.product;
}

setProduct(product: Product) {
   this.product = product;
}
}
```

The *Item* entity contains an *id* , a *quantity* , a *price* , and two attributes (*order* and *product*) representing relations to other entities. *@ManyToOne* is used to decorate the *order* and *product* attributes since an *Order* contains multiple *Items* , and a *Product* contains multiple *Items* too. And an *Item* is only assigned to one *Order* and to one *Product* . We complete the *Item* entity with getters and setters.

Modifying Product Entity

In *src/models/product.entity.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Entity, Column, PrimaryGeneratedColumn, OneToMany } from 'typeorm';
import { Item } from './item.entity';



@Entity()
export class Product {
   ...

   @Column()
   price: number;

   @OneToMany(() => Item, (item) => item.product)
   items: Item[];
```

```
...

setPrice(price: number) {
   this.price = price;
}

getItems(): Item[] {
   return this.items;
}

setItems(items: Item[]) {
   this.items = items;
}

...
```

We import the *Item* entity and use it to define the *items* attribute. Note that we use the *@OneToMany* decorator since a *Product* contains multiple *Items* . Then, we define the proper getter and setter for the new attribute.

Analyzing class diagram

Fig. 26-2 shows our initial class diagram. We have implemented almost all the elements in that diagram.

- We implemented all the classes (in the form of entities).
- We implemented all the classes' attributes.
- We implemented all the classes' associations.
- We implemented all the classes' getters and setters.
- We implemented many CRUD methods (in the form of services). However, we will need to implement the Order service later.

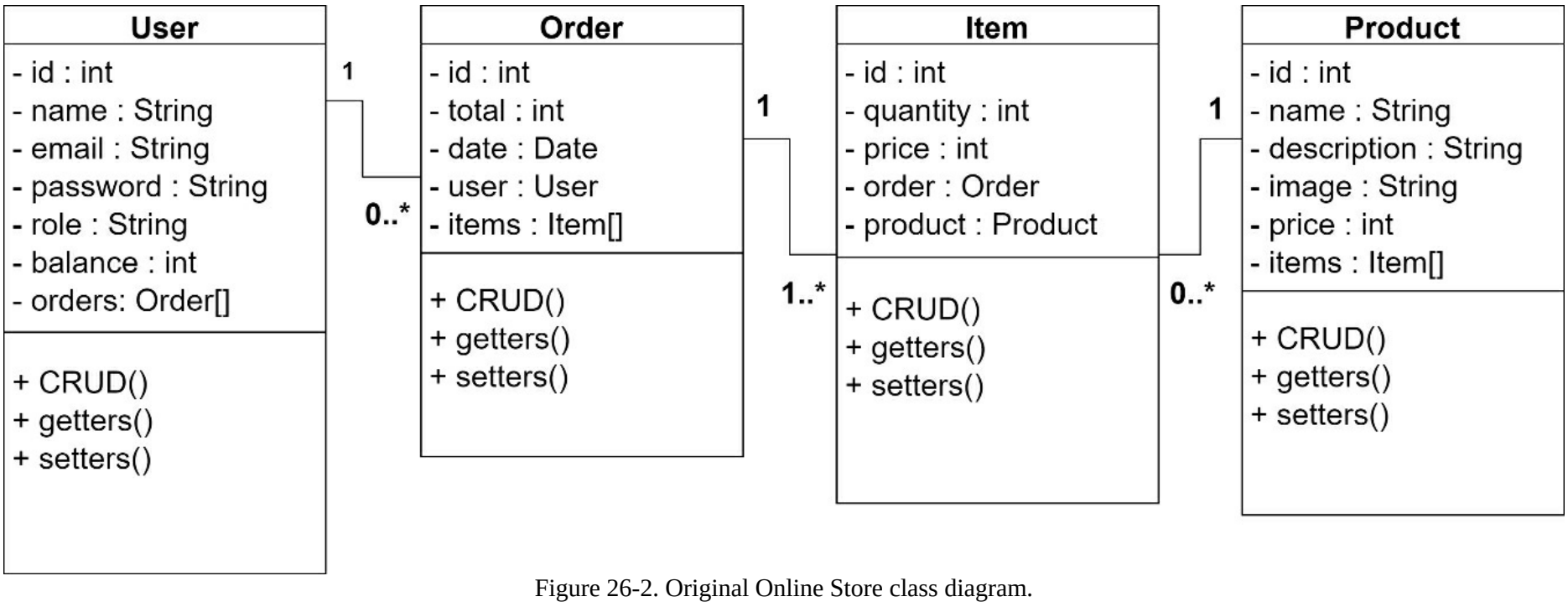

Figure 26-2. Original Online Store class diagram.

Synchronizing the database

Let's synchronize our database. **First, stop the server.** Then, In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

You can see the new *order* and *item* tables in your *online_store* database (see Fig. 26-3).

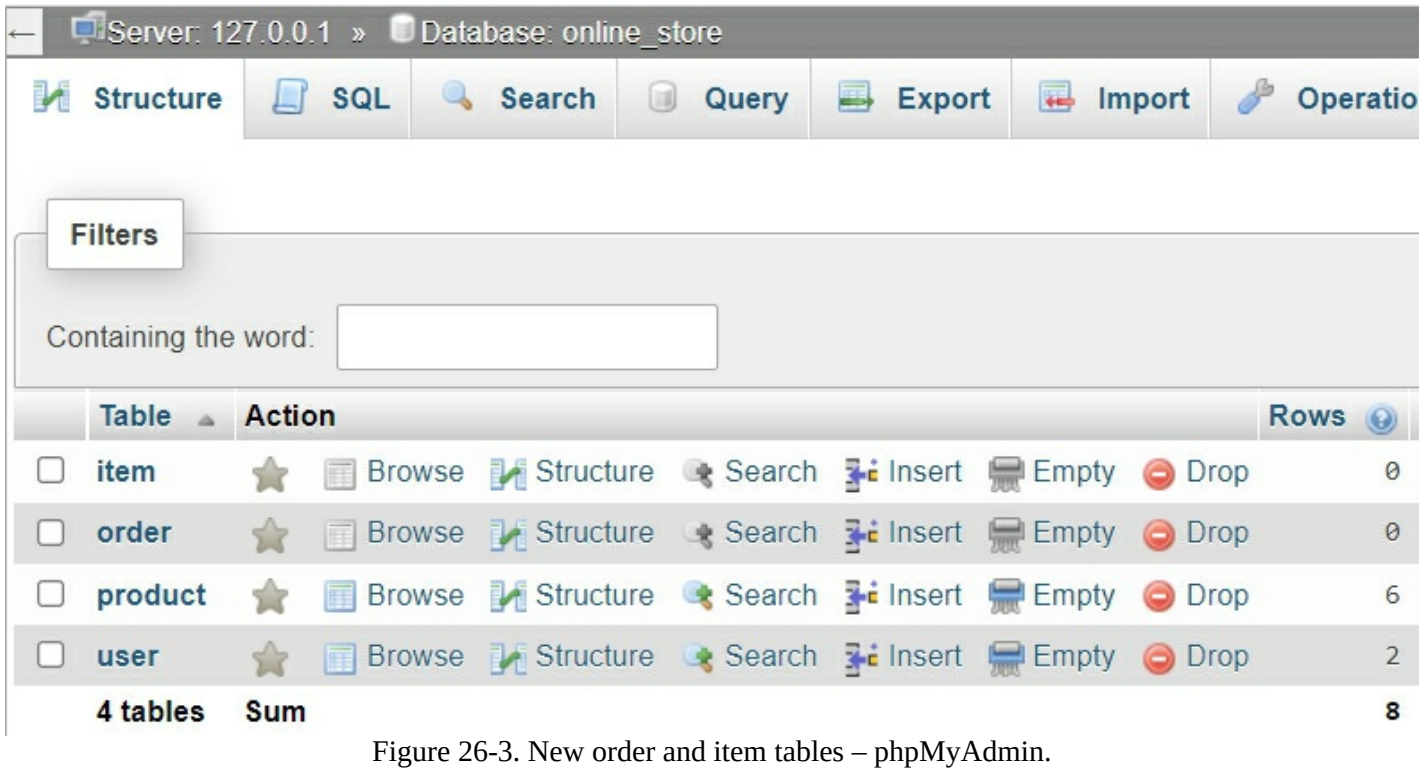

Figure 26-3. New order and item tables – phpMyAdmin.

Now, we are ready to implement the purchase system.

Chapter 27 – Product Purchase

Let’s make some changes to implement the product purchase.

Modifying UsersService

In *src/models/users.service.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
@Injectable()
export class UsersService {
  ...

  findOne(id: string): Promise<User> {
    return this.usersRepository.findOne(id);
  }

  updateBalance(id: number, balance: number) {
    return this.usersRepository.update(id, {balance: balance});
  }
}
```

We extend the *UsersService* capabilities, by including a *findOne* method that allows searching users by *id* . We also include a *updateBalance* method that updates the user *balance* based on the user *id* .

Creating OrdersService

In *src/models* , create a new file called *orders.service.ts* , and fill it with the following code.

Add Entire Code

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Order } from './order.entity';

@Injectable()
export class OrdersService {
  constructor(
    @InjectRepository(Order)
    private ordersRepository: Repository<Order>,
  ) {}

  createOrUpdate(order: Order): Promise<Order> {
    return this.ordersRepository.save(order);
  }
}
```

We implement a *OrdersService* class with a *createOrUpdate* method to save an *order* into the database.

Registering Order and OrdersService in AppModule

Let’s register *Order* and *OrdersService* in the application root module (*AppModule*). In *src/app.module.ts* , make the following changes in **bold**.

Modify Bold Code

```
…
import { UsersService } from './models/users.service';
import { OrdersService } from './models/orders.service';
import { Product } from './models/product.entity';
import { User } from './models/user.entity';
import { Order } from './models/order.entity';
import { AdminModule } from './admin/admin.module';
…

@Global()
@Module({
  imports: [
    TypeOrmModule.forRoot(),
    TypeOrmModule.forFeature([Product, User, Order]),
    AdminModule,
    AuthModule,
    CartModule,
  ],
  controllers: [AppController, ProductsController],
```

```
  providers: [ProductsService, UsersService, OrdersService],
  exports: [ProductsService, UsersService, OrdersService],
})
export class AppModule {}
```

We imported and configured the *OrdersService* and the *Order* entity. It will make the *OrdersService* and the TypeORM *Order Repository* available to be injected across all modules.

Modifying Order Entity

In *src/models/order.entity.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
@Entity()
export class Order {
  ...

  @ManyToOne(() => User, (user) => user.orders)
  user: User;

  @OneToMany(() => Item, (item) => item.order, {
    cascade: ['insert']
  })
  items: Item[];

  ...
```

We set the *cascade* option to enable *insert* in the *@OneToMany items* relation. This means that when we save the *order* instance, the array of *items* of that *order* will be saved too. We will illustrate this behavior later.

Modifying cart/index view

In *views/cart/index.hbs* , make the following changes in **bold**.

Modify Bold Code

```
...
        <div class="row">
          <div class="text-end">
                <a class="btn btn-outline-secondary mb-2"><b>Total to pay:</b> ${{viewData.total}}</a>
                {{#if viewData.productsInCart}}
                <a href="/cart/purchase" class="btn bg-primary text-white mb-2">Purchase</a>
                <a href="/cart/delete" class="btn btn-danger mb-2">
                    Remove all products from Cart
                </a>
                {{/if}}
          </div>
        </div>
…
```

We modified the *cart/index* view. We only show the *Purchase* button and the “Remove all products from Cart” button if the user has products in session. And we link the *Purchase* link with the (“cart/purchase”) route.

Cart purchase view

In *views/cart* , create a new file *purchase.hbs* and fill it with the following code.

Add Entire Code

```
{{#> app}}
  {{#*inline "content"}}
    <div class="card">
      <div class="card-header">
        Purchase Completed
      </div>
      <div class="card-body">
        <div class="alert alert-success" role="alert">
          Congratulations, purchase completed. Order number is <b>#{{viewData.orderId}}</b>
        </div>
      </div>
    </div>
  {{/inline}}
{{/app}}
```

It is a simple view that displays a message and shows the order id. This view will be displayed when the user completes the purchase.

Modifying CartController

In *src/cart/cart.controller.ts* , make the following changes in **bold**.

Modify Bold Code

```
import { Controller, Get, Render, Req, Redirect, Param,
   Body, Post, Res } from '@nestjs/common';
import { ProductsService } from '../models/products.service';
import { OrdersService } from '../models/orders.service';
import { UsersService } from '../models/users.service';
import { Product } from '../models/product.entity';
import { Order } from '../models/order.entity';
import { Item } from '../models/item.entity';

@Controller('/cart')
export class CartController {
   constructor(
      private readonly productsService: ProductsService,
      private readonly usersService: UsersService,
      private readonly ordersService: OrdersService,
   ) {}

   …

   @Get('/purchase')
   async purchase(@Req() request, @Res() response) {
      if (!request.session.user) {
         return response.redirect('/auth/login');
      } else if (!request.session.products) {
         return response.redirect('/cart');
      } else {
         const user = await this.usersService.findOne(request.session.user.id);
         const productsInSession = request.session.products;
         const productsInCart = await this.productsService.findByIds(
            Object.keys(productsInSession),
         );

         let total = 0;
         const items: Item[] = [];
         for (let i = 0; i < productsInCart.length; i++) {
            const quantity = productsInSession[productsInCart[i].getId()];
            const item = new Item();
            item.setQuantity(quantity);
            item.setPrice(productsInCart[i].getPrice());
            item.setProduct(productsInCart[i]);
            items.push(item);
            total = total + productsInCart[i].getPrice() * quantity;
         }
         const newOrder = new Order();
         newOrder.setTotal(total);
         newOrder.setItems(items);
         newOrder.setUser(user);
         const order = await this.ordersService.createOrUpdate(newOrder);

         const newBalance = user.getBalance() - total;
         await this.usersService.updateBalance(user.getId(), newBalance);

         request.session.products = null;

         const viewData = [];
         viewData['title'] = 'Purchase - Online Store';
         viewData['subtitle'] = 'Purchase Status';
         viewData['orderId'] = order.getId();
         return response.render('cart/purchase', { viewData: viewData });
      }
   }
}
```

Let’s analyze the previous method by parts.

Analyze Code

```
   @Get('/purchase')
   async purchase(@Req() request, @Res() response) {
      if (!request.session.user) {
         return response.redirect('/auth/login');
      } else if (!request.session.products) {
         return response.redirect('/cart');
      } else {
```

We define the *purchase* method that is the most complex in this book. In the beginning, we check if there is user information in session (the user is logged in), if there is not, we redirect the user to the ("/auth/login") route. Then, we check if the user has products in session. If there are no products in session, we redirect the user to the ("/cart") route.

Analyze Code

```
const user = await this.usersService.findOne(request.session.user.id);
const productsInSession = request.session.products;
const productsInCart = await this.productsService.findByIds(
  Object.keys(productsInSession),
);


let total = 0;
const items: Item[] = [];
```

If the user is logged in and there are products in session, we start the purchase process. We extract the user information from the *user* variable using the request.session.user.id data. We extract the products info from the session and extract the corresponding products from the database. Then, we define a *total* variable with a value of *0* and create an empty array of items.

Analyze Code

```
for (let i = 0; i < productsInCart.length; i++) {
  const quantity = productsInSession[productsInCart[i].getId()];
  const item = new Item();
  item.setQuantity(quantity);
  item.setPrice(productsInCart[i].getPrice());
  item.setProduct(productsInCart[i]);
  items.push(item);
  total = total + productsInCart[i].getPrice() * quantity;
}
```

We iterate through the *productsInCart* array. For each product in *productsInCart* , we create a new *Item* , and set the corresponding *quantity* (based on the values stored in session), *price* , and *product* . We then add the item to the *items* array, and update the *total* value.

Analyze Code

```
const newOrder = new Order();
newOrder.setTotal(total);
newOrder.setItems(items);
newOrder.setUser(user);
const order = await this.ordersService.createOrUpdate(newOrder);
```

We create a *newOrder* instance, set *total* , *items* , *user* and store it in the database. Since we set to *newOrder* an array of items with *cascade insert* , the order and all its items will be added into the database.

Analyze Code

```
const newBalance = user.getBalance() - total;
await this.usersService.updateBalance(user.getId(), newBalance);


request.session.products = null;


const viewData = [];
viewData['title'] = 'Purchase - Online Store';
viewData['subtitle'] = 'Purchase Status';
viewData['orderId'] = order.getId();
return response.render('cart/purchase', { viewData: viewData });
```

We calculate and set the new user's *balance* . We then remove the products in session and show the *cart/purchase* view with *orderId* .

Note: we have not verified if the user has enough money to purchase. Try to include that validation in the previous code. You can use the discussion zone of the book repository to show us your solution.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now, log in to the application, go to the ("/products") route, and add some products to the cart (see Fig. 27-1). Click the *Purchase* button and the application will show a confirmation message (see Fig. 27-2). Next, you can open phpMyAdmin and check that the *orders* table (see Fig 27-3), and *items* table (see Fig. 27-4) contain the new data.

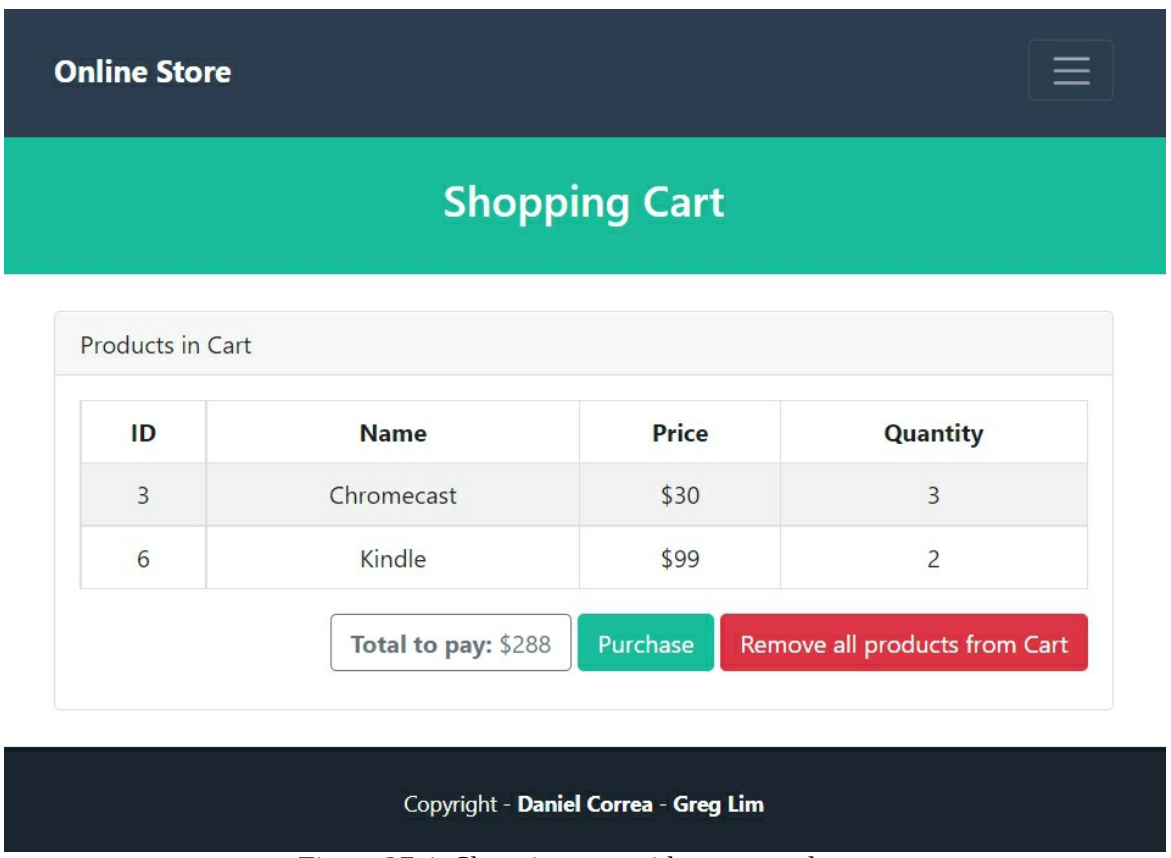

Figure 27-1. Shopping cart with some products.

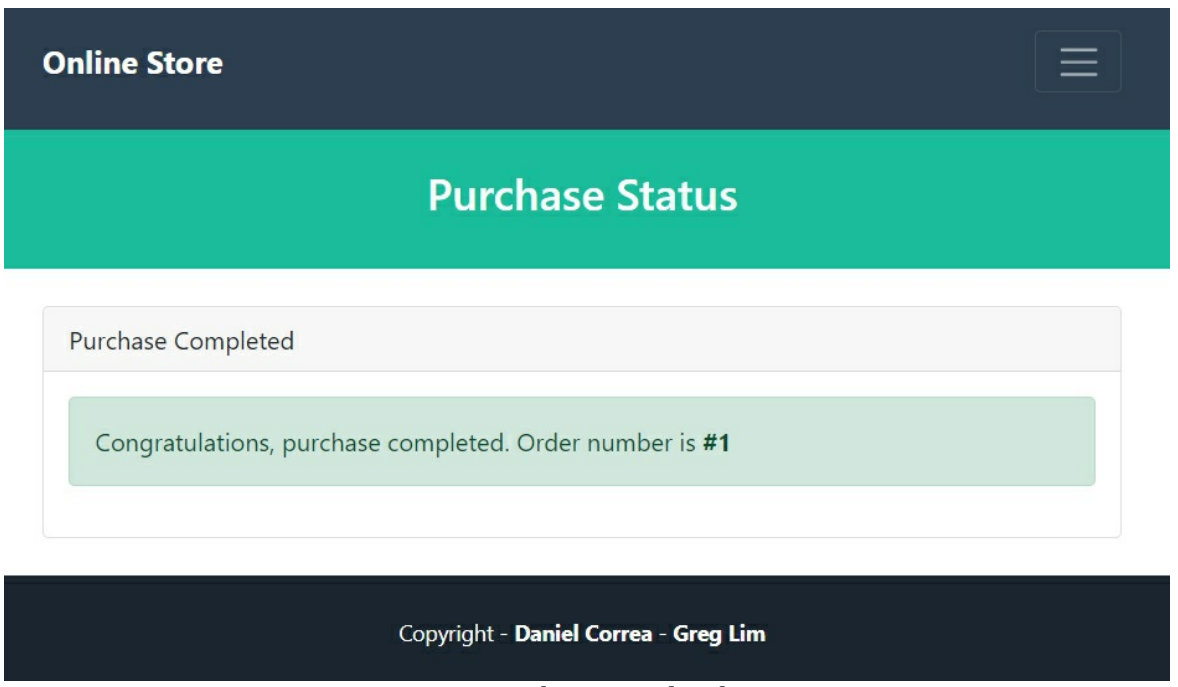

Figure 27-2. Purchase completed message.

id	total	date	userId
1	288	2022-01-13 14:51:41.423611	1

Figure 27-3. New order row included in the database.

id	quantity	price	orderId	productId
1	3	30	1	3
2	2	99	1	6

Figure 27-4. New item rows included in the database.

Chapter 28 – Orders Page

Let’s finish our Online Store application. We will create a page where users can list their orders.

Account Module

The *AccountModule* will be used to manage user accounts. In this book, we will only implement listing user orders, but in real-world applications, you can extend this module to enable users to modify their information or set preferences. Let’s begin.

Creating AccountController

In *src/* , create a subfolder called *account* . Then, in *src/account* create a new file *account.controller.ts* and fill it with the following code.

Add Entire Code

```
import { Controller, Get, Render, Req } from '@nestjs/common';
import { OrdersService } from '../models/orders.service';

@Controller('/account')
export class AccountController {
  constructor(private readonly ordersService: OrdersService) {}

  @Get('/orders')
  @Render('account/orders')
  async orders(@Req() request) {
    const viewData = [];
    viewData['title'] = 'My Orders - Online Store';
    viewData['subtitle'] = 'My Orders';
    viewData['orders'] = await this.ordersService.findByUserId(
      request.session.user.id,
    );
    return {
      viewData: viewData,
    };
  }
}
```

We have an *AccountController* with an *orders* method linked to the (“/account/orders”) route. This is a simple method that searches the orders of the logged in user (based on the *request.session.user.id* data). We use *this.ordersService.findByUserId* method to extract those orders (will be implemented later). Finally, we render the account/orders view.

Modifying Orders Service

In *src/models/orders.service.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
@Injectable()
export class OrdersService {
  ...

  findByUserId(id: number): Promise<Order[]> {
    return this.ordersRepository.find({
      where: {
        user: { id: id },
      },
      relations: ['items', 'items.product'],
    });
  }
}
```

We include the *findByUserId* method. This method receives a user *id* and finds the *orders* related to that user *id* . Note that we pass a *relations* option to the TypeORM *Repository find* method. The *relations* option is used to inform the repository that we need to extract not only the *orders* information, but also the *items* related to those *orders* , and even *products* related to the *items* related to the *orders* . This is the power of the TypeORM relations. If properly implemented, it allows us to navigate across our class diagram. We will navigate across these relations later in the orders view.

Creating Account Module

In *src/account* create a new file *account.module.ts* and fill it with the following code.

Add Entire Code

```
import { Module } from '@nestjs/common';
import { AccountController } from './account.controller';


@Module({
  controllers: [AccountController],
})
export class AccountModule {}
```

We created *AccountModule* and registered *AccountController* .

Modifying AppModule

Let's register *AccountModule* in the application root module (*AppModule*). In *src/app.module.ts* , make the following changes in **bold**.

Modify Bold Code

```
…
import { AuthModule } from './auth/auth.module';
import { CartModule } from './cart/cart.module';
import { AccountModule } from './account/account.module';

@Global()
@Module({
  imports: [
    TypeOrmModule.forRoot(),
    TypeOrmModule.forFeature([Product, User]),
    AdminModule,
    AuthModule,
   CartModule,
    AccountModule,
  ],
  …
```

Account orders view

In *views/* , create a subfolder called *account* . Then, in *views/account* create a new file *orders.hbs* and fill it with the following code.

Add Entire Code

```
{{#> app}}
  {{#*inline "content"}}
    {{#each viewData.orders}}
    <div class="card mb-4">
      <div class="card-header">
        Order #{{getId}}
      </div>
      <div class="card-body">
        <b>Date:</b> {{getDate}}<br />
        <b>Total:</b> ${{getTotal}}<br />
        <table class="table table-bordered table-striped text-center mt-3">
          <thead>
            <tr>
              <th scope="col">Item ID</th>
              <th scope="col">Product Name</th>
              <th scope="col">Price</th>
              <th scope="col">Quantity</th>
            </tr>
            <tbody>
              {{#each getItems}}
                <tr>
                  <td>{{getId}}</td>
                  {{#with getProduct}}
                  <td>
                    <a class="link-success" href="/products/{{getId}}">
                      {{getName}}
                    </a>
                  </td>
                  {{/with}}
                  <td>${{getPrice}}</td>
                  <td>{{getQuantity}}</td>
                </tr>
              {{/each}}
            </tbody>
      </thead>

    </table>
```

```
        </div>
      </div>
      {{else}}
      <div class="alert alert-danger" role="alert">
        Seems to be that you have not purchased anything in our store =(.
      </div>
      {{/each}}
    {{/inline}}
{{/app}}
```

We use a *each* hbs helper to iterate through the *viewData.orders* . If the user has no orders, we use the *else* hbs helper to show a message saying "you have not purchased anything". Otherwise, we show each order's *date* and *total* (using the respective getters). Then, we have a second *each* helper. This time, we iterate through the items of each order (using *getItems*). We show each item's *id* , *price* , and *quantity* . Finally, we use the *with* hbs helper to access the product properties of each item (using *getProduct*) and display the product *name* and link to the respective product page.

Updating link in app layout

Now that we have the proper orders route, let's include it in the *app* layout. In *views/layouts/app.hbs* , make the following changes in **bold**.

Modify Bold Code

```
<!doctype html>
            ...
            {{#if session.user}}
            <a class="nav-link active" href="/account/orders">My Orders</a>
            <a class="nav-link active" href="/auth/logout">Logout ({{session.user.name}})</a>
            {{else}}
            <a class="nav-link active" href="/auth/login">Login</a>
            <a class="nav-link active" href="/auth/register">Register</a>
            {{/if}}
            ...
```

We added a new link to My Orders page.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now, log in to the application, go to the ("/account/orders") route, and you will see your orders (see Fig. 28-1).

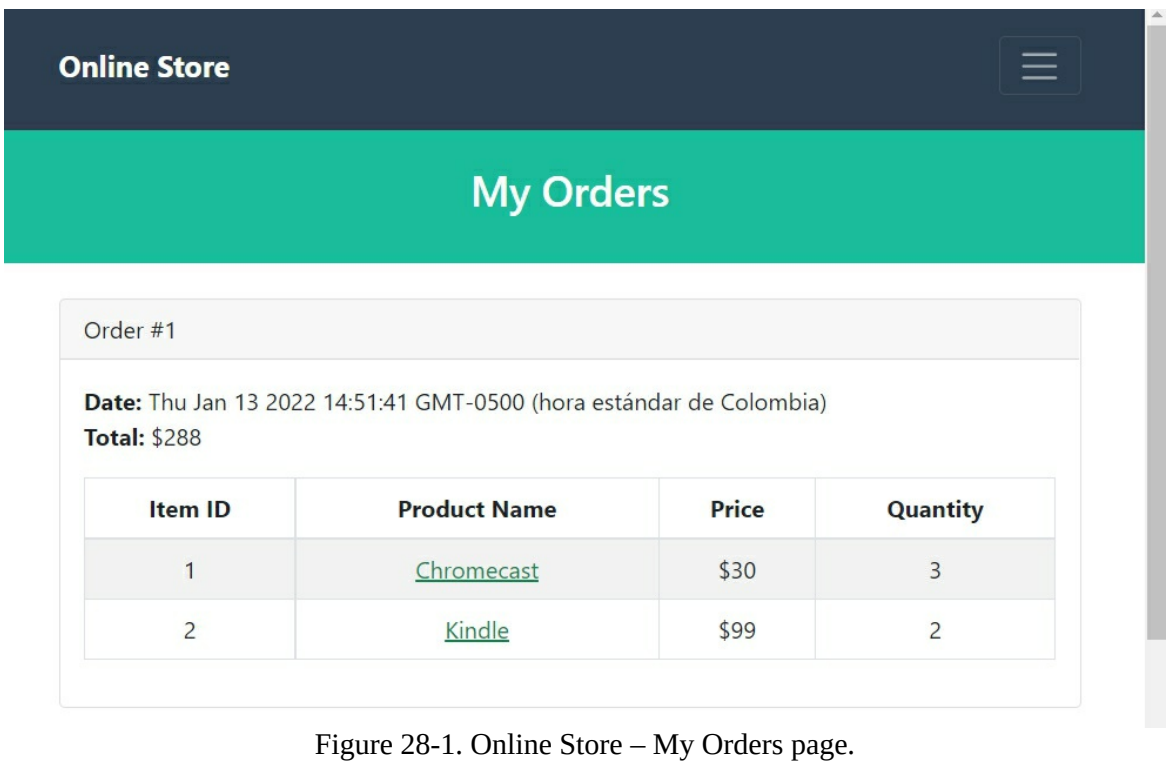

Figure 28-1. Online Store – My Orders page.

Refactoring My Orders page

The current date format looks messy. Another problem is that anyone (logged user or not) can access the ("/account/orders") route (by placing that route in the browser). Let's fix these two issues before completing our application.

Modifying Order Entity

In *src/models/order.entity.ts* , make the following changes in **bold**.

Modify Bold Code

```
...
@Entity()
export class Order {
   ...

   getDate(): string {
      return this.date.toISOString().split('T')[0];
   }

   ...
```

We modify the *getDate* method to return a *string* , and use the *toISOString* method to return a date object as a string using the ISO standard(YYYY-MM-DDTHH:mm:ss). We remove the time portion (THH:mm:ss) of that standard.

Modifying main.ts

In *src/main.ts* , make the following changes in **bold**.

Modify Bold Code

```
   ...
   });
   app.use('/account*', function (req, res, next) {
      if (req.session.user) {
         next();
      } else {
         res.redirect('/');
      }
   });

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

We create a new middleware which applies for routes that begins with the (“/account”) string (all our account routes). We check if there is *a req.session.user* property. If true, it means that the current user is logged in and can access the current route. We then call the *next()* method to continue to the next middleware. Otherwise, redirect the user to the home page.

Running the app

In the Terminal, go to the project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now, log in to the application, go to the (“/account/orders”) route, and you will see your orders with the new date format (see Fig. 28-2). Again, this illustrates the usefulness of getters in action.

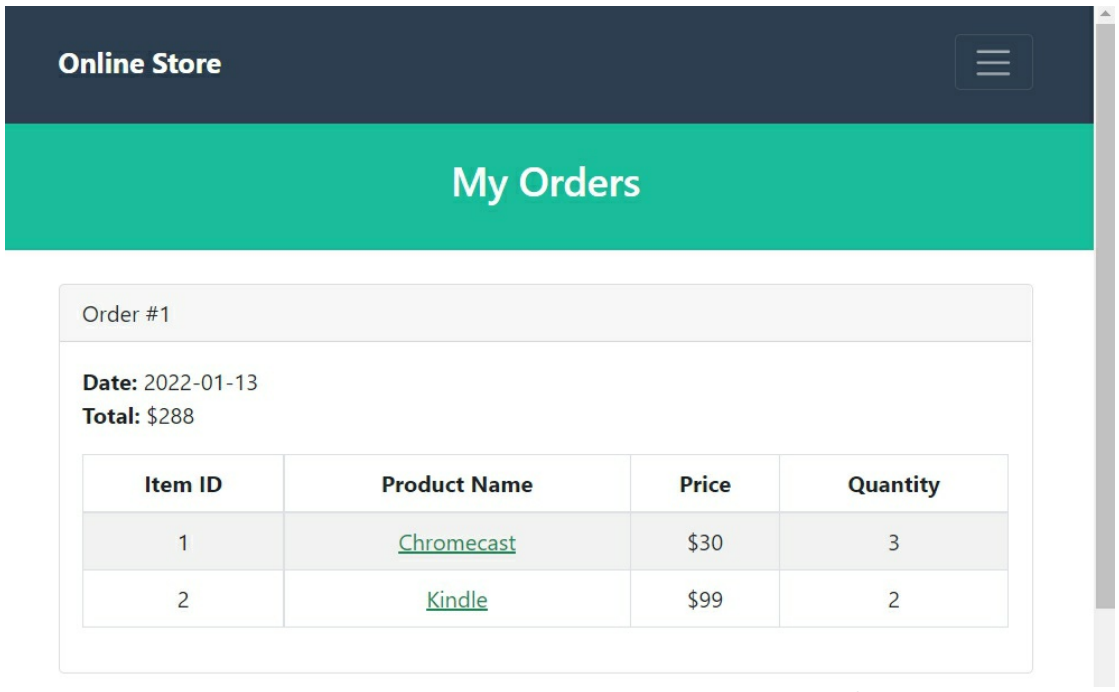

Figure 28-2. Online Store – Orders page with new date format.

We have finished our application. Congratulations! We hope you now better understand our architecture diagram and its elements (see Fig. 28-3).

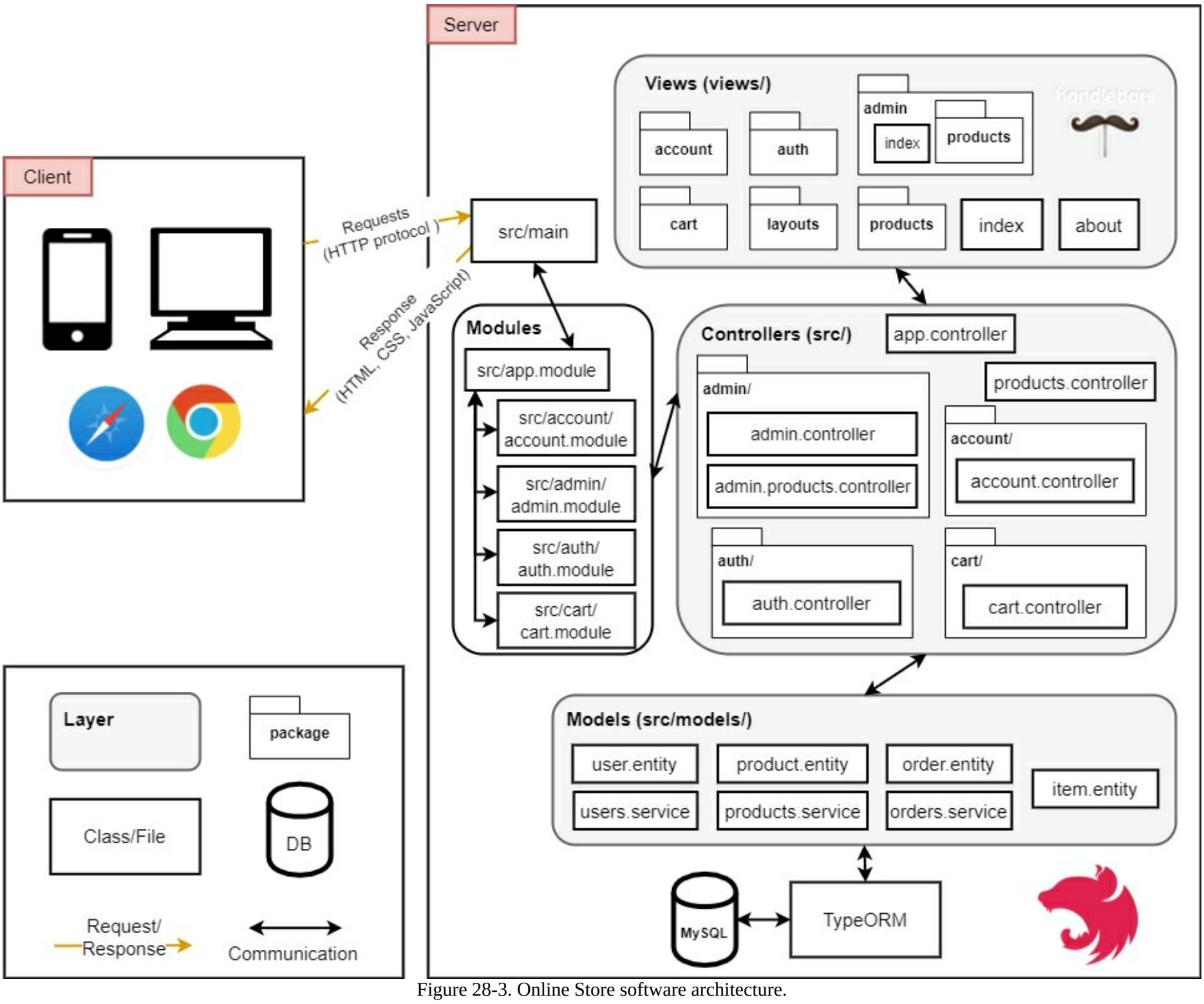

Figure 28-3. Online Store software architecture.

Let’s have a quick final analysis.

- We integrated MVC into our software architecture.
- We have proper consistency between the architecture diagram, class diagram, and the developed code. For example, the architecture diagram shows four entities, corresponding to the class diagram classes, and those entities are coded adequately into the *src/models* folder.
- The diagram serves to understand how the application is connected, how each layer or file invokes other layers or files. You can even deduce some architectural rules from that diagram. For example, you cannot invoke views or models from a module file. Or it is not allowed to invoke views from a model file. This is useful to keep our application code clean and flexible to evolving changes.

TIP: Use what you learnt from the diagram to complete your architectural rules document. You can use some of the rules described. Share the architecture and class diagrams with your colleagues. For example, store these documents in a repository that is easily found and accessible.

What an exciting journey! We have completed our Online Store with an MVC architecture. Let’s have a final tip and discussion, before deploying our application to the cloud.

TIP: A final quote from the (*2019 - Thomas, D., & Hunt, A. - The Pragmatic Programmer: your journey to mastery*) book. “Critically Analyze What You Read and Hear”. That video you saw on YouTube, that post you read in StackOverFlow, this paragraph that you are reading in this book. Does it make sense? Is it a good strategy? Does it apply to your context? Is it true?

Quick discussion: Have you seen the “Harry Potter and the Half-Blood Prince” film? In the film, Harry Potter was enrolled in a potions class. The class had an “Advanced Potion-Making” textbook which contained a variety of recipes for various potions. Harry had not bought his own textbook. So, the potions professor loaned one of the older books (left behind by previous

students) to Harry. This book belonged to Severus Snape whose nickname was the "Half-Blood Prince". Snape improved many of the potions by means of including procedures and scribbling notes in the margins. Harry tried Snape's methods and achieved the best results in the class. We all should be like the Half-Blood Prince, trying to critically analyze everything and make our own judgements. Book authors make mistakes, experts make mistakes, we all make mistakes. There is still room for improvement.

Chapter 29 – Deploying to the Cloud – Clever-Cloud – MySQL Database

Now, we will learn how to deploy our application to the cloud using a free cloud server.

Introduction to Clever-Cloud

Clever Cloud is a ‘Platform as a Service’ company that helps developers deploy and run their apps over the cloud. In addition, Clever Cloud offers free service trials, such as the MySQL service. With this service, we will deploy, run, and monitor our MySQL databases and tables.

Creating a Clever Cloud account

Let’s create a Clever Cloud account. Go to https://api.clever-cloud.com/v2/session/signup and complete the “Sign up with Email” information (see Fig. 29-1).

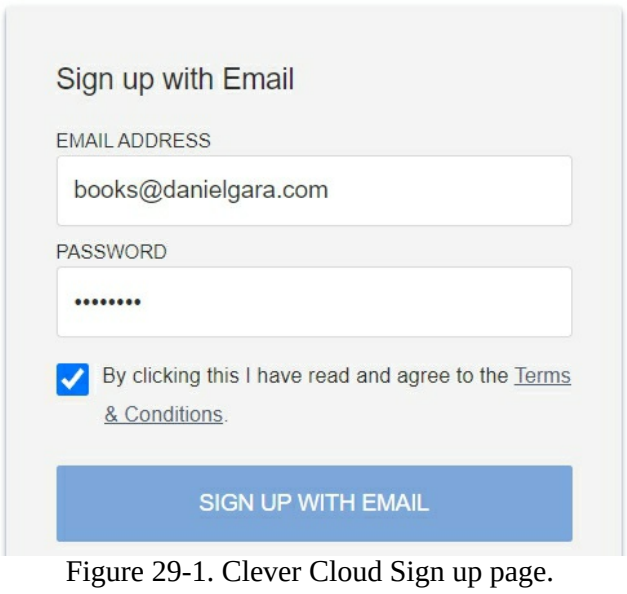

Figure 29-1. Clever Cloud Sign up page.

Validate your email. After that, access https://console.clever-cloud.com/, click *Personal space* , *Create…* , and click *an add-on* . After that, select *MySQL* , choose the *DEV plan* option (the first one), which costs 0.00€, and click *next* . Then, put *onlineStore* as the add-one name, leave the region as it appears, and click *next* (see Fig. 29-2). We have created our MySQL cloud service.

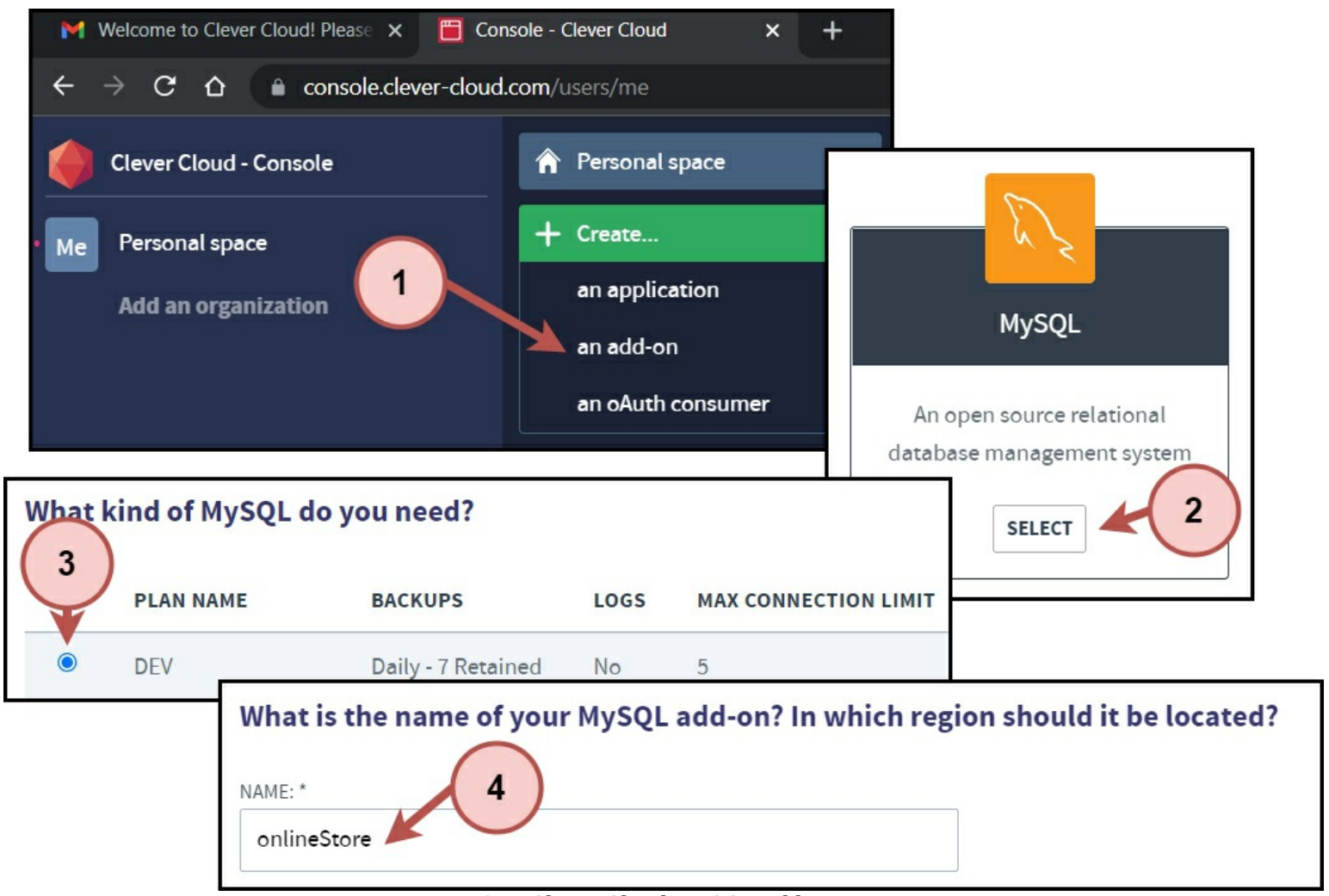

Figure 29-2. Clever Cloud MySQL add-on creation.

After the MySQL creation, we will see a dashboard (see Fig. 29-3), which contains our “Database Credentials” and a link to “phpMyAdmin”. We will use this information next.

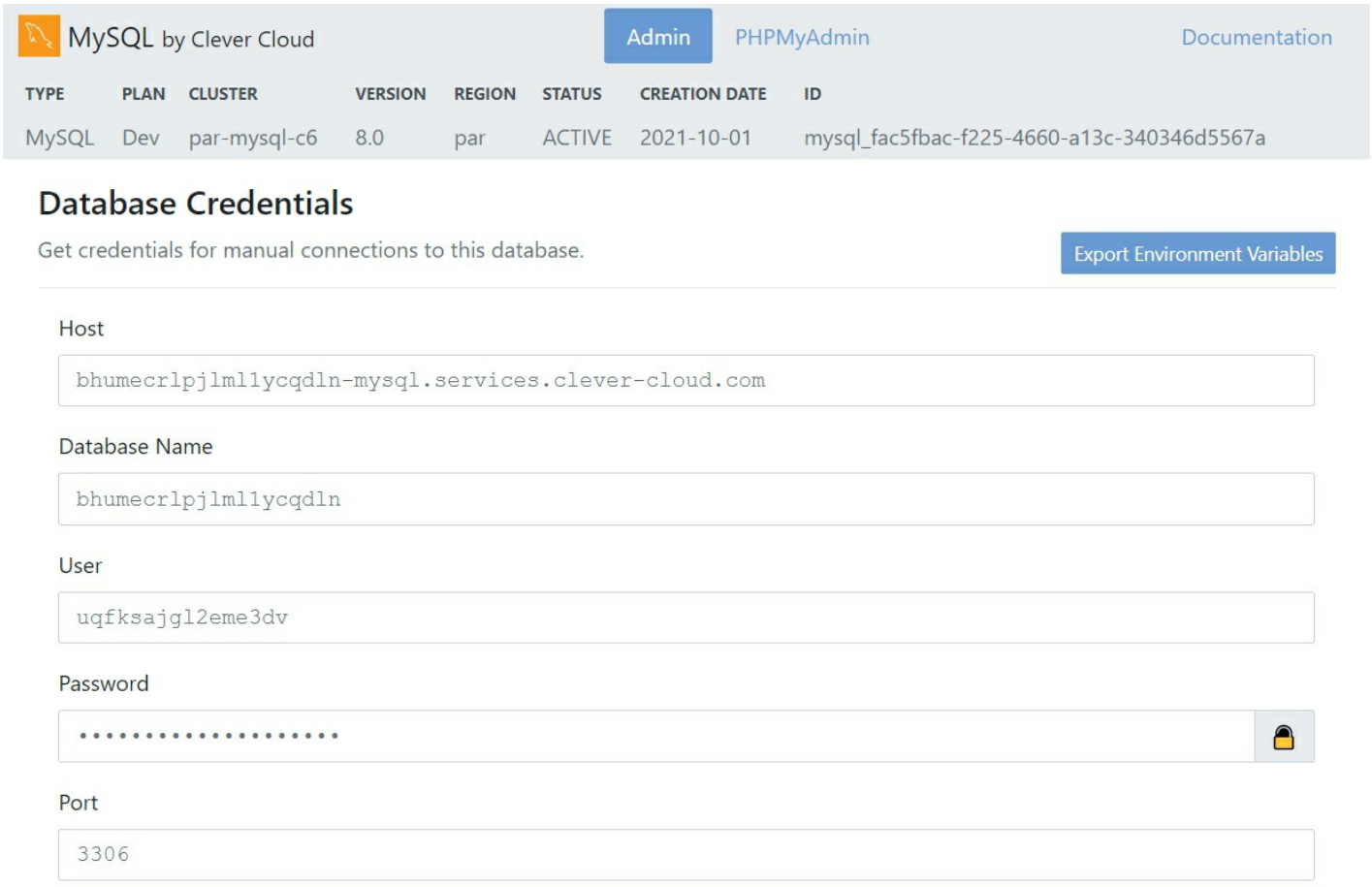

Figure 29-3. Clever Cloud MySQL add-on creation.

Note: if for any reason, Clever Cloud disables the *DEV free plan* , just create a new topic in the discussion zone of the book repo, and we will make a tutorial with an alternative platform.

Cloning our application

Let’s make a copy of our application code. Copy all the content of your *online-store* folder to a new folder called *online-store-cloud* (in a location of your choice).

Modifying the .env file

To execute the database queries over our cloud database, we need to modify the *.env* file (located at the cloud project root folder).

Go to your *online-store-cloud* folder, and in the *ormconfig.json* file, make the following changes in **bold**. Replace the *DATABASE_CREDENTIALS_HOST* , *DATABASE_CREDENTIALS_DATABASE_NAME* , *DATABASE_CREDENTIALS_USER* , and *DATABASE_CREDENTIALS_PASSWORD* with your values (from the “Clever Cloud addon dashboard”, presented previously).

Modify Bold Code with Your Values

```
{
   "type": "mysql",
      "host": "DATABASE_CREDENTIALS_HOST",
   "port": 3306,
   "username": "DATABASE_CREDENTIALS_USER",
   "password": "DATABASE_CREDENTIALS_PASSWORD",
   "database": "DATABASE_CREDENTIALS_DATABASE_NAME",
   "entities": ["dist/**/*.entity{.ts,.js}"],
   "synchronize": true
}
```

You will get something like the following.

Analyze Code

```
{
      "type": "mysql",
      "host": "b6ejrgireod7ueb9yryu-mysql.services.clever-cloud.com",
      "port": 3306,
      "username": "u1ta1it4vk9ffz38",
      "password": "Vz0QqTGQaZ5fmAa8n…",
      "database": "b6ejrgireod7ueb9yryu",
      "entities": ["dist/**/*.entity{.ts,.js}"],
      "synchronize": true
}
```

Synchronizing the database

Let’s synchronize our database. In the Terminal, go to the cloud project directory, and execute the following:

Execute in Terminal

```
npm run start:dev
```

Now, if you go to the Clever Cloud “phpMyAdmin” tab, you will see the tables properly created (see Fig. 29-4).

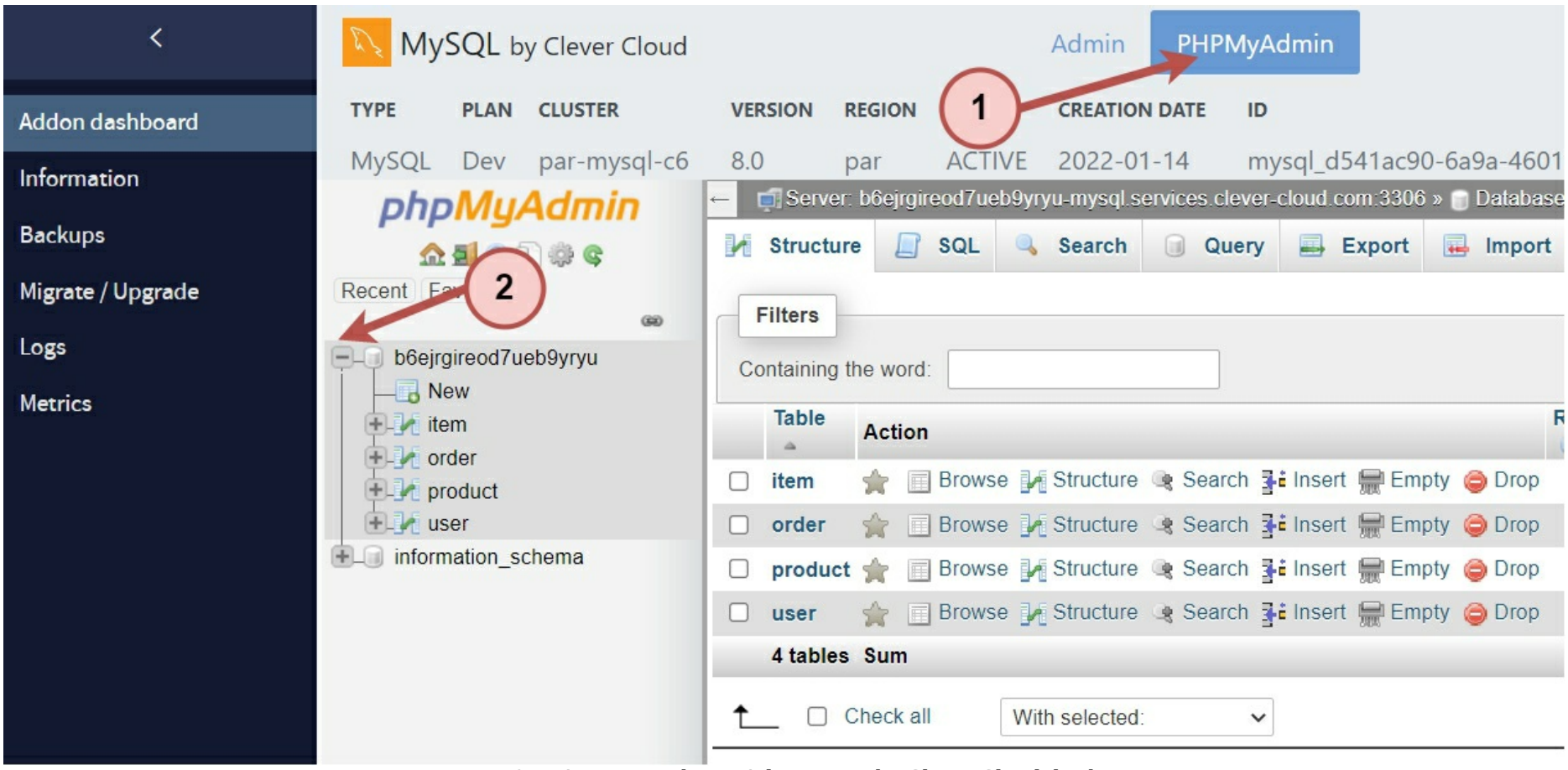

Figure 29-4. Accessing phpMyAdmin over the Clever Cloud database.

Exporting and Importing our data

We currently have empty tables. Let’s copy the data from our local database to our cloud database. Open “phpMyAdmin” from your local computer. Go to the *online_store* database, go to the *Export* tab, and select “export method” as *Custom* . In tables, deselect all the *Structure* options, and in *Data* , select all. Finally, click *Go* (see Fig. 29-5).

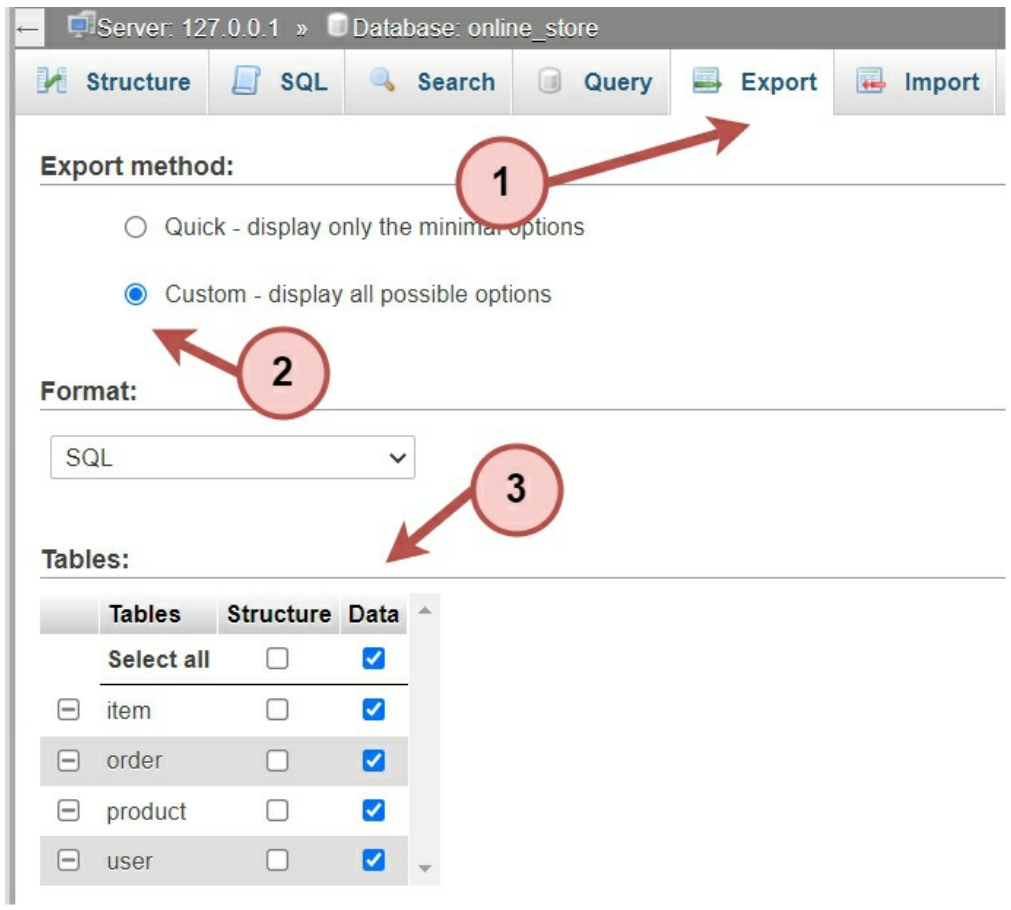

Figure 29-5. Exporting the data from the local database.

You will get a SQL file. This file contains our application data. Then, go to the cloud database, go to “phpMyAdmin”, go to your cloud database, click the *Import* tab, select the SQL file, deselect the *Enable foreign key checks* option, and click *Go* (see Fig. 29-6).

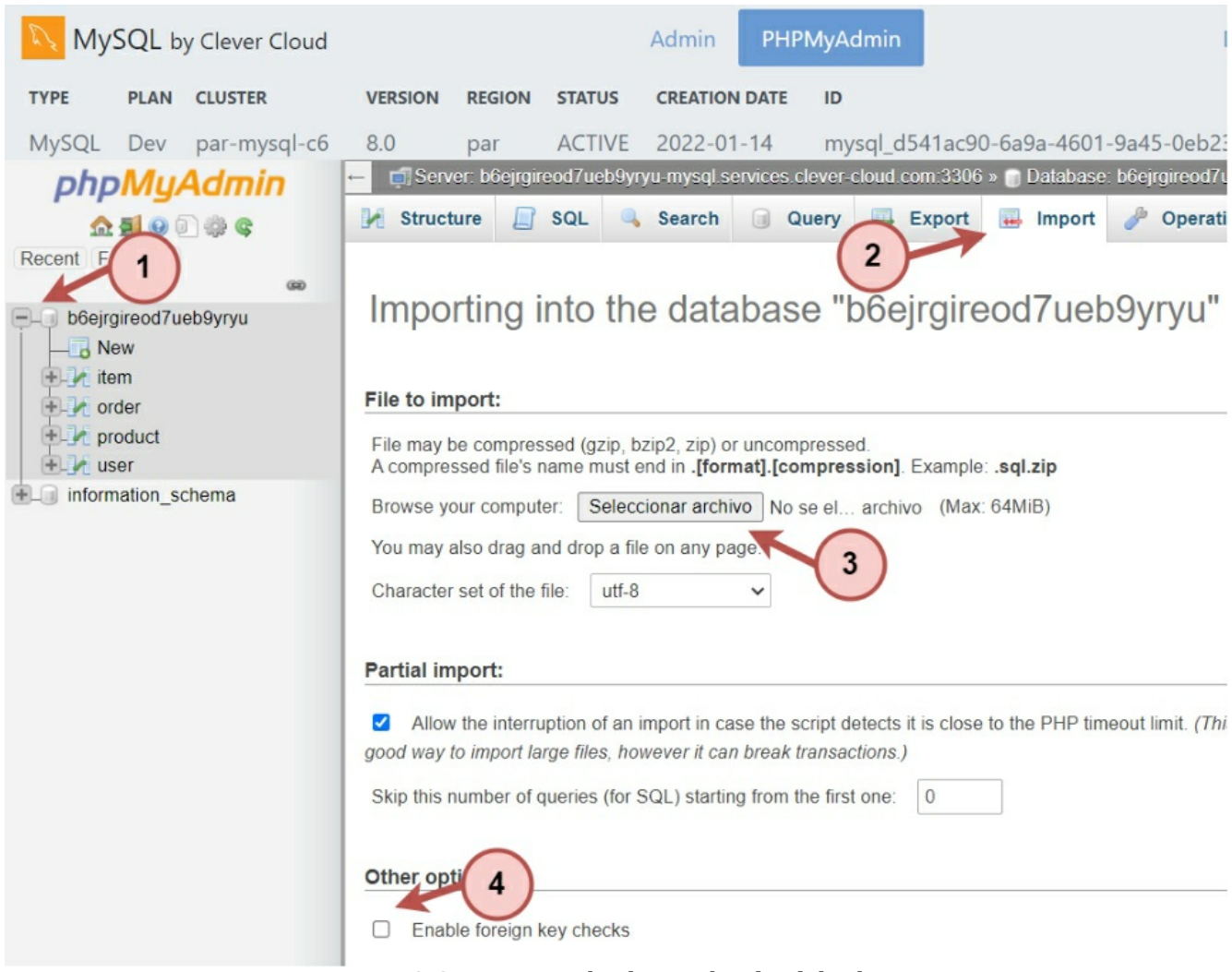

Figure 29-6. Importing the data to the cloud database.

Congratulations, we have our database and data in the cloud. You can check that your data was successfully uploaded by opening the *product* or *user* table (see Fig. 29-7).

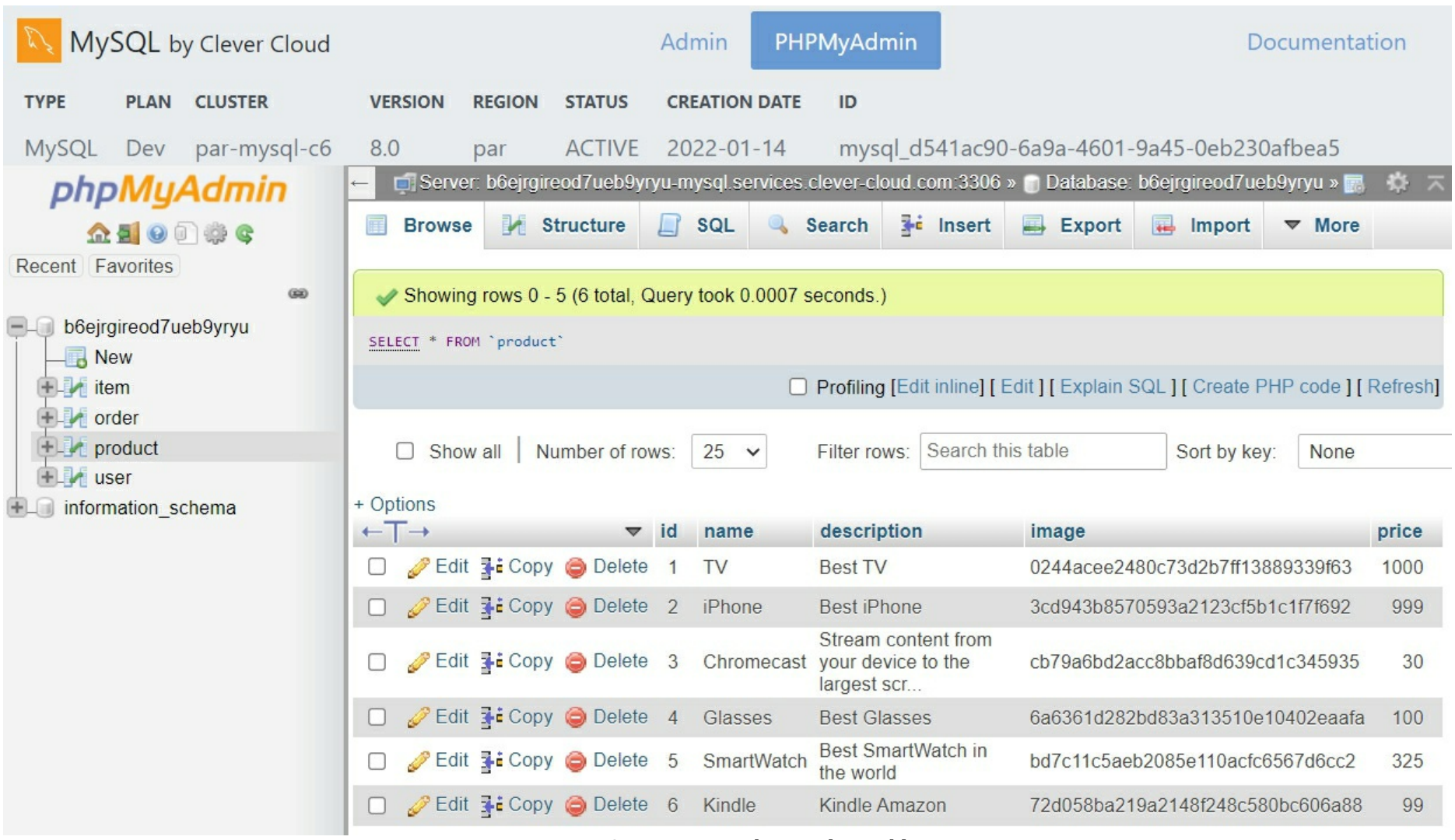

Figure 29-7. Browsing the *product* table.

Chapter 30 – Deploying to the Cloud – Heroku – Nest Application

Now, let’s deploy our Nest code over the cloud.

Creating a Heroku account

Let’s create a Heroku account. First, access https://signup.heroku.com/, complete the information, and click “Create Free Account”. Next, confirm your email address and set up your password.

Installing Git

Install git by following the instruction in this link: https://git-scm.com/.

Initializing a Git repository

Go to the **cloud project** directory, and in the Terminal, execute the following commands.

Execute in Terminal

```
git init
git add .
git commit -m "new nest project"
```

The commands initialize a Git repository and commit the current state.

Installing Heroku CLI

The Heroku Command Line Interface (CLI) makes it easy to create and manage our Heroku apps directly from the terminal. Follow the installation instructions from this link: https://devcenter.heroku.com/articles/heroku-cli.

Deploying to Heroku

Procfile

To deploy our application to Heroku, you must create a *Procfile* file, which tells Heroku what settings to use to launch the web server. In the cloud project root folder, create a new file called *Procfile* (without any extension), and fill it with the following code.

Add Entire Code

```
web: npm run start:prod
```

Heroku executes the command in *start:prod* script defined in *package.json* .

Modifying main.ts

In *src/main.ts* , make the following changes in **bold**.

Modify Bold Code

```
…
  await app.listen(process.env.PORT || 8080);
}
bootstrap();
```

When we developed the Nest application, we used the port 3000, but this won't work with Heroku. So, we need to use an environment variable (if defined) or the classic port 8080.

Modifying package.json

Heroku needs to know the version of your Node.js runtime to be able to run it on Heroku. You can check for your node version by running the following command in your terminal:

Execute in Terminal

```
node -v
```

In our case, it showed *v16.13.1* . Based on that, we need to modify our *package.json* . In *package.json* , make the following changes in **bold**.

Modify Bold Code

```
  …
    "testEnvironment": "node"
  },
```

```
  "engines": {
    "node": "16.x"
  }
}
```

We set the node engine to 16.x, meaning it will take one of the node 16.x versions.

Committing changes

Since we added a new file and modified others, we need to include them in a new commit. So, go to the cloud project directory, and in the Terminal, execute the following commands.

Execute in Terminal

```
git add .
git commit -m "configuring app for heroku deployment"
```

Creating a new application on Heroku

To create a new Heroku application, we use the heroku create command. Go to the **cloud project** directory, and in the Terminal, execute the following command.

Execute in Terminal

```
heroku create
```

This command creates a Heroku application. Sometimes, the command will open a browser tab to prompt us to log in to the Heroku website (if that is your case, please complete the log-in process). At the end, it provides a link with our new Heroku application deployed in the cloud (see Fig. 30-1).

```
Logged in as books@danielgara.com
Creating app... done, ⬢ infinite-stream-65085
https://infinite-stream-65085.herokuapp.com/ | https://git.heroku.com/infinite-stream-65085.git
```

Figure 30-1. Creating a Heroku application.

If you access the link (in our case it was https://infinite-stream-65085.herokuapp.com/), you will see a default application with a welcome message (see Fig. 30-2).

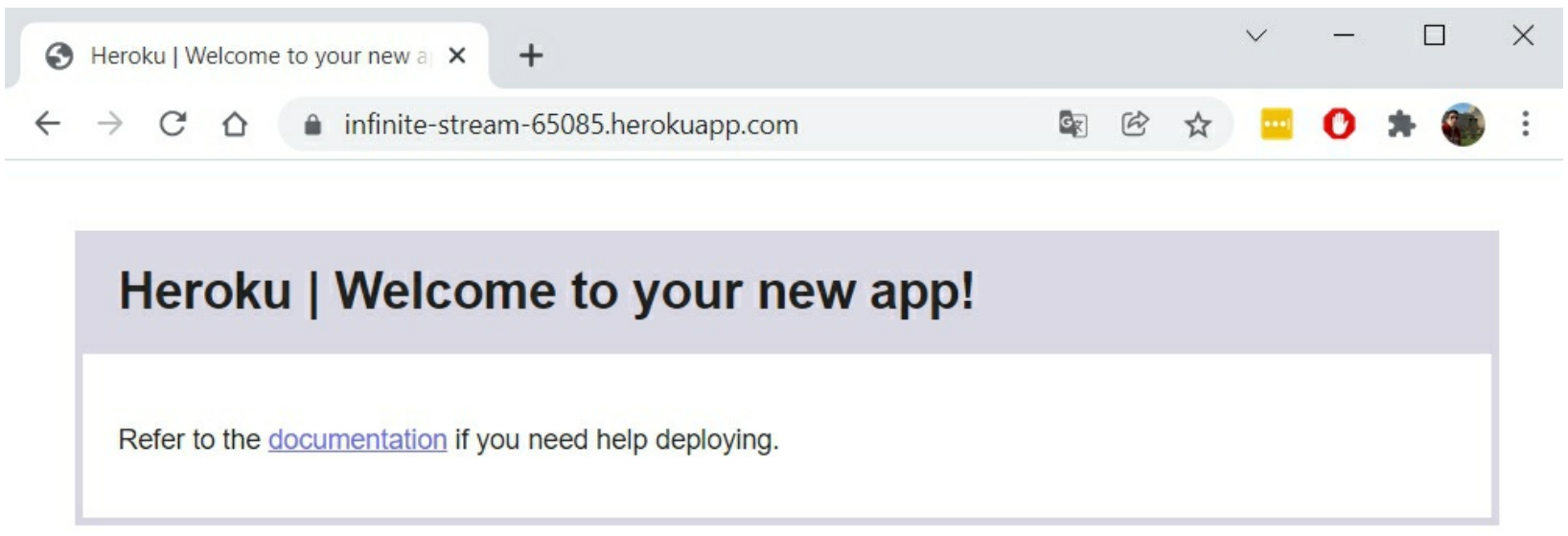

Figure 30-2. Accessing the Heroku application.

Pushing to Heroku

Next, it’s time to deploy our application to Heroku. Go to the cloud project directory, and in the Terminal, execute the following command.

Execute in Terminal

```
git push heroku master
```

Open your application with the following command.

Execute in Terminal

```
heroku open
```

You will see the complete Online Store application running over the cloud (see Fig. 30-3).

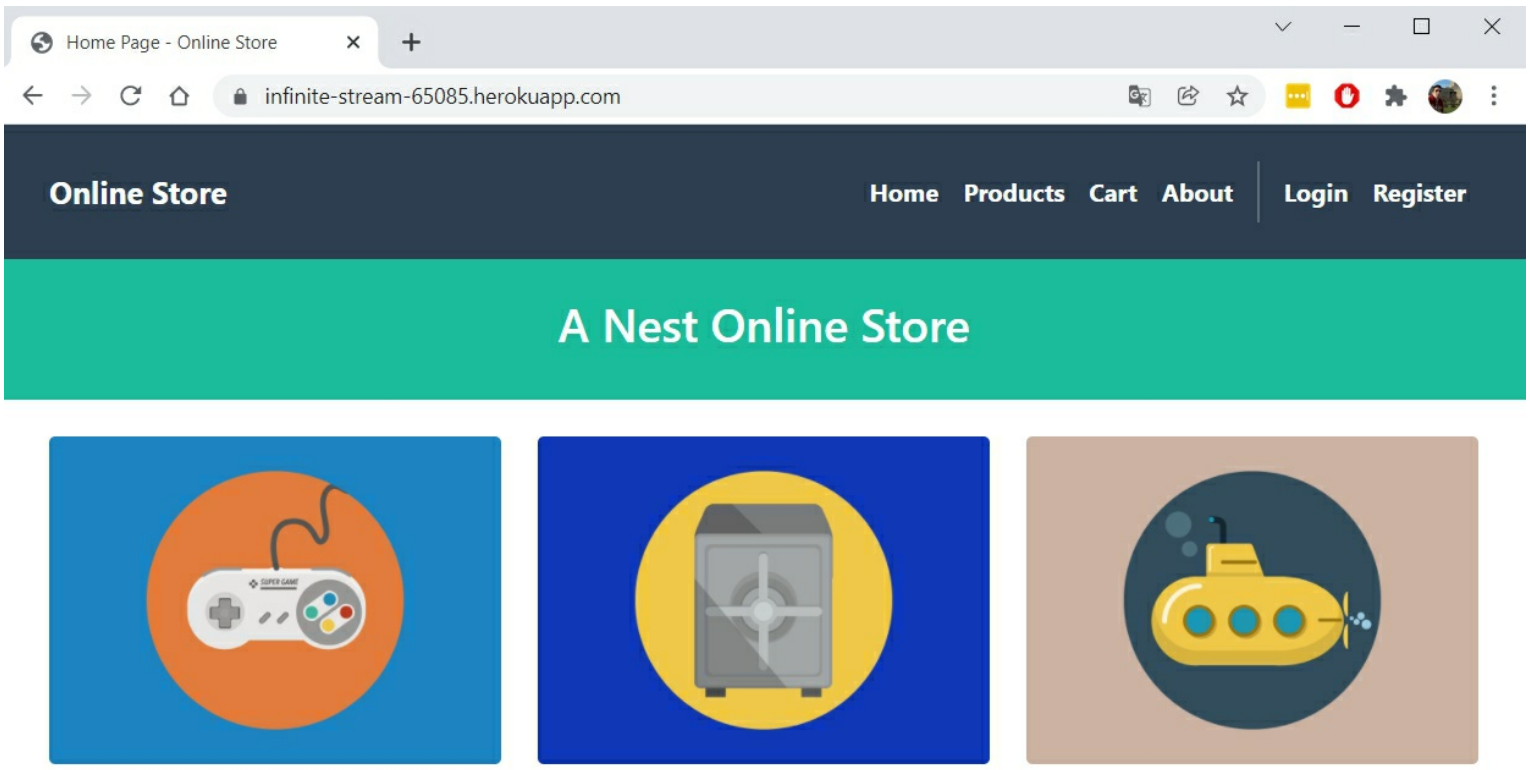

Figure 30-3. Online Store running over the cloud.

Note: If you get a 500 error, it means you didn't configure your database credentials properly.

Chapter 31 – Continue your Nest Journey

We have learned a lot since we started. We took a practical journey to master the design and implementation of MVC web applications with Nest. We developed a real-world project in which we applied a lot of different Nest functionalities. We purposely made code mistakes, but we fixed it and learned some practical tips about clean code (not only for Nest projects but for any web MVC project).

There are missing elements we didn't address in this book. So, we will show you additional interesting links and concepts of Nest where you can learn independently. We will also give you some book recommendations.

Other Nest interesting features

- **Testing.** Do you want to test your Nest application? Check Nest testing https://docs.nestjs.com/fundamentals/testing.
- **Cache.** Do you want to cache some parts of your application and improve your app's performance? Check Nest cache https://docs.nestjs.com/techniques/caching.
- **Custom decorators.** Do you want to create your own decorators? Check Nest custom decorators https://docs.nestjs.com/custom-decorators.
- **Passport.** This is a recommended authentication system for Nest API applications. https://docs.nestjs.com/security/authentication.
- **Pipes.** This is a recommended validation system for Nest API applications. https://docs.nestjs.com/pipes.
- **Guards.** This is a recommended authorization system for Nest API applications. https://docs.nestjs.com/guards.

Recommended books

Do you want to learn other frameworks and other types of software architectures? We have some recommendations.

- **Beginning Django 4: Build Full Stack Python Web Applications (By Greg Lim and Daniel Correa)** https://www.amazon.com/gp/product/B09M2N778S - this is a book where we design an MVT architecture with Django (Python). We don't have some clean code strategies and diagrams as we did in this book, but it is a great book for beginners.
- **Beginning Vue Stack: Build and Deploy a Full Stack MongoDB, Express, Vue.js, Node.js App (By Greg Lim and Daniel Correa)** https://www.amazon.com/gp/product/B09G9V44MN - Do you want to split your application into two parts (front-end and back-end)? You can use this book as a base. We will design two applications: a front-end with Vue.js (which follows an MVVM architecture) and a backend with Express (Node.js). In the process, you will learn another database system (MongoDB).
- Finally, go to Amazon and look for Greg Lim's books (https://www.amazon.com/Greg-Lim/e/B06XP6FHDK).

Final remarks and future books

Across our books, we have developed an extensive range of web applications with different architectures (not only MVC). E.g., applications with separated front-end and back-end, service-oriented architectures (SOA), and even micro-services. We are developing books similar to this on Laravel (PHP), Django (Python), and Spring (Java), so keep checking Daniel's Twitter account (**@danielgarax**) and Greg's Twitter account (**@greglim81**) for updates.

We would love to get your feedback. Contact us at practicalbooksco@gmail.com. Please let us know what you liked and what you didn't. That's the way all of us improve as web developers. It is something that we do very constantly, writing to book authors, with questions, suggestions, critics, and both author and readers learn a lot in this process.

Finally, please try to leave an honest review on Amazon. This is vital so that other potential readers can see and use your unbiased opinion to buy the book and understand what people like and didn't like about the book. It will only take a few minutes of your time but is very valuable to us.

"Hecho en Medellín"