The Definitive Guide to AdonisJs

Building Node.js Applications with JavaScript

—

Christopher Pitt

Apress®

The Definitive Guide to AdonisJs

Building Node.js Applications with JavaScript

Christopher Pitt

Apress®

The Definitive Guide to AdonisJs: Building Node.js Applications with JavaScript

Christopher Pitt
Cape Town, Western Cape, South Africa

ISBN-13 (pbk): 978-1-4842-3389-4 ISBN-13 (electronic): 978-1-4842-3390-0
https://doi.org/10.1007/978-1-4842-3390-0

Library of Congress Control Number: 2018937100

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York,
233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484233894. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Christopher Pitt is a developer and writer working at Over. He usually works on application architecture, though sometimes you'll find him building compilers or robots. He is also the author of several web development books and is a contributor on various open source projects such as AdonisJs.

Acknowledgments

Many people helped to make this book a reality. I'd like to thank them, in no particular order.

Harminder and the rest of the core team have written some beautiful software. I am continually amazed by how beautiful AdonisJs code can look. I enjoy JavaScript, and it seems particularly suited to a framework like this. Harminder also gave insightful technical feedback for this book.

Taylor began a wonder with Laravel. It's clear that much of AdonisJs is inspired by Laravel, and with good reason. Laravel kept me engaged in PHP, and AdonisJs will keep me engaged in server-side JavaScript.

Mark, the coordinating editor, was fantastic to work with. He never got in the way and was always helpful. That's uncommon in his line of work. Matt, the development editor, was similarly helpful and let me set the vision for this project, unencumbered. These folks are a credit to Apress.

Matt, the writer, is an inspiration to me. He wrote what I consider to be the best Laravel book on the market. Throughout this book, I sought to emulate his wisdom and skill.

Bruno, Christian, and Wayne are good friends and supporters of my work. I miss writing for Bruno. I miss having coffee with Christian. I will miss taking walks with Wayne.

Liz, my love, has given me space and coffee in abundance. She is the inspiration for the sample application, and though she finds me altogether strange, she loves me effortlessly.

Introduction

This book is designed to teach you how to use AdonisJs. To get the most out of it, you should have a firm grasp of modern JavaScript and some knowledge of how to work with relational databases and the command line.

I explain new and interesting bits of syntax, but this book isn't primarily about teaching you how to use JavaScript. It's about teaching you how to build real applications by using AdonisJs.

The application we're going to build is called Threadbear. It's a play on words, meant to conjure up thoughts of patchy, knitted teddy bears. We're going to make an application through which sellers can register and upload their knitted patterns and products. Customers will then be able to register and purchase these things.

We're not going to focus much on design. Instead, we're going to focus on the mechanics of commerce applications. Beyond adding a CSS framework, everything is as the good browser vendors intended. That said, I welcome you to add your own style to the application

We'll create a secure registration and login system. We'll add profile and product management. We'll connect the front end to the back end by using WebSockets and the Fetch API. We'll design a shopping cart in React and we'll package static files with a custom build chain. Finally, we'll learn how to deploy the application to a virtual server, and install custom domains and SSL certificates.

It is my hope that by the time you are finished reading this book, you'll know all you need to know in order to build your online business.

CHAPTER 1

Getting Started

Deciding where to start is often the hardest part, don't you find? In this chapter, we're going to get our development tools set up. We're also going to create a new AdonisJs application. We'll finish up by taking a look at the folders of our application and making a few small customizations.

Note Complex and exhaustive installation instructions are one of the easiest ways to age a book. I'm not going to belabor the steps for every operating system and configuration. I'm guessing you know enough about Google to find the help you need. If you get stuck, ask me for help on Twitter (`https://twitter.com/assertchris`) or e-mail (`cgpitt@gmail.com`).

Installing Node.js

As the name implies, AdonisJs is a JavaScript framework. There have been a few runtimes for JavaScript on the server, but the clear winner is Node.js (`https://nodejs.org/en`). You can see what the Node.js web site looks like in Figure 1-1. The latest version of AdonisJs requires Node 8.0 or newer. That's because it uses new language features to make writing code easier and more expressive. We'll learn about those later.

Node.js is relatively painless to install, despite its size and intricacy. On macOS, you can run the following:

```
brew install node
```

And on Debian or Ubuntu, you can run this:

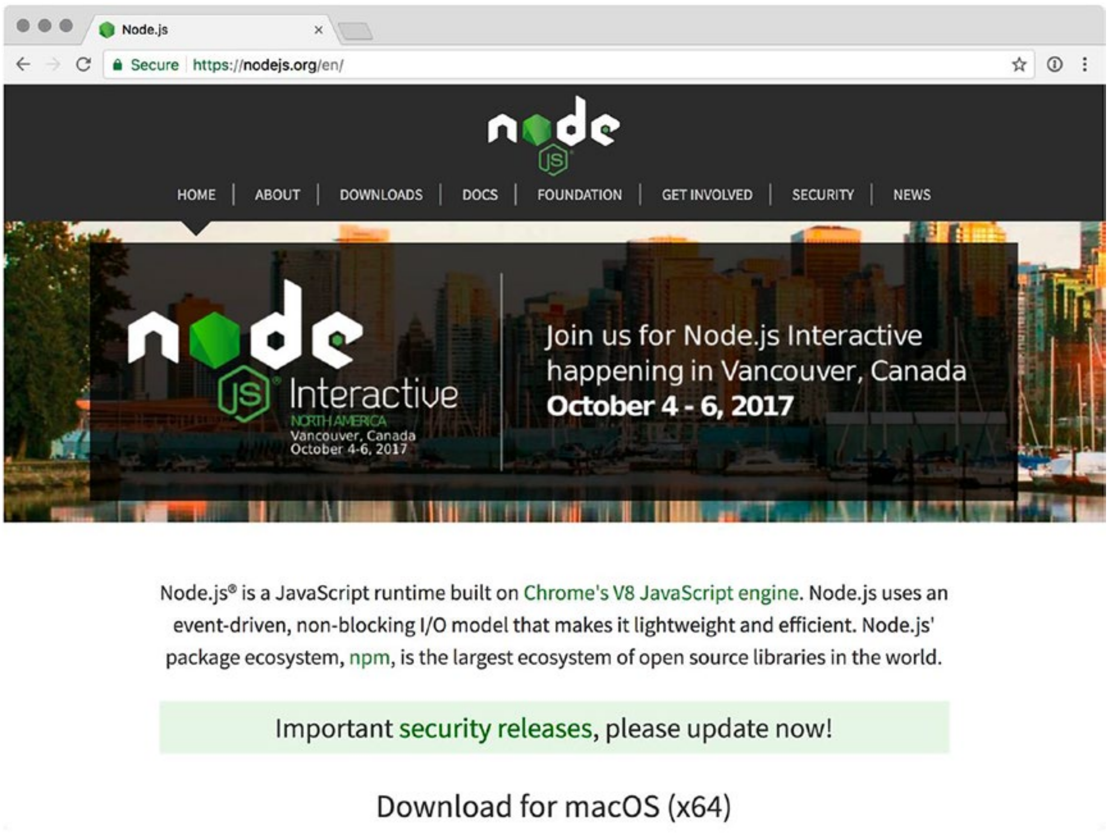
```
apt-get install node
```

C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_1

Note Alternatively, you can install Node.js from a binary at `https://nodejs.org/en/download`. There are installers for Windows, so you don't have to use macOS or Debian if you don't want to.

You can check that Node.js is correctly installed by running the following:

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



Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

Important security releases, please update now!

Download for macOS (x64)

Figure 1-1. *The Node.js web site*

Installing the AdonisJs Command-Line Tool

Setting up a new AdonisJs application requires quite a few steps. Fortunately, there's a tool to perform all of those steps for us. We need to install that tool, which also will help us to create all future AdonisJs applications:

```
npm install --global @adonisjs/cli
```

This installs the command-line tool globally, which means we'll be able to use it from anywhere on our system. Similarly to the way we check Node.js, we can check to see whether this tool has been successfully installed by running the following:

```
adonis --help
```

This also gives as a sneak peek at the kinds of things we can start to do to an application. Various commands are available, from creating and running applications to adding new functionality in existing applications.

For now, we're just going to create a new AdonisJs application.

Creating a New Application

If you read through the output of that `help` command, you may have noticed the `new` command. It's what we're going to use to create a new application:

```
adonis new threadbear
```

It'll take a minute or two to fully install and configure the new application. The only step I want to talk about for the moment is "generated unique `APP_KEY`." AdonisJs applications have an `.env` configuration file, which we can use to store sensitive configuration variables. One of these is called `APP_KEY`, which is a private key unique to the application.

Whenever text is encrypted or hashed, this value is used to seed the operation. The same password, when hashed in two applications (with different `APP_KEY` values), will result in different hashes. This is important to know because at times you might need to compare hashes or decrypt values that have been generated using a different `APP_KEY` than the one the command-line tool just made for you.

Note It's possible to reuse an APP_KEY in multiple applications, but it's definitely not recommended. Sharing an APP_KEY should be done only in exceptional situations and for a limited amount of time. If you reuse keys, you're increasing the likelihood that all the applications reusing the key are compromised at the same time.

Serving the Application

Now that we've created an application, we need a way to view it in the browser. Some server-side languages use third-party web servers (such as Nginx or Apache) as a kind of load balancer. PHP and Perl are examples of this.

Things work a bit differently with server-side JavaScript. The foundational example of how to use Node.js (https://nodejs.org/en/docs/guides/getting-started-guide) demonstrates how we can make our own HTTP server:

```
const http = require("http")

const hostname = "127.0.0.1"
const port = 3333

const server = http.createServer((req, res) => {
    res.statusCode = 200
    res.setHeader("Content-Type", "text/plain")
    res.end("hello world")
})

server.listen(port, hostname, () => {
    // server is now running
})
```

This is quite similar to what AdonisJs does under the hood. This means our server(s) need to keep the Node.js HTTP script running as what is usually referred to as a daemon. We'll unpack that later, but for now we'll need to make sure `adonis serve` is running while we code.

As the message when we run `adonis serve --dev` says, we can see the new application by loading `127.0.0.1:3333` in a web browser. We can see what this looks like in Figure 1-2.

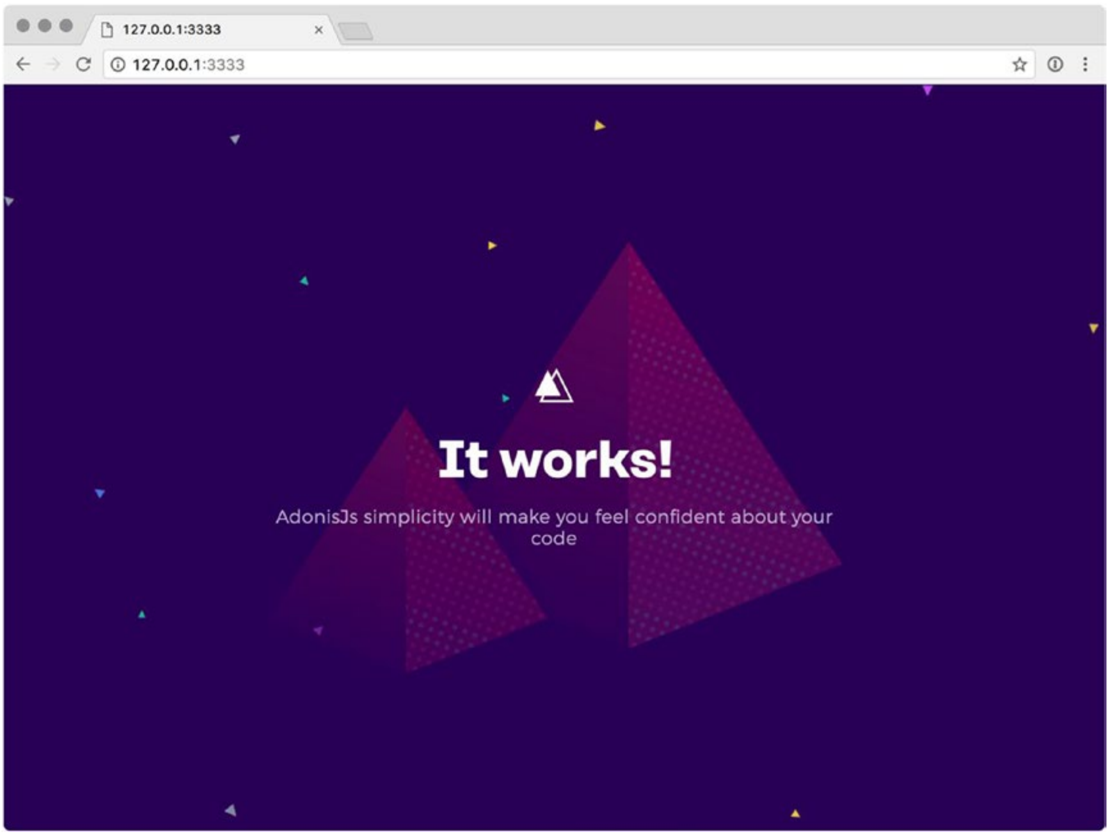

127.0.0.1:3333
127.0.0.1:3333
It works!
AdonisJs simplicity will make you feel confident about your code

Figure 1-2. *A new AdonisJs application*

Exploring the Code

New AdonisJs applications are as slim as the repository's name on which they are based.[1] They look something like this:

```
threadbear
 ↳ node_modules
 ↳ public
 ↳ start
 ↳ .env
 ↳ .env.example
 ↳ .gitignore
```

[1]https://github.com/adonisjs/adonis-slim-app

```
↳ ace
↳ package.json
↳ server.js
```

As usual, NPM (or Node Package Manager) dependencies are stored in the `node_modules` folder. When we run `adonis serve --dev`, it's doing little more than running `node server.js`.

Static files, such as client-side stylesheets and scripts, are usually placed in the `public` folder. More complex applications tend to have build chains, which take source stylesheets and scripts and combine/compress them into smaller production versions.

AdonisJs favors convention over configuration. This means that it is designed to be useful out of the box. Sometimes we might need to deviate from the standard configuration. In new applications, we can do that by customizing the files in the `start` folder.

Let's make a few customizations to the application. We're going to learn about routes in a while, but we'll add one in the meantime. The starting routes file resembles Listing 1-1.

Listing 1-1. This is from `threadbear/start/routes.js`

```
"use strict"

const Route = use("Route")

Route.get("/", ({ request }) => {
    return "...some html"
})
```

See that `Route.get`? We're going to add another, as shown in Listing 1-2.

Listing 1-2. This is from `threadbear/start/routes.js`

```
Route.get("/register", () => {
    return "...form for customers to make new profiles"
})
```

For now, restart the `serve` command and go to the address you see in the terminal window. We can see the text we entered: text that we're returning from the route, matching the `/register` path we specified.

Larger applications require more files than just the ones we have currently. You may remember that the `adonis --help` command suggested quite a few `make` commands.

We can use these to create any number of project files. Though we're also going to learn about them later, let's create a controller:

```
adonis make:controller PageController
```

This creates a nested folder structure: `app` ➤ `Controllers` ➤ `Http` ➤ `PageController.js`.

Similarly, the other `make` commands usually put the files they create in the `app` folder, and create more nested folders if they don't already exist. There's quite a rich structure built up as needed. The pattern is always the same, leading back to the convention-over-configuration paradigm.

Note The command-line tools create files in a predictable location. That's not to say that we couldn't change the position and name of these files—just that by default, they will always be in a predictable configuration.

Let's make a new command so we can explore more of the starter configuration:

```
adonis make:command SendReminderCommand
```

This command creates a new command-line command class, located in `app` ➤ `Commands` ➤ `SendReminder.js`. It automatically strips the `Command` suffix and gives some help for how to add the command to Ace.

Ace is a local version of the `adonis` global command-line helper. We can run it with the following:

```
node ace
```

The output is similar to (though much less than) that of `adonis --help`. These commands are similarly built, but the global `adonis` command has more subcommands registered within it. `ace` has only the commands registered for your app.

Sometimes it's easier not to have to specify that you want to run the command by using `node`. In that case, we can make the Ace script executable:

```
chmod +x ace
```

We also need to add a *hash-bang* (`#!`) to the top of the Ace script, as in Listing 1-3.

Listing 1-3. This is from `threadbear/ace`

```
#!/usr/bin/env node

"use strict"

// ...other code
```

Now we can run Ace by using the following:

```
./ace
```

It's not a huge improvement, but it is an interesting technique to be aware of. Hash-bang directives (like this one) can be useful for hiding unnecessary details, such as the binary we need to run a script, so that these kinds of stand-alone scripts are easier for others to use.

We're going to use this trick later, when we work out how to restart the server with a new routing configuration. In the meantime, let's follow the advice we received when we created the command. Let's change `start/app.js` to include the new command, as shown in Listing 1-4.

Listing 1-4. This is from `threadbear/start/app.js`

```
// ...other code

const commands = [
    "App/Commands/SendReminder"
]

module.exports = {
    providers, aceProviders, aliases, commands
}
```

The command code Ace generated resembles Listing 1-5.

Listing 1-5. This is from `threadbear/app/Commands/SendReminder.js`

```
"use strict"

const { Command } = require("@adonisjs/ace")
```

```
class SendReminder extends Command {
    static get signature() {
        return "send:reminder"
    }

    static get description() {
        return "..."
    }

    async handle(args, options) {
        this.info("...")
    }
}

module.exports = SendReminder
```

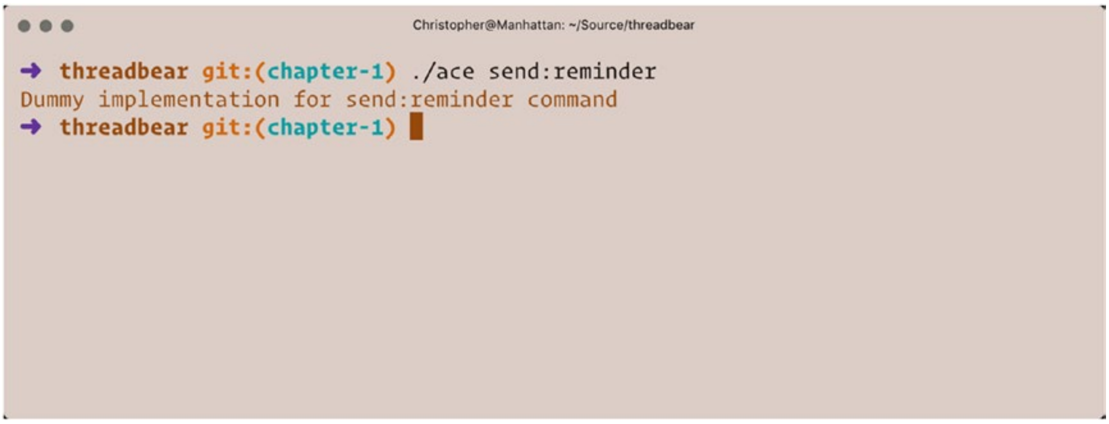
`SendReminder` extends the built-in `Command` class. `Command` does most of the heavy lifting, but it does require a few properties/getters to be defined:

`signature` is the name by which the command will be called, but also the arguments and extra parameters the command accepts.

`description` is useful when we run `./ace --help`. It shows what the command does and describes what the arguments and parameters are for.

`handle` is where we put all the code this command needs to use. We're going to use it to send reminder e-mails to our customers, but for now it can log text to the console.

We can see the results of this code in Figure 1-3.



```
➜  threadbear git:(chapter-1) ./ace send:reminder
Dummy implementation for send:reminder command
➜  threadbear git:(chapter-1) █
```

Figure 1-3. *The output of `./ace send:reminder`*

Summary

In this chapter, we installed all the requirements for AdonisJs. We created a new application, and learned how routes and shared code (such as Ace commands) can be configured in the files within the `start` folder.

We've touched lightly on a few parts of the AdonisJs application structure. In the next chapter, we're going to learn far more about defining application routes.

CHAPTER 2

Routing

In the previous chapter, we set up our environment and got our first look at the structure of an AdonisJs application. Now it's time to add HTTP endpoints to the application. This is done by adding routes. We'll learn about all the ways to customize routes, use the data they hold, and respond to any request.

What Are Routes?

Routes are like glue between the HTTP requests your browser makes and the server code meant for each request. This isn't the first time we're seeing them, either. In the previous chapter, we created a `/registration` route (though we didn't do any meaningful work through it).

Routes are an essential part of every application, because every application has at least one of them. In fact, applications can have many routes, and you're not limited in the length, complexity, or number of them your application can have.

The Web is built on top of a communications protocol called *HTTP*. It has quite a lengthy definition, but the important bits describe how browsers (and other clients) can make text-based requests to servers. Those requests include details such as the type of request being made (the request method) and the server address to which the requests should be sent.

Let's take another look at the route we defined in the previous chapter, in Listing 2-1.

Listing 2-1. This is from `threadbear/start/routes.js`

```
Route.get("/register", () => {
  return "...form to make new profiles"
})
```

© Christopher Pitt 2018
C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_2

We can tell a couple of things about this route, even if we remove all the code inside `() => {...}`. The first is that this route responds to `GET` requests. There are other kinds of requests, such as `POST`, `PUT`, `DELETE`, and `HEAD`. The second thing we can tell is that the request should go to `/register`.

Routes can also have values embedded in the path, as we'll see in this chapter.

Planning Routes

We're going to define a few more customer routes alongside the `/registration` route we already defined. Customers will need these to manage their accounts and gain access to protected profile and product information:

- `GET /login`: `GET` is the most common request method on the Internet. It's the go-to method for fetching information in a read-only way. Similarly to the way we used `GET` to show the registration form, we're using it here to show the login form.
- `POST /login`: `POST` is more common than `GET` when it comes to creating things. In this case, we'd be creating new customer sessions (and getting back security tokens). It's not crucial to understand how we do that right now. All you need to know is that we want to use `POST` because we're making something.
- `PUT /logout`: `PUT` is similar to `POST` in that it's used for a write operation. However, whereas `POST` is often used to indicate that a whole new thing is being created, `PUT` is often used to indicate that a change is being made to something. As you'll see, we use `PUT` when we want to update something that has already been created.
- `POST /register`: We already have a route for showing the registration, but this one is for creating a new customer profile. Notice that we use the `POST` method, as we're creating a whole new customer account.
- `GET /forgot-password`: When customers forget or lose their passwords, we need a way for them to request to set a new one. The only practical way for us to allow this is to send an e-mail to their address on record, which contains a link (and security token) they can click. This endpoint should show users a form with which to do that.

- `POST /forgot-password`: After the customer has entered their e-mail address, the form should be posted to this endpoint. The server code running here will then send them the e-mail. You should start to see a pattern here: we use `GET` to show forms, and `POST` to create new things from those forms.
- `GET /reset-password/{token}`: Similarly to the way the customer requests to set a new password, this endpoint shows the form they can use to enter a new password. It has an added `{token}` parameter, which we'll learn about shortly.
- `PUT /reset-password/{token}`: After the customer has entered a new password, the form can be posted to this endpoint. Because the customer's profile already exists (and we want to make an update to it), we use the `PUT` request method.
- `GET /{user-name}`: After the customer has a profile, we want them to be able to see what information is visible on it. This is where the customer will see their protected information, and where other customers will be able to see details marked as open to the public.
- `PUT /{user-name}`: Should the customer wish to change details about their account, they would be able to do so through forms that send their data to this endpoint. Because it's a partial update, we use the `PUT` method.
- `DELETE /{user-name}`: Customers should also be able to delete their profiles, which is a good opportunity for us to use the `DELETE` method. It acts a lot like the `GET` method, except that it is not read-only. Deletes are serious business, so we should confirm whether the customer initiated the process by accident or on purpose.
- `GET /{user-name}/products`: In our design, customers are also sellers. They'll be able to create new products (which is an entirely optional part of their experience), and those products will be accessible through this endpoint.
- `POST /{user-name}/products`: Should the customer want to create a new product, they'll be able to do so via this endpoint. Because they're creating a whole new product, we use the `POST` method.

- `GET /{user-name}/{product-name}`: If a customer wants to find out more details about a specific product, they will be able to do so by using this product-specific, customer-specific endpoint.
- `PUT /{user-name}/{product-name}`: Similarly, if a customer wants to update the details of a product, they should be able to do so via a `PUT` request to the same endpoint. It's a partial update, or else we'd use the `POST` method.
- `DELETE /{user-name}/{product-name}`: Finally, if a customer wants to delete a product, they should use the `DELETE` request method on this endpoint.

Note There's a fine line between respecting a seller's desire for privacy and other customers' right to access content they've purchased. It is right for us to immediately delete products and customer profiles at the will of the customer. It's also important for us to preserve the ability to download purchased content after a profile has been deleted or a product removed from sale. We'll cross this bridge when we get to it, but it's important to remember these rights and responsibilities when we design the application.

Defining Routes

Now that we've planned the routes we'll begin our application with, it's time to define them in code. We'll begin with the profile routes, shown in Listing 2-2.

Listing 2-2. This is from `threadbear/start/routes.js`

```
Route.get("/login", () => {
  // show login form
  return "GET /login"
})

Route.post("/login", () => {
  // create new customer session
  return "POST /login"
})

Route.put("/logout", () => {
  // expire current customer session
  return "PUT /logout"
})

Route.get("/register", () => {
  // show registration form
  return "GET /register"
})

Route.post("/register", () => {
  // create new customer profile
  return "POST /register"
})

Route.get("/forgot-password", () => {
  // show forgot password form
  return "GET /forgot-password"
})

Route.post("/forgot-password", () => {
  // create new password reset token and send e-mail
  return "POST /forgot-password"
})
```

Each of these routes follows the pattern I just described. They will grow to contain the functionality of the application, but in the meantime they just return strings. This is immediately useful for testing, covered next.

Testing Routes

We're going to take a brief detour from our route definitions to testing. This book has no chapter dedicated to testing, because I believe it's better to always be asking ourselves how we plan to test the code we're writing.

There are many schools of thought when it comes to testing. I'm not going to dwell on any of them (because we don't have the time, and we need to stay focused), but I will say that no one approach to testing is the best. Having any tests is better than having none, and thinking about tests early and continuously has caused me to write better code. I recommend this approach.

The jury is still out on which testing framework is the best. For now, we're going to use a framework called Mocha (`https://mochajs.org`). To install it, we need to run the following command:

```
npm install --save-dev mocha
```

This gives us access to a few tools we can use for testing. Now let's create a test file to hold our first test, as shown in Listing 2-3.

Listing 2-3. This is from `threadbear/test/routes.js`

```
const assert = require("assert")
const http = require("http")

require("dotenv").config()

const shouldBeOk = (method, path, done) => {
  http
    .request(
      {
        host: process.env.HOST,
        port: process.env.PORT,
        method,
        path,
      },
      response => {
        assert.equal(200, response.statusCode)
        done()
      },
    )
    .end()
}
```

```
const shouldHaveMessage = (
  method,
  path,
  message,
  done,
) => {
  http
    .request(
      {
        host: process.env.HOST,
        port: process.env.PORT,
        method,
        path,
      },
      response => {
        let data = ""

        response.on("data", chunk => {
          data += chunk
        })

        response.on("end", () => {
          assert.equal(message, data)
          done()
        })
      },
    )
    .end()
}

describe("GET /login", () => {
  it("should have the correct status (200)", done => {
    shouldBeOk("GET", "/login", done)
  })

  it("should have the correct message", done => {
    shouldHaveMessage(
      "GET",
      "/login",
      "GET /login",
      done,
    )
  })
})
```

We begin by importing the assertions and HTTP libraries via `require` statements. We also import the `dotenv` library and immediately call the `config` method. This loads all the environment variables in our `.env` file so we can build a URL from the `HOST` and `PORT` variables.

We follow this up by creating the `shouldBeOk` and `shouldHaveMessage` helper functions. The first checks whether a request to an address returns a response with a `200` status code. This would indicate that the request returned successfully, which tells us that the route is defined.

The `shouldHaveMessage` function tells us whether a request to an address returns a message we expect. We've defined our routes to return simple messages that describe their method and path, so we can use `shouldHaveMessage` to make sure the requests match the messages we expect them to.

We've also defined our first set of tests using the `describe` and `it` functions. Mocha provides these for us, so we don't have to import them. We can duplicate these tests for each route, as demonstrated in Listing 2-4.

Listing 2-4. This is from `threadbear/test/routes.js`

```
describe("POST /login", () => {
  it("should have the correct status (200)", done => {
    shouldBeOk("POST", "/login", done)
  })

  it("should have the correct message", done => {
    shouldHaveMessage(
      "POST",
      "/login",
      "POST /login",
      done,
    )
  })
})

// ...tests omitted for brevity

describe("POST /forgot-password", () => {
  it("should have the correct status (200)", done => {
    shouldBeOk("POST", "/forgot-password", done)
  })

  it("should have the correct message", done => {
    shouldHaveMessage(
      "POST",
      "/forgot-password",
      "POST /forgot-password",
      done,
    )
  })
})
```

We could use a loop to define these tests, but that becomes problematic for several reasons. First, if we write faulty tests in a loop, all of our tests are faulty. Perhaps we make a mistake in how we request data from the server, or how we inspect the results. We may never know about the fault because all the tests are faulty in the same way. They may appear to execute successfully or make valid assertions, and fool us into thinking our tests are OK when they are not.

Second, these are simple tests, and they are bound to change. Some of these endpoints are going to take wildly different input. Some will render forms, whereas others will return redirection instructions to the browser.

It's best to write out each test (which isn't too much of a problem because we have abstracted the helper methods), and change them as we evolve our application.

To run these tests, we need the following command:

```
node_modules/mocha/bin/mocha
```

We can shorten this by defining an NPM test script, shown in Listing 2-5.

Listing 2-5. This is from `threadbear/package.json`

```
"scripts": {
  "test": "mocha"
},
```

This allows us to run the test just by typing the following:

```
npm run test
```

Defining Route Parameters

Let's continue defining routes, this time with parameters. There are a few ways to define these parameters, but the most common form can be seen in Listing 2-6.

Listing 2-6. This is from `threadbear/start/routes.js`

```
Route.get(
  "/reset-password/:token",
  ({ params }) => {
    // show forgot password form
    return "GET /reset-password " + params.token
  },
)

Route.post(
  "/reset-password/:token",
  ({ params }) => {
    // create new password reset token and send e-mail
    return "POST /reset-password " + params.token
  },
)
```

Route parameters are defined with a leading colon, as seen in `:token`. Route closures are provided with a context object, which contains a `params` key. Using ES6 object destructuring, we can shorten the path to these parameters.

Note It's also possible to define optional parameters, as seen in `:token?`. However, our tokens aren't going to be optional, so we don't need that syntax.

Let's create the tests for these in Listing 2-7.

Listing 2-7. This is from `threadbear/test/routes.js`

```
describe("GET /reset-password", () => {
  it("should have the correct status (200)", done => {
    shouldBeOk(
      "GET",
      "/reset-password/token123",
      done,
    )
  })

  it("should have the correct message", done => {
    shouldHaveMessage(
      "GET",
      "/reset-password/token123",
      "GET /reset-password token123",
      done,
    )
  })
})

describe("POST /reset-password", () => {
  it("should have the correct status (200)", done => {
    shouldBeOk(
      "POST",
      "/reset-password/token123",
      done,
    )
  })
  it("should have the correct message", done => {
    shouldHaveMessage(
      "POST",
      "/reset-password/token123",
      "POST /reset-password token123",
      done,
    )
  })
})
```

These tests follow a pattern that's similar to that of the previous tests. The only difference is that we provide an example token so that we call the routes that require it, and check that their response matches it. Let's finish up this section by defining the remaining routes in Listing 2-8.

Listing 2-8. This is from `threadbear/start/routes.js`

```
Route.get("/:customer", ({ params }) => {
  // show customer profile
  return "GET /:customer " + params.customer
})

Route.put("/:customer", ({ params }) => {
  // update customer profile
  return "PUT /:customer " + params.customer
})

Route.delete("/:customer", ({ params }) => {
  // delete customer profile
  return "DELETE /:customer " + params.customer
})

Route.get("/:customer/products", ({ params }) => {
  // show customer's products
  return (
    "GET /:customer/products " + params.customer
  )
})
```

```
Route.post(
  "/:customer/products",
  ({ params }) => {
    // create a new product
    return (
      "POST /:customer/products " +
      params.customer
    )
  },
)

Route.get("/:customer/:product", ({ params }) => {
  // show customer profile
  return (
    "GET /:customer/:product " +
    params.customer +
    " " +
    params.product
  )
})

Route.put("/:customer/:product", ({ params }) => {
  // update customer profile
  return (
    "PUT /:customer/:product " +
    params.customer +
    " " +
    params.product
  )
})

Route.delete(
  "/:customer/:product",
  ({ params }) => {
    // delete customer profile
    return (
      "DELETE /:customer/:product " +
      params.customer +
      " " +
      params.product
    )
  },
)
```

These are all much the same as before. The only thing we're doing differently is defining multiple parameters for a few of them. Let's also add tests for these, as shown in Listing 2-9.

Listing 2-9. This is from `threadbear/test/routes.js`

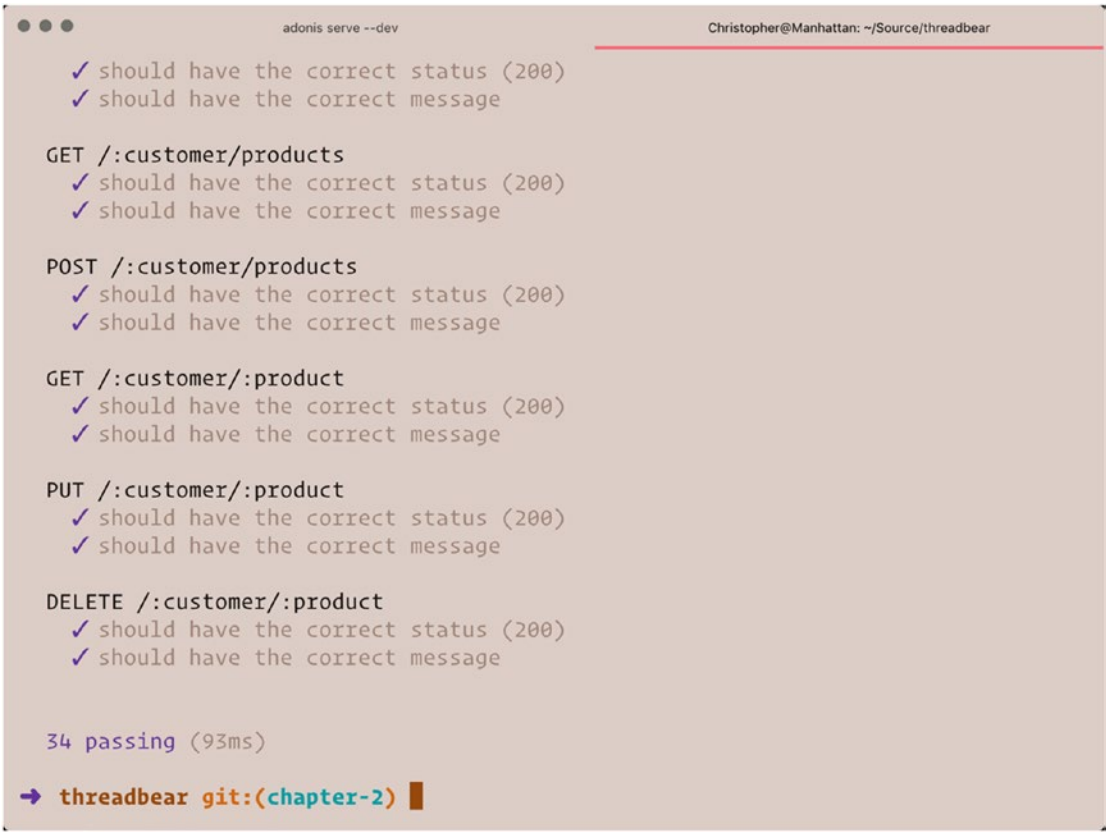
```
describe("GET /:customer", () => {
  it("should have the correct status (200)", done => {
    shouldBeOk("GET", "/assertchris", done)
  })

  it("should have the correct message", done => {
    shouldHaveMessage(
      "GET",
      "/assertchris",
      "GET /:customer assertchris",
      done,
    )
  })
})

// ...tests omitted for brevity


describe("DELETE /:customer/:product", () => {
  it("should have the correct status (200)", done => {
    shouldBeOk(
      "DELETE",
      "/assertchris/teddy",
      done,
    )
  })
  it("should have the correct message", done => {
    shouldHaveMessage(
      "DELETE",
      "/assertchris/teddy",
      "DELETE /:customer/:product assertchris teddy",
      done,
    )
  })
})
```

Now we have 17 routes, leading to various parts of our application, and 34 tests. Not bad for our first real work inside an AdonisJs application! Figure 2-1 shows what running these tests looks like.



```
    ✓ should have the correct status (200)
    ✓ should have the correct message

  GET /:customer/products
    ✓ should have the correct status (200)
    ✓ should have the correct message

  POST /:customer/products
    ✓ should have the correct status (200)
    ✓ should have the correct message

  GET /:customer/:product
    ✓ should have the correct status (200)
    ✓ should have the correct message

  PUT /:customer/:product
    ✓ should have the correct status (200)
    ✓ should have the correct message

  DELETE /:customer/:product
    ✓ should have the correct status (200)
    ✓ should have the correct message


  34 passing (93ms)

➜ threadbear git:(chapter-2) ▌
```

Figure 2-1. *Running tests with Mocha*

Summary

In this chapter, we defined many routes. We learned how to combine request methods and paths to target specific parts of our application code.

We also learned about how to test these routes by using Mocha. Testing is an important part of any maintainable and scalable application. Testing helps us know when things are broken and gives us clues about where they are breaking.

CHAPTER 3

Views

In the previous chapter, we created many new routes. These will direct traffic to many parts of our application. We looked at how to match specific request methods and how to add parameters to the endpoint URLs.

Let's take this a step further by rendering HTML for some of those endpoints. We could take a few approaches, including returning a string of HTML directly from a route handler or rendering a template. We're going to focus on the latter, as we learn new Ace commands and the Edge template syntax.

Creating Views

Let's ease into this topic by creating a new template file. You may recall: we've already seen how to render HTML by returning a string from a route, as demonstrated in Listing 3-1.

Listing 3-1. This is from `threadbear/start/routes.js`

```
Route.get("/", ({ request }) => {
  return `
    <html>
      <head>
        <link rel="stylesheet" href="/style.css" />
      </head>
      <body>
        <section>
          <div class="logo"></div>
          <div class="title"></div>
          <div class="subtitle">
            <p>AdonisJs ...</p>
          </div>
```

© Christopher Pitt 2018
C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_3

```
        </section>
      </body>
    </html>
    `
})
```

Although this approach works—for simple cases—it doesn't scale well. We may want to use some or all of the HTML to respond to other routes. We may want to compose responses out of many template files or isolate specific presentation behavior.

So, instead, we're going to create a separate template file and use it to display a new home page. We can do this with the command in Listing 3-2.

Listing 3-2.

```
adonis make:view page/home
```

This creates a new file at `threadbear/resources/views/page/home.edge`, which we can fill with the sample content in Listing 3-3.

Listing 3-3. This is from `threadbear/resources/views/page/home.edge`

```
<h1>Welcome to Threadbear</h1>
<p>
  Looking for a cuddly friend or your next hobby
  project? Our extensive range of lovingly crafted
  bears and clearly written patterns will delight
  you.
</p>
```

Registering the View Provider

Before we can see this code in the browser, we need to register a new provider. *Providers* are like maps to greater functionality, hidden in other parts of the framework or in third-party libraries. They describe how to register new commands and container resources (which we'll learn about later).

The AdonisJs application we created previously is a minimal application. In it, all nonessential things are disabled by default, including the view renderer. We have to add the `ViewProvider` to the application before we can directly access the view renderer in our routes. This can be seen in Listing 3-4.

Listing 3-4. This is from `threadbear/start/app.js`

```
const providers = [
  "@adonisjs/framework/providers/AppProvider",
  "@adonisjs/framework/providers/ViewProvider",
]
```

It's a bit too early to learn about the exact structure of a provider. But we can assume that `ViewProvider` adds a new key to the object each route gets. We can use this new `view` key to render view files from inside route handlers, as shown in Listing 3-5.

Listing 3-5. This is from `threadbear/start/routes.js`

```
Route.get("/", ({ view }) => {
  return view.render("page/home")
})
```

We've replaced that big multiline string of HTML with a call to `view.render`. Just as when we created the view file, we have no need of the `.edge` extension. AdonisJs adds that in for us. We just need to reference the view files relative to the `resources/views` folder, and AdonisJs will add the rest.

Note This is one of those times when running `adonis serve --dev` is really useful. We can change the `start` and `resources` files, and we don't have to restart the server after each change to see its effect.

Creating Layouts

At times, the view code we want to share will be common outer code (structural markup). Think about larger sites and how they share the same stylesheets and visual styles across many pages. They often have common stylesheet files that they include in many rendered pages.

We *could* achieve this by copying and pasting the stylesheet links in every view file. Or we could be smart and put them in a layout file. Then, each view file could *inherit* those structural elements from the layout without copy-and-paste.

Let's try this by creating a new layout view file, as in Listing 3-6.

Listing 3-6. This is from `threadbear/resources/views/layout.edge`

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    @!section("content")
  </body>
</html>
```

This is basic HTML markup that you may expect to see on any page of your application. It declares a doctype and defines container markup in which page-specific content will sit. What's interesting is the `@!section("content")` tag. If we look at the documentation at http://edge.adonisjs.com/docs/layouts#_basic_example, we'll see that this `@!tag` syntax is shorthand for the `@tag/@endtag` syntax that can contain data. It's a self-closing shorthand syntax.

It's important to understand that the `section` tag is like a named placeholder for future content. We can place default content inside it, on the layout, but it's far more useful for marking where child pages will display their content.

I used the `adonis make:view` command to create this layout view file, but you could just as easily have created the file by hand. The `make:view` command does have another parameter to it, which is useful for creating new view files that extend a layout view:

```
adonis make:view user/login --layout layout
```

The `--layout` parameter generates the new view file and includes `@layout("layout-name")` at the top of it. That `@layout` tells AdonisJs to render this view file as an extension of the named layout view file. We can embellish this somewhat, by adding the markup in Listing 3-7.

Listing 3-7. This is from `threadbear/resources/views/user/login.edge`

```
@layout("layout")
@section("header")
  <h1>Login</h1>
@endsection
@section("content")
```

```
  <form method="post">
    <div class="form-group">
      <label for="email">Email address</label>
      <input type="email" class="form-control" id="email">
    </div>
    <div class="form-group">
      <label for="password">Password</label>
      <input type="password" class="form-control" id="password">
    </div>
    <button type="submit" class="btn btn-primary">
      Log in
    </button>
  </form>
@endsection
```

I've opted to use Bootstrap markup (`https://getbootstrap.com`) for this form, because I like the default styling and accessibility it uses. Classes such as `form-group` and `form-control` tell Bootstrap how to style these elements. In order for that styling to reach the browser, we need to add more markup to the layout view file, as shown in Listing 3-8.

Listing 3-8. This is from `threadbear/resources/views/layout.edge`

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1,
    shrink-to-fit=no">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
    bootstrap/4.0.0-beta/css/bootstrap.min.css">
  </head>
  <body>
    <div class="container">
      <header class="row">
        <div class="col">
          @!section("header")
        </div>
      </header>
      <main class="row">
        <div class="col">
          @!section("content")
        </div>
      </main>
    </div>
    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js">
    </script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/
    umd/popper.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/js/
    bootstrap.min.js"></script>
  </body>
</html>
```

We've added Bootstrap's CSS and JS files (from a content delivery network, or CDN) to the layout view file. We've also introduced some Bootstrap classes around the sections, to arrange them into rows and columns. We could reduce this code somewhat by using the Edge global functions and relative protocols in Listing 3-9.

Listing 3-9. This is from `threadbear/resources/views/layout.edge`

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1,
    shrink-to-fit=no">
    {{ css("//maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/css/bootstrap.
    min.css") }}
  </head>
  <body>
    <div class="container">
      <header class="row">
        <div class="col">
          @!section("header")
        </div>
      </header>
      <main class="row">
        <div class="col">
          @!section("content")
        </div>
      </main>
    </div>
    {{ script("//code.jquery.com/jquery-3.2.1.slim.min.js") }}
    {{ script("//cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/umd/
    popper.min.js") }}
    {{ script("//maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/js/bootstrap.
    min.js") }}
  </body>
</html>
```

If we wanted to reference a local CSS or JS file, we would just need to omit the protocol (`http` or `https`) and give a path relative to the public directory. Figure 3-1 shows what these styles look like when applied to the markup.

Note Don't forget to replace the previous `login` route's handler code with a call to `view.render`, or you won't see these changes in your app.

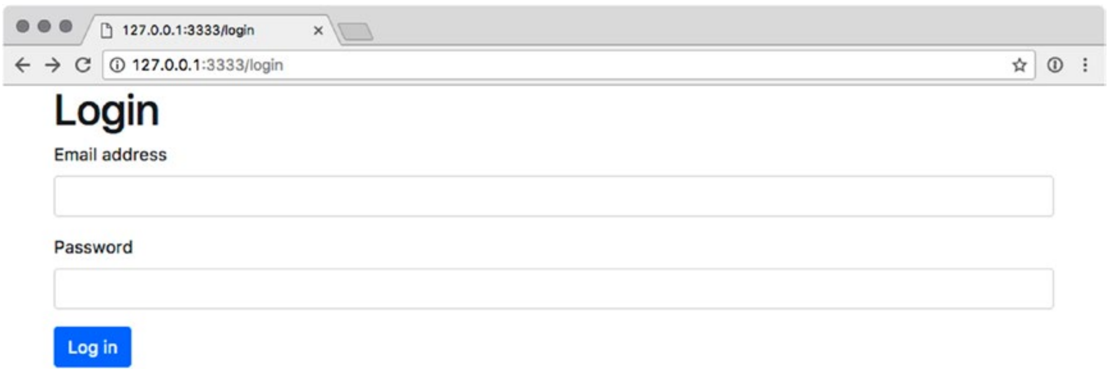


Figure 3-1. Login view with Bootstrap styles

Similarly, we can create a register view (and form), as shown in Listing 3-10.

Listing 3-10. This is from `threadbear/resources/views/user/register.edge`

```
@layout("layout")
@section("header")
  <h1>Register</h1>
@endsection
@section("content")
  <form method="post">
    <div class="form-group">
      <label for="email">Email address</label>
      <input type="email" class="form-control" id="email">
    </div>
    <div class="form-group">
      <label for="password">Password</label>
      <input type="password" class="form-control" id="password">
    </div>
    <div class="form-group">
      <label for="confirm-password">
        Confirm password
      </label>
      <input type="password" class="form-control" id="confirm-password">
    </div>
    <button type="submit" class="btn btn-primary">Register</button>
  </form>
@endsection
```

We've changed the heading and added a new field to the form. We're using the same layout file, so we don't have to add or change any of the shared code for this new view.

Learning Template Syntax

Edge provides a ridiculous number of template tags and general bits of syntax, to control all aspects of the way templates render. We're going to use most of them throughout this book, but I want to introduce them in this section so you know what's there.

Note I'm following the outline of the official docs, which you can find at http://edge.adonisjs.com/docs/getting-started. I'll go through the important tags, briefly describing them. I'll follow each description with an example (code listing). For the sake of brevity, I don't label the listings or refer to them in the preceding description—so pay attention to the order and syntax of the examples as they relate to the descriptions they follow. And remember that this isn't an exhaustive description of each. We will see many more examples of these as we use them in our application.

Interpolation

Interpolation is a fancy word for replacing variable placeholders (in a string) with the value of the matching variable. You can think of it as filling in a form, where you put your name next to the blank space called *Name*.

This looks like the following:

```
<h1>Hello, {{ name }}</h1>
```

The way this works is by treating everything inside {{ and }} as a JavaScript expression. That means we can also call methods inside these braces:

```
<h1>Hello, {{ name.toUpperCase() }}</h1>
```

Or we can even do this:

```
<h1>Hello, {{ name.replace(/^(.)|\s+(.)/g, match => match.toUpperCase()
}}</h1>
```

In both cases, we call functions on the `name` string variable. Even though we can use standard JavaScript inside these braces, the returned value is always escaped by Edge. To use unescaped HTML values, we need to use triple-braces:

```
<h1>Hello, {{{ `<strong>${name}</strong>` }}}</h1>
```

Sometimes you might want to use client-side template renderers. Template languages like those in AngularJS (https://angularjs.org) and Vue.js (https://vuejs.org) use a similar brace syntax to interpolate their variables. Using these braces in Edge means that they'll likely be replaced long before the markup gets to the browser.

There's a special Edge syntax for this case:

```
This variable will be sent directly to the @{{ browser }}
```

This seems like such a simple bit of syntax (interpolation in general), but it's the kind of thing we're going to see all over our apps. To get that data there, we need to pass it in where we render the template:

```
Route.get("/:customer", ({ view, params }) => {
  return view.render("user/profile", {
    name: params.customer,
  })
})
```

In fact, while we're here, we can reduce this code (which is returning only one thing):

```
Route.get("/:customer", ({ view, params }) =>
  view.render("user/profile", {
    name: params.customer,
  }),
)
```

This works because JavaScript fat-arrow functions return the result of their expression if the expression isn't inside a set of braces. It's not a requirement, but it may lead to cleaner code in your projects.

Conditionals

Conditionals are things such as `if` statements. Edge allows a few variations of these, including these two:

```
@if(name)
  <h1>Hi, {{ name }}</h1>
@elseif(email)
  <h1>Hi, {{ email }}</h1>
@else
  <h1>Who are you?!</h1>
@endif
```

… and …

```
@unless(user.isBanned)
  <p>Your account is in good standing</p>
@else
  <p>Please contact a system administrator</p>
@endif
```

You're free to use either of these, as best suits your template. They're functionally similar, but sometimes using `unless` makes your intentions clearer. We are also allowed to use most JavaScript expressions between the parentheses of these tags.

Loops

Loops are an integral part of all nontrivial applications. There are many kinds of loops in JavaScript, but only one kind supported by Edge:

```
@each(product in user.products)
  <li>{{ product.name }}</li>
@else
  <li>No products found</li>
@endeach
```

Where this tag shines is that it allows an `@else` similar to the conditionals. That means we don't have to wrap looping code in an `if` statement, in case there are no items in the array. We can display empty-state messages alongside non-empty-state data.

This syntax works well for arrays, but we need a slightly different syntax for objects:

```
@each((value, key) in user.preferences)
  <li>{{ key }}: {{ value }}</li>
@endeach
```

We express the `value` and `key` as `(value, key)`, so that Edge knows how to destructure the object. We can use most JavaScript expressions after the `in` keyword—but before it, we need to be careful of the syntax we use. Either plain `value` or `(value, key)` are recommended.

Note There are some helpful "magic" constants inside the `@each` tags, such as `$loop.index` and `$loop.total`, but you'll have to read up about them in the docs (http://edge.adonisjs.com/docs/iteration#_loop_variable_properties).

If you find yourself including a partial for each iteration, there's a handy shortcut:

```
@!each(product in user.products, include = "user/profile/product")
```

This is the same as calling `@include` between `@each` tags, but it's significantly shorter to write.

Partials

We've already seen `@layout`, which makes the current view file a child of another view file. The opposite of this is the `@include` tag:

```
@include("user/profile/menu")
@section("header")
  <h1>Hi, {{ name }}</h1>
@endsection
@include("user/profile/timeline")
```

`@include` gets the content of a view and inlines it. It's not specific to named containers, like `@section,` but rather you can think of it as an automatic copy-and-paste feature. Includes are automatically given access to the variables you pass to `view.render`, so you don't need to do anything extra to use them.

Yield

It's pretty common to check whether a value is present and then output that value, or to otherwise output a default value. Think of situations like this:

```
@if(name)
  <h1>Hi, {{ name }}</h1>
@else
  <h1>Who are you?!</h1>
@endif
```

It's so common, in fact, that Edge has a shortcut for it:

```
<h1>
  @yield(name)
    Who are you?!
  @endyield
</h1>
```

The content inside the `@yield` tags is returned if the expression inside the parentheses is falsy (can be equated to `false`). There is no point to having a `@!yield(name)` form, because this is the same as `{{ name }}`, where name is falsy.

Functions

At times we may want to reuse functions in many view files that we don't want to keep on passing through `view.render`. In these times, we can define global view functions in a provider or one of the starter files. Listing 3-11 demonstrates how.

Listing 3-11. This is from `threadbear/app/Providers/ViewProvider.js`

```
"use strict"

const {
  ServiceProvider,
} = require("@adonisjs/fold")

class ViewProvider extends ServiceProvider {
  boot() {
    const View = this.app.use("Adonis/Src/View")

    View.global("toTitleCase", string =>
      string.replace(/^(.)|\s+(.)/g, match =>
        match.toUpperCase(),
      ),
    )
  }
}

module.exports = ViewProvider
```

If we want this provider loaded for every request, we need to include it as shown in Listing 3-12.

Listing 3-12. This is from `threadbear/start/app.js`

```
const providers = [
  "@adonisjs/framework/providers/AppProvider",
  "@adonisjs/framework/providers/ViewProvider",
  __dirname + "/../App/Providers/ViewProvider",
]
```

There are a number of predefined global view functions (including `capitalize`, which does exactly what this function does). You can learn more about them at `http://edge.adonisjs.com/docs/globals`.

Summary

In this chapter, we learned how to render isolated view markup. We looked at compositional elements such as layouts and includes. We also learned about many common behavioral elements including loops, conditionals, and functions.

It's important to stress that this is only the beginning of our view work. We're going to use these things in almost every other chapter. Take some time to become familiar with this syntax by creating your own view files.

In the next chapter, we're going to learn how to pull data out of the forms we created here. We'll look at how requests are created and how to use the data they contain to make informed choices about what the server should do and what the user wants to see.

CHAPTER 4

Requests

Now that we're able to render HTML forms, it's time to think about what we do with the data users send to us. In this chapter, we're going to take a look at what makes a `Request` object, and how we extract data from each request.

It's also a great opportunity for us to extend our tests to be able to make requests to the application and verify that the responses we get back represent the data we expect. That way, we'll know the moment we break something that a browser is using.

Updating Tests

Since we added the markup in the previous chapter, the tests have been broken. When we run `npm run test`, we notice that the profile, register, and login endpoints all return different content than that which the tests expect.

This is good because it shows us that behavior is changed, and that consuming those endpoints (in something like a browser) is going to be affected. We need to update these tests to reflect the new behavior they have: rendering markup for each task.

But there's a catch! We can't easily check the markup because Node.js servers don't naturally understand it. Sure, they can produce it via string concatenation, and they can understand posted form data. The most natural way for us to check the response markup is to walk through the Document Object Model (DOM), and Node.js servers are ill-equipped for this task.

We're going to introduce a library that can help with this:

```
npm install --save-dev cheerio
```

Our routes test file is a little busy, so let's move the helper functions to their own file, as in Listing 4-1.

© Christopher Pitt 2018
C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_4

Listing 4-1. This is from `threadbear/test/helpers.js`

```
const assert = require("assert")
const http = require("http")

require("dotenv").config()

const shouldBeOk = (method, path, done) => {
  // ...snip
}

const shouldHaveMessage = (
  method,
  path,
  message,
  done,
) => {
  // ...snip
}

module.exports = {
  shouldBeOk,
  shouldHaveMessage,
}
```

The methods remain the same, but we need to export them by adding them to `module.exports`. This means we can export them in our tests file, as shown in Listing 4-2.

Listing 4-2. This is from `threadbear/test/routes.js`

```
const assert = require("assert")
const http = require("http")

const {
  shouldBeOk,
  shouldHaveMessage,
} = require("./helpers")

require("dotenv").config()
```

```
describe("GET /login", () => {
  it("should have the correct status (200)", done => {
    shouldBeOk("GET", "/login", done)
  })

  // ...snip
})
```

As we add more helper functions, we can keep them in `helpers.js` and import them here. Let's change the register test to inspect the markup that endpoint returns. We need to import Cheerio (`https://cheerio.js.org`) for that and make a custom request, as in Listing 4-3.

Listing 4-3. This is from `threadbear/test/routes.js`

```
describe("GET /register", () => {
  it("should have the correct status (200)", done => {
    shouldBeOk("GET", "/register", done)
  })

  // it("should have the correct message", done => {
  //     shouldHaveMessage(...snip)
  // })

  it("should have the correct markup", done => {
    http
      .request(
        {
          host: process.env.HOST,
          port: process.env.PORT,
          method: "GET",
          path: "/register",
        },
        response => {
          let buffer = ""

          response.on("data", data => {
            buffer += data
          })
          response.on("end", () => {
            callback(buffer, response)
          })
        },
      )
      .end()
  })
})
```

In Node.js, we typically receive responses as asynchronous streams of data (as opposed to a single body of data). To get the full response body, we need to add event listeners to the response. Each time we get data, we add it to the buffer. When the response ends (and is complete), we can work with the body content.

This seems a bit verbose (especially if we're going to need to do it often). Let's make a helper function out of this, as in Listing 4-4.

Listing 4-4. This is from `threadbear/test/helpers.js`

```
const querystring = require("querystring")

const request = (
  method,
  path,
  callback,
  options = {},
) => {
  let data = ""

  const methods = ["POST", "PUT", "PATCH"]

  if (
    options.data &&
    methods.indexOf(method) > -1
  ) {
    data = querystring.stringify(options.data)
  }

  const parameters = Object.assign(
    {
      headers: {
        "Content-Type":
          "application/x-www-form-urlencoded",
        "Content-Length": Buffer.byteLength(data),
      },
      host: process.env.HOST,
      port: process.env.PORT,
      method,
      path,
    },
    options,
  )

  const request = http.request(
    parameters,
    response => {
      let buffer = ""

      response.on("data", data => {
        buffer += data
      })

      response.on("end", () => {
        callback(buffer, response)
      })
    },
  )

  if (data) {
    request.write(data)
  }

  request.end()
}

module.exports = {
  request,
  // ...snip
}
```

In addition to accepting a callback (which is given the buffered markup), this helper function accepts an optional `options` object. The other helper functions never need to change things such as headers, but custom test requests may need to. Using `Object.assign`, we can allow the initial options object to be overridden by any required custom properties in the test. We've also added special consideration for requests with bodies, so that their data will be appropriately serialized and added to the request.

Now we can use Cheerio to inspect the returned markup, as shown in Listing 4-5.

Listing 4-5. This is from `threadbear/test/routes.js`

```
it("should have the correct markup", done => {
  request("GET", "/register", markup => {
    const find = cheerio.load(markup)

    assert.equal(find("h1").text(), "Register")
    // assert.equal(find("input[...]").length, 1)
    // assert.equal(find("input[...]").length, 1)
    // assert.equal(find("input[...]").length, 1)

    done()
  })
})
```

Cheerio works by creating a context in which to search, using methods quite similar to those found in jQuery (`http://jquery.com`). This allows us to target elements by using CSS selectors and asserting things about them.

We begin by loading the markup into a Cheerio context and then finding an `h1` element. We can then assert that the text contained in the h1 element matches our expectations. When we run this code, the test starts to pass again.

Do you see those commented assertions? They are there to check that the registration form contains fields appropriately named. If we uncomment them, we'll discover a bug in our markup. When we created the registration form, we forgot to give the fields names. Listing 4-6 shows what the markup should look like.

Listing 4-6. This is from `threadbear/resources/views/user/register.edge`

```
<form method="post">
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email" class="form-control" id="email" name="email">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control" id="password"
    name="password">
  </div>
  <div class="form-group">
    <label for="confirm-password">Confirm password</label>
    <input type="password" class="form-control" id="confirm-password"
    name="confirm-password">
  </div>
  <button type="submit" class="btn btn-primary">
    Register
  </button>
</form>
```

Now we can uncomment those lines, and the test will continue to pass. This shows one of the best things about writing tests: they're useful for identifying where our code breaks, but they're also extremely useful for validating our expectations about our code. Tests help us design better systems by stating what we expect the code to do and letting the tests tell us whether we've written code that actually works like that.

Spend a few minutes trying to fix the login and profile tests. You'll have to do the same kind of markup inspection. It may reveal similar issues with the markup we created in the previous chapter. If you get stuck, look at the chapter's source code to see how I've done it.

Understanding Requests

Let's take a look at the `Request` class. More specifically, let's look at how we get it and how we use it. AdonisJs has allowed us to create unique URLs in our application. It's allowed us to respond to requests by returning string data. But what about when those requests are form submissions or other kinds of requests?

To parse those, we need to add another dependency. This time, it's a request body parser library:

```
adonis install @adonisjs/bodyparser
```

This library introduces middleware and configuration files. Ordinarily, we'd use `npm install`, but we want the config files to be copied over to our application. `adonis install` uses `npm install`, but it also arranges config files for us. We haven't looked at middleware in detail, but what we need to do is add it to the bootstrapping code. The first half is shown in Listing 4-7, and the second half is shown in Listing 4-8.

Listing 4-7. This is from `threadbear/start/app.js`

```
const providers = [
  "@adonisjs/bodyparser/providers/BodyParserProvider",
  // ...snip
]
```

Listing 4-8. This is from `threadbear/start/kernel.js`

```
const globalMiddleware = [
  "Adonis/Middleware/BodyParser",
]
```

Now that this middleware is configured, we can post data from a browser and inspect it on the server. Let's change one of the registration endpoints to show us what is being sent, as shown in Listing 4-9.

Listing 4-9. This is from `threadbear/start/routes.js`

```
Route.post("/register", ({ request }) => {
  // ...create new customer profile
  return JSON.stringify(request.all())
})
```

Note We don't technically need the `JSON.stringify` function call, as AdonisJs does this automatically. I still prefer to add it, to be clear about what my response is. Feel free to leave it out if you prefer.

One of the properties of the context parameter given to all route handlers is the `Request` object. It's only one step removed from the built-in Node.js request object (which can be reached through `request.request`), but AdonisJs adds a lot of utility to it.

If we take the given request and fetch all the key/value pairs, we can stringify and pass them straight back. This will show a JSON representation of the posted form data, no matter what the form's `enctype` attribute says. That's already a step up from working with requests in built-in Node.js code.

In the next chapter, we're going to look at how to respond to the values we see. In the meantime, let's look at what other information we can get from the request.

Note I'm going to be showing many examples again. Omitting the listing numbers is slightly less distracting, so pay particular attention to the order of the examples. I'll explain a concept and then show example code.

Getting Parameters

We may need to get the value of a query string parameter or a posted form field. For this, we can use the `input` method:

```
request.input("page", 1)
```

The `input` method allows an optional second parameter, which will be the default value if no query string parameter or form field matches the name given as the first parameter. Pagination is a good example of this, because the current page can be inferred from a query string parameter (or even a named route parameter), but may not be defined before the user has started using pagination links.

We may also want to include only a subset of all fields, which we can do with the `only` method:

```
request.only(["email", "password"])
```

This is good when we need lots of information for validation, but want to persist only some of that information to the database. Alternatively, we can exclude fields and allow everything else through:

```
request.except(["confirm-password"])
```

Getting Headers and Cookies

Headers are slightly less complicated than parameters. There's one method to get a specific header, and one method to get all headers:

```
request.header(
  "content-type",
  /* default */ "text/html",
)
request.headers() // {"content-type": ...}
```

It's also possible to use these request methods with the underlying Lodash library (https://lodash.com):

```
_.get(request.headers(), "content-type")
```

Note AdonisJs requires Lodash to perform certain useful functional programming tasks. It's still a good idea to require Lodash as a dependency if your code uses it, because future versions of AdonisJs may not.

Cookies follow a similar pattern, in that there are two methods we can use to access them:

```
request.cookie("language", /* default */ "en-gb")
request.cookies() // {"language": ...}
```

AdonisJs automatically encrypts and signs cookies, meaning you can't read cookies set via the browser. We *can* use less secure cookies via a couple of corresponding methods:

```
request.plainCookie("language", /* default */ "en-gb")
request.plainCookies() // {"language": ...}
```

Getting "accepts" and "language"

I've used `content-type` and `language` as examples, but these are ideas strongly represented in request headers. There are even a few methods that give us quick access to them:

```
const preferred = request.accepts([
  "json",
  "html",
])

if (preferred === "json") {
  // ...return a JSON response
}

// ...return an HTML response
```

...And:

```
const preferred = request.language(["en", "fr"])

if (preferred === "en") {
  // ...return an English response
}

// ...return a French response
```

These make it easier to decide how and with what information to respond to a request.

Finding Out More

There's an incredible amount of metadata to discover, which is less about the data that a user submits and more about how the request was made. I'm not going to list all of the options, but I recommend you look at the documentation for the `Request` class at `http://dev.adonisjs.com/docs/4.0/request`.

Responding to POST, PUT, PATCH, and DELETE

Most of the web works through `GET` and `POST` requests, but the HTTP 1.1 specification allows so much more. It's entirely possible to express different intentions through the use of different request methods, such as `POST`, `PUT`, `PATCH`, and `DELETE`. In fact, it's commonly understood that each of these refers to a specific kind of action:

- `GET` can be a read-only request to one or many resources, such as products or people.

- POST can be an operation that creates a resource.
- PUT can be an operation that entirely replaces a resource's attributes.
- PATCH can be an operation that replaces only a few of a resource's attributes, leaving all the others intact.
- DELETE can be an operation that deletes a resource.

The HTTP 1.1 specification doesn't enforce these rules, but they're helpful for creating understandable web services. If a developer sees that a request needs the DELETE method, they know to be cautious when using it (because it may be deleting resources). If they have to choose between a PUT and a PATCH request method, they may pick the one that best suits the information they already have or want to persist.

We can use this too, for things like password resetting. Let's make a couple of new HTML forms, beginning with the Forgot Password form in Listing 4-10.

Listing 4-10. This is from `threadbear/resources/views/user/forgot-password.edge`

```
@layout("layout")
@section("header")
  <h1>Forgot password</h1>
@endsection
@section("content")
  <form method="post">
    <div class="form-group">
      <label for="email">Email address</label>
      <input type="email" class="form-control" id="email" name="email">
    </div>
    <button type="submit" class="btn btn-primary">Send</button>
  </form>
@endsection
```

The idea behind this form is that users who forget their passwords can submit their e-mail address. The application should generate a password-reset token and send it (as part of a link, in an e-mail) to the user's e-mail address.

Because we're creating something, it seems natural to use the POST request method. In theory, each time the user submits this form (whether or not they successfully reset their password), a new password-reset token should be created.

We're going to learn how to send e-mail later. For now, let's just ignore the requirement to create new password-reset tokens (or validate against them). The next step is to make the password-reset form, as in Listing 4-11.

Listing 4-11. This is from `threadbear/resources/views/user/reset-password.edge`

```
@layout("layout")
@section("header")
  <h1>Reset password</h1>
@endsection
@section("content")
  <form method="post">
    <div class="form-group">
      <label for="password">Password</label>
      <input type="password" class="form-control" id="password"
      name="password">
    </div>
    <div class="form-group">
      <label for="confirm-password">Confirm password</label>
      <input type="password" class="form-control" id="confirm-password"
      name="confirm-password">
    </div>
    <button type="submit" class="btn btn-primary">Reset</button>
  </form>
@endsection
```

This is similar to the other forms we've made, but there's a conceptual problem here. When a user completes this form, that user's account will be partially updated. The best HTTP method to describe this is `PATCH`, but browsers typically don't support `PATCH` request methods on HTML forms.

Fortunately, this isn't a new problem. Following the lead of many other web frameworks, AdonisJs provides a way for us to indicate that we want `PATCH` while still using `POST`. We just need to add a query string parameter to indicate the request method we mean, as in Listing 4-12.

Listing 4-12. This is from `threadbear/resources/views/user/reset-password.edge`

```
<form method="post" action="?_method=PATCH">
```

We still post the form, but AdonisJs picks up that we've defined `_method=PATCH` and reroutes the request accordingly. We need to adjust the route definition to account for this, as shown in Listing 4-13.

Listing 4-13. This is from `threadbear/start/routes.js`

```
Route.patch(
  "/reset-password/:token",
  ({ request, params }) => {
    // create new password reset token and send e-mail

    return JSON.stringify(request.all())
  },
)
```

Even though it depends on the `_method` query string parameter to know where to direct the request, AdonisJs is still smart enough to require `POST` requests for this behavior to take place. Even if we refresh a page with the `_method` parameter in the URL, AdonisJs will direct the request to the `GET` route handler.

Note Remember to update the routes and tests files as we go. I am doing it (and the source code will show you what to do if you get stuck), but showing all that work here would be boring. It's healthy to keep the tests passing, or you might not notice when something big breaks.

We need to do one more thing before this will work. In order for method spoofing (which is what this bit of trickery is called) to work, we need to enable it via some configuration changes. When we created the project, using the `adonis` command-line tool, we were given the "slim" application boilerplate. This means we got only the essential files to make the application work (and none of the extra stuff). The method-spoofing configuration may live inside that extra stuff.

I'll show you what the config file looks like (in Listing 4-14), and then I'll tell you where to find the other config files.

Listing 4-14. This is from `threadbear/config/app.js`

```
"use strict"

const Env = use("Env")

module.exports = {
  /*
  |--------------------------------------
  | App Key
  |--------------------------------------
  |
  | App key is a randomly...snip
  |
  */
  appKey: Env.get("APP_KEY"),

  http: {
    /*
    |--------------------------------------
    | Allow Method Spoofing
    |--------------------------------------
    |
    | Method spoofing allows to...snip
    |
    */
    allowMethodSpoofing: true,

    /*
    |--------------------------------------
    | Trust Proxy
    |--------------------------------------
    |
    | Trust proxy defines whether...snip
    |
    */
    trustProxy: false,

    /*
    |------------------------------------
    | Subdomains
    |------------------------------------
    |
    | Offset to be used for...snip
    |
    */
    subdomainOffset: 2,

    /*
    |------------------------------------
    | JSONP Callback
    |------------------------------------
    |
    | Default jsonp callback to be...snip
    |
    */
    jsonpCallback: "callback",
  },

  views: {
    /*
    |------------------------------------
    | Cache Views
    |------------------------------------
    |
    | Define whether or not to...snip
    |
    */
    cache: Env.get("CACHE_VIEWS", true),
  },

  static: {
    /*
    |-------------------------------------
    | Dot Files
    |-------------------------------------
    |
    | Define how to treat dot files...snip
    |
    */
    dotfiles: "ignore",

    /*
    |-------------------------------------
    | ETag
    |-------------------------------------
    |
    | Enable or disable etag generation
    |
    */
    etag: true,

    /*
    |-------------------------------------
    | Extensions
    |-------------------------------------
    |
    | Set file extension fallbacks...snip
    |
    */
    extensions: false,
  },

  locales: {
    /*
    |-------------------------------------
    | Driver
    |-------------------------------------
    |
    | The driver to be used for...snip
    |
    | file, database
    |
    */
    driver: "file",

    /*
    |------------------------------------
    | Default Locale
    |------------------------------------
    |
    | Default locale to be used by...snip
    |
    */
    locale: "en",

    /*
    |------------------------------------
    | Fallback Locale
    |------------------------------------
    |
    | Fallback locale to be used...snip
    |
    */
    fallbackLocale: "en",
  },
}
```

The config property we're interested in is `allowMethodSpoofing`. If this is set to `true`, AdonisJs will facilitate the method spoofing, which means we can use the correct (conceptually) request method but still support current browsers.

AdonisJs config files are usually quite well-documented. You can read through the comments and figure out what each config option does (and what values it accepts). Installing new libraries via `adonis install library-name` will also install the config files it has. For all other config files, we can head over to the GitHub repository for the fuller application boilerplate (`https://github.com/adonisjs/adonis-fullstack-app/tree/develop/config`).

Summary

In this chapter, we learned about how a request is put together. We looked at the correlation between HTTP 1.1 request methods and the actions they should perform, and how this relates to routing and forms in our application.

We also updated our tests, adding the ability to inspect markup and JSON response data. We're going to dig deeper into these topics in the next chapter. Specifically, we're going to see how to return different kinds of responses, redirections, and formats.

CHAPTER 5

Responses

In the previous chapter, we learned about requests. The flip side of that coin is responses. For each request that demands something of our application, a well-crafted JSON, XML, or HTML response is the answer. Perhaps the request is for a list of users or an API authentication. In this chapter, let's learn the ins and outs of responses.

Sending Simple Responses

We've already seen several ways to respond to HTTP requests. We started off by returning strings from our route closures. These strings were then rendered as HTML responses in the browser.

We reused this idea when we started to return JSON strings of request inputs. We did the same when we returned rendered Edge views. The process we used to render templates may have been a bit more complex than returning simple strings, but the result was the same.

That's because AdonisJs inspects the result of the closure, and if it sees a string, it sends this to the `Response` object behind the scenes. It's that response object we want to look at in more detail. We can access it directly by referencing another property of the router context, as demonstrated in Listing 5-1.

Listing 5-1. This is from `threadbear/start/routes.js`

```
Route.get(
  "/:customer",
  ({ response, view, params }) => {
    // return view.render("user/profile", {
    //     name: params.customer,
    // })
    return JSON.stringify(Object.keys(response))
  },
)
```

The exact keys returned aren't important, but the results show us that the response object lives in the context. When we return a string—and if the response object hasn't been modified in such a way that it would be prevented—then AdonisJs injects the string into the response object. It's similar to if we were to inject it manually, as in Listing 5-2.

Listing 5-2. This is from `threadbear/start/routes.js`

```
Route.get(
  "/:customer",
  ({ response, view, params }) => {
    // return view.render("user/profile", {
    //     name: params.customer,
    // })

    response.send(
      view.render("user/profile", {
        name: params.customer,
      }),
    )
  },
)
```

Sending JSON Responses

So, what else can we do with this `Response` object? For a start, we can set headers. Our JSON endpoints could define their `content-type` header so that they are correctly rendered, as in Listing 5-3.

Listing 5-3. This is from `threadbear/start/routes.js`

```
Route.post(
  "/register",
  ({ request, response }) => {
    // ...create new customer profile

    response.safeHeader(
      "content-type",
      "application/json",
    )
    response.send(JSON.stringify(request.all()))
  },
)
```

We use `safeHeader` instead of `header` because other parts of our application may already have set the content type. `safeHeader` sets the header only if it hasn't already been set. As you may imagine, returning JSON responses is quite common, and this code is verbose for something we would commonly do. Instead, we can use a special method, shown in Listing 5-4.

Listing 5-4. This is from `threadbear/start/routes.js`

```
Route.post(
  "/register",
  ({ request, response }) => {
    // ...create new customer profile

    response.json(request.all())
  },
)
```

The `json` method sets the content type and automatically serializes the parameters so that they are rendered as a well-formed JSON response.

Note If we want to remove a header for some reason, we can do it by using the `removeHeader` function. We can also use the `type` method as a shortcut to set the `content-type` header.

Working with Cookies

In the previous chapter, we looked at how to set cookies. Unsurprisingly, the response object has similarly named methods to be able to set them. We have access to both the `cookie` and `plainCookie` variants, as shown in Listing 5-5.

Listing 5-5.

```
response.cookie("language", "en-gb") // is encrypted
response.plainCookie("...", "...") // is unencrypted
```

There's also a `clearCookie` method, which will clear a cookie by name. If you plan to access the cookies (or set the cookies) in the browser, you should use `plainCookie`. The added encryption of `request.cookie` and `response.cookie` is great, but its use is limited if the browser isn't able to decrypt it.

Redirecting Users

Some operations require redirection. Perhaps we want to redirect after a successful form submission, or we want to allow users to rename their accounts without breaking their old account links.

It's this latter example we're going to try to implement now. Let's create a lookup table of new and old usernames that we'll update manually (at least for now). This could look similar to Listing 5-6.

Listing 5-6. This is from `threadbear/start/routes.js`

```
const redirects = {
  assertchris: "christopher",
  thetutlage: "harminder",
}

Route.get(
  "/:customer",
  ({ response, view, params }) => {
    const redirect = redirects[params.customer]
    if (redirect) {
      return response.redirect("/" + redirect)
    }

    // ...render normal profile view
  },
)
```

By passing a new URL to `response.redirect`, we can automatically redirect the browser to that new URL. In fact, we can even name routes and redirect to them by name. This is demonstrated in Listing 5-7.

Listing 5-7. This is from `threadbear/start/routes.js`

```
Route.get(
  "/:customer",
  ({ response, view, params }) => {
    const redirect = redirects[params.customer]

    if (redirect) {
      return response.route("profile", {
        customer: redirect,
      })
    }

    // ...render normal profile view
  },
).as("profile")
```

When we redirect to named routes that expect parameters, we need to provide those parameters in the form of an object. The benefit of naming routes is that we can change the format in a single place, and everything that refers to the route (or builds a URL from it) will automatically use the new format.

We could change `Route.get("/:customer", ...)` to `Route.get("/profile/:customer", ...)`, and our application would continue to function. Obviously, people linking to the old format will encounter 404 errors unless you set up redirects for the old format.

By default, the action at the other end of the redirection won't have access to the parameters that were present in the action that performed the redirection. Posted form parameters and query string parameters will be lost unless the second argument is set to `true`.

There's also a third argument, which defines which HTTP status code (301 or 302) should be used. The signature looks like `response.redirect("/url", sendParams = false, status = 302)`.

Responding with Attachments

There's one other trick specific to the `Response` object. It can respond with attachments, allowing users to download files of our choosing without being redirected to them. It's a powerful tool, but best left for when we look at how to work with the filesystem. For now, just know that it's there and look forward to when we use it in Threadbear!

Extending with Macros

Remember when we added methods to Edge in order to be able to use inside templates? Macros are a similar but more universal concept. Often our applications will grow to contain repeated operations. And as with templates, we'd benefit from having ways of extending `Request` and `Response` with these reusable methods.

Both of these classes extend the `Macroable` class. This little class allows macro functions and getters to be defined on other classes without extending them. Imagine we want to serve XML from some of our routes, something similar to `response.json(...)`. To begin with, let's install a helper library, which will convert from JSON objects to XML:

```
npm install --save js2xmlparser
```

Next, we need to add a macro function so that we can use `response.xml` as easily as we could `response.json`. Listing 5-8 shows one way we might achieve this.

Listing 5-8. This is from `threadbear/start/routes.js`

```
const parser = use("js2xmlparser")
const Response = use("Adonis/Src/Response")

Response.macro("xml", function(
  data,
  root = "root",
) {
  this.type("application/xml")
  this.send(parser.parse(root, data))
})

Route.get(
  "/:customer/products",
  ({ params, response }) => {
    const products = [
      { price: 4.99, title: "Teddy Bear" },
    ]

    if (params.format === ".xml") {
      return response.xml(
        {
          product: products.map(product => ({
            "@": { price: product.price },
            "#": product.title,
          })),
        },
        "products",
      )
    }

    if (params.format === ".json") {
      return response.json({
        products,
      })
    }

    // ...render normal view
  },
).formats(["xml", "json"])
```

We have yet to learn about the dependency injection container, but for now, you can think of `use("Adonis/Src/Response")` as a special kind of `require`. In this case, we're loading the `Response` object and adding a new macro function on it.

We do so by also loading the `js2xmlparser` library and converting whatever data we receive into an XML string. All XML documents must have a root element, and we can default that to `root` if no other is defined.

Note You can define these macros directly in the routes file (though we'll learn of better places to do this in the future). I have chosen to place the macros in `threadbear/start/hooks.js` instead. I've also gone ahead and used the hook system. We'll learn about that system later, but what's important is that the macro code is the same no matter which file you choose to put it in. The only difference is how that code is loaded and when.

Then we define a new route, which will show a list of products. We want to allow users to request XML, JSON, and HTML responses. We can tell these apart by the extension that is used in the URL. If the extension is `.xml`, we respond with XML, and so forth. This comes from the `params` context key. We could just as easily create a couple of macros to make this code cleaner, as shown in Listing 5-9.

Listing 5-9. This is from `threadbear/start/routes.js`

```
Response.macro("for", function(params, handlers) {
  if (params.format === ".xml") {
    const handler = handlers.xml
    const data = handler()

    return this.xml(...data)
  }

  if (params.format === ".json") {
    const handler = handlers.json
    const data = handler()

    return this.json(...data)
  }
  return (handlers.default || function() {})()
})

Route.get(
  "/:customer/products",
  ({ params, response }) => {
    const products = [
      { price: 4.99, title: "Teddy Bear" },
    ]

    return response.for(params, {
      xml: () => [
        {
          product: products.map(product => ({
            "@": { price: product.price },
            "#": product.title,
          })),
        },
        "products",
      ],
      json: () => [
        {
          products,
        },
      ],
      default: () => "...render normal view",
    })
  },
).formats(["xml", "json"])
```

Now we can short-circuit the `params.format` logic and respond to different request types with a switch-like macro. We could probably enhance the macro to be able to handle *any* type of extension, but these are the most common ones we'll see.

Summary

In this chapter, we learned all about the response object. We learned how to send simple responses, manage headers and cookies, and even how to redirect users. We also looked at how to define reusable functions on commonly used framework objects such as `Request` and `Response`.

In the next chapter, we're going to take a step back and learn about generators. We need to understand them and the promises that come after them before we can unlock the true power of AdonisJs. We'll tackle these two topics, one after the other, and come back to writing new code for our application.

CHAPTER 6

Generators

We're going to sidetrack for a bit. Recent changes to Node.js have introduced the `async` and `await` keywords. They're going to be useful for our application, but before we can use them, we should understand how they work.

In this chapter (and the next), we're going to learn how these keywords work. We'll explore how JavaScript has evolved to deal with asynchronous streams of events and data, and we'll see how to take full advantage of this evolution.

Arrays: Where It All Began

Arrays are the bread and butter of programming. Unlike the other variable types, arrays are perfect for managing lists of data, as shown in Listing 6-1.

Listing 6-1.

```
const items = ["foo", "bar", "baz"]

items.forEach((value, key) => {
  console.log(key, value)
})
```

There are a few kinds of loops, and each interacts with the array in subtly different ways. Arrays also have methods that can iterate, map, filter, and reduce the values of an array. Because JavaScript is a dynamic language, it's sometimes useful to be able to tell whether the value of a variable is an array. We can do that with the code in Listing 6-2.

Listing 6-2.

```
Array.isArray(items) // true
```

© Christopher Pitt 2018
C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_6

We could use these methods and language constructs almost exclusively for dealing with lists and iteration. But what if we want more control over what happens during iteration? Let me show you what I mean in the next section.

Then Came Iterators

Imagine we want to create something that works directly in a loop, but for each iteration, it makes a request to a remote server. Well, iterators are a step between normal arrays and being able to modify the behavior of loops. Imagine we have something that works like Listing 6-3.

Listing 6-3.

```
"use strict"

const fetch = require("node-fetch")

const requests = function(urls) {
  // ...make the requests here
}

const urls = [
  "http://adonisjs.com",
  "http://threadbear.store",
]
const iterator = requests(urls)

iterator.next().value
iterator.next().value
iterator.next().done // true
```

For this code to work, we need to return an object from `requests` that has a `next` method. This is the heart of what makes an iterator—something that knows how to access the next item in a list. The `requests` code may look something like Listing 6-4.

Listing 6-4.

```
const requests = function(urls) {
  // ...make the requests here
  let index = 0

  return {
    next: function() {
      if (index < urls.length) {
        return {
          value: fetch(urls[index++]),
          done: false,
        }
      }

      return {
        done: true,
      }
    },
  }
}
```

Although the index is less than the length of the array, we return the array item at that index (and remember to increment the index after). We also indicate that our iterator isn't done iterating yet. When that time comes, we return an object with `done` equal to `true`.

This is OK, but it doesn't help us to use the iterator in a loop. For that to work, we need to beef up our implementation, as shown in Listing 6-5.

Listing 6-5.

```
class Requests {
  constructor(urls) {
    this.urls = urls
    this[Symbol.iterator] = () =>
      new RemotesIterator(this)
  }
}

class RequestsIterator {
  constructor(remotes) {
    this.remotes = remotes
    this.index = 0
  }
  next() {
    if (this.index < this.remotes.urls.length) {
      return {
        value: fetch(
          this.remotes.urls[this.index++],
        ),
        done: false,
      }
    }

    return {
      done: true,
    }
  }
}

const iterableRequests = new Requests(urls)

for (let request of iterableRequests) {
  // ...do something with request
}
```

This code is similar to the `requests` function, but it places all the iterator code inside a class. The iterator class walks through the URLs in `Requests` and continues to maintain an `index` position. The reason for this change is that the `for...of` loop calls `Requests[Symbol.Iterator]` on the `iterableRequests` instance, and expects that method to return an iterable thing.

Now we've created custom classes, which we can use directly in loops. This example makes requests for every URL in the array, but you can probably think of many applications for this technique. And when you think about it, we've created a subtle shift from lists of static data to lists of operations to get static data.

When the iterator is created, none of the iterable data exists. We have the URLs but not the responses (or requests) we care about. The list of responses has become a list of operations to get each response.

Generators

What does this have to do with generators? They're not entirely dissimilar to iterators, but their implementation definitely is, as shown in Listing 6-6.

Listing 6-6.

```
const generateRequests = function*(urls) {
  for (let url of urls) {
    yield fetch(url)
  }
}

const generatedRequests = generateRequests(urls)

for (let request of generatedRequests) {
  // ...do something with request
}
```

See how much easier that was? Generators are a handy shortcut for the full iterator implementation we did. We don't need to make a special iterator handler class or even worry about `Symbol.iterator`. All we need to do is add a `yield` keyword, and Node.js knows to treat this function differently.

If we wanted, we could use generators in place of that iterator handler class, as shown in Listing 6-7.

Listing 6-7.

```
class Remotes {
  constructor(urls) {
    this.urls = urls

    this[Symbol.iterator] = function*() {
      for (let url of this.urls) {
        yield fetch(url)
      }
    }
  }
}
```

```
const remotes = new Remotes(urls)

for (let request of remotes) {
  console.log(request)
}
```

This is the best of both worlds. We can define well-structured list-like objects, which are expressive and concise (thanks to generator syntax). But this is only one facet of generators. I'm excited about an entirely different use of them, described next.

Cooperative Multitasking

As it happens, we can use generators to model asynchronous architecture. Suspend your disbelief for a moment and consider what it might look like to write code that looks synchronous (and simple) but is every bit as asynchronous as the callback-based code you're familiar with.

Let's try to build a task runner that alternates between tasks, giving each a little bit of time to do their work. In this model, each task does a bit of its work and then gives control back to the task running so that the task runner can give control over to another task. In this way, we can simulate parallel execution in the context of a single thread.

First, we need a task class, as in Listing 6-8.

Listing 6-8.

```
class Task {
  constructor(generator) {
      this.generator = generator()
      this.complete = false
  }

  run() {
      const result = this.generator.next()
      this.complete = result.done
  }
  incomplete() {
      return !this.complete
  }
}
```

Let's assume that the constructor argument will always be a generator function. We could add checks to make sure that this is the case, but they're unimportant for what I want to show you. When the `run` method is called, the generator is run for a single tick. We can think of this as one bit of work being done.

And because generators are similar to iterators, we can inspect the `done` property (on the result) to see whether more values are left in the generator. We can create a few tasks, as in Listing 6-9.

Listing 6-9.

```
const task1 = new Task(function*() {
  for (let i = 0; i < 5; i++) {
    console.log("task 1, iteration " + i)
    yield
  }
})

const task2 = new Task(function*() {
  for (let i = 0; i < 6; i++) {
    console.log("task 2, iteration " + i)
    yield
  }
})

const task3 = new Task(function*() {
  for (let i = 0; i < 7; i++) {
    console.log("task 3, iteration " + i)
    yield
  }
})
```

We're not doing any complex processing inside each of these generator functions. We're just writing to the console and yielding (so that the function becomes a generator). Next, let's create a task manager like the one in Listing 6-10.

Listing 6-10.

```
class Manager {
  constructor() {
    this.tasks = []
  }

  add(task) {
    this.tasks.push(task)
  }

  run() {
    while (this.tasks.length) {
      const next = this.tasks.shift()
      next.run()

      if (next.incomplete()) {
        this.add(next)
      }
    }
  }
}
```

The `Manager` class has a list of tasks, which we can `add` new tasks to. It also has a `run` method, which will step through each of the remaining tasks (in a first-in-first-out, or FIFO) order. Each task is then run. If there's more work for the task to do, we add it back into the manager's list. This way, each task will be run, in order, until it is out of work.

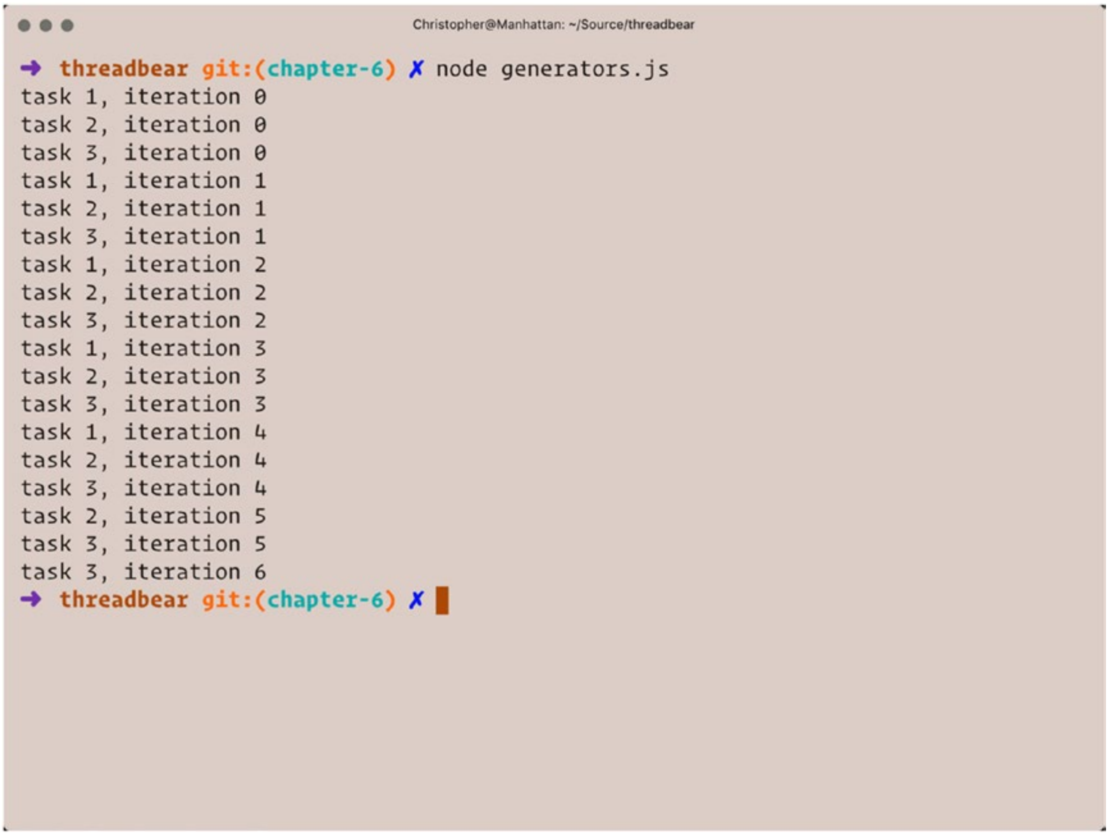
Finally, we can run the tasks by using the code in Listing 6-11.

Listing 6-11.

```
const manager = new Manager()
manager.add(task1)
manager.add(task2)
manager.add(task3)

manager.run()
```

That's it! All that's left to do is run the code. Figure 6-1 demonstrates what the output of the task runner looks like.



```
➜ threadbear git:(chapter-6) ✗ node generators.js
task 1, iteration 0
task 2, iteration 0
task 3, iteration 0
task 1, iteration 1
task 2, iteration 1
task 3, iteration 1
task 1, iteration 2
task 2, iteration 2
task 3, iteration 2
task 1, iteration 3
task 2, iteration 3
task 3, iteration 3
task 1, iteration 4
task 2, iteration 4
task 3, iteration 4
task 2, iteration 5
task 3, iteration 5
task 3, iteration 6
➜ threadbear git:(chapter-6) ✗
```

Figure 6-1. *Cooperative multitasking with generators*

Summary

It may be a little tricky to see how this relates to `async` and `await`, but trust me, it does. Generators are a powerful tool for shifting the focus from "list of items" to "list of operations." By using iterators and generators, we can break up a list of operations (which we'd like to execute asynchronously) and allow them to be iterated over.

In the next chapter, we're going to learn about promises. We'll look at what they are and how to use them. We'll also come to understand how they fit together with generators, to explain `async` and `await`.

CHAPTER 7

Promises

In the previous chapter, we learned about generators: how they work, and one or two things they can be useful for. We ended with a demonstration of building a multitasking system. In particular, we looked at how to use generators as interruptible functions. Other programming languages call this kind of function a *coroutine* (short for *cooperative routine*), because it cooperates to allow for multitasking.

In the grand scheme of things, promises come before coroutines. They're slightly harder to understand, though. They also don't resolve (if you'll pardon the pun) into a nice system until we start to use them in conjunction with coroutines.

In this chapter, we're going to take a look at callback-based code. We'll learn what's going on underneath and how callbacks fit in. Then we'll look at how generators allow us to execute small parts of an asynchronous application. Finally, we'll learn how to use promises and coroutines to create our own asynchronous libraries. This chapter is gonna be wild!

Reacting to Events

Have you ever worked on the front end? If you have, you'll know that no order of execution is guaranteed. That may be a bit of an exaggeration, but often you might expect a linear execution of code, only to be surprised by out-of-order execution.

This leads to strange race conditions, and the need for a complex mental model. Take the code in Listing 7-1, for example.

Listing 7-1.

```
const body = document.querySelector("body")
const image = new Image()

image.onload = function() {
  const element = document.createElement("img")
  element.setAttribute("src", this.src)
  body.appendChild(element)
}

body.addEventListener("click", function() {
  image.src = "https://placekitten.com/200/300"
})
```

We create an `Image` object and assign an `onload` callback; when the image is loaded, it will create a new `img` element and append this to the document body.

However, we initiate the load only when the body is clicked. Nothing happens as a result of running this script. Yet something *can* happen when someone clicks the body element.

This kind of reactivity is possible because browsers (and JavaScript runtimes) have event loops. These event loops continuously cycle, checking for user input and situations in which events have happened and should be dispatched to event listeners.

These event loops are the cornerstone enabling browsers to handle mountains of event listeners and the code they contain. Node.js has been built with an event loop, so it is capable of asynchronous, reactive architecture.

Entering Callback Hell

One of the problems with asynchronous code is the mountain of callbacks that tend to be involved. It's a situation known commonly as *Callback Hell*, though it's often exaggerated. Callbacks (or *closures*, as they're commonly referred to) are at the heart of how Listing 7-2 works.

Closures are a way of encapsulating the state of code you'd like to run at a future time. They're behavior packaged in a neat box. You can reason about them easily: "What do I want to happen when the domain name is resolved to an IP address? I can provide a closure, and the resolver can execute it when the results."

But such code can be heavily nested, as events depend on other events completing, in a sort of sideways-code-pyramid. Enough people thought this was a problem to think about a neat abstraction for it: promises.

Understanding Promises

Promises are a way of organizing closures so there's never more than one level of closure nesting. Listing 7-2 shows what promises look like.

Listing 7-2.

```
const promise = new Promise((resolve, reject) => {
  // ...perform some async action
  // ...either resolve(data) or reject(reason)
})

promise.then(data => {
  // ...do something with data
})

promise.catch(reason => {
  // ...do something with reason
})
```

What we learn from this is that promises don't help us to avoid all closures. They just provide a framework for organizing them and for deciding when and in which order to resolve them (in an asynchronous system).

Promises represent eventual values. A function may return a promise, but that promise doesn't necessarily have a real value behind it yet. The promise can eventually resolve to a real value and then trigger the callbacks given to `then`.

Let's create a function that returns a promise and resolves after a few seconds. It might look something like Listing 7-3.

Listing 7-3.

```
const delay = (
  shouldResolve,
  having,
  after = 1000,
) => {
  return new Promise((resolve, reject) => {
    if (shouldResolve) {
      setTimeout(() => resolve(having), after)
    } else {
      setTimeout(() => reject(having), after)
    }
  })
}

delay(true, "some data").then(data => {
  console.log("resolved with " + data)
})

delay(false, "a reason").catch(reason => {
  console.log("rejected with " + reason)
})
```

When we create a new promise, we define when it should be resolved or rejected by calling the callbacks given to the constructor. Though we attach a `then` and `catch` to the results of `delay`, those callbacks are executed only after the delay we specify. The callbacks are attached immediately, but they aren't executed immediately.

There's also no limit to the number of `then` or `catch` callbacks that can be attached to a single promise. Given a promise, we can go on attaching as many new callbacks as we want. One application of this is that functions can return promises from other functions.

Using Promises with Coroutines

Many of the libraries you're likely to use these days will be geared to return promises. You may not have to write a lot of your own promise code as a result. But you will need to compose many promise-making libraries, and that can get tricky.

We've seen how generators can lead to multitasking systems, and how promises can help us arrange callback-based code. Now let's combine them!

Let's change some of Listing 7-3 to use new keywords. Listing 7-4 introduces `async` and `await`.

Listing 7-4

```
const delay = (
  shouldResolve,
  having,
  after = 1000,
) => {
  return new Promise((resolve, reject) => {
    if (shouldResolve) {
      setTimeout(() => resolve(having), after)
    } else {
      setTimeout(() => reject(having), after)
    }
  })
}

// delay(true, "some data").then(data => {
//     console.log("resolved with " + data)
// })

// delay(false, "a reason").catch(reason => {
//     console.log("rejected with " + reason)
// })

const run = async () => {
  const data = await delay(true, "some data")
  console.log("resolved with " + data)

  try {
    await delay(false, "a reason")
  } catch (reason) {
    console.log("rejected with " + reason)
  }
}

run()
```

I've kept the `delay` function because I think it's important to see that it's exactly the same as it was before. `async` and `await` don't require that we change the libraries or frameworks we use. So long as they return promises, `async` and `await` will work!

Inside a function marked as `async`, we can start to use the `await` function as a way to help resolve promises. We can write code that looks synchronous and has far fewer callbacks, but still works asynchronously.

When we want to get to the value that would have been passed to the `resolve` callback, we can assign it to a variable/constant—as long as we also use the `await` keyword. If we forget that, the promise is assigned to the variable/constant.

If we want to get to the reason for a rejection, we need to wrap the code that will cause the rejection inside a try-catch block.

Future versions of the JavaScript language will probably support a global, or top-level, `await` keyword, but for now, we need to wrap `await` inside functions marked with `async`. I've used the fat-arrow function syntax, but we could also use `const foo = async function() {...}`.

Making Promises

We've already looked at how to create new functions, but one area we need to study is how to convert traditional callback-based functions into those that return promises. Consider the Node.js filesystem code in Listing 7-5.

Listing 7-5.

```
const fs = require("fs")

fs.readFile(".gitignore", (error, data) => {
  if (error) {
    throw error
  }

  console.log(data.toString())
})
```

This function, from the Node.js standard library, still uses callbacks to act after a file has been read. How could we convert this to a function that returns a promise? We could probably use something similar to Listing 7-6.

Listing 7-6.

```
const readFile = path => {
  return new Promise((resolve, reject) => {
    fs.readFile(path, (error, data) => {
      if (error) {
        reject(error)
      } else {
        resolve(data)
      }
    })
  })
}

readFile(".gitignore").then(data =>
  console.log(data.toString()),
)
```

Now that this is returning a promise, we can use `async` and `await`, as in Listing 7-7.

Listing 7-7.

```
const run = async () => {
  const data = await readFile(".gitignore")
  console.log(data.toString())
}

run()
```

In fact, this format of Node.js callback functions is so common that a few libraries can perform this transformation for us. We can use a library such as promisify-node (`www.npmjs.com/package/promisify-node`):

```
npm install promisify-node
```

This will allow us to import Node.js modules so that they produce promises, as shown in Listing 7-8.

Listing 7-8

```
const promisify = require("promisify-node")
const fs = promisify("fs")

const run = async () => {
  const data = await fs.readFile(".gitignore")
  console.log(data.toString())
}

run()
```

Bringing It Back to AdonisJs

We've had to take this detour—into generators and promises—because much of AdonisJs uses `async` and `await`. Look at the database code example in Listing 7-9.

Listing 7-9.

```
const Database = use("Database")

// ...some time later
const users = await Database.table("users").select("*")
```

Many of the actions we take as back-end developers are asynchronous actions: reading and writing from the database, making HTTP requests to remote servers, moving files around in the filesystem, and more.

If we had to run these synchronously, we'd be wasting resources that could have been spent responding to more browser requests. If we had to do all of them with traditional callback functions, we'd have a mountain of hard-to-read code. If we had to use promises alone, we'd still not get away from all the callbacks.

Using promises together with a coroutine-like execution model (like the one we saw with generators) gets us much closer to clean code.

And we can use `async` and `await` alongside promise functions to have incredible flow control. Want to run multiple things in parallel and resolve only after all of them have? Check out Listing 7-10.

Listing 7-10.

```
const run = async () => {
  // const data = await fs.readFile(".gitignore")
  // console.log(data.toString())

  try {
    const [gitignore, env] = await Promise.all([
      fs.readFile(".gitignore"),
      fs.readFile(".env.example"),
    ])
    console.log(
      gitignore.toString(),
      env.toString(),
    )
  } catch (e) {
    // ...one of the files couldn't be read
  }
}

run()
```

We can use promise-making functions with the promise static functions. Take a look at the docs at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#Methods` to discover other variations and applications of this idea. For now, it's time for us to wrap up.

Summary

We've just about finished our sidetracking. While we've been away from the main Threadbear application, we've learned about generators and how to use them to model sequences of asynchronous behavior. We've also learned about promises, and how they help us to reduce and repackage callbacks into more readable asynchronous code.

In this chapter, we learned how to connect the two. Wonderful asynchronous code is possible when you understand how the `async` and `await` keywords work and how to stitch them together with `Promise` functions.

In the next chapter, we're going to get back to the Threadbear application. We'll start to organize our application better with controllers and learn how to apply global rules to our application with middleware.

CHAPTER 8

Controllers and Middleware

In the previous chapter, we wrapped up the theory behind `async` and `await`. Let's get back to the main Threadbear application, beginning with a reorganization of code!

In this chapter, we're going to learn about controllers and middleware. These are two powerful concepts for organizing and reusing code. The first will help us to slim down our routes file, and the second will help us to add reusable things such as authentication and content-negotiation features.

Making Controllers

So far, we've been putting all of our request and response code into the routes file. That's OK for small projects, but it doesn't scale well. We don't want all of our application code sitting in one file, and we certainly don't want to have to go searching for code in a huge file.

So let's create a new controller:

```
adonis make:controller CustomerController
```

This creates an almost-empty controller file, in `threadbear/app/Controllers/Http`, as shown in Listing 8-1.

Listing 8-1. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
"use strict"

class UserController {}

module.exports = UserController
```

© Christopher Pitt 2018
C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_8

We can begin to move some of our route handlers to this class so that they're no longer cluttering up the routes file. Listing 8-2 demonstrates some of this.

Listing 8-2. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
"use strict"

const redirects = {
  assertchris: "christopher",
  thetutlage: "harminder",
}

class CustomerController {
  showLogin({ view }) {
    return view.render("customer/login")
  }

  doLogin() {
    // create new customer session
    return "POST /login"
  }

  logout() {
    // expire current customer session
    return "PUT /logout"
  }

  showRegister({ view }) {
    return view.render("customer/register")
  }

  doRegister({ request, response }) {
    // ...create new customer profile
    response.json(request.all())
  }

  showForgotPassword({ view }) {
    return view.render("customer/forgot-password")
  }
```

```
doForgotPassword({ request }) {
  // create new password reset token and send e-mail
  return JSON.stringify(request.all())
}

showResetPassword({ view, params }) {
  return view.render(
    "customer/reset-password",
    {
      token: params.token,
    },
  )
}

doResetPassword({ request }) {
  // create new password reset token and send e-mail
  return JSON.stringify(request.all())
}

showProfile({ params, response, view }) {
  const redirect = redirects[params.customer]

  if (redirect) {
    return response.route("profile", {
      customer: redirect,
    })
  }

  response.send(
    view.render("customer/profile", {
      name: params.customer,
    }),
  )
}

updateProfile({ params }) {
  // update customer profile
  return "PUT /:customer " + params.customer
}
```

```
  deleteProfile({ params }) {
    // delete customer profile
    return "DELETE /:customer " + params.customer
  }
}

module.exports = CustomerController
```

All we've done is move the route handlers from the routes file to `CustomerController`. We're accepting the same handler callback parameters as before, but we're using class method syntax. We've also had to declare those redirects as a constant before the class.

The real benefit can be seen in Listing 8-3.

Listing 8-3. This is from `threadbear/start/routes.js`

```
Route.get(
  "/login",
  "CustomerController.showLogin",
)
Route.post("/login", "CustomerController.doLogin")
Route.put("/logout", "CustomerController.logout")
Route.get(
  "/register",
  "CustomerController.showRegister",
)
Route.post(
  "/register",
  "CustomerController.doRegister",
)
Route.get(
  "/forgot-password",
  "CustomerController.showForgotPassword",
)
Route.post(
  "/forgot-password",
  "CustomerController.doForgotPassword",
)
```

```
Route.get(
  "/reset-password/:token",
  "CustomerController.showResetPassword",
)
Route.patch(
  "/reset-password/:token",
  "CustomerController.doResetPassword",
)
Route.get(
  "/:customer",
  "CustomerController.showProfile",
).as("profile")
Route.put(
  "/:customer",
  "CustomerController.updateProfile",
)
Route.delete(
  "/:customer",
  "CustomerController.deleteProfile",
)

// Route.get("/login", ({ view }) => {
//     return view.render("customer/login")
// })

// ...snip

// Route.delete("/:customer", ({ params }) => {
//     return "DELETE /:customer " + params.customer
// })
```

We've replaced about 60 lines of route-handler code with 11 lines of routing code. All of these routes reference the `CustomerController` class we've created, and AdonisJs is smart enough to add the directory prefix to the class name so that everything is neatly resolved.

Note I've taken the opportunity to rename `user` to `customer` in the view paths, because I think this fits better with the business logic of Threadbear. You can call your entities whatever you want, but be aware of this change in the example code.

Reusing Code

Apart from the improvement to our routes file, controllers also present us with a means to share code. Imagine we want to reuse the `response.json(request.all())` code. One way we could do that is to inherit from a common controller, which we create in Listing 8-4.

Listing 8-4. This is from `threadbear/app/Controllers/Http/Controller.js`

```
"use strict"

class Controller {
  showRequestParameters({ request, response }) {
    response.json(request.all())
  }
}

module.exports = Controller
```

We can inherit from this by resolving it out of the IoC (or Inversion of Control) container. Don't worry about how this works for now. Just think of it as a way for AdonisJs to find and return this class in another file. Let's use this as our base controller, shown in Listing 8-5.

Listing 8-5. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
"use strict"

const Controller = use(
  "App/Controllers/Http/Controller",
)

class CustomerController extends Controller {
  doRegister(context) {
    // ...create new customer profile
    this.showRequestParameters(context)
  }

  doForgotPassword(context) {
    // create new password reset token and send e-mail
    this.showRequestParameters(context)
  }

  doResetPassword(context) {
    // create new password reset token and send e-mail
    this.showRequestParameters(context)
  }

  // ...snip
}

module.exports = CustomerController
```

It's not earth-shatteringly important, but this illustrates one of the ways in which we can reuse code. When we get to validating input, shared controller functions become quite useful.

AdonisJs gives us all these ways to reuse code. Whether we use macros or common base controllers or even middleware depends on which fits our use case best and which we are most comfortable with. No one place is best for keeping shared functionality, so use what works best for you and your application.

Using Middleware

I've mentioned middleware a few times now, so let's learn about it. One way to think about middleware is as a series of functions that happen as a request is made to the application. If browser requests are at one end of a transaction, and route handlers are at the other, middleware is in the middle.

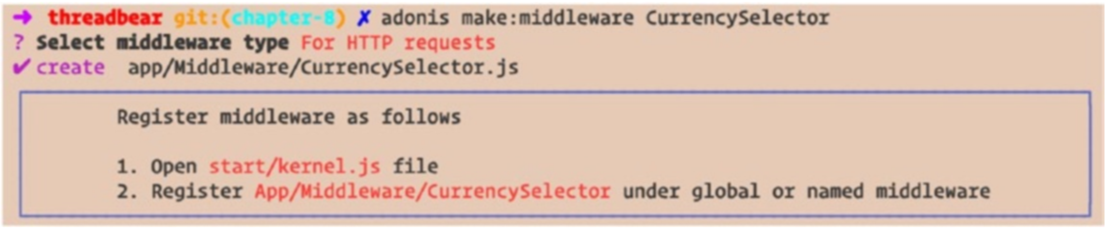
Parts of AdonisJs (which we haven't yet seen) depend heavily on middleware. When we start to create customer sessions, we'll use the authentication middleware to protect against unauthorized access to parts of our application. The very act of logging in to the application will make use of the session middleware to store the users' profile information.

If we were to create a full JSON API, we'd probably look at ways to secure that. Again, we'd use some sort of authentication middleware to make sure the endpoints were being accessed by an authorized client. We'd also look at ways to throttle API traffic (to prevent scraping and/or denial-of-service attacks). This we could also do with middleware.

Let's create some simple middleware to explore its structure. We can create this by running another Ace command:

```
adonis make:middleware CurrencySelector
```

This displays a prompt, asking what kind of middleware we'd like to make. We're working on the HTTP layer of our application, so we can select that. When we select this, Ace makes the new middleware file and displays instructions, as we can see in Figure 8-1.



```
→ threadbear git:(chapter-8) ✗ adonis make:middleware CurrencySelector
? Select middleware type For HTTP requests
✔ create  app/Middleware/CurrencySelector.js

        Register middleware as follows

        1. Open start/kernel.js file
        2. Register App/Middleware/CurrencySelector under global or named middleware
```

Figure 8-1. *Instructions for registering the middleware*

Let's add `CurrencySelector` to `kernel.js`, as in Listing 8-6.

Listing 8-6. This is from `threadbear/start/kernel.js`

```
const globalMiddleware = [
  "Adonis/Middleware/BodyParser",
  "App/Middleware/CurrencySelector",
]
```

As the name suggests, we're going to use this middleware to detect the user's location and to select the appropriate currency for displaying product pricing. To do this, we're also going to need to know a bit about where the customer is from. Let's install a library to help with this:

```
npm install --save geoip-lite
```

We can use this library to find the location of a user, based on the user's IP address. Listing 8-7 demonstrates how this code can look.

Listing 8-7. This is from `threadbear/app/Middleware/CurrencySelector.js`

```
"use strict"

const geoip = use("geoip-lite")

class CurrencySelector {
  async handle({ request }, next) {
    const ip = request.ip()

    const reference = geoip.lookup(
      "207.97.227.239",
    )

    if (!reference) {
      request.country = "Unknown"
      request.currency = "US"
    } else {
      request.country = reference.country
    }

    await next()
  }
}

module.exports = CurrencySelector
```

We fetch the `geoip` library (using that function similar to `require`). In the `handle` method, we have access to the same context we've been using in our route handlers. So, we can access the request and response objects.

We get the IP address from the request and use it to guess the location, via the `geoip` library. Finally, we await the next middleware in line. After all the middleware has run, the request is sent for routing.

This code is similar to that of the middleware docs. Let's take it a step further by mapping currencies (and their symbols to the discovered country). We'll need another library to help us with this:

```
npm install --save currency-map-country
```

This gives us access to a few functions, which we can use to figure out the currency and even the currency format we should use. Listing 8-8 shows how we might go about this.

Listing 8-8. This is from `threadbear/app/Middleware/CurrencySelector.js`

```
"use strict"

const geoip = use("geoip-lite")

const {
  getCountry,
  getCountryByAbbreviation,
  getCurrency,
} = use("currency-map-country")

class CurrencySelector {
  async handle({ request }, next) {
    const ip = request.ip()
    const reference = geoip.lookup(ip)

    let currency


    if (!reference) {
      request.country = "Unknown"
      request.currency = "USD"

      currency = getCurrency("USD")
    } else {
      request.country = reference.country

      const name = getCountryByAbbreviation(
        request.country,
      )

      const country = getCountry(name)
      request.currency = country.cur

      currency = getCurrency(country.cur)
    }
    request.currencyFormat = currency.symbolFormat

    await next()
  }
}

module.exports = CurrencySelector
```

With `country`, `currency`, and `currencyFormat` in hand, we can now display accurate local pricing for customers. We can access these properties as would we would any others in our route handlers.

We're also seeing more of `async` and `await`, as we use and create asynchronous code in AdonisJs. In this case, it's because any middleware can be asynchronous. So, we call `next` with that in mind.

We'll revisit the authentication and session middleware as we start to store and verify customer credentials from the database. They work similarly, in that they are run before routing and can alter the context properties. Middleware can even terminate requests with responses or exceptions. A failed authorization check could display an error before any routing takes place!

Summary

In this chapter, we learned about various ways to reuse code and organize our application. We learned a bit about controllers, which act as the glue between routes and views. They're also where we'll start to initiate our database work and form validation.

We also learned about middleware. We created our own middleware to figure out and store localization data for the user. From here, we'll be able to display accurate currency and country information to them, and we'll have access to this data everywhere, because global middleware is run on every request.

Note In future chapters, we'll see how to selectively run middleware on a route-by-route basis.

In the next chapter, we're going to start to use the database. We'll store and retrieve data from it, and we'll use this data in interesting ways as we continue our journey.

CHAPTER 9

Databases

In the previous chapter, we started to organize our code into controllers and to reuse code with middleware. I mentioned we'd be seeing authentication and session middleware just as soon as we started working with the database. That time has come.

In this chapter, we're going to learn about schema migrations. We'll use these to create a code blueprint of the database structures we want our application to use. Then we'll start to write and read from these structures by using SQL (or Structured Query Language) queries.

Finally, we'll take a look at Lucid, which is an abstraction that takes care of a lot of manual SQL work. At the end of this chapter, we should have all the bits we need to allow customers to create new accounts and authenticate with them.

Installing Lucid

Before we get going, we need to install some dependencies. These give us access to things like the schema migration creator and the database driver:

```
adonis install @adonisjs/lucid
```

Because we want to make migrations *and* use the database directly, we need to register two providers, as shown in Listing 9-1.

Listing 9-1. This is from `threadbear/start/app.js`

```
const providers = [
  "@adonisjs/lucid/providers/LucidProvider",
  // ...other global providers
]
```

C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_9

```
const aceProviders = [
  "@adonisjs/lucid/providers/MigrationsProvider",
  // ...other ace providers
]
```

Ace added a new database configuration file as part of the installation process. We can change it to reflect our database details or to pick the provider we want. It resembles Listing 9-2.

Listing 9-2. This is from `threadbear/config/database.js`

```
"use strict"

const Env = use("Env")
const Helpers = use("Helpers")

module.exports = {
  /*
  |--------------------------------------------------
  | Default Connection
  |--------------------------------------------------
  |
  | ...
  */
  connection: Env.get("DB_CONNECTION", "sqlite"),

  /*
  |--------------------------------------------------
  | Sqlite
  |--------------------------------------------------
  |
  | ...
  */
  sqlite: {
    client: "sqlite3",
    connection: {
      filename: Helpers.databasePath(
        "development.sqlite",
      ),
    },
    useNullAsDefault: true,
  },

  /*
  |--------------------------------------------------------
  | MySQL
  |--------------------------------------------------------
  |
  | ...
  */
  mysql: {
    client: "mysql",
    connection: {
      host: Env.get("DB_HOST", "localhost"),
      port: Env.get("DB_PORT", ""),
      user: Env.get("DB_USER", "root"),
      password: Env.get("DB_PASSWORD", ""),
      database: Env.get("DB_DATABASE", "adonis"),
    },
  },

  /*
  |--------------------------------------------------------
  | PostgreSQL
  |--------------------------------------------------------
  |
  | ...
  */
  pg: {
    client: "pg",
    connection: {
      host: Env.get("DB_HOST", "localhost"),
      port: Env.get("DB_PORT", ""),
      user: Env.get("DB_USER", "root"),
      password: Env.get("DB_PASSWORD", ""),
      database: Env.get("DB_DATABASE", "adonis"),
    },
  },
}
```

By default, Lucid uses a SQLite database. This won't work out of the box, as we need to also install a Node.js library to support this database (which it tells us to do in the comment, in the SQLite portion of the configuration file):

```
npm i --save sqlite3
```

You don't have to stick to using SQLite, but you do need to install the appropriate alternative on your server. If you like MySQL, install that server and the corresponding Node.js libraries into your application.

If you stick with SQLite, be sure to also create the empty database file:

```
mkdir threadbear/database
touch threadbear/database/development.sqlite
```

Note Lucid may automatically create this directory and file as soon as we start interacting with the database.

This creates an empty file, which the database driver will then be able to turn into a valid SQLite database. It's probably also a good idea to ignore this `.sqlite` file, so it isn't committed to GitHub. Your `.gitignore` file may resemble Listing 9-3.

Listing 9-3. This is from `threadbear/database/.gitignore`

```
*.sqlite
```

We can test the connection to the database with the code in Listing 9-4.

Listing 9-4. This is from `threadbear/start/routes.js`

```
const Route = use("Route")
const Database = use("Database")

Route.get("/", async ({ view }) => {
  const result = await Database.raw(
    "SELECT CURRENT_TIME as time",
  )
  console.log(
    "when you hear the beep, it will be " +
      result[0].time,
  )

  return view.render("page/home")
})

// ...other routes
```

We'll come back to this `Database.raw` behavior, but just know that it's executing a SQL query against the SQLite database connection. `CURRENT_TIME` is a kind of *magic constant* that returns the current time of the database. `Database.raw` returns an array of rows. We pick the first one and get the `time` property, which is what we aliased `CURRENT_TIME` to.

Note I don't want to make this a chapter about writing SQL queries. I assume you have some experience with writing SQL queries or that you're capable of learning how they work from one of the many great resources on this topic.

If you'd like to learn about MySQL, check out *Beginning MySQL Database Design and Optimization* by Jon Stephens and Chad Russell (Apress, 2004).

Making Migrations

Before we can store any products, we need a database in which to store them. We now have a database file, but we need to define the tables in which those products (and customers) will be stored.

Migrations are a good way to define this structure. They give us a blueprint for the database structures we want to be able to use, and we can commit this blueprint to GitHub or any other code repository.

Migrations also help our teammates start off on a good footing. Setting up a project no longer requires that they search (or ask) for a database dump file. Given a working database connection and the migration files, they can run the migrations. After that, they'll have the full database working and ready.

Remember how we used a JavaScript object to store redirects for the profile page? Let's store those redirects in the database, in a table we create from a migration. We can use Ace to create the empty migration for us:

```
adonis make:migration redirects_schema
```

This creates a file similar to Listing 9-5.

Listing 9-5. This is from `threadbear/database/...redirects_schema.js`

```
"use strict"

const Schema = use("Schema")

class RedirectsSchema extends Schema {
  up() {
    this.create("redirects", table => {
      table.increments()
      table.timestamps()
    })
  }

  down() {
    this.drop("redirects")
  }
}

module.exports = RedirectsSchema
```

There are two parts to a migration file. The first is what must happen when the table is created or modified. The second is what to do when that modification must be undone. It's entirely reasonable to ignore the second part (or to make it remove the table altogether). It depends on how you want to use the migration system.

Note If you want to be able to roll back to any valid state of the database, you want to put more time and energy into the down method. If you only want to set and/or reset the database to a "good state," you can be less gentle inside down.

By default, the created table will have an `id` column, as well as columns for `created_at` and `updated_at`. If you'd like to view this structure, consider using a SQLite database browser (`http://sqlitebrowser.org`). Let's add a couple of string columns to this, so we can store the profile name we're redirecting from and the profile name we're redirecting to. We can see how this is done in Listing 9-6.

Listing 9-6. This is from `threadbear/database/..._redirects_schema.js`

```
this.create("redirects", table => {
  table.increments()
  table.string("from")
  table.string("to")
  table.timestamps()
})
```

To persist these migrations to the database, we need to run the following:

```
adonis migrate:run
```

But, if we want to alter a migration and rerun it (or need to roll back all migrations), we can use this:

```
adonis migrate:reset
```

Next, let's add some rows to this table. We can create a temporary route for this, as shown in Listing 9-7.

Listing 9-7. This is from `threadbear/start/routes.js`

```
Route.get("/add-redirects", async () => {
  const created_at = Database.raw("CURRENT_TIME")

  await Database.insert({
    from: "assertchris",
    to: "christopher",
    created_at,
  }).into("redirects")

  await Database.insert({
    from: "thetutlage",
    to: "harminder",
    created_at,
  }).into("redirects")

  return "done"
})
```

Here, again, we see `Database.raw`. We also see a series of `Database.insert(...).into(...)` statements, which add new rows to the table. These are asynchronous operations, so we have to `await` each one.

By default, all the values we try to insert will be escaped. That's why we need to use `Database.raw` : so we can insert the *magic constant* `CURRENT_TIME` and not a literal string. In this context, `Database.raw` isn't an entire query, as we saw before, but rather a special query value. If we try to `await` it, we'll get an error because it's not a valid SQL query.

Now let's replace the lookup table with a database lookup, as in Listing 9-8.

Listing 9-8. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async showProfile({ params, response, view }) {
  const rows = await Database.select(
    "from",
    "to",
  ).from("redirects")

  const redirects = rows.reduce(
    (accumulator, row) => {
      accumulator[row.from] = row.to
      return accumulator
    },
    {},
  )

  const redirect = redirects[params.customer]

  // ...redirect to new location
}
```

We begin by fetching all the redirects from the database. We need only `from` and `to`, so we limit our query to those columns. Then we reduce the array of rows by adding each `from` as a key and each `to` as a value, to an empty object. The rest of our code stays the same.

We can start managing these redirects from the database. We could even create a form through which customers can create their own redirects (in cases where they want to rename their profiles).

Seeding the Database

Our temporary route, to insert the redirects we want, works. It's not optimal, though. What if it is run more than once? There's no code to prevent duplicates from being inserted. And do we really want a public endpoint to perform database work, without any validation or throttling?

There's a better solution, called *seeding*. Persisting a database schema is only half of the process. Seeding is the other half, which lets us populate the database with initial data. We can create a new seeder with the following command:

```
adonis make:seeder
```

This creates an empty seeder for us, similar to Listing 9-9.

Listing 9-9. This is from `threadbear/database/seeds/DatabaseSeeder.js`

```
"use strict"

/*
|--------------------------------------------------------
| DatabaseSeeder
|--------------------------------------------------------
|
| ...
|
*/

const Factory = use("Factory")

class DatabaseSeeder {
  async run() {}
}

module.exports = DatabaseSeeder
```

We can move the code that created the redirects to this file, as in Listing 9-10.

Listing 9-10. This is from `threadbear/database/seeds/DatabaseSeeder.js`

```
async run() {
  const created_at = Database.raw(
    "CURRENT_TIME",
  )

  await Database.insert({
    from: "assertchris",
    to: "christopher",
    created_at,
  }).into("redirects")

  await Database.insert({
    from: "thetutlage",
    to: "harminder",
    created_at,
  }).into("redirects")
}
```

Now this code is no longer exposed to the world, and we can run it every time we reset the database:

```
adonis migrate:reset
adonis migrate:run
adonis seed
```

The `DatabaseSeeder` also mentions factories, but we're not going to look at them yet. Instead, we're going to create another couple of migrations and seed them with test data. To start, let's create migrations for customers and products, as shown in Listings 9-11 and 9-12.

Listing 9-11. This is from `threadbear/database/..._customers_schema.js`

```
"use strict"

const Schema = use("Schema")

class CustomersSchema extends Schema {
  up() {
    this.create("customers", table => {
      table.increments()
      table.string("first_name")
      table.string("last_name")
      table.string("email").unique()
      table.string("password")
      table.string("nickname").unique()
      table.timestamps()
    })
  }

  down() {
    this.drop("customers")
  }
}

module.exports = CustomersSchema
```

We want e-mails and nicknames to be unique, because customers will use e-mail to log in, and nicknames will be used to display profile pages.

Listing 9-12. This is from `threadbear/database/..._products_schema.js`

```
"use strict"

const Schema = use("Schema")

class ProductsSchema extends Schema {
  up() {
    this.create("products", table => {
      table.increments()
      table.string("name")
      table.integer("price")
      table.integer("customer_id")
      table.timestamps()
    })
  }
  down() {
    this.drop("products")
  }
}

module.exports = ProductsSchema
```

Each product needs a name and a price. It's more reliable (thanks to rounding errors and storage mechanisms) to store price in cents, so we use an `integer` field type for this. We also need to link products to customers, so we can display them on the appropriate profile page.

Next, let's populate these tables with test data, as shown in Listing 9-13.

Listing 9-13. This is from `threadbear/database/seeds/DatabaseSeeder.js`

```
"use strict"

const Database = use("Database")
const Factory = use("Factory")
const Hash = use("Hash")

class DatabaseSeeder {
  async run() {
    const created_at = Database.raw(
      "CURRENT_TIME",
    )

    // ...insert redirects

    await Database.insert({
      first_name: "Harminder",
      last_name: "Virk",
      email: "virk.officials@gmail.com",
      password: await Hash.make("harminder123"),
      nickname: "harminder",
      created_at,
    }).into("customers")
    const ids = await Database.insert({
      first_name: "Christopher",
      last_name: "Pitt",
      email: "cgpitt@gmail.com",
      password: await Hash.make("christopher123"),
      nickname: "christopher",
      created_at,
    }).into("customers")

    await Database.insert({
      name: "Soft Teddy",
      price: 499,
      customer_id: ids[0],
      created_at,
    }).into("products")
  }
}

module.exports = DatabaseSeeder
```

When we insert customers into the database, we use the `Hash.make` method. This generates a secure password hash. It's a good idea to hash passwords (never to store them in plain text) in the database.

After each insert, we get the identifier for the row that was inserted. We can access this identifier to insert related records, as we do with the seeded product.

Working with Routes and the Database

Until now, the profile route has displayed a profile page, whether or not the related profile is stored in the database. Let's make this dynamic! Listing 9-14 shows how we might change the route handler, to check the database or display an error page.

Listing 9-14. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async showProfile({ params, response, view }) {
  // ...redirect if needed
  const customer = await Database.select("*")
    .from("customers")
    .where("nickname", params.customer)
    .first()

  if (!customer) {
    return view.render("oops", {
      type: "PROFILE_MISSING",
    })
  }

  return view.render("customer/profile", {
    name: params.customer,
  })
}
```

Assuming there's no redirect for the profile named in the URL, we fetch the corresponding customer row from the database. The `first` method reduces the array we would have gotten to a single object (and it also adds a `LIMIT 1` to the underlying query). If none exists, we display an error template. This error page can say just about anything. Mine resembles Listing 9-15.

Listing 9-15. This is from `threadbear/resources/views/oops.edge`

```
@layout("layout")
@section("header")
  <h1>Oops</h1>
@endsection
@section("content")
  @if(type === "PROFILE_MISSING")
    <p>Looks like that profile doesn't exist. Try another...</p>
  @endif
@endsection
```

Showing Products

Now that the profiles are database driven, we can get a list of each profile's products. Let's fetch them, as shown in Listing 9-16.

Listing 9-16. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async showProfile({ params, response, view }) {
  // ...redirect or fetch customer

  if (!customer) {
    return view.render("oops", {
      type: "PROFILE_MISSING",
    })
  }

  const products = await Database.select("*")
    .from("products")
    .where("customer_id", customer.id)

  return view.render("customer/profile", {
    customer,
    products,
  })
}
```

This is the second time we're seeing the `where` query method. There are many variations of this, all of which relate to a specific kind of SQL `where` clause. You can use `whereNot`, `whereIn`, `whereNull`, and even `whereRaw`. Check the documentation (`http://dev.adonisjs.com/docs/4.0/query-builder#_where_clauses`) to get more details on how to use each of these.

Given the array of products, we can render these in the template. Listing 9-17 shows one way we can do this.

Listing 9-17. This is from `threadbear/resources/views/customer/profile.edge`

```
@layout("layout")
@section("header")
  <h1>
    {{ toTitleCase(customer.first_name) }}'s profile
  </h1>
@endsection
@section("content")
```

```
  @each(product in products)
    <p>
      {{ toTitleCase(product.name) }}
      <strong>${{ product.price / 100 }}</strong>
    </p>
  @else
    <p>
      {{ toTitleCase(customer.first_name) }} doesn't have any products
    </p>
  @endeach
@endsection
```

We can use the @each, @else, and @endelse tags to render a loop or provide information when the loop is empty. For each product, we can render a paragraph that includes the product name and price. If we visit Harminder's profile (http://127.0.0.1:3333/harminder), we'll see a message telling us that he has no products. If we visit my profile (http://127.0.0.1:3333/Christopher), we'll see one product and its price.

Note Here's a good opportunity to use that currency information we discovered in the previous chapter. Perhaps we could hook into a currency conversion service to show what each product is probably going to cost the customer, in their own currency.

Making Models

We could stop here, but we'd be missing a trick. We can encapsulate a lot of the raw database queries and relationships inside *models*. These are JavaScript class representations of individual database rows, which follow the Active Record pattern.

Let's make a few of these:

```
adonis make:model Redirect
adonis make:model Customer
adonis make:model Product
```

These create empty models, which resemble Listing 9-18.

Listing 9-18. This is from `threadbear/app/Models/Customer.js`

```
"use strict"

const Model = use("Model")

class Customer extends Model {

}

module.exports = Customer
```

Given these three new models, we can simplify all of our queries. Take a look at how we do this in Listing 9-19.

Listing 9-19. This is from `threadbear/app/Models/Customer.js`

```
async showProfile({ params, response, view }) {
  // const rows = await Database.select("from", "to")
  //   .from("redirects")

  const rows = await Redirect.all()

  const redirects = Array.from(rows).reduce(
    (accumulator, row) => {
      accumulator[row.from] = row.to
      return accumulator
    },
    {},
  )

  // ...

  // const customer = await Database.select("*")
  //   .from("customers")
  //   .where("nickname", params.customer)
  //   .first()

  const customer = await Customer.query()
    .where("nickname", params.customer)
    .first()
  // ...

  // const products = await Database.select("*")
  //   .from("products")
  //   .where("customer_id", customer.id)

  const products = await Product.query().where(
    "customer_id",
    customer.id,
  )

  // ...
}
```

We can take this a step further, but accessing the productions from the `Customer` model. We'd need to define a relation method on `Customer`, as in Listing 9-20.

Listing 9-20. This is from `threadbear/app/Models/Product.js`

```
const Model = use("Model")

class Customer extends Model {
  products() {
    return this.hasMany("App/Models/Product")
  }
}
```

This, in turn, would give us the ability to query products directly on a `Customer` instance, as in Listing 9-21.

Listing 9-21. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async showProfile({ params, response, view }) {
  // ...

  // const products = await Database.select("*")
  //   .from("products")
  //   .where("customer_id", customer.id)

  // const products = await Product.query()
  //    .where("customer_id", customer.id)

  const products = await customer.products()

  return view.render("customer/profile", {
    customer,
    products,
  })
}
```

The `hasMany` relation method connects the customer primary key with the product foreign key. It automatically adds the `where` clause so that `customer.products()` returns only products related to the customer.

Note There are many other kinds of relationships, including `belongsTo`, `hasOne`, and `belongsToMany`. Take some time to review the documentation (`http://dev.adonisjs.com/docs/4.0/relationships`) so you can become familiar with the differences and usage of these.

Registering Customers

Now that we can read and write to the database, let's expand Threadbear to allow customers to create new profiles. We already set up a registration page—when we were busy with routing—but we need to add a few fields. Listing 9-22 shows what we need to add.

Listing 9-22. This is from `threadbear/resources/views/customer/register.edge`

```
<form method="post">
  <div class="form-group">
    <label for="email">First name</label>
    <input type="text" class="form-control" id="first_name" name="first_name">
  </div>
  <div class="form-group">
    <label for="email">Last name</label>
    <input type="text" class="form-control" id="last_name" name="last_name">
  </div>
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email" class="form-control" id="email" name="email">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control" id="password" name="password">
  </div>
  <div class="form-group">
    <label for="confirm-password">Confirm password</label>
    <input type="password" class="form-control" id="confirm-password"
    name="confirm-password">
  </div>
  <div class="form-group">
    <label for="email">Nickname</label>
    <input type="text" class="form-control" id="nickname" name="nickname">
  </div>
  <button type="submit" class="btn btn-primary">Register</button>
</form>
```

Specifically, we need to add `first_name`, `last_name`, and `nickname` fields, as we defined these in the `customers_schema` database migration. Next, we need to store these properties in a new instance of the model, as shown in Listing 9-23.

Listing 9-23. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async doRegister({ request, response }) {
  // ...create new customer profile
  // this.showRequestParameters(context)

  const customer = await Customer.create(
    request.only([
      "first_name",
      "last_name",
      "email",
      "password",
      "nickname",
    ]),
  )

  return "done"
}
```

AdonisJs models have a neat `create` static method, into which we can pass an object of key/value pairs. In addition, the request class has an `only` method, which generates an object of key/value pairs, for every key specified.

Together, these reduce the code we need to write in order to insert records into the database. We could use an alternative syntax, as Listing 9-24 shows.

Listing 9-24.

```
const customer = new Customer()

customer.first_name = request.input("first_name")
customer.last_name = request.input("last_name")
customer.email = request.input("email")
customer.password = request.input("password")
customer.nickname = request.input("nickname")

await customer.save()
```

These do the same thing. The first is a lot quicker (and just as clear, in my opinion). The second form is closer to the code we'll use to update profiles. That's a subject for another time, though. For now, let's turn our attention to passwords. If you check the database, you'll see that the password is saved in plain text. We want to hash our passwords, but it would be great if we didn't even have to think about the process.

One way to change data before it is written to the database is to use customer setters. They allow us to modify values set on a model object so that the internal state is what we want it to be. Listing 9-25 shows an example of this.

Listing 9-25. This is from `threadbear/app/Models/Customer.js`

```
const Model = use("Model")

class Customer extends Model {
  products() {
    return this.hasMany("App/Models/Product")
  }
  setNickname(nickname) {
    return nickname.toLowerCase()
  }
}
```

As `customer.nickname = "..."` is executed, this setter is called. The nickname is transformed, and the new value is set on that property. Unfortunately, setters are synchronous only, which means we can't do asynchronous things to get the new value. This includes generating hashes. If we want to automatically hash passwords, as they are set on the model, we need to use hooks. Listing 9-26 shows what these look like.

Listing 9-26. This is from `threadbear/app/Models/Customer.js`

```
const Model = use("Model")
const Hash = use("Hash")

class Customer extends Model {
  products() {
    return this.hasMany("App/Models/Product")
  }

  setNickname(nickname) {
    return nickname.toLowerCase()
  }

  static boot() {
    super.boot()

    this.addHook(
      "beforeCreate",
      async customer => {
        customer.password = await Hash.make(
          customer.password,
        )
      },
    )
  }
}
```

The boot method is run whenever a new model object is created. We can use it to hydrate (fill the object with special data) and attach hooks. This hook should run before the model object is persisted to the database, and it allows us to give an asynchronous callback. Inside it, we can hash the password.

As the nickname is set, it will be converted to all lowercase, thanks to the customer setter. Before the model is saved to the database, the password will be hashed, thanks to the customer hook.

Note In the next chapter, we'll look at adding validation to this route handler. We need to make sure that the e-mail and nickname are unique and that the password fields match.

Authenticating Customers

Now that our registration form is working, let's make the login form work! The existing markup is good enough, so we just have to check the credentials, as in Listing 9-27.

Listing 9-27. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async doLogin({ request, response }) {
  // create new customer session
  // return "POST /login"

  const email = request.input("email")
  const password = request.input("password")

  const customer = await Customer.findByOrFail(
    "email",
    email,
  )

  const matches = await Hash.verify(
    password,
    customer.password,
  )
  if (matches) {
    return "valid"
  } else {
    return "invalid"
  }
}
```

Here we can see another useful model method: `findByOrFail`. It takes a key/value pair and tries to find a matching database row. If it can't find a matching row, it displays an error page. A similar method, called `findOrFail`, matches a value against the primary-key column.

Given a customer row, we can verify that a plain-text password (such as the one we get from the login form) matches the hashed version. Later, we'll learn how to create the new session and protect certain routes from unauthenticated access.

We could even hide this authentication logic inside the model, as shown in Listing 9-28.

Listing 9-28. This is from `threadbear/app/Models/Customer.js`

```
static async authenticate(email, password) {
  const customer = await Customer.findByOrFail(
    "email",
    email,
  )

  const matches = await Hash.verify(
    password,
    customer.password,
  )

  if (matches) {
    return customer
  }

  throw new Error("invalid credentials")
}
```

This is another example of why `async` and `await` are so much nicer than writing purely promise-based code. We can throw errors and declare `async` functions. We don't have to worry about instantiating promises or using `then` and `catch` in our controller. Listing 9-29 shows how much nicer that code looks.

Listing 9-29. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async doLogin({ request, response }) {
  // create new customer session
  // return "POST /login"

  const email = request.input("email")
  const password = request.input("password")

  try {
    const customer = await Customer.authenticate(
      email,
      password,
    )
  } catch (e) {
    return "invalid"
  }

  return "valid"
}
```

We still need to start a session and add other bits of validation, but this code is already looking great and working well. What's more, the controller doesn't even know that the passwords get hashed, or how. We could completely change the hashing algorithm or scheme, and we wouldn't have to touch this controller code.

Summary

This was an intense chapter! We learned about migrations, seeders, direct database queries, and even Lucid models. Database work is essential to any nontrivial application, and we're going to see a lot more of it before we're through.

Take some time to review the database and Lucid documentation, and to become more familiar with the Lucid database and model methods. I've skipped over many smaller details, but we'll definitely revisit these and other database concepts, so you're going to become quite the database expert.

In the next chapter, we're going to round out the registration and login forms with validation and sessions. We'll protect parts of our application, and work on the profile and commerce sections.

CHAPTER 10

Validation and Errors

In the previous chapter, we learned about databases. This got us most of the way to building our registration and login forms. One topic we didn't spend enough time on was how to respond in the absence or poor formatting of data.

In this chapter, we're going to start validating our forms. We'll learn how to make fields required, how to enforce the format of the data we do get, and how to display validation errors to the user. We'll also look at how to customize error pages for all the common types of errors users are bound to see.

Installing Validation

Validation is common, but not enough to be included in the slim application skeleton. To use it, we need to install it:

```
adonis install @adonisjs/validator
```

Once this is complete, we also need to register a provider (just as we did with views). This is shown in Listing 10-1.

Listing 10-1. This is from `threadbear/start/app.js`

```
const providers = [
  // ...
  "@adonisjs/validator/providers/ValidatorProvider",
]
```

Now we can modify our controller to check some of the data being posted from the form. Let's first look at this in the context of a single controller, and then move it to the base controller. Listing 10-2 shows what I mean.

© Christopher Pitt 2018
C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_10

Listing 10-2. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
const Controller = use(
  "App/Controllers/Http/Controller",
)
const Database = use("Database")
const { validate } = use("Validator")

// ...some time later

async doLogin({ request, response }) {
  const rules = {
    email: "required|email",
    password: "required",
  }

  const validation = await validate(
    request.all(),
    rules,
  )

  if (validation.fails()) {
    return response.json(validation.messages())
  }

  // ...normal doLogin code
}
```

This is the simplest form of validation. We define a list of rules, where the key is the name of the field that will be posted, and the value is a pipe-delimited list of validation rules, per field. There's no limit to the number of rules we can apply to each field.

Note Just as the schema builder is based on another underlying database library (called Knex, http://knexjs.org), so too validation is based on another underlying library. This one is called Indicative (http://indicative.adonisjs.com). If you want to learn more about the rules already provided, you can check out the documentation at http://indicative.adonisjs.com/#indicative-basics-rules.

`validate` is async because it could do any number of asynchronous things. We could check that the e-mail address exists in the database as part of validation. We could add an authenticate validation rule so that authentication happened as part of validation. These would need to be asynchronous, as they'd interact with the database.

We're going to try to make the first one into a new validation rule. We can call the rule `exists` and model it on an existing validation rule. The rule we can model it on is called the `unique` rule. We can use the unique rule to check that a value we want to insert is unique to the table we want to insert it into. We'll use that when we get to the registration validation. For now, let's look at how we could create our new validation rule, shown in Listing 10-3.

Listing 10-3. This is from `threadbear/start/hooks.js`

```
const Validator = use("Validator")
const Database = use("Database")

const existsFn = async (
  data,
  field,
  message,
  args,
  get,
) => {
  const value = get(data, field)

  if (!value) {
    return
  }
  const [table, column] = args

  const row = await Database.table(table)
    .where(column, value)
    .first()

  if (!row) {
    throw message
  }
}
```

This code is similar to an example in the documentation (see http://adonisjs.com/docs/4.0/validator#_extending_validator). We created `hooks.js` in a previous chapter. We can add this code inside that file, in the `providersBooted` hook, so that it's available throughout our application.

It begins by getting the posted field value. `field` is the name of the posted field (the key of the `rules` list). `get` is a global function, which pulls the named value out of another object (in this case, `data`). If the value doesn't exist, we exit early from the validator. If the field is required, we should add a `required` rule instead of validating that here.

If the value is defined, we inspect the `args` array. This is made up of the values following `exists`: `customers` and `email`. We'll see how those are defined in just a moment. The first `args` value is the name of the table we want to check, and the second is the name of the column in that table. We want to check whether the value exists in the column of a database table.

So, we perform that query. If no rows are found, we throw the error message. If a row is found, we do nothing (just as if the value didn't exist). We can now use `exists` in our rules, as shown in Listing 10-4.

Listing 10-4. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
const rules = {
  email: "required|email|exists:customers,email",
  password: "required",
}
```

Reusing Validation

I mentioned we were going to move the validation logic to the base controller. There's not much work involved, but I think the result is clearer. Listing 10-5 shows what the extraction looks like.

Listing 10-5. This is from `threadbear/app/Controllers/Http/Controller.js`

```
const { validateAll } = use("Validator")

// ...some time later

async validate({
  request,
  response,
  session,
  rules,
}) {
  const validation = await validateAll(
    request.all(),
    rules,
  )

  if (validation.fails()) {
    session
      .withErrors(validation.messages())
      .flashAll()

    return response.redirect("back")
  }
}
```

This function uses a lot of parts. We're using the `validateAll` method, as the `validate` method stops at the first error. `validateAll` continues so that the list of error messages includes all errors at once.

We still use the `fails` method, but if validation `fails`, we now persist some things to the session. We're going to explore these methods in more detail in the next chapter. For now, you just need to understand that we're saving the validation errors *and* the posted form data to the session.

We do this so that we can redirect back to the form, and display the errors and posted form data there again. How do we do this? Listing 10-6 shows how.

Listing 10-6. This is from `threadbear/resources/views/customer/login.edge`

```
<form method="post">
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email" class="form-control" id="email" name="email"
    value="{{ old(" email ", " ") }}">
    {{ hasErrorFor("email") ? getErrorFor("email") : "" }}
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control" id="password" name="password">
    {{ hasErrorFor("password") ? getErrorFor("password") : "" }}
  </div>
  <button type="submit" class="btn btn-primary">Log in</button>
</form>
```

The validation library adds a view helper called `getErrorFor`. This returns the error message for that field from those stored in the session. We also see an `old` method, which is meant to return the old posted form value, for that field. Before `old` will work, we need to install the session library:

```
adonis install @adonisjs/session
```

And as usual, we also need to register the session provider (as seen in Listing 10-7).

Listing 10-7. This is from `threadbear/start/app.js`

```
const providers = [
  // ...
  "@adonisjs/validator/providers/ValidatorProvider",
  "@adonisjs/session/providers/SessionProvider",
]
```

The session library also needs middleware, so that it can begin the session for each request. We can register that, as shown in Listing 10-8.

Listing 10-8. This is from `threadbear/start/kernel.js`

```
const globalMiddleware = [
  // ...
  "Adonis/Middleware/Session",
]
```

The first time we visit the login page in the browser, we should see nothing out of the ordinary. It should still look like it did before. To see the results of this new `validate` method, we need to modify the customer controller, as shown in Listing 10-9.

Listing 10-9. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async doLogin({ request, response, session }) {
  const rules = {
    email:
      "required|email|exists:customers,email",
    password: "required",
  }

  if (
    !await this.validate({
      request,
      response,
      session,
      rules,
    })
  ) {
    return
  }

  // ...normal doLogin code
}
```

Our validation logic is significantly cleaner. We don't have to worry about what to do if validation fails. We can design our forms around the idea that they will be redirected to if any validation errors occur.

Note We've skipped over quite a few session details. That's OK; we'll learn about sessions in the next chapter. For now, we're using them only as a temporary store between `doLogin` and the login form.

Let's take a moment to add validation to our registration page. Listing 10-10 shows what this might look like.

Listing 10-10. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async doRegister({
  request,
  response,
  session,
}) {
  const rules = {
    first_name: "required",
    last_name: "required",
    email:
      "required|email|unique:customers,email",
    password: "required",
    confirm_password: "required|same:password",
    nickname: "required",
  }

  if (
    !await this.validate({
      request,
      response,
      session,
      rules,
    })
  ) {
    return
  }
  // ...normal doRegister code
}
```

Most of these fields are just `required`. We're using the `unique` rule, I mentioned, which works similarly to the way `exists` works—that is, it checks the `customers` table and the `email` field. Whereas `exists` wants there to be a row, `unique` wants no rows to have the same e-mail value.

We're also checking that the `confirm_password` matches the `password` field, using the `same` rule. Of course, this means we also have to update our registration form, as shown in Listing 10-11.

Listing 10-11. This is from `threadbear/resources/views/customer/register.edge`

```
<form method="post">
  <div class="form-group">
    <label for="email">First name</label>
    <input type="text" class="form-control" id="first_name" name="first_
    name" value="{{ old(" first_name ", " ") }}">
    {{ hasErrorFor("first_name") ? getErrorFor("first_name") : "" }}
  </div>
  <div class="form-group">
    <label for="email">Last name</label>
    <input type="text" class="form-control" id="last_name" name="last_name"
    value="{{ old(" last_name ", " ") }}">
    {{ hasErrorFor("last_name") ? getErrorFor("last_name") : "" }}
  </div>
  <div class="form-group">
    <label for="email">Email address</label>
    <input type="email" class="form-control" id="email" name="email"
    value="{{ old(" email ", " ") }}">
    {{ hasErrorFor("email") ? getErrorFor("email") : "" }}
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control" id="password" name="password">
    {{ hasErrorFor("password") ? getErrorFor("password") : "" }}
  </div>
  <div class="form-group">
    <label for="confirm_password">Confirm password</label>
    <input type="password" class="form-control" id="confirm_password"
    name="confirm_password">
    {{ hasErrorFor("confirm_password") ? getErrorFor("confirm_password") :
    "" }}
  </div>
  <div class="form-group">
    <label for="email">Nickname</label>
    <input type="text" class="form-control" id="nickname" name="nickname"
    value="{{ old(" nickname ", " ") }}">
    {{ hasErrorFor("nickname") ? getErrorFor("nickname") : "" }}
  </div>
  <button type="submit" class="btn btn-primary">Register</button>
</form>
```

We're using the same `old` and `getErrorFor` methods as before. There's nothing new to learn from this code. However, the error messages include the raw field names. It would be nice if we could customize the messages that we display to customers. Listings 10-12 and 10-13 show how we can achieve this.

Listing 10-12. This is from `threadbear/app/Controllers/Http/Controller.js`

```
async validate({
  request,
  response,
  session,
  rules,
  messages,
}) {
  const validation = await validateAll(
    request.all(),
    rules,
    messages,
  )
  if (validation.fails()) {
    session
      .withErrors(validation.messages())
      .flashAll()

    return response.redirect("back")
  }
}
```

Listing 10-13. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
const rules = {
  first_name: "required",
  last_name: "required",
  email:
    "required|email|unique:customers,email",
  password: "required",
  confirm_password: "required|same:password",
  nickname: "required",
}

const messages = {
  "first_name.required":
    "you must provide a first name",
  "last_name.required":
    "you must provide a last name",
  "email.required":
    "you must provide an email address",
  "email.unique":
    "your email address must be unique",
  "password.required":
    "you must provide a password",
  "confirm_password.required":
    "you must confirm the password",
  "confirm_password.same":
    "passwords must match",
  "nickname.required":
    "you must provide a nickname",
}

if (
  !await this.validate({
    request,
    response,
    session,
    rules,
    messages,
  })
) {
  return
}
```

Indicative lets us provide a `messages` object, mapping fields (and their rules) to specific error messages. We can specify messages for each rule, for each field. Later, we'll learn how to allow these messages to be translated into other languages.

Displaying Custom Error Pages

Now that we're handling validation errors, we should also look at customizing the error pages customers are likely to see. Validation is easy, because we redirect to an existing form. What about missing pages or server errors?

Our routes are designed so that if none of the others are matched, we'll look for a profile matching the route name. If someone goes to `/does-not-exist`, we'll look for a profile with the nickname `does-not-exist`. But, imagine we commented those routes out, so that `/does-not-exist` leads to a 404 error. We could respond to that error by adding the hook shown in Listing 10-14.

Listing 10-14. This is from `threadbear/start/hooks.js`

```
const Exception = use("Exception")

Exception.handle(
  "HttpException",
  async (error, { response, session }) => {
    // respond to the 404 error
    return
  },
)
```

If we comment out those profile routes , we can respond with a custom error message (or view) inside the handler. We don't have to use a hook for this, either. We could create a whole new handler class:

```
adonis make:ehandler
```

This creates a new exception handler class, which we can move this `HttpException` logic into, as demonstrated in Listing 10-15.

Listing 10-15. This is from `threadbear/app/Exceptions/Handler.js`

```
"use strict"

class ExceptionHandler {
  async handle(error, { request, response }) {
    if (error.name === "HttpException") {
      response
        .status(error.status)
        .send("not found")
      return
    }

    // response
    //     .status(error.status)
    //     .send(error.message)
  }

  async report(error, { request }) {
    // write the error to a log
  }
}

module.exports = ExceptionHandler
```

Let's refactor our validation logic to be handled via this exception handler. Unfortunately, we can't use the `ExceptionHandler` class for `ValidationException` errors because of how the order of precedence works. We have to go back to hooks—Listings 10-16, 10-17, and 10-18 show how we might achieve this.

Listing 10-16. This is from `threadbear/app/Controllers/Http/Controller.js`

```
const {
  ValidationException,
} = require("@adonisjs/validator/src/Exceptions")

// ...some time later

async validate(request, rules, messages) {
  const validation = await validateAll(
    request.all(),
    rules,
    messages,
  )

  if (validation.fails()) {
    // session
    //   .withErrors(validation.messages())
    //   .flashAll()
    //
    // return response.redirect("back")

    throw ValidationException.validationFailed(
      validation.messages(),
    )
  }
}
```

Listing 10-17. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async doRegister({
  request,
  response,
  session,
}) {
  const rules = {
    first_name: "required",
    last_name: "required",
    email:
      "required|email|unique:customers,email",
    password: "required",
    confirm_password: "required|same:password",
    nickname: "required",
  }

  const messages = {
    "first_name.required":
      "you must provide a first name",
    "last_name.required":
      "you must provide a last name",
    "email.required":
      "you must provide an email address",
    "email.unique":
      "your email address must be unique",
    "password.required":
      "you must provide a password",
    "confirm_password.required":
      "you must confirm the password",
    "confirm_password.same":
      "passwords must match",
    "nickname.required":
      "you must provide a nickname",
  }

  await this.validate(request, rules, messages)

  // ...normal doRegister code
}
```

Listing 10-18. This is from `threadbear/start/hooks.js`

```
Exception.handle(
  "ValidationException",
  async (error, { response, session }) => {
    session.withErrors(error.messages).flashAll()

    await session.commit()

    return response.redirect("back")
  },
)
```

Instead of dealing with the session and response objects inside the `Controller` class, we can move that behavior to the `ValidationException` exception handler hook. This leads to cleaner `this.validate` code. We can forget about the `if` statements and the early `return`. We don't even need to change how our forms work, because they're getting exactly the same data.

Creating Custom Exceptions

We're reusing the `HttpException` and `ValidationException` classes. What if we want to create our own exceptions but still want AdonisJs to handle them in the same way? We can use an Ace command to generate a new exception:

```
adonis make:exception ProfileMissing
```

This gives us a new, empty exception class (similar to Listing 10-19).

Listing 10-19. This is from `threadbear/app/Exceptions/ProfileMissingException.js`

```
"use strict"

const {
  LogicalException,
} = require("@adonisjs/generic-exceptions")

class ProfileMissingException extends LogicalException {
  async handle() {
    // ...handle this exception
  }
}

module.exports = ProfileMissingException
```

This is slightly different from what is generated:

- We've changed the handle method to be async. We might do async things in it!
- We've uncommented the handle method. I'll explain why in a moment.

We could use hooks or our `ExceptionHandler` class to work out what to do when we see these kinds of exceptions. Or we could declare a `handle` method (like the one we uncommented) so that AdonisJs uses that instead. It's an automatic way of keeping the handle logic together with the custom exception type.

Let's move the "oops!" error code to this handler, shown in Listing 10-20.

Listing 10-20. This is from `threadbear/app/Exceptions/ProfileMissingException.js`

```
async handle(error, { response, view }) {
  const content = view.render("oops", {
    type: "PROFILE_MISSING",
  })

  response.status(404).send(content)

  return
}
```

This means we can make our controller code simpler (shown in Listing 10-21).

Listing 10-21. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
const ProfileMissingException = use(
  "App/Exceptions/ProfileMissingException",
)
```

```
// ...some time later

const customer = await Customer.query()
  .where("nickname", params.customer)
  .first()

if (!customer) {
  // return view.render("oops", {
  //   type: "PROFILE_MISSING",
  // })

  throw new ProfileMissingException()
}
```

This is a powerful concept! We can respond to errors without mixing that error-handling code into our controllers and routes. We can reuse error-handling code so we don't have to keep repeating the same `if (!customer)` and `return view.render("oops"...)` code everywhere.

Summary

In this chapter, we learned all about how to validate our forms. We added validation code to the registration and login forms, and rendered those errors (with custom error messages) to the browser. We also learned how to handle built-in and custom exceptions. We rendered custom content for 404 errors and refactored our validation errors to be handled in this way. We also created a custom error type and refactored the profile handler to be cleaner. We refactored it not to care about how the error is handled—just to throw the error and let AdonisJs take care of the rest.

Note We could take this a step further by creating custom error page views. I didn't do that for the sake of time, but you're welcome to try this as an exercise.

In the next chapter, we're going to learn how to create user sessions tied to our authentication code. We'll learn more about the session code we saw in this chapter. We'll also learn how to work with cookies to create a "remember me" function for our login page.

CHAPTER 11

Sessions

In the previous chapter, we learned how to validate form data and present custom error pages to customers. We're well on our way to having a professional, polished commerce web site. But there's one area we're still a bit behind in: we need to make a way for customers to log in and update their profiles. We created the registration and login forms in previous chapters. We also saw how to persist those details to the database and validate login credentials.

In this chapter, we're going to learn how to create customer sessions. We'll use these to ensure that customers see only the pages and products they're supposed to. We'll start to capture orders and display customer details. We'll also learn more about Edge and Lucid.

Making the Dashboard

Now that we're validating login credentials, all we have to do (to enable customer sessions) is to store the customer details in a session cookie. Let's make a dashboard page—one that can be seen by only an authenticated customer. We'll call it the *dashboard*. Listing 11-1 shows what it could look like.

Listing 11-1. This is from `threadbear/resources/views/customer/dashboard.edge`

```
@layout("layout")
@section("header")
  <h1>
    Hello, {{ customer.displayName }}
  </h1>
@endsection
@section("content")
  <h2>Your products</h2>
  @each(product in products)
    <p>
      {{ product.displayName }}
      <strong>${{ product.price / 100 }}</strong>
    </p>
    @else
    <p>
      You haven't created any products
    </p>
  @endeach
  <h2>Pending orders</h2>
  @if(pendingOrders.length)
    <ol>
      @!each(order in pendingOrders, include = "customer/partial/order")
    </ol>
  @else
    <p>
      There are no pending orders.
    </p>
  @endif
  <h2>Complete orders</h2>
  @if(completeOrders.length)
    <ol>
      @!each(order in completeOrder, include = "customer/partial/order")
    </ol>
  @else
    <p>
      There are no complete orders.
    </p>
  @endif
@endsection
```

This looks similar to our profile page, but it uses more advanced Lucid and Edge functionality. For starters, we're using the `displayName` property on the `Customer` model. We could add this as a Lucid getter. Before we do that, though, let's create a base model.

That way, we can share common model extensions without repeating a lot of code. Listing 11-2 shows what I mean.

Listing 11-2. This is from `threadbear/app/Models/BaseModel.js`

```
"use strict"

const Model = use("Model")
const startCase = use("lodash/startCase")

class BaseModel extends Model {
  titleCase(...params) {
    return startCase(...params)
  }
}

module.exports = BaseModel
```

The first bit of shared functionality is a `titleCase` method. It does the same thing as our `toTitleCase` global view function, but this time we're using a `Lodash` function. This is what Edge uses (in its `capitalize` function).

Note We should probably add Lodash (`https://lodash.com`) as a dependency if we use it in our codebase. AdonisJs already includes it, and given the structure of NPM 3, it will be available to us regardless.

Now we can add a getter to the `Customer` model, as shown in Listing 11-3.

Listing 11-3. This is from `threadbear/app/Models/Customer.js`

```
// ...rest of the imports

const BaseModel = use("App/Models/BaseModel")

class Customer extends BaseModel {
  static get computed() {
    return ["displayName"]
  }
  getDisplayName({ first_name, last_name }) {
    return this.titleCase(
      first_name + " " + last_name,
    )
  }

  // ...rest of the methods
}
```

Most of the time, we're going to be accessing model data after also calling the `toJSON` method. In order to add custom (computed) properties to this, we need to export them with the `computed` static getter. This is ES6 syntax, used to produce a dynamic static class property. Lucid uses it in the process of generating the results for `toJSON`.

We can define getters as `getDisplayName`, where `DisplayName` is the title case property name. The method is called with an object of database keys and values. `first_name` and `last_name` are columns on the `customers` table, so, we can dereference them to compute a full name.

Next, we reference a bunch of customer-related data: `products`, `pendingOrders`, and `completeOrders`. We've learned a bit about model relationships. These are defined as shown in Listing 11-4.

Listing 11-4. This is from `threadbear/app/Models/Customer.js`

```
class Customer extends BaseModel {
  products() {
    return this.hasMany("App/Models/Product")
  }

  pendingOrders() {
    return this.hasMany(
      "App/Models/Order",
      "id",
      "seller_id",
    ).where("status", "pending")
  }
  completeOrders() {
    return this.hasMany(
      "App/Models/Order",
      "id",
      "seller_id",
    ).where("status", "complete")
  }

  // ...rest of the methods
}
```

pendingOrders and completeOrders are just slightly more complicated hasMany relationships. We can add where clauses so that the method names more accurately reflect the kinds of filtered data we want them to return.

Finally, we have a couple of @!each tags. The @! syntax is a way to tell Edge to ignore (or rather, not to expect) closing @endeach tags. The second argument is the name of a partial view file to repeat for each item iteration. This partial resembles Listing 11-5.

Listing 11-5. This is from threadbear/resources/views/customer/partial/order.edge

```
<li>
  {{ order.buyer ? order.buyer.displayName : order.id }}
  @if(order.items)
    <ul>
      @each(item in order.items)
        <li>
          {{ item.quantity }} x {{ item.product.displayName }} at
          <strong>
            ${{ item.price / 100 }}
          </strong>
        </li>
      @endeach
    </ul>
  @endif
</li>
```

This partial is meant to render a single order. If the order has a buyer property, I want to show the buyer's name. If that's missing, the template should just display the order ID.

Then, if there are items, the template should render a list item for each order item. A few more relationships and computed properties are in use here. Listings 11-6 and 11-7 show how these can be added.

Listing 11-6. This is from `threadbear/app/Models/Order.js`

```
"use strict"

const BaseModel = use("App/Models/BaseModel")

class Order extends BaseModel {
  items() {
    return this.hasMany("App/Models/OrderItem")
  }

  buyer() {
    return this.belongsTo(
      "App/Models/Customer",
      "buyer_id",
    )
  }

  seller() {
    return this.belongsTo(
      "App/Models/Customer",
      "seller_id",
    )
  }
}

module.exports = Order
```

Listing 11-7. This is from `threadbear/app/Models/OrderItem.js`

```
"use strict"

const BaseModel = use("App/Models/BaseModel")
```

```
class OrderItem extends BaseModel {
  product() {
    return this.belongsTo("App/Models/Product")
  }
}

module.exports = OrderItem
```

I skipped over the creation of these, but let's look at how these are persisted to the database. We can create a couple of migrations, resembling Listings 11-8 and 11-9.

Listing 11-8. This is from `threadbear/database/migrations/*_orders_schema.js`

```
"use strict"

const Schema = use("Schema")

class OrdersSchema extends Schema {
  up() {
    this.create("orders", table => {
      table.increments()
      table.integer("buyer_id")
      table.integer("seller_id")
      table.string("status")
      table.timestamps()
    })
  }

  down() {
    this.drop("orders")
  }
}

module.exports = OrdersSchema
```

Listing 11-9. This is from `threadbear/database/migrations/*_order_items_schema.js`

```
"use strict"

const Schema = use("Schema")

class OrderItemsSchema extends Schema {
  up() {
    this.create("order_items", table => {
      table.increments()
      table.integer("order_id")
      table.integer("product_id")
      table.integer("quantity")
      table.integer("price")
      table.text("comments")
      table.timestamps()
    })
  }

  down() {
    this.drop("order_items")
  }
}

module.exports = OrderItemsSchema
```

Each order has a status—for now, these are `pending` and `complete`. Orders are linked to two customers: the buyer and the seller. We saw how those relationships are linked in the preceding code. We also need to store the current price and the desired quantity. Finally, we probably want to add a `comments` field, in case the buyer wants to give special instructions for the order.

Creating the Customer Session

When we left the `doLogin` action, we were validating login credentials. We could tell whether the customer should be logged in. We also added the session dependency (so that we could flash validation errors and display them after redirecting).

Let's use these things to create a user session, shown in Listing 11-10.

Listing 11-10. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async doLogin({ request, response, session }) {
  const rules = {
    email:
      "required|email|exists:customers,email",
    password: "required",
  }

  await this.validate(request, rules)

  const email = request.input("email")
  const password = request.input("password")

  const customer = await Customer.authenticate(
    email,
    password,
  )

  session.put("customer", customer.id)
  await session.commit()

  return response.route("dashboard")
}
```

This is much the same as before. The difference is that a successful validation state means the customer's ID needs to be saved. We put it in the session, wait for the session to be saved, and then redirect to the dashboard route.

Behind the scenes, I've updated the `authenticate` method to throw a `ValidationException`. Listing 11-11 demonstrates this.

Listing 11-11. This is from `threadbear/app/Models/Customer.js`

```
static async authenticate(email, password) {
  const customer = await Customer.findByOrFail(
    "email",
    email,
  )
  const matches = await Hash.verify(
    password,
    customer.password,
  )

  if (matches) {
    return customer
  }

  throw ValidationException.validationFailed({
    email: "invalid credentials",
  })
}
```

We could use a different kind of error for this. We were using `Error` before, but I think a `ValidationException` is more useful for us because it allows the failed login to be handled as if an ordinary validation error happened.

Note Don't forget to import the `ValidationException` class, using `const { ValidationException } = use("@adonisjs/validator/src/Exceptions")`.

Additionally, we've referenced a `dashboard` route. Listing 11-12 shows where to add this.

Listing 11-12. This is from `threadbear/start/routes.js`

```
Route.get(
  "/dashboard",
  "CustomerController.dashboard",
).as("dashboard")
```

We should now be able to log in. The problem is that there's no `dashboard` action yet. Let's create it, as shown in Listing 11-13.

Listing 11-13. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async dashboard({
  request,
  response,
  session,
  view,
}) {
  const customerId = session.get("customer")

  if (!customerId) {
    return response.route("login")
  }

  const customer = await Customer.find(
    customerId,
  )

  const products = await customer
    .products()
    .fetch()

  const pendingOrders = await customer
    .pendingOrders()
    .with("buyer")
    .with("items.product")
    .fetch()

  const completeOrders = await customer
    .completeOrders()
    .with("buyer")
    .with("items.product")
    .fetch()

  return view.render("customer/dashboard", {
    customer: customer.toJSON(),
    products: products.toJSON(),
    pendingOrders: pendingOrders.toJSON(),
    completeOrders: completeOrders.toJSON(),
  })
}
```

We begin by checking whether the customer session property exists. If not, we redirect back to the login page. We're going to learn about a more robust way to do this in the following chapter. This approach works, for now.

If the customer ID is present, we fetch the customer record from the database. Using relationships, we also fetch the products, pending orders, and complete orders. We're lazy-loading multiple related records (including the nested items.product relationship).

It's easier for us to access the computed properties if we convert each fetched record to JSON.

Logging Customers Out

At this point, you should be able to register and log in. If you're lacking good example data, you're welcome to use my improved seeder class (shown in Listing 11-14).

Listing 11-14. This is from threadbear/database/seeds/DatabaseSeeder.js

```
"use strict"

const Database = use("Database")
const Factory = use("Factory")
const Hash = use("Hash")
const moment = use("moment")

class DatabaseSeeder {
  async run() {
    const timestamps = {
      created_at: moment().format(
        "YYYY-MM-DD HH:mm:ss",
      ),
      updated_at: moment().format(
        "YYYY-MM-DD HH:mm:ss",
      ),
    }

    await Database.insert([
      {
        from: "assertchris",
        to: "christopher",
        ...timestamps,
      },
      {
        from: "thetutlage",
        to: "harminder",
        ...timestamps,
      },
    ]).into("redirects")

    await Database.insert([
      {
        first_name: "Harminder",
        last_name: "Virk",
        email: "virk.officials@gmail.com",
        password: await Hash.make("harminder123"),
        nickname: "harminder",
        ...timestamps,
      },
      {
        first_name: "Christopher",
        last_name: "Pitt",
        email: "cgpitt@gmail.com",
        password: await Hash.make(
          "christopher123",
        ),
        nickname: "christopher",
        ...timestamps,
      },
    ]).into("customers")

    const customerIds = await this.ids(
      "customers",
    )

    await Database.insert([
      {
        name: "Soft Teddy",
        price: 499,
        customer_id: customerIds[1],
        ...timestamps,
      },
    ]).into("products")

    const productIds = await this.ids("products")

    await Database.insert([
      {
        buyer_id: customerIds[0],
        seller_id: customerIds[1],
        status: "pending",
        ...timestamps,
      },
    ]).into("orders")

    const orderIds = await this.ids("orders")

    await Database.insert([
      {
        order_id: orderIds[0],
        product_id: productIds[0],
        quantity: 2,
        price: 499,
        ...timestamps,
      },
    ]).into("order_items")
  }

  async ids(table) {
    return await Database.select("id")
      .from(table)
      .orderBy("id", "asc")
      .map(next => next.id)
  }
}

module.exports = DatabaseSeeder
```

I've started combining multiple objects (of the same type) into batch insert statements. Normally, we'd be able to return only a single inserted ID, so I've created an `ids` method. This fetches all IDs from the desired table in ascending order.

Of course, it would be good to allow the customer to close their browser session. Let's modify the customer controller, as shown in Listing 11-15.

Listing 11-15. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async logout({ session, response }) {
  session.forget("customer")
  await session.commit()

  return response.route("login")
}
```

All we need to do here is forget the session value. We can redirect back to the login or even to the home page. I had to change the logout route to `Route.get("/logout", "CustomerController.logout")`, since it was previously `PUT`. It will be again, after we get to Chapter 12.

Note As we've been using saving sessions, the data has been stored in browser cookies. We saw many cookie-related methods when we learned about the `Request` and `Response` classes. We could work with them directly, but I think the session abstraction makes things a lot simpler. We can put, get, and forget session values without caring how the cookies are stored or associated with the browser.

If you'd like a cookie refresher, refer to Chapter 4 and Chapter 5, where we look at the cookie methods for requests and responses.

Summary

In this chapter, we learned much more about Lucid and Edge. We created new relationships and finished off the login and logout actions. We saw a few concepts repeated and changed slightly. This wasn't a particularly complex chapter, but it was work that needed to be done.

In the next chapter, we’re going to learn about encryption, middleware, and preventing cross-site request forgery. In other words, we’re going to review the built-in security libraries and mechanisms of AdonisJs.

CHAPTER 12

Security

In the previous chapter, we completed the login and logout pages of our application. We learned a lot more about the Lucid and Edge. We also set up orders, order items, and improved products a bit.

We also protected the dashboard by checking for a customer session ID before allowing a customer to view the page. It was a subtle, albeit important, step in securing our application.

That's what this chapter is all about. We're going to improve that check by making the authorization layer robust and ubiquitous. We'll learn about more of the security aspects of AdonisJs and how to use them to great effect.

Recap

Let's look at how we secured the dashboard and think of ways we could improve it; see Listing 12-1.

Listing 12-1. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async dashboard({
  request,
  response,
  session,
  view,
}) {
  const customerId = session.get("customer")

  if (!customerId) {
    return response.route("login")
  }
```

© Christopher Pitt 2018
C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_12

```
  const customer = await Customer.find(
    customerId,
  )

  // ...rest of the dashboard code
}
```

We needed the `customerId` to be able to display the correct orders on the dashboard. We doubled up on this dependency by using the absence of the ID to tell us when the customer wasn't authenticated.

The thing is, everywhere we're likely to care about the customer being authenticated, we're also likely to need to know something about the customer. In other words, if the page is protected, it's probably going to show or use information about the customer to function correctly.

Therefore, whatever we replace this with should store a reference to the customer. It should at least store the customer's ID, although it probably wouldn't hurt to store the user model as well.

If we kept up this approach (fetching `customerId` from the session and redirecting if it's falsy), we'd probably repeat this exact code every time. We should try to reduce the repetition to one or two lines of code. Perhaps we could use the base controller or middleware to share code.

If we opt for middleware, we can apply the authorization check in the routes files. We can even try to group multiple routes under the same middleware check.

Creating More Middleware

Remember when we created the country and currency middleware? We're going to follow the same process:

```
adonis make:middleware Auth
```

This makes another empty middleware class. Let's modify it to resemble Listing 12-2.

Listing 12-2. This is from `threadbear/app/Middleware/Auth.js`

```
"use strict"

const Customer = use("App/Models/Customer")
const key = "customer"
const redirectTo = "login"

class Auth {
  async handle(
    { request, session, response },
    next,
  ) {
    const id = session.get(key)

    if (!id) {
      return response.route(redirectTo)
    }

    const customer = await Customer.find(id)

    if (!customer) {
      return response.route(redirectTo)
    }

    request.customer = customer

    await next()
  }
}

module.exports = Auth
```

As we learned before, middleware has access to all the context that route callbacks and controller actions do. We can use the `session` and `response` objects to check and store customer data. Because we'll often need the customer record (on protected pages), we're fetching that too. Actions can access the `customer` property on the controller, as shown in Listing 12-3.

Listing 12-3. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async dashboard({
  request,
  response,
  session,
  view,
}) {
  // const customerId = session.get("customer")

  // if (!customerId) {
  //     return response.route("login")
  // }

  // const customer = await Customer.find(customerId)


  const customer = request.customer


  const products = await customer
    .products()
    .fetch()

  // ...rest of the dashboard code
}
```

We need to do a couple of things before this code will work. When we made the country (and currency) selector middleware, we registered it as global middleware. This is ideal for middleware we want to run on every request. It won't work for the auth middleware, because we want that to run on only certain routes.

We still need to register it, but instead of global, we'll register it as named middleware. This means we can apply it by name to a specific set of routes. We can register it as shown in Listing 12-4.

Listing 12-4. This is from `threadbear/start/kernel.js`

```
const namedMiddleware = {
  auth: "App/Middleware/Auth",
}
```

Now we can apply the `Auth` middleware class with the `"auth"` string. To do this, we group protected routes, as in Listing 12-5.

Listing 12-5. This is from `threadbear/start/routes.js`

```
Route.group(() => {
  Route.get(
    "/dashboard",
    "CustomerController.dashboard",
  ).as("dashboard")

  Route.put(
    "/:customer",
    "CustomerController.updateProfile",
  ).as("updateProfile")

  Route.delete(
    "/:customer",
    "CustomerController.deleteProfile",
  ).as("deleteProfile")
}).middleware(["auth"])
```

Route groups work by allowing a number of nested routes to be defined, which then inherit rules applied to the group. This means we can add prefixes, apply middleware, and specify formats for multiple routes within the group.

These three routes look like they should be protected. With this code applied, the dashboard is secure and working again.

Note An official AdonisJs Auth library can be installed with `adonis install @adonisjs/auth`. It does much of what we've done here and then some. I wanted us to go through this process so we could understand what that library (and similar ones) do in order to protect routes. Extensive documentation (`http://adonisjs.com/docs/4.0/authentication`) indicates how to install and use the official auth library.

The framework author is always looking for ways to help the community learn more about AdonisJs. To that end, he created `adonisjs.com/screencasts`. There you'll find high-quality screencasts showing how to use AdonisJs effectively.

Securing Forms

All our forms, including the login form, allow requests from any kind of browser (or HTTP client). This can lead to a security vulnerability called *cross-site request forgery* (CSRF). We could require that form submissions include some sort of proof that they come from someone using the site (instead of posting directly to the site, without the user's knowledge).

Note A Wikipedia article talks more about this particular vulnerability: `https://en.wikipedia.org/wiki/Cross-site_request_forgery`.

AdonisJs has a mechanism for this, so we can protect our forms against this kind of attack. We need to install the Shield library:

```
adonis install @adonisjs/shield
```

This gives us access to a new provider, which we can install by following Listing 12-6.

Listing 12-6. This is from `threadbear/start/app.js`

```
const providers = [
  "@adonisjs/shield/providers/ShieldProvider",
  // ...rest of the providers
]
```

Shield generates and checks special tokens. For this to work, it needs access to the session and must be applied as global middleware. Let's add it to the global middleware array, as shown in Listing 12-7.

Listing 12-7. This is from `threadbear/start/kernel.js`

```
const globalMiddleware = [
  "Adonis/Middleware/Session",
  "Adonis/Middleware/Shield",
  // ...rest of the middleware
]
```

Because it uses the session library, it needs to be added *after* that middleware. The session needs to be started before it can be used, after all.

Shield also has a configuration file (which was automatically added to our project). A lot of things are configured there, but the important bits can be seen in Listing 12-8.

Listing 12-8. This is from `threadbear/config/shield.js`

```
csrf: {
  enable: true,
  methods: ["POST", "PUT", "DELETE"],
  filterUris: [],
  cookieOptions: {
    httpOnly: false,
    sameSite: true,
    path: "/",
    maxAge: 7200,
  },
},
```

This config specifies that CSRF protection should be applied to `POST`, `PUT`, and `DELETE` request methods. These are typically used to perform destructive changes (for example, deleting database records or changing their content), so they're good candidates for protection.

When we submit a form, we should start to see a new kind of error. This looks like the "not found" `HttpException` error we're already handling, but we can customize it to show a specific page or error message. Listing 12-9 shows how.

Listing 12-9. This is from `threadbear/app/Exceptions/Handler.js`

```
async handle(error, { request, response }) {
  if (error.code === "EBADCSRFTOKEN") {
    response.forbidden(
      "Cannot process your request.",
    )
    return
  }

  // ...previous HttpException code
}
```

So, how do we reenable form posting? We have to add a new hidden field to each of our forms, as shown in Listings 12-10 to 12-13.

Listing 12-10. This is from `threadbear/resources/views/customer/register.edge`

```
@section("header")
  <h1>Register</h1>
@endsection
@section("content")
  <form method="post">
    {{ csrfField() }}
    // ...rest of the form
  </form>
@endsection
```

Listing 12-11. This is from `threadbear/resources/views/customer/login.edge`

```
@section("header")
  <h1>Login</h1>
@endsection
@section("content")
  <form method="post">
    {{ csrfField() }}
    // ...rest of the form
  </form>
@endsection
```

Listing 12-12. This is from `threadbear/resources/views/customer/forgot-password.edge`

```
@section("header")
  <h1>Forgot password</h1>
@endsection
@section("content")
  <form method="post">
    {{ csrfField() }}
    // ...rest of the form
  </form>
@endsection
```

Listing 12-13. This is from `threadbear/resources/views/customer/reset-password.edge`

```
@section("header")
  <h1>Reset password</h1>
@endsection
@section("content")
  <form method="post" action="?_method=PATCH">
    {{ csrfField() }}
    // ...rest of the form
    <button type="submit" class="btn btn-primary">Reset</button>
  </form>
@endsection
```

Each of these `csrfField` invocations generates a hidden field, which resembles `<input name="_csrf" value="gmoclIzW-2efhRg6rw-08okQnCO2pItp2Y7s" type="hidden">`. The value is the special token, and it is randomly generated and stored in the session each time the form is viewed.

This means the browser has to load the form before it can submit to the form's action. This makes it much harder for a malicious third-party to impersonate the customer and piggyback on their session.

Note We haven't worked on the password-reset features yet. We still need to work out how to send e-mail. We'll get there soon.

Encrypting Values

We've been storing hashed customer passwords in the database. This protects the plain-text versions the customers use. Should the database be compromised, hackers won't gain knowledge of the plain-text passwords. We could improve this by using a per customer salt, as explained in a great Stack Overflow answer (see `https://security.stackexchange.com/a/36838`).

One-way hashes (like the type we've been using) are great for storing verifiable data. Using the same salts and algorithms, we can check whether a customer-supplied value hashes to the same value as the stored hash.

Sometimes, for values other than passwords, we might want to make obfuscated values into plain-text values. This could be for any number of reasons. Perhaps the customer needs to store sensitive information through our application. We don't want unauthorized employees to be able to read the sensitive information, so we can encrypt it.

The code for this is relatively straightforward:

```
const Encryption = use("Encryption")
const encrypted = Encryption.encrypt("plain-text value")
```

The `encrypt` method uses the `appKey` config variable, so we'd have to make sure unauthorized employees can't see that either. Fortunately, that's defined as follows:

```
appKey: Env.get("APP_KEY"),
```

Decrypting the data is done similarly:

```
const Encryption = use("Encryption")
const decrypted = Encryption.decrypt("encrypted value")
```

Note Do not encrypt passwords. You do not need the plain-text values of passwords. Use a one-way hash, and think about adding a per user salt to the process.

Summary

In this chapter, we learned about adding authorization as part of the middleware process. We can now define groups of protected routes, with almost no code repetition. We also have access to the customer model, so future customer-specific routes should be much easier to handle.

We also learned about protecting our forms from CSRF attacks, and added a way to return custom error messages when those checks fail. Finally, we learned about how to encrypt values.

In the next chapter, we're going to start building the front end. We'll set up a build chain and start using React and Sass to make our app look and work better!

CHAPTER 13

Front-End Tools

In the previous chapter, we looked at a few methods to improve the security of our application. We added anti-CSRF tokens to our forms, created an authentication middleware layer, and learned how to encrypt and decrypt values when needed.

Now we're going to change gears and learn about front-end build tools. This topic is fraught with change and uncertainty. Unless you're constantly in the front-end space, it's difficult to know which tools to use and how to correctly configure them. Don't worry—we'll figure things out together.

We'll start by setting up an easy-to-use build tool. We'll follow this by installing Bootstrap and testing out the Sass-to-CSS transformation process. Finally, we'll install React and make sure the modern JavaScript transforms into syntax most browsers can handle.

Installing Mix

Mix (`https://laravel.com/docs/5.5/mix`) is a wonderful outworking of the Laravel PHP framework community (`https://laravel.com`). PHP developers aren't usually comfortable writing front-end code, so Mix has been designed to be easy to use and hard to break.

Mix is the most stable and friendly front-end asset management library I have ever used, and we're going to use it to create the build chain for our application. Even though it was built for a large PHP audience, it's not dependent on any PHP tools or technologies to function. It is a JavaScript tool, so it'll work perfectly for our needs.

Note You don't have to use Mix for your applications. You could use Grunt, Gulp, or even webpack directly. Mix ultimately generates a webpack config, so it's not a replacement, but rather an improvement over standard webpack configuration.

I personally find webpack hard to use, yet it has become popular. Mix adds a layer of abstraction on top of webpack, vastly simplifying configuration. We can get started with the following:

```
npm install --save-dev webpack laravel-mix
```

Mix uses webpack, so we need to configure it by using JavaScript. Mix provides a neat configuration language, shown in Listing 13-1.

Listing 13-1. This is from `threadbear/webpack.mix.js`

```
let mix = require("laravel-mix")

mix.setPublicPath("public")

mix.js("resources/js/app.js", "public/js/app.js")
mix.sass(
  "resources/css/app.scss",
  "public/css/app.css",
)
```

With a reference to `mix`, we can do things such as convert modern JavaScript syntax to older JavaScript syntax, using the `js` method. Similarly, we can convert Sass files to CSS files, using the `sass` method.

We can run Mix with the following command:

```
webpack --progress --hide-modules--config=node_modules/laravel-mix/setup/
webpack.config.js
```

Running this command over and over is tiring. Instead, we can use NPM scripts to shorten what we need to type. Listing 13-2 demonstrates what these scripts look like.

Listing 13-2. This is from `threadbear/package.json`

```
"scripts": {
"build": "$npm_package_config_webpack",
"build:watch":
    "$npm_package_config_webpack -w",
"build:production":
    "$npm_package_config_webpack -p"
},
"config": {
"webpack":
    "webpack --progress --hide-modules --config=node_modules/laravel-mix/
    setup/webpack.config.js"
}
```

Here, we're using the magic `$npm_package_config_webpack` variable to remove repetition from our scripts. NPM scripts are an easy way to store detailed commands so we don't have to remember them ourselves. Yarn still uses `package.json`, which is an NPM file. So, we can still use these commands.

The first runs Mix to produce a developer-friendly bundle of code. It includes source maps (so browsers can find the true location of errors) and is not minified in any way.

The second command produces the same files but is a watcher script. That means we can leave it running, and only the file we change will be re-added to the bundle.

The third command creates a production bundle of code. All the JavaScript and CSS are minified, and source maps are removed. The files are a lot smaller but also much harder to debug.

Note While building the application, I highly recommend the first or second script. If you run the watcher script, you won't have to keep on remembering to rebuild the bundle. It'll just happen automatically. You'll even get notifications to show that it has been updated to reflect your new code.

The commands then become the following:

```
npm run-script build
npm run-script build:watch
npm run-script build:production
```

I can't stress enough that this is only the beginning of what Mix has to offer. We're still going to look at one additional feature, but there's much more it can do. You'll have to go to the documentation to learn more, though (`https://laravel.com/docs/5.5/mix`).

This webpack and Babel build chain will let us combine JavaScript files by using `import` and `export` statements. If we want to support older browsers and newer keywords (for example, `async` and `await` in the browser), we also need to add compatibility layers:

```
npm install --save-dev babel-preset-env babel-plugin-transform-async-to-
generator babel-plugin-transform-class-properties
```

`babel-preset-env` transforms modern JavaScript syntax (commonly known as ES2015+) down to ES5, which is syntax most browsers can understand. This means we can use all the new syntax, and customers can use all their old browsers.

`babel-plugin-transform-async-to-generator` transforms `async` and `await` into syntax that uses generators as coroutines. Chapter 6 explained how generators can do this. We're doing exactly the same thing here, with this polyfill, but in a browser environment.

Note Some versions of `babel-preset-env` include `babel-plugin-transform-async-to-generator`. I'm including it again for safety.

Finally, `babel-plugin-transform-class-properties` transforms class properties into a form of syntax that older browsers understand. We're going to be using React to build some front-end components, and React components can use class properties to great effect. That's why we need this last plug-in.

In order for Babel to use these presets and plug-ins, we need to create a configuration file resembling Listing 13-3.

Listing 13-3. This is from `threadbear/.babelrc`

```
{
  "presets": ["env"],
  "plugins": [
    "transform-class-properties",
    "transform-async-to-generator"
  ],
  "ignore": ["node_modules"]
}
```

Now our modern JavaScript code will work in the most common browsers. We can see this transformation in action by modifying our JavaScript file to resemble Listing 13-4.

Listing 13-4. This is from `threadbear/resources/js/app.js`

```
import "regenerator-runtime/runtime"

const load = async () => {
  const response = await fetch(
    "https://threadbear.store",
  )

  const text = await response.text()

  console.log(text)
}

load()
```

We need to import the regenerator runtime into our entry JavaScript file. The async-to-generator plug-in rewrites `async` and `await` to use it, but doesn't import it automatically. The generated code is quite messy, but you can check it out at `threadbear/public/js/app.js`.

Installing Bootstrap (Sass)

Another benefit of Mix is the ability to use Sass for styling our application. Sass is an enhanced version of CSS, with support for nesting and variables (among many other features). We're not going to dwell too much on Sass and CSS, but we can get a lot of design for free by using Bootstrap. Bootstrap (`https://getbootstrap.com`) is a front-end framework with many good defaults and built-in component styles.

We can already compile Sass files, but we still need to install Bootstrap:

```
npm install --save-dev bootstrap@4.0.0-beta.2
```

We can import Bootstrap into our application's stylesheet with the `@import` statement. Listing 13-5 demonstrates this.

Listing 13-5. This is from `threadbear/resources/css/app.scss`

```
@import "node_modules/bootstrap/scss/bootstrap";

body {
  margin: 0;
}
```

The generated CSS file includes all of the Bootstrap CSS (converted from the original Sass). You can check it out at `threadbear/public/css/app.css`.

Note At the time of writing, Bootstrap v4 was in beta. Odds are that v4 is now stable, and newer versions may exist. If these versions still use Sass, this same approach will work, though you may need to adjust the import path to reflect a new folder structure inside `node_modules`.

Mix also supports languages such as Less and tools such as PostCSS. I like using Sass and Bootstrap, so this is the approach I usually take. Feel free to experiment, and refer to the Mix documentation when you get stuck.

Let's add our stylesheet to the layout template, as shown in Listing 13-6.

Listing 13-6. This is from `threadbear/resources/views/layout.edge`

```
<!doctype html>
<html lang="en">
  <head>
    {{ css("/css/app.css") }}
    <!-- rest of the meta tags -->
  </head>
  <body>
    <!-- rest of the body -->
    {{ script("/js/app.js") }}
  </body>
</html>
```

We should also migrate our home-page template to use this layout, as shown in Listing 13-7.

Listing 13-7. This is from `threadbear/resources/views/page/home.edge`

```
@layout("layout")
@section("header")
  <h1>Welcome to ThreadBear</h1>
@endsection
@section("content")
  <p>
```

```
    <!-- intro text -->
  </p>
@endsection
```

We're also linking to the built JavaScript file instead of the external scripts. We don't really need jQuery or the Bootstrap JavaScript, so, we won't bother to import them into our `app.js` file.

Installing React

Now that our home page is looking better, and our Sass and JavaScript files are being transformed into versions most browsers can handle, it's time to learn a bit about React (`https://reactjs.org`). I love it, and given time, I think you will too.

React appeals to people for many reasons. React appeals to me for two reasons.

The first is that React supports a kind of HTML-in-JS syntax and a style of building large interfaces by thinking of them in their smallest parts. In other words, when I start to build an interface (and I'm using React), I think about it from bottom to top: "What is the smallest part of this, and how can I make it well, so that larger components can nest it easily?" I don't start building a shopping cart, creating a `.cart` div to hold everything. I start with each of the interaction points (usually links, buttons, or input fields) of each of the products. I think of the Clear and Checkout buttons. I think of everything as a small, indivisible thing. Then I start to nest these small components inside larger components.

The second thing I like is that React promotes a unidirectional flow of data. Interfaces are stateful things; they contain state about where you are and what you've already done, what you're busy doing, and what your available interaction options are. We can transmit this data from the top down, and let React work out which nested components should be updated to reflect the new state.

These are difficult concepts to grasp (and master), and they're not the only reasons people come to like React. Clearly, we'd be better served by looking at how they apply to the code we write. Before we can, though, we need to install React:

```
npm install --save-dev babel-preset-react react react-dom
```

`babel-preset-react` imports all the bits that React needs, for JSX (that HTML-in-JS syntax) to be compiled into regular JavaScript. It does more than that, but we don't need to worry about anything else.

We also need to update `.babelrc`, as shown in Listing 13-8.

Listing 13-8. This is from `threadbear/.babelrc`

```
{
  "presets": ["env", "react"],
  "plugins": [
    "transform-class-properties",
    "transform-async-to-generator"
  ],
  "ignore": ["node_modules"]
}
```

`react` and `react-dom` take care of the rest. To understand how they work, take a look at Listing 13-9.

Listing 13-9. This is from `threadbear/resources/js/app.js`

```
import "regenerator-runtime/runtime"

// const load = async () => {
//   const response = await fetch(
//     https://threadbear.store
//   )
//
//   const text = await response.text()
//
//   console.log(text)
// }
//
// load()

import React from "react"
import { render } from "react-dom"

render(<span>hello world</span>, document.body)
```

`react-dom` takes the `<span>hello world</span>` syntax and renders a span to the body element. We can build larger components out of regular HTML elements. Listing 13-10 demonstrates what one of these larger components might look like.

Listing 13-10. This is from `threadbear/resources/js/app.js`

```
import React, { Component } from "react"
import { render } from "react-dom"

class Heading extends Component {
  render() {
    const { style, children } = this.props

    return (
      <h1 className={"heading-" + style}>
        {children}
      </h1>
    )
  }
}

render(
  <Heading style="bright">
    Welcome to ThreadBear
  </Heading>,
  document.body,
)
```

When we build these kinds of components, we define a `render` method. This method returns what the component should look like with any given state. We can pretend this is being run only once, but React could be calling it constantly (with updates to properties or state).

We'll start to build more and more complex components in future chapters. For now, I encourage you to experiment with React (and the JSX syntax) until you have the hang of building and rendering elements. It's important that your build chain is working, so you can follow along when we write more browser JavaScript.

Summary

In this chapter, we set up a complete build chain. We installed and configured Laravel Mix. We saw how it enables us to define input and output files, and how easy it is to work with Sass and modern JavaScript syntax.

We installed Bootstrap so we can use local builds and customize them with our own Sass variables and styles. We also installed React and tested to see that it can compile JSX syntax to regular old JavaScript.

In the next chapter, we're going to connect WebSockets in the browser to our application back end. We'll start to store and retrieve data through these WebSockets and build a shopping cart in the process.

CHAPTER 14

WebSockets

In the previous chapter, we started work on the front end. We set up a build chain and installed a few front-end frameworks. We're now able to build Sass stylesheets and parse JSX!

In this chapter, we're going to start to put that to use. We'll set up WebSockets (from the server to the browser) and start to store and retrieve data through them. We'll connect them to a React shopping cart and learn about how to manage state in a deeply-nested component hierarchy. Finally, we'll use the checkout process to create orders.

Installing Socket.io

AdonisJs 4.0 is full of awesome new features and improvements. But one of the areas that still needs to be addressed is WebSockets. AdonisJs 3.x had some support for them, but with 4.0 the core team decided to assess where that support needs to be improved. While work continues in that area, some excellent alternatives fit right into an AdonisJs app.

My favorite alternative is called Socket.io (`https://socket.io`). It's a library we can install as follows:

```
npm install --save socket.io
```

The Getting Started documentation shows how to take an instance of the standard Node.js HTTP server and extend it with WebSocket functionality, as shown in Listing 14-1.

Listing 14-1

```
const app = require("express")()
const http = require("http").Server(app)
const io = require("socket.io")(http)

app.get("/", function(req, res) {
  res.sendFile(__dirname + "/index.html")
})
```

© Christopher Pitt 2018
C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_14

```
io.on("connection", function(socket) {
  console.log("a user connected")
})

http.listen(3000, function() {
  console.log("listening on *:3000")
})
```

Of course, this demonstrates using ExpressJS as well, which we're not going to do. Instead, we're going to tap into the HTTP instance that AdonisJs is using. We'll do this as part of the `hooks` file, shown in Listing 14-2.

Listing 14-2. This is from `threadbear/start/hooks.js`

```
"use strict"

const { hooks } = use("@adonisjs/ignitor")

hooks.after.httpServer(() => {
  const server = use("Server")
  const socket = use("socket.io")(
    server.getInstance(),
  )

  socket.on("connection", connection => {
    console.log("browser connected")
  })

  console.log("set up socket.io")
})

// ...rest of the hooks
```

`use("Server")` returns the single, shared instance of the AdonisJs HTTP server. It's a wrapper for the standard Node.js HTTP server, with a helpful `getInstance` method. If we needed to, we could even insert our own HTTP server object into the AdonisJs Server object, using the `setInstance` method. We'd have to do that before the application is launched, though.

We can pass the HTTP server instance to Socket.io, and then we can begin to attach events to the WebSocket server. Aside from handling all WebSocket traffic for us, Socket.io extends the HTTP server object with a browser runtime endpoint. We should add the exposed script to our layout template, shown in Listing 14-3.

Listing 14-3. This is from `threadbear/resources/views/layout.edge`

```
<!doctype html>
<html lang="en">
  <head>
    <!-- rest of the head -->
  </head>
  <body>
    <!-- rest of the body -->
    {{ script("/socket.io/socket.io.js") }}
    {{ script("/js/app.js") }}
  </body>
</html>
```

Then we can make a new WebSocket connection inside our `app.js`, resembling Listing 14-4.

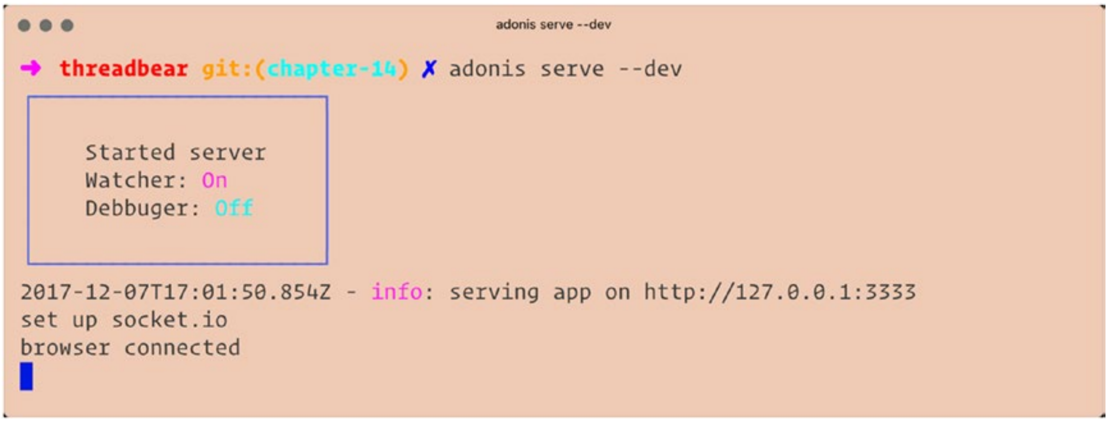
Listing 14-4. This is from `threadbear/resources/js/app.js`

```
const socket = io()

socket.on("server message", message => {
  console.log("server message:", message)
})
```

`/socket.io/socket.io.js` introduces a global function called `io`. This is an easy way for us to start a WebSocket connection. Many browsers still don't support the WebSocket protocol, so Socket.io also ships with workarounds. All of them, and the official WebSocket protocol, can be started in this same way. Socket.io will pick the one that works.

After a connection is made, we can see the console messages. Figure 14-1 shows what this looks like in the development server console.



```
➜ threadbear git:(chapter-14) ✗ adonis serve --dev

    Started server
    Watcher: On
    Debbuger: Off

2017-12-07T17:01:50.854Z - info: serving app on http://127.0.0.1:3333
set up socket.io
browser connected
```

Figure 14-1. *Browser connecting to server*

We can begin to attach new events—for example, for times when the browser disconnects or sends a message to the server. Listing 14-5 shows what this looks like.

Listing 14-5. This is from `threadbear/start/hooks.js`

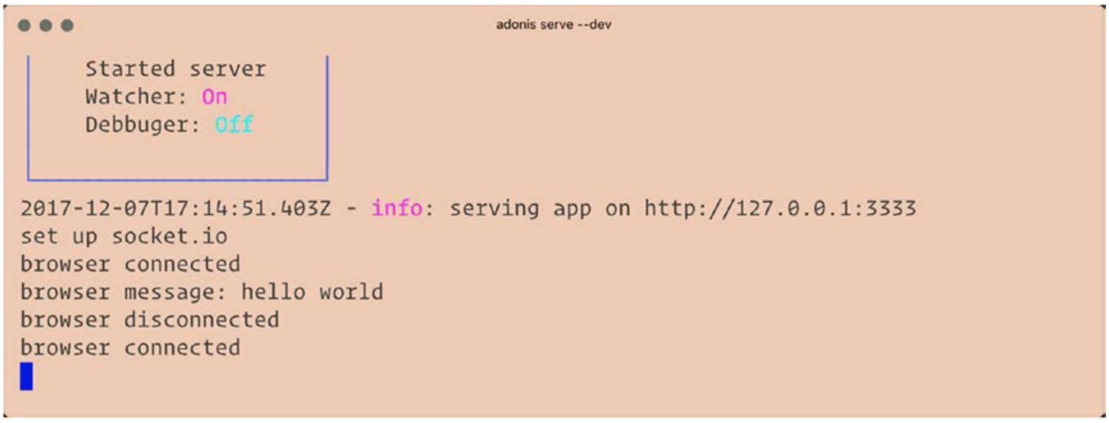
```
socket.on("connection", connection => {
  console.log("browser connected", connection.id)

  connection.on("disconnect", () => {
    console.log("browser disconnected")
  })

  connection.on("browser message", message => {
    console.log("browser message:", message)
  })

  connection.emit(
    "server message",
    "hello browser",
  )
})
```

When we refresh the browser or send a message (using `socket.emit("browser message", "hello world")`), we can see the messages in Figure 14-2.



```
    Started server
    Watcher: On
    Debbuger: Off

2017-12-07T17:14:51.403Z - info: serving app on http://127.0.0.1:3333
set up socket.io
browser connected
browser message: hello world
browser disconnected
browser connected
```

Figure 14-2. *Listening for new socket events*

Each time the browser page is refreshed, a new socket connection will automatically be made to the server. That's good, but it could be better. We could try to maintain a single session each time the server is restarted or the browser page is refreshed.

Unfortunately, this will take us further into the guts of Socket.io than I would like us to go. For the sake of simplicity, we're going to assume that the server won't restart and that the browser won't refresh while we're working with the cart. That's obviously not going to be the case in a real application, and I'll point you in the direction of how to maintain sessions a little later.

Let's add a Buy Now button to the Products list. Let's also add the shopping cart on the same page so we can see the communication between the Buy Now button, WebSockets, and React. Listing 14-6 shows the Buy Now button and the container into which we'll put the shopping cart.

Listing 14-6. This is from `threadbear/resources/views/customer/profile.edge`

```
@section("header")
<h1>
  {{ customer.displayName }}'s profile
</h1>
@endsection
@section("content")
  @each(product in products)
    <p>
      {{ toTitleCase(product.name) }}
      <strong>${{ product.price / 100 }}</strong>
      <button data-buy-id="{{ product.id }}" data-buy-name="{{ product.name
      }}" data-buy-price="{{ product.price }}">buy now</button>
    </p>
    @else
    <p>
      {{ toTitleCase(customer.first_name) }} doesn't have any products
    </p>
  @endeach
  <div class="shopping-cart">
    Shopping cart goes here
  </div>
@endsection
```

We've had to update part of this template to match the changes we made in the dashboard template. Similarly, we also need to change how we get the dashboard data, as demonstrated in Listing 14-7.

Listing 14-7. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async showProfile({ params, response, view }) {
  const redirect = await this.redirectForCustomer(
    params.customer,
  )

  if (redirect) {
    return response.route("profile", {
      customer: redirect,
    })
  }

  const customer = await Customer.query()
    .where("nickname", params.customer)
    .first()

  if (!customer) {
    throw new ProfileMissingException()
  }
  const products = await customer
    .products()
    .fetch()

  return view.render("customer/profile", {
    customer: customer.toJSON(),
    products: products.toJSON(),
  })
}

async redirectForCustomer(customer) {
  const row = await Redirect.query()
    .where("from", customer)
    .first()

  if (row) {
    return row.to
  }

  return false
}
```

Building a React Shopping Cart

Now let's get the shopping cart started. We need to check whether the page should display the shopping cart, and then render into it a React component. Listing 14-8 shows how this is done.

Listing 14-8. This is from `threadbear/resources/js/app.js`

```
class Cart extends Component {
  componentDidMount() {
    this.socket = io()

    this.socket.on(
      "server message",
      this.onServerMessage,
    )
  }
  onServerMessage = message => {
    console.log("server message:", message)
  }

  render() {
    return <div>Shopping cart is here</div>
  }
}

const container = document.querySelector(
  ".shopping-cart",
)

if (container) {
  render(<Cart />, container)
}
```

This component resembles the previous `Heading` component. After it is mounted, being rendered into the DOM, it connects to the WebSocket. This will only happen on pages that include the `.shopping-cart` container, and the element will only be unmounted when the customer navigates away from the page. For this reason, we don't need to remove the `server message` event listener or close the socket connection.

Note We could remove the event listeners by using `this.socket.off("event name", fn)`. Furthermore, we could close the socket connection by using `this.socket.close()`. If there was any chance of the `Cart` component being unmounted, removing the event listener would be a very good idea. It would be best to do this inside the `componentWillUnmount` method, which is invoked just before the component is unmounted.

Methods such as `componentDidMount` are called *life-cycle methods*. Without going into too much detail, these methods are called at various stages of the component's life. `componentDidMount` is useful for initiating asynchronous loading operations or connections to other services.

If everything is working, "Shopping cart goes here" should be replaced by "Shopping cart is here." That means the component is being rendered on the correct page. Let's exercise that React muscle and start thinking of the cart in terms of its constituent parts.

Note This is not the book for learning React from scratch. We've started in the deep end, and things will only get deeper from here. If you'd like to learn React from scratch, read a good book about it: *Introduction to React* by Cory Gackenheimer (Apress, 2015).

The cart should show each item's quantity and price. It should show the total value of the cart. It should have a way to add (increase the number of) or remove items from the cart. It should have a button for proceeding to payment.

Let's create each row (with controls), followed by the Total and Proceed to Payment components. Listings 14-9 and 14-10 show each of these.

Listing 14-9. This is from `threadbear/resources/js/app.js`

```
const dollars = cents =>
  "$" + (cents / 100).toFixed(2)

const CartItem = ({
  name,
  price,
  quantity,
  onPlus,
  onMinus,
}) => (
  <div className="item">
    {name}: <strong>{quantity}</strong> x{" "}
    <strong>${(price / 100).toFixed(2)}</strong>{" "}
    <button onClick={onPlus}>+</button>
    <button onClick={onMinus}>-</button>
  </div>
)
```

You'd be forgiven for thinking that this is a function, because it is a function. React can render using components (with a `render` method) just as easily as it can render functions. This function uses some pretty advanced ES2016+ syntax, so let's look at it piece by piece.

We define the function as a constant by using `const CartItem`. This means `CartItem` cannot be redefined. An alternative would be to define the function as `function CartItem`, but I prefer the constant approach.

To this, we assign a short-arrow function. This takes the form of `(...arguments) => {...body}`. For the list of arguments, we're destructuring an input object—that is to say, we're taking the first function argument (which is the `props` object) and assigning those variables the values for matching keys in the object. It's similar to when we say `{ name, price, quantity, onPlus, onMinus } = props`.

Finally, we're omitting the function body, and instead we're immediately returning an object.

This results in a function we can call with a `props` object, which will return some markup. Those named `props` keys are required for the function to work. We have to define a `name`, `price`, `quantity`, `onPlus` (function), and `onMinus` (function).

Note In some cases, React doesn't render spaces between things, as you might be used to in HTML. We force it to by adding `{" "}` in places where we definitely want a space. This workaround is weird, but worth knowing about.

The first three variables are rendered as text. The two functions are used to do something when the buttons are clicked.

Listing 14-10. This is from `threadbear/resources/js/app.js`

```
const CartTotal = ({ items, onCheckout }) => (
  <div className="total">
    Total:{" "}
    <strong>
      {dollars(
        items
          .map(item => item.quantity * item.price)
          .reduce(
            (total, item) => total + item,
            0,
          ),
      )}
    </strong>
    <br />
    <button onClick={onCheckout}>
      Check Out
    </button>
  </div>
)
```

The `CartTotal` component is slightly simpler, though the formatting could use work. This component accepts an array of `items` and reuses the `dollars` function to format their totals. We work out the total for each by first mapping (their quantities and prices) and then reducing each item into a total value.

Note `map` and `reduce` are powerful tools. There's no practical way we could have produced the same output without making it into a normal function body. If you're not already familiar with them, I suggest you take some time to learn these two array functions. Throw in some time with `filter` and you'll be halfway to learning functional programming.

We can put these together by adding them to the `Cart` render process. Listing 14-11 demonstrates this.

Listing 14-11. This is from `threadbear/resources/js/app.js`

```
class Cart extends Component {
  state = {
    items: {},
  }

  componentDidMount() {
    this.initSocket()
    this.initButtons()
  }

  initSocket = () => {
    this.socket = io()
    this.socket.on(
      "server message",
      this.onServerMessage,
    )
  }

  onServerMessage = message => {
    console.log("server message:", message)
  }

  initButtons = () => {
    const buttons = document.querySelectorAll(
      "[data-buy-id]",
    )

    for (let button of buttons) {
      button.addEventListener("click", event =>
        this.onBuy(event),
      )
    }
  }

  onBuy = event => {
    const {
      buyId,
      buyName,
      buyPrice,
    } = event.target.dataset

    const key = "_" + buyId
    const previous = this.state.items[key]
    const quantity = previous
      ? previous.quantity + 1
      : 1

    this.setState({
      items: {
        ...this.state.items,
        [key]: {
          id: buyId,
          name: buyName,
          price: buyPrice,
          quantity,
        },
      },
    })
  }

  render() {
    const items = Object.values(this.state.items)

    return (
      <div className="items">
        {items.map(this.renderItem)}
        <CartTotal
          items={items}
          onCheckout={() =>
            alert("check out time!")
          }
        />
      </div>
    )
  }

  renderItem = item => {
    if (item.quantity < 1) {
      return
    }

    return (
      <CartItem
        {...item}
        key={item.id}
        onPlus={() => this.onPlusItem(item)}
        onMinus={() => this.onMinusItem(item)}
      />
    )
  }
```

```
  onPlusItem = item => {
    const key = "_" + item.id

    this.setState({
      items: {
        ...this.state.items,
        [key]: {
          ...this.state.items[key],
          quantity: item.quantity + 1,
        },
      },
    })
  }

  onMinusItem = item => {
    const key = "_" + item.id

    this.setState({
      items: {
        ...this.state.items,
        [key]: {
          ...this.state.items[key],
          quantity: item.quantity - 1,
        },
      },
    })
  }
}
```

Most of the new code is there to add products to the items object, and to increase or decrease their quantity. Take some time to study this. I want to focus on a couple of other things, though:

- The product buttons aren't controlled by React or the `Cart` component. This means we need to get a reference to them and add event listeners to them. This isn't great, but we'll investigate a better alternative shortly.

- The render method returns the markup for each CartItem (mapped to the renderItem method, as well as the CartTotal component). We were able to create CartItem and CartTotal in complete isolation. We had to decide what variables and functions we needed beforehand, but we certainly didn't need to build the Cart component before the others. That's the bottom-up thinking I've been describing.

Connecting the UI

For the sake of better understanding WebSockets, we're going to do something a little bit silly. We're going to give up control of the Buy Now buttons (in the Cart), and instead we're going to create React components for each of them.

Then we'll connect the Buy Now buttons to the Cart through the WebSockets. Normally, we wouldn't incur network traffic just to connect multiple things on the same page. I think it's an interesting exercise for the moment. Listing 14-12 shows what the new BuyNow button component could look like.

Listing 14-12. This is from threadbear/resources/js/app.js

```
class BuyNow extends Component {
  onBuyNow = () => {
    const { id, name, price, socket } = this.props
    socket.emit(
      "buy",
      JSON.stringify({ id, name, price }),
    )
  }

  render() {
    return (
      <button onClick={this.onBuyNow}>
        buy now
      </button>
    )
  }
}
```

This new component accepts `id`, `name`, `price`, and `socket` as required attributes. When the button is clicked, it emits a message through the WebSocket. We might seek to create these new components by adjusting the profile template and adding new render code to the main JavaScript file. Listings 14-13 and 14-14 show what I'm talking about.

Listing 14-13. This is from `threadbear/resources/views/customer/profile.edge`

```
@section("content")
  @each(product in products)
    <p>
      {{ product.displayName }}
      <strong>${{ product.price / 100 }}</strong>
      <!--
        <button ...>buy now</button>
      -->
      <div class="buy-now" data-id="{{ product.id }}" data-name="{{
      product.displayName }}" data-price="{{ product.price }}">
      </div>
    </p>
  @else
    <p>
      {{ customer.displayName }} doesn't have...
    </p>
  @endeach
  <div class="cart">
    Shopping cart goes here
  </div>
@endsection
```

The new element is similar, but it has a `class` attribute we can use to identify it. I've also chosen to drop the buy prefix from the data attributes. This makes it easier to handle the data.

Listing 14-14. This is from `threadbear/resources/js/app.js`

```
const socket = io()

const cartContainer = document.querySelector(
  ".cart",
)

if (cartContainer) {
  render(<Cart socket={socket} />, cartContainer)
}

const buyNowContainers = document.querySelectorAll(
  ".buy-now",
)

for (let buyNowContainer of buyNowContainers) {
  const {
    id,
    name,
    price,
  } = buyNowContainer.dataset

  render(
    <BuyNow
      id={id}
      name={name}
      price={price}
      socket={socket}
    />,
    buyNowContainer,
  )
}
```

Instead of creating the socket inside the `Cart` component, we can create a single (shared) WebSocket connection. We can pass this as a parameter to each component that needs it.

Here, we're fetching each `buy-now` div, and rendering a new `BuyNow` button component into it. We're providing the `id`, `name`, and `price` required to send the buy message to the server. Listing 14-15 shows how we can intercept that buy message and send it back to the `Cart`.

Listing 14-15. This is from `threadbear/start/hooks.js`

```
connection.on("buy", message => {
  connection.emit("bought", message)
})
```

The `Cart` component becomes a lot simpler now. We can remove the `initButton` method we had before and simplify the `onBuy` method too! Listing 14-16 demonstrates this.

Listing 14-16. This is from `threadbear/resources/js/app.js`

```
class Cart extends Component {
  componentDidMount() {
    this.props.socket.on("bought", this.onBought)
  }

  onBought = data => {
    this.onBuy(JSON.parse(data))
  }

  onBuy = ({ id, name, price }) => {
    const key = "_" + id
    const previous = this.state.items[key]
    const quantity = previous
      ? previous.quantity + 1
      : 1

    this.setState({
      items: {
        ...this.state.items,
        [key]: {
          id,
          name,
          price,
          quantity,
        },
      },
    })
  }

  // ...rest of Cart
}
```

Creating Orders

Let's finish up by creating new orders. We know who is purchasing (because we can force customers to log in before checking out). We know what items should go in each order. That's all we need to know.

Let's make the customer log in before they can check out. Listings 14-17 and 14-18 show how we can achieve this.

Listing 14-17. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async showProfile({
  params,
  session,
  response,
  view,
}) {
  // ...rest of showProfile

  return view.render("customer/profile", {
    customer: customer.toJSON(),
    products: products.toJSON(),
    session: session.get("customer"),
  })
}
```

We can check the session to see if there is a stored `customer`. If you're wondering why we don't just reuse the `auth` middleware, remember that it will automatically redirect customers who aren't logged in. We don't want people redirected away from here. We just want them to log in first.

Listing 14-18. This is from `threadbear/resources/views/customer/profile.edge`

```
@section("content")
  @each(product in products)
    <p>
      {{ product.displayName }}
      <strong>${{ product.price / 100 }}</strong>
      <!--
        <button ...>buy now</button>
      -->
      @if(session)
      <div class="buy-now" data-id="{{ product.id }}" data-name=
      "{{ product.displayName }}" data-price="{{ product.price }}">
      </div>
      @endif
    </p>
  @else
    <p>
      {{ customer.displayName }} doesn't ...
    </p>
  @endeach
  @if(session)
    <div class="cart">
      Shopping cart goes here
    </div>
  @else
  <div>
    <a href="{{ route(" login ") }}">Log in</a>...
  </div>
  @endif
@endsection
```

When the session exists, we show the normal buttons and cart. When it doesn't, we instead show a way for them to log in. What about when they are logged in and have added items to the shopping cart? How do we let them check out?

We can add another socket event, as shown in Listing 14-19.

Listing 14-19. This is from `threadbear/start/hooks.js`

```
connection.on("check out", async message => {
  const { seller, buyer, items } = JSON.parse(
    message,
  )

  const Database = use("Database")
  const moment = use("moment")

  const timestamps = {
    created_at: moment().format(
      "YYYY-MM-DD HH:mm:ss",
    ),
    updated_at: moment().format(
      "YYYY-MM-DD HH:mm:ss",
    ),
  }

  const orderId = await Database.table(
    "orders",
  ).insert({
    buyer_id: seller,
    seller_id: buyer,
    status: "pending",
    ...timestamps,
  })

  await Database.from("order_items").insert(
    items.map(item => ({
      order_id: orderId[0],
      product_id: item.id,
      quantity: item.quantity,
      price: item.price,
      ...timestamps,
    })),
  )
})
```

This is similar to the database seeder work we did in an earlier chapter. We're also using a `buyer` and `seller` value, which we need to store somewhere in the template, as shown in Listing 14-20.

Listing 14-20. This is from `threadbear/resources/views/customer/profile.edge`

```
@if(session)
  <div class="cart" data-seller={{ customer.id }} data-buyer={{ session }}>
    Shopping cart goes here
  </div>
@else
  <div>
    <a href="{{ route(" login ") }}">Log in</a>...
  </div>
@endif
```

Finally, we need to stitch these two bits of code together in the `onCheckout` method, as shown in Listing 14-21.

Listing 14-21. This is from `threadbear/resources/js/app.js`

```
onCheckout = () => {
  const { buyer, seller, socket } = this.props
  const items = Object.values(this.state.items)

  socket.emit(
    "check out",
    JSON.stringify({ buyer, seller, items }),
  )

  this.setState({ items: {} })
}

render() {
  const items = Object.values(this.state.items)

  return (
    <div className="items">
      {items.map(this.renderItem)}
      <CartTotal
```

```
                items={items}
                onCheckout={this.onCheckout}
            />
        </div>
    )
}
```

When the Checkout button is clicked, a new socket message is emitted (and the items object is reset). The message we send includes the buyer's ID, the seller's ID, and the array of order items. Spend some time reviewing this code and becoming familiar with the messages we're sending back and forth, and the components that are reacting to changes in state.

Note I'm avoiding payment gateways for now. We'll tackle them in chapter 16.

Summary

In this chapter, we took a whirlwind tour of WebSockets. We built many React components and threaded them all together by sending messages back and forth through WebSockets. It can be a tricky subject, and there's lots of code to read through and understand.

In the next chapter, we're going to learn about deploying our application to the Web. We'll learn about virtualized servers, provisioning services, SSL certificates, and domains.

CHAPTER 15

Deployment

In the previous chapter, we got knee-deep into WebSockets. We set up the hooks and the Socket.io code so that our application could accept WebSocket connections. Then we created React components and the client-side code so that we could send and receive data from the server. We made a rudimentary shopping cart and checkout process so that customers could purchase products from other customers.

In this chapter, we're going to cover topics related to deploying our application to the Web. This is arguably the least important, and potentially the most complex, step to creating a commerce application, but there are plenty of tools to make the job easier for us.

Note This chapter is full of recommendations for third-party services. We can't host a commerce application on a laptop in the garage. That's unreliable and difficult. Any significant application needs infrastructure, and the companies I recommend here are ones that have helped me host things in the past.

That doesn't mean you have to use the same companies. You can buy your domains anywhere, host your code anywhere, get your SSL certificates from anywhere. I've made my instructions as provider-independent as possible, but you're probably better off using the same services as I do while you're still learning the basics of hosting.

It's also important to remember that these providers could change their interfaces (and the services they provide), so the figures might be out-of-date as you read this book.

Setting Up DigitalOcean

DigitalOcean (www.digitalocean.com) is a virtual private server hosting platform. In the old days, web sites were hosted on dedicated, physical hardware. If you wanted to set something up, you rented a server (or shared a server with many other people). Apache or Nginx was installed on that server and was configured to serve one or more virtual hosts.

These days, it's far more common to run a web application in a virtual server environment—one in which we control more than a sandboxed virtual host environment, and can install any language or binary or service we want to install.

DigitalOcean makes creating and maintaining these kinds of virtual servers easier. It's still very bare-bones but provides ways to manage DNS, monitor the server, and manage installation and configuration of some things.

OK, that's enough waffling. Let's set up an account and configure a domain name. Go to www.digitalocean.com and click Sign Up, as shown in Figure 15-1.

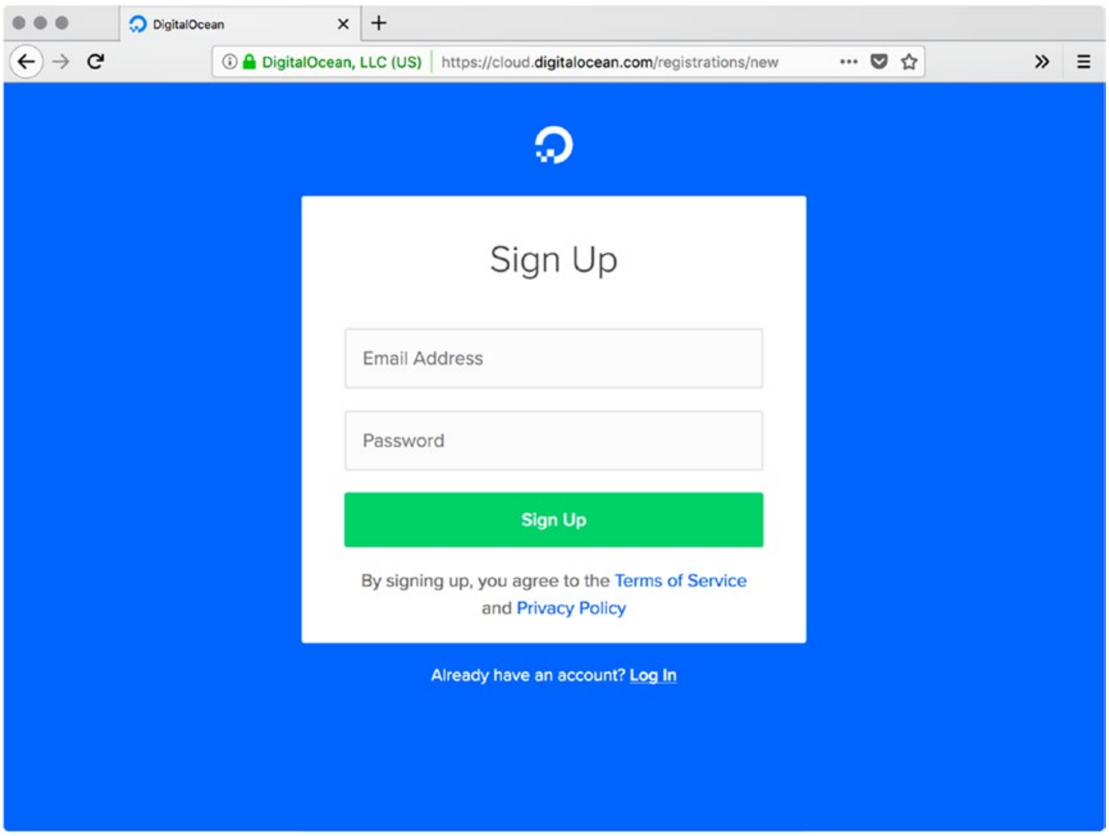

DigitalOcean
DigitalOcean, LLC (US)
https://cloud.digitalocean.com/registrations/new
Sign Up
Email Address
Password
Sign Up
By signing up, you agree to the Terms of Service and Privacy Policy
Already have an account? Log In

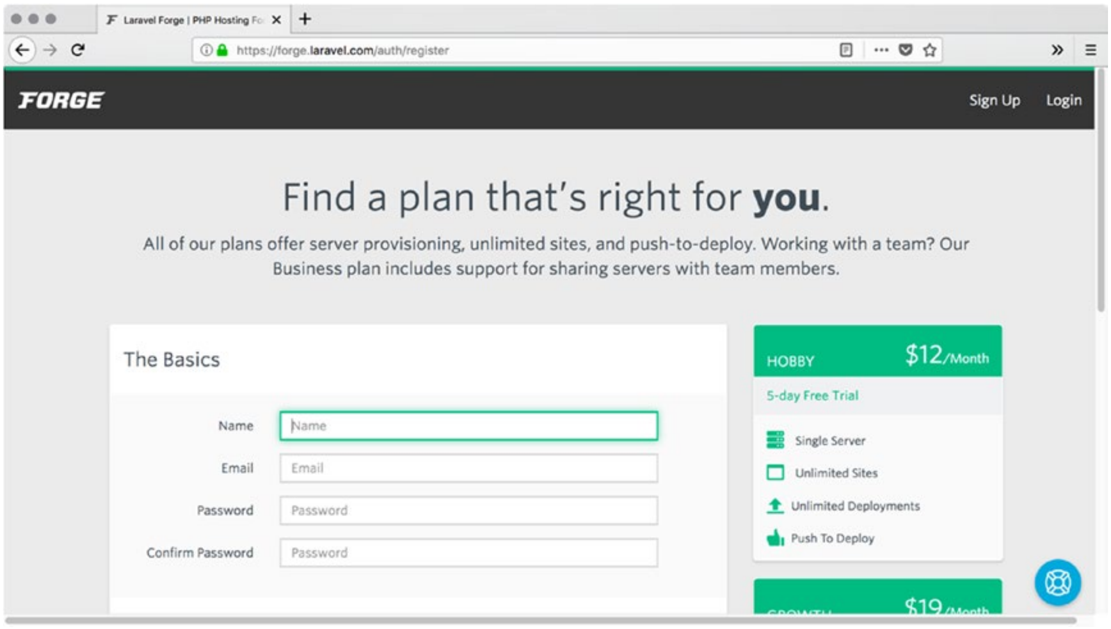
Figure 15-1. *Signing up to DigitalOcean*

Once you're finished setting up your account, we can start to set up new virtual servers. We could do that through the DigitalOcean interface (which is quite friendly), or we could use a provisioning service. Let me tell you about one that I like.

Setting Up Forge

I've mentioned before how similar AdonisJs is to the Laravel PHP framework. I was building Laravel applications long before I ever heard about AdonisJs. Taylor Otwell, the creator of Laravel, has also created a provisioning service. It's called *Forge* (`https://forge.laravel.com`).

Let's create a Forge account, as shown in Figure 15-2.


Laravel Forge | PHP Hosting Fo
https://forge.laravel.com/auth/register
FORGE
Sign Up
Login

Find a plan that's right for **you**.

All of our plans offer server provisioning, unlimited sites, and push-to-deploy. Working with a team? Our Business plan includes support for sharing servers with team members.

The Basics

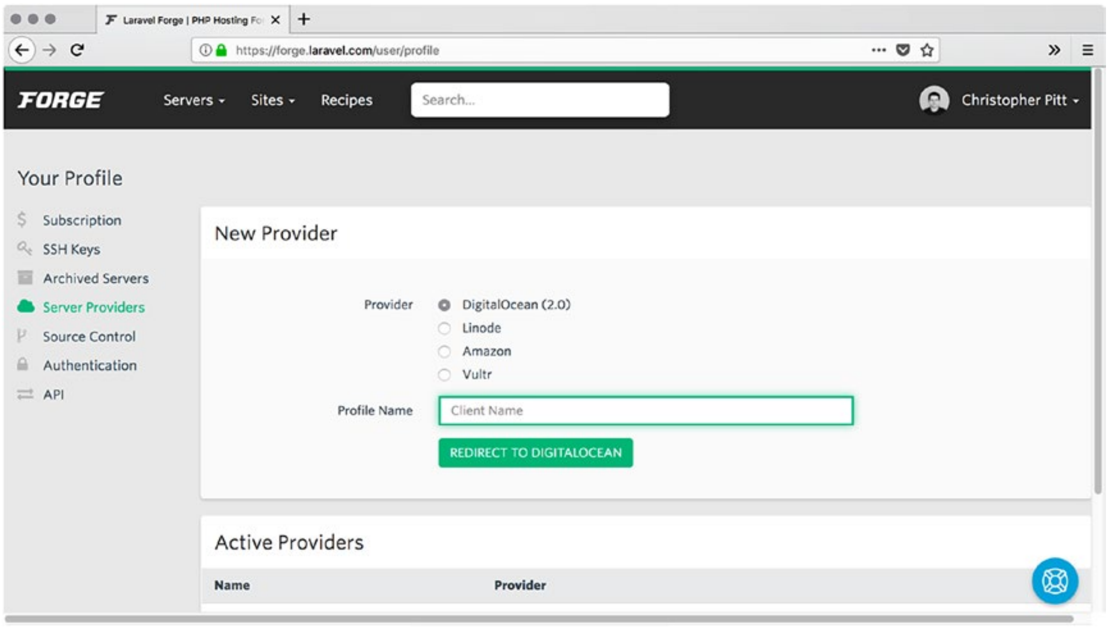
Name: Name
Email: Email
Password: Password
Confirm Password: Password

HOBBY $12/Month
5-day Free Trial
Single Server
Unlimited Sites
Unlimited Deployments
Push To Deploy

$19/Month

Figure 15-2. *Creating a Forge account*

Once that's set up, click your name and go to My Account. Then link your Forge account to your DigitalOcean account, as shown in Figure 15-3.



New Provider

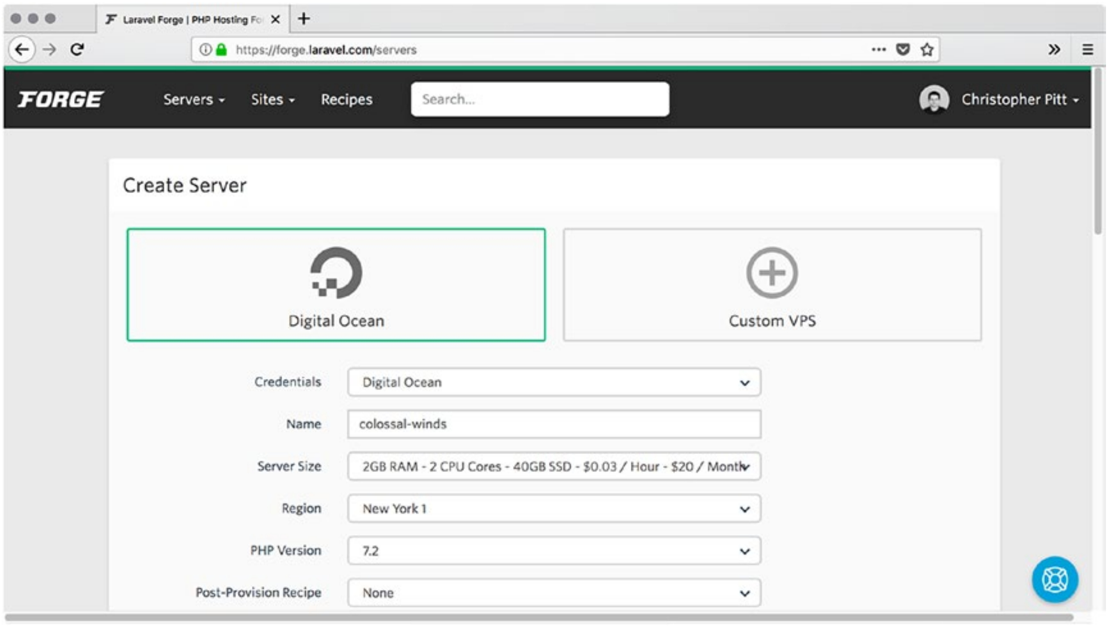
Provider [✓ DigitalOcean (2.0)] Linode Amazon Vultr

Profile Name: Client Name

REDIRECT TO DIGITALOCEAN

Figure 15-3. *Linking Forge to DigitalOcean*

Once that's done, you can go back to the Forge dashboard (click the Forge logo) and create a new DigitalOcean server, as shown in Figure 15-4.



Credentials	Digital Ocean
Name	colossal-winds
Server Size	2GB RAM - 2 CPU Cores - 40GB SSD - $0.03 / Hour - $20 / Month
Region	New York 1
PHP Version	7.2
Post-Provision Recipe	None

Figure 15-4. *Creating a new DigitalOcean server*

The defaults should be fine. This process performs the first bit of setup. It'll install everything we need to host a PHP application. We're not making a PHP application, so some of that software will not be used. It will give us MySQL and network security, though. That's a good start!

Note Keep this browser tab open—we'll need it soon.

After a few minutes, we'll see a screen where we can add new applications to the server. Before we do that, we should try to get a domain name. It's time to set up another account.

Getting a Domain

Note If you've bought a domain before, you can skip this section.

There are many places to buy a domain. If you've built things for the Web, you probably have a sense of how to go about getting a domain. If not, head over to a reseller such as Namecheap, as shown in Figure 15-5. I've never found a pleasant or wholesome domain reseller, but I think Namecheap is the best I've tried.

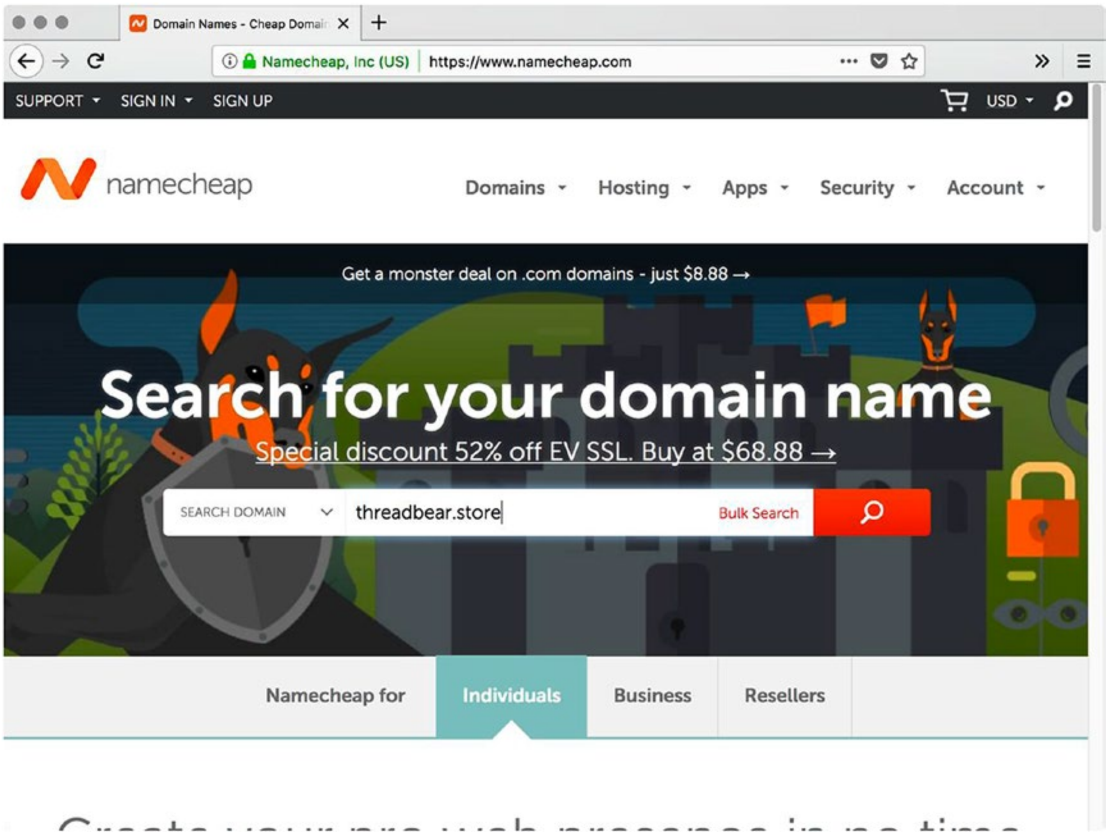
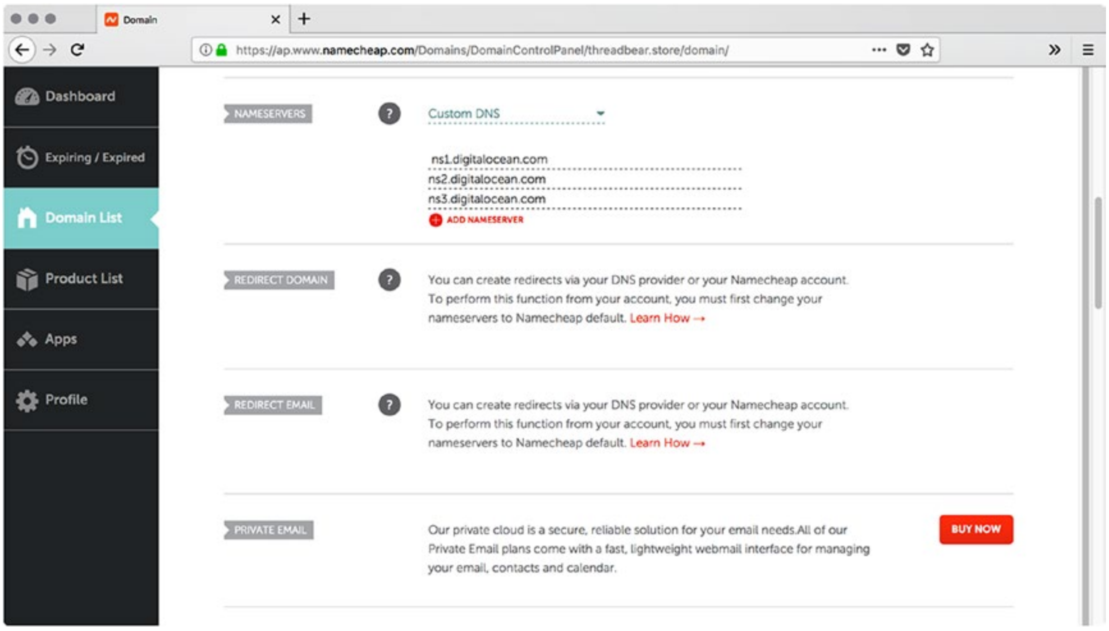

Domain Names - Cheap Domain
Namecheap, Inc (US)
https://www.namecheap.com
SUPPORT
SIGN IN
SIGN UP
USD
namecheap
Domains
Hosting
Apps
Security
Account
Get a monster deal on .com domains - just $8.88 →
Search for your domain name
Special discount 52% off EV SSL. Buy at $68.88 →
SEARCH DOMAIN
threadbear.store
Bulk Search
Namecheap for
Individuals
Business
Resellers

Figure 15-5. *Searching for a domain name*

Search for a domain name you like. Don't be surprised if it is already taken—domain names are like that. Also, don't be surprised if your preferred domain name is taken and is not being used. Domain squatting is an insufferable part of the Web. Find a domain name you don't hate and snag it.

Part of the process will include creating a new account. You should be getting the hang of creating these new accounts by now. After the account is created and the domain is purchased, you'll be able to log in to a management console.

Go to the Domain List and click Manage next to the domain you've just bought. Scroll down to Nameservers and select Custom DNS from the drop-down. Enter `ns1.digitalocean.com`, `ns2.digitalocean.com`, and `ns3.digitalocean.com`; as in Figure 15-6.



ns1.digitalocean.com
ns2.digitalocean.com
ns3.digitalocean.com

You can create redirects via your DNS provider or your Namecheap account. To perform this function from your account, you must first change your nameservers to Namecheap default. Learn How →

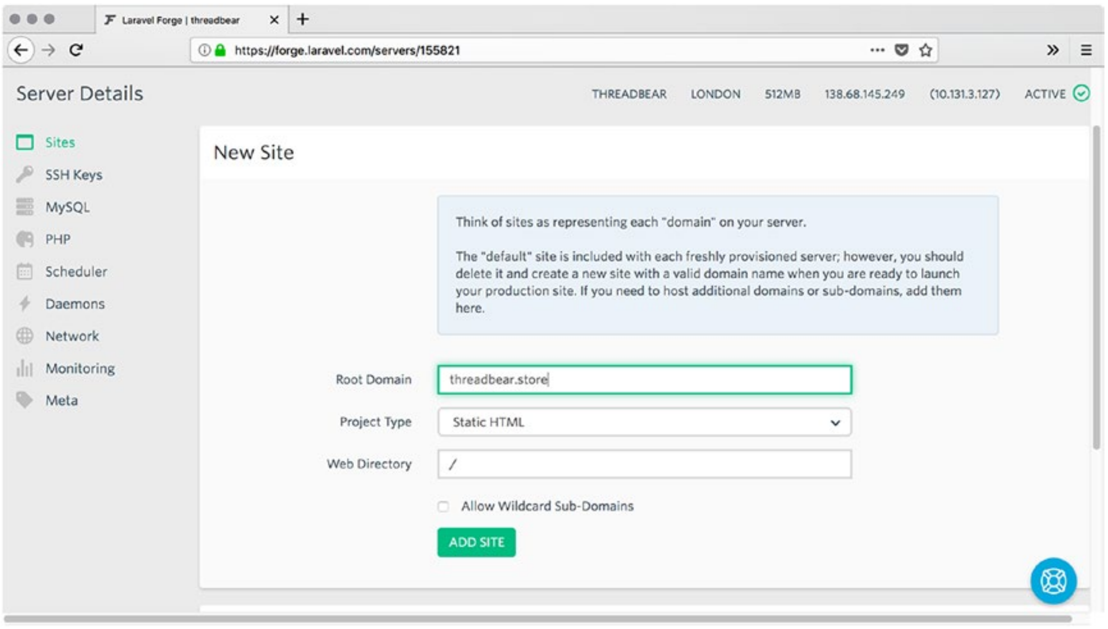
You can create redirects via your DNS provider or your Namecheap account. To perform this function from your account, you must first change your nameservers to Namecheap default. Learn How →

Our private cloud is a secure, reliable solution for your email needs.All of our Private Email plans come with a fast, lightweight webmail interface for managing your email, contacts and calendar.

Figure 15-6. *Setting custom DNS options*

Adding the Domain Name in Forge

It's time to add the domain name in Forge. Hopefully, you kept the Forge tab open on the server page. If not, navigate back to the virtual server you created. Enter the domain name you bought, and select Static HTML from the drop-down, as shown in Figure 15-7. Our AdonisJs application isn't exactly static HTML, but that's the best option for now.



Laravel Forge | threadbear

https://forge.laravel.com/servers/155821

Server Details

THREADBEAR LONDON 512MB 138.68.145.249 (10.131.3.127) ACTIVE

New Site

Think of sites as representing each "domain" on your server.

The "default" site is included with each freshly provisioned server; however, you should delete it and create a new site with a valid domain name when you are ready to launch your production site. If you need to host additional domains or sub-domains, add them here.

Root Domain: threadbear.store

Project Type: Static HTML

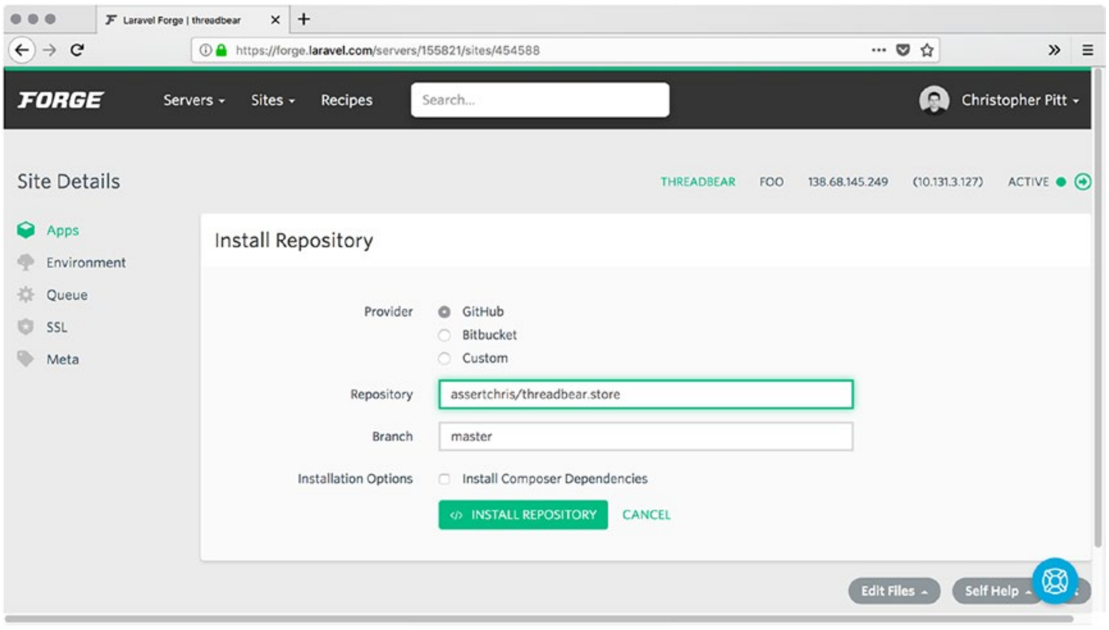
Web Directory: /

Allow Wildcard Sub-Domains

ADD SITE

Figure 15-7. *Adding a domain to Forge*

That'll take a few minutes to finish. When it does, click the domain name you added and you'll be taken to a page that asks whether you want WordPress or a Git Repository. Choose Git Repository and enter the details of the GitHub repository containing your application code, as shown in Figure 15-8.



Install Repository

Provider: [✓ GitHub] Bitbucket Custom

Repository: assertchris/threadbear.store

Branch: master

Installation Options: Install Composer Dependencies

INSTALL REPOSITORY CANCEL

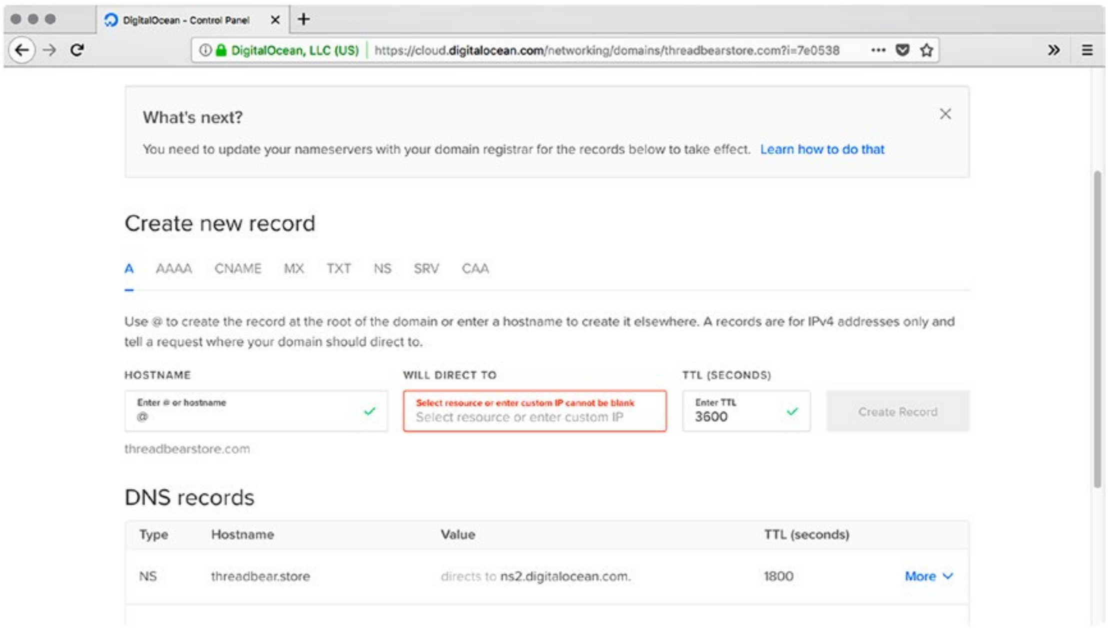
Figure 15-8. Linking a GitHub repository

Remember the IP address in the top-right corner of the screen. We're going to use it to complete the domain name setup, over in DigitalOcean's control panel.

Setting Up DNS Servers

The Namecheap DNS server configuration was only half of the equation. We pointed the domain name to DigitalOcean's servers, but we still have to set up those servers. Go back to the DigitalOcean tab, click Networking, and enter your domain name in the Enter Domain field.

When you click Add Domain, you'll be taken to the configuration page for that domain. Add an "A" record with the hostname of "@" and select the server you created in Forge from the drop-down field next to it. Figure 15-9 demonstrates this.


DigitalOcean - Control Panel
DigitalOcean, LLC (US) https://cloud.digitalocean.com/networking/domains/threadbearstore.com?i=7e0538
A
AAAA
CNAME
MX
TXT
NS
SRV
CAA
Create Record
More

What's next?

You need to update your nameservers with your domain registrar for the records below to take effect. Learn how to do that

Create new record

Use @ to create the record at the root of the domain or enter a hostname to create it elsewhere. A records are for IPv4 addresses only and tell a request where your domain should direct to.

HOSTNAME: Enter @ or hostname — @ — threadbearstore.com

WILL DIRECT TO: Select resource or enter custom IP cannot be blank — Select resource or enter custom IP

TTL (SECONDS): Enter TTL — 3600

DNS records

Type	Hostname	Value	TTL (seconds)
NS	threadbear.store	directs to ns2.digitalocean.com.	1800

Figure 15-9. *Creating an @ record*

DigitalOcean and Namecheap (and most other services you could use instead) will have a delay between creating these records and you being able to use that domain. This is what is represented by *TTL* (short for *time to live*). You may have to wait up to 48 hours before you can type your domain name in a browser and see a response from the correct server—the one you've created.

However, all the pieces should be in place now. When that delay is over, your domain name should match up to the IP address of your DigitalOcean server. All that's left is to run your application through Forge.

Adding a Server Daemon

So far, we've been using the AdonisJs command-line utility as a global utility. That's OK for local development, but we need something a little more predictable in production. Let's install it as a dependency of our application:

```
npm install --save-dev @adonisjs/cli
```

This creates a binary in `node_modules/.bin`, and we can make it easier to run this by adding new `package.json` scripts. Listing 15-1 demonstrates this.

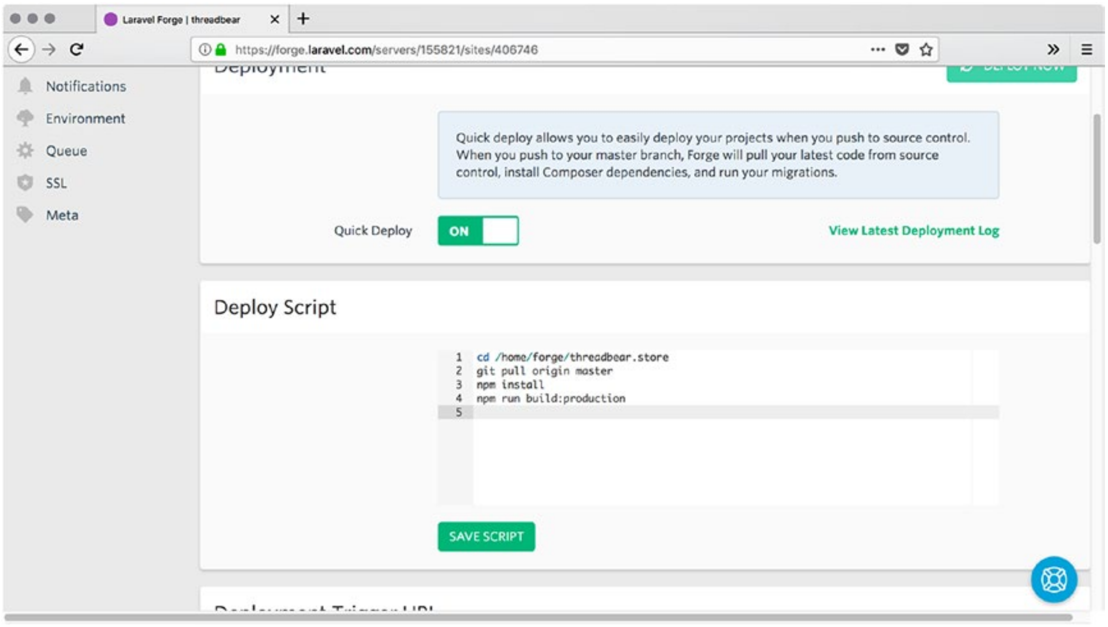
Listing 15-1. This is from `threadbear/package.json`

```
"scripts": {
  "serve": "adonis serve --dev",
  "serve:production": "adonis serve"
  // ...rest of the scripts
}
```

Now we can serve the application just by typing `npm run serve`, or `npm run serve:production` in the case of our production server. We also need Forge to install our NPM dependencies and build our static assets. Go to the Forge dashboard, select the server you created, and then select the application/domain you added to the server.

Note If you're planning to need clustering or any other advanced configuration, consider using PM2 (`http://pm2.keymetrics.io`) instead. It requires a bit more configuration than just using `adonis serve` directly.

On this site dashboard page, you should see a box that has a Deploy Script in it. These are instructions that will be run every time the application is deployed. Remove the PHP-FPM line and add a couple of NPM lines, as shown in Figure 15-10.


Laravel Forge | threadbear
https://forge.laravel.com/servers/155821/sites/406746
Notifications
Environment
Queue
SSL
Meta
Quick Deploy
ON
View Latest Deployment Log
Deploy Script
SAVE SCRIPT

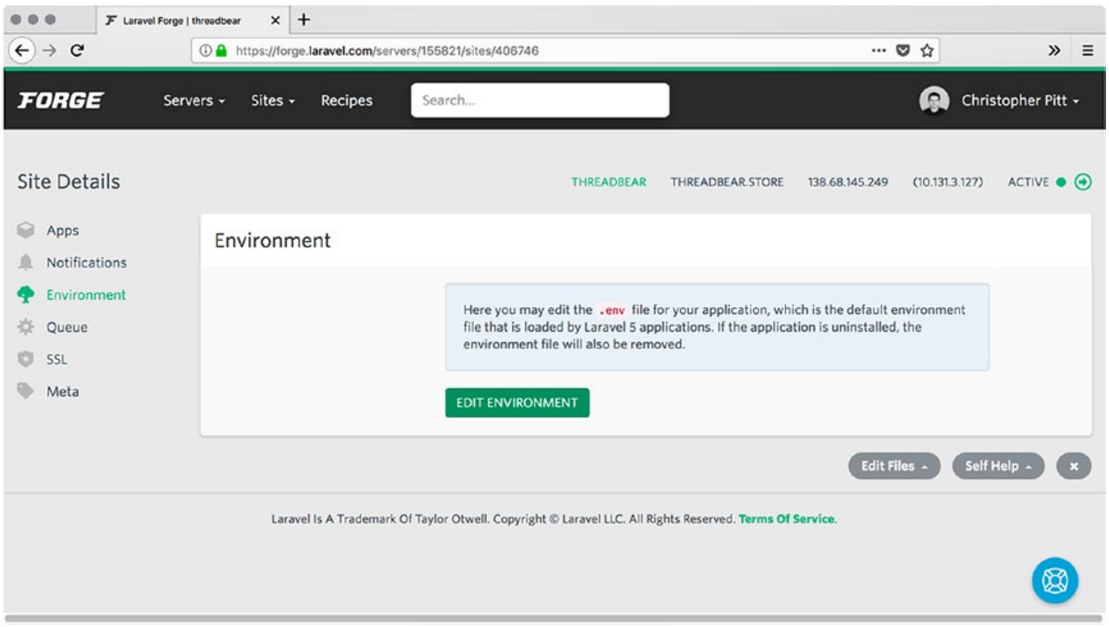
Quick deploy allows you to easily deploy your projects when you push to source control. When you push to your master branch, Forge will pull your latest code from source control, install Composer dependencies, and run your migrations.

```
cd /home/forge/threadbear.store
git pull origin master
npm install
npm run build:production
```

Figure 15-10. *Deploy Script*

These lines tell Forge to install the NPM dependencies and build the static assets, so the front end will look and work as it does on our local system. Now let's set up the environment variables, as shown in Figure 15-11. Click Environment and then Edit Environment to edit the `.env` file that's on the server. You can copy and paste your local settings, but change `NODE_ENVIRONMENT` to `production`.

Note You'll probably want to change the database settings to use something more robust than SQLite. AdonisJs supports many types (see `http://adonisjs.com/docs/4.0/database#_supported_databases`), so check out the documentation for how to connect to the one you want. Even if your database engine isn't connected correctly, you should be able to see parts of your application on the remote server by the end of this chapter.

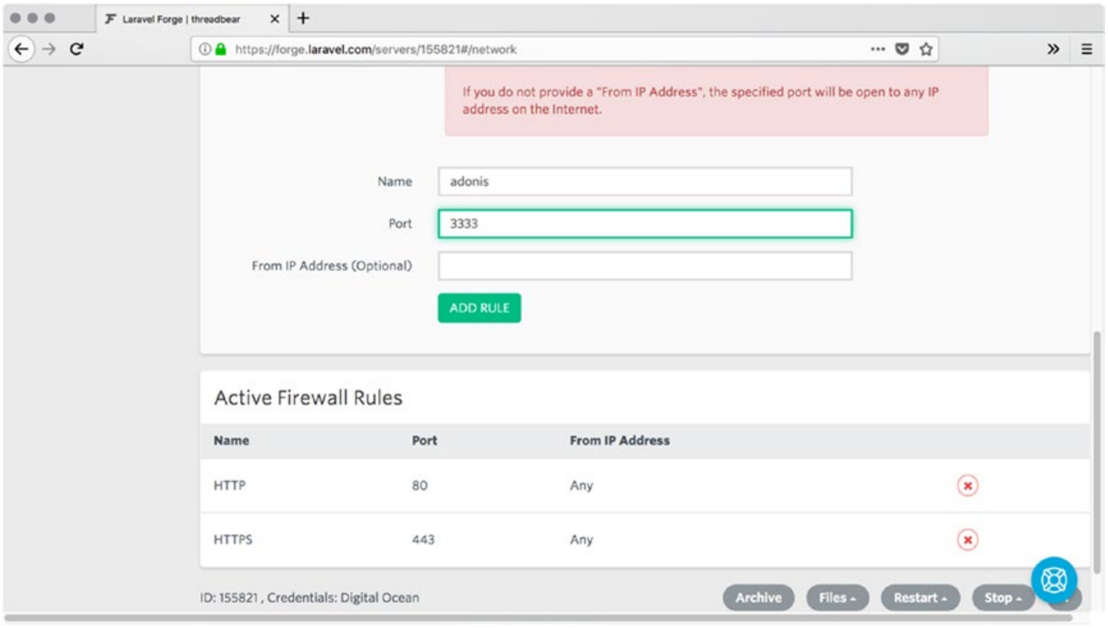


Here you may edit the .env file for your application, which is the default environment file that is loaded by Laravel 5 applications. If the application is uninstalled, the environment file will also be removed.

Laravel Is A Trademark Of Taylor Otwell. Copyright © Laravel LLC. All Rights Reserved. Terms Of Service.

***Figure 15-11.** Adding environment variables*

We've been using the local 3333 port to access the AdonisJs site locally. We need to allow that through the server's network security. Go back to the Forge dashboard and select your server. Click Network and add *3333* with any name (preferably something like *Adonis port,* as shown in Figure 15-12.



https://forge.laravel.com/servers/155821#/network

If you do not provide a "From IP Address", the specified port will be open to any IP address on the Internet.

Name	adonis
Port	3333
From IP Address (Optional)	

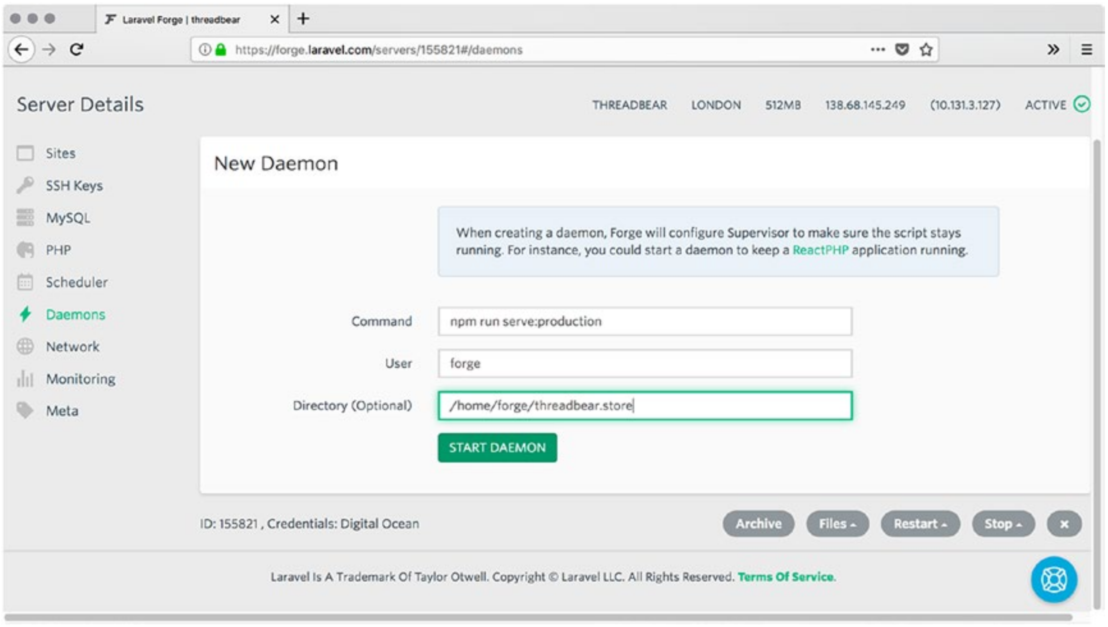
Active Firewall Rules

Name	Port	From IP Address
HTTP	80	Any
HTTPS	443	Any

ID: 155821 , Credentials: Digital Ocean

Figure 15-12. *Adding a network rule*

Finally, click Daemons and add a new daemon, as shown in Figure 15-13.



https://forge.laravel.com/servers/155821#/daemons

Server Details THREADBEAR LONDON 512MB 138.68.145.249 (10.131.3.127) ACTIVE

New Daemon

When creating a daemon, Forge will configure Supervisor to make sure the script stays running. For instance, you could start a daemon to keep a ReactPHP application running.

Command	npm run serve:production
User	forge
Directory (Optional)	/home/forge/threadbear.store

ID: 155821 , Credentials: Digital Ocean

Laravel Is A Trademark Of Taylor Otwell. Copyright © Laravel LLC. All Rights Reserved. Terms Of Service.

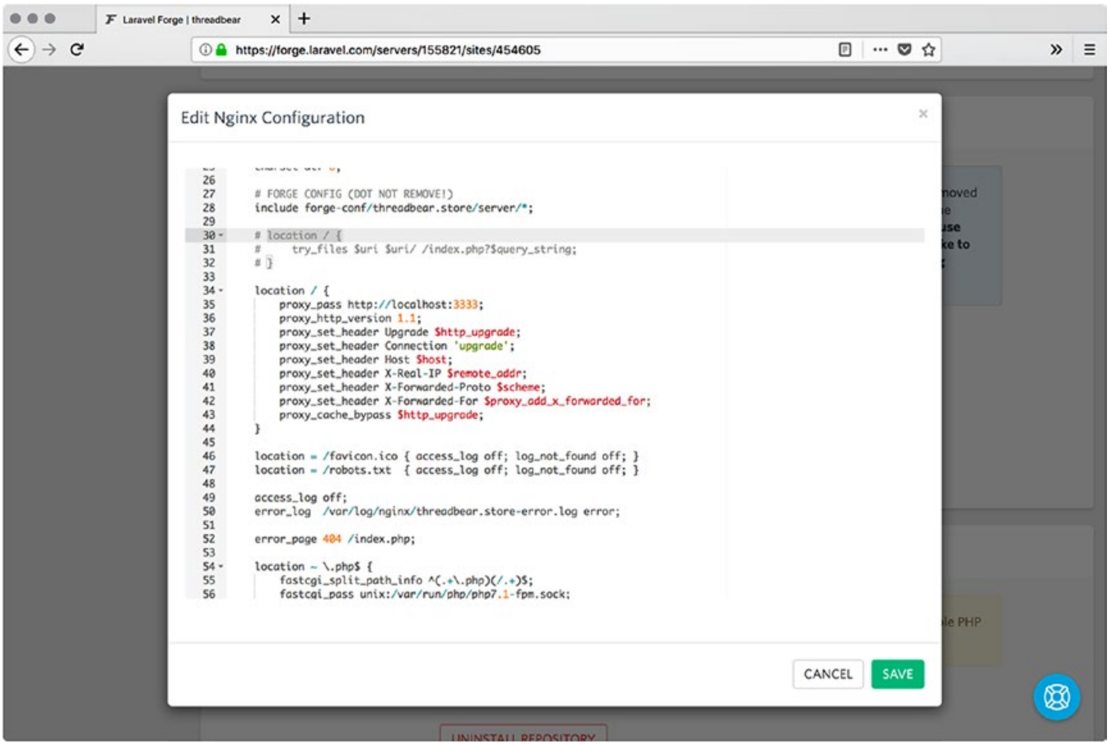
Figure 15-13. *Adding a daemon*

When you click Start Daemon, Forge will create a Supervisor daemon (http://supervisord.org) to run the server. If you go to your server's IP address (and don't forget to add 3333 to the URL), you should see your AdonisJs application!

Serving on Port 80

Our site would be even better if you didn't need to use 3333—and you could go straight to the domain name and it would serve your AdonisJs application. Let's add a bit of Nginx configuration to proxy requests to the correct port.

Go to the Forge dashboard and click the server. Click the site/domain and scroll to the bottom of the page. Click Edit Files, followed by Edit Nginx Configuration. Scroll down to where you see `location / {`, as shown in Figure 15-14. Comment this block out and add the recommended AdonisJs configuration (http://adonisjs.com/recipes/4.0/nginx-proxy) below it.



Edit Nginx Configuration

```

    # FORGE CONFIG (DOT NOT REMOVE!)
    include forge-conf/threadbear.store/server/*;

    # location / {
    #     try_files $uri $uri/ /index.php?$query_string;
    # }

    location / {
        proxy_pass http://localhost:3333;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_cache_bypass $http_upgrade;
    }

    location = /favicon.ico { access_log off; log_not_found off; }
    location = /robots.txt  { access_log off; log_not_found off; }

    access_log off;
    error_log  /var/log/nginx/threadbear.store-error.log error;

    error_page 404 /index.php;

    location ~ \.php$ {
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass unix:/var/run/php/php7.1-fpm.sock;
```

Laravel Forge | threadbear
https://forge.laravel.com/servers/155821/sites/454605
CANCEL
SAVE

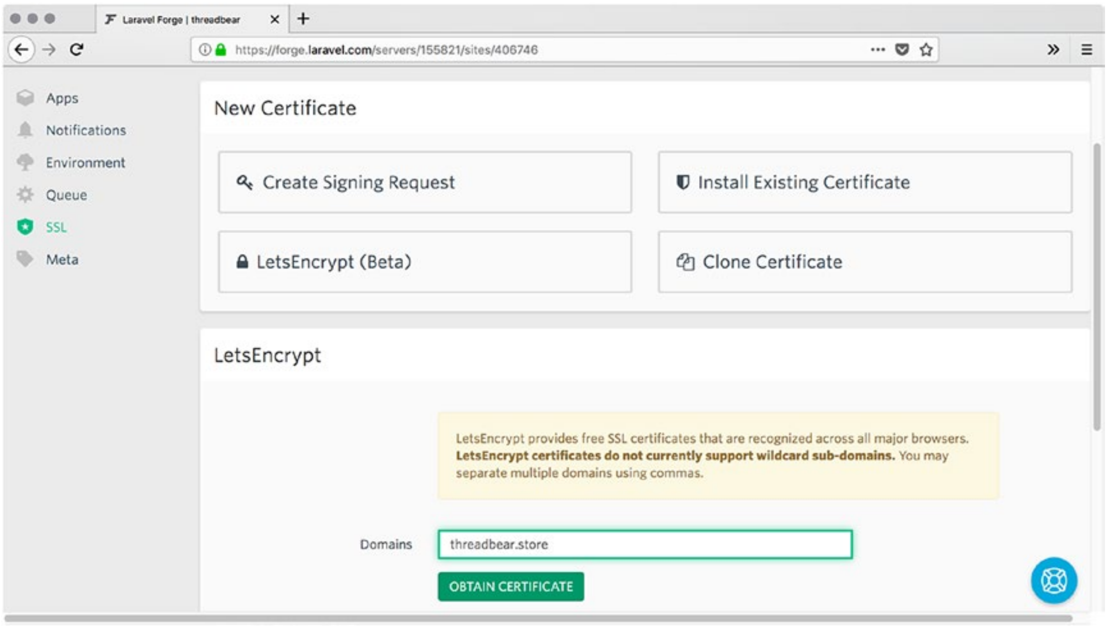
Figure 15-14. Nginx configuration

Don't remove or change any of the other configuration unless you know exactly what you're doing. Instead, go back to the dashboard and click the server. Scroll to the bottom of the page and click Restart, followed by Restart Nginx.

You should be able to go to `http://your-ip-address` and see the AdonisJs application. The last thing we are going to try is setting up an SSL certificate for the application.

Setting Up an SSL Certificate

Go to the Forge dashboard and click the server. Click the site/domain and then click SSL. You should see four options, one of which is LetsEncrypt, as shown in Figure 15-15. Click that. Enter your domain name (without the `www` option, unless you've already set that up) and click Obtain Certificate.



LetsEncrypt provides free SSL certificates that are recognized across all major browsers. **LetsEncrypt certificates do not currently support wildcard sub-domains.** You may separate multiple domains using commas.

Domains: threadbear.store

Figure 15-15. *Getting a certificate*

This will take a minute or two to complete. When it does, you can choose to activate the certificate.

This will also update your Nginx configuration, so that requests to `http://your-domain` are redirected to `https://your-domain`. After all the DNS delays are finished, this will be the moment where you see your application go from a local hobby project to a secure online store.

Summary

In this chapter, we learned about a few tools to help us host our application. This is a good time to share the link with a few trusted friends so they can help you test the application from their computers and Internet connections.

Find out whether your friends got to the application and whether the SSL certificate is working for them. Find out how quickly the files loaded for them, and whether they could see all the stylesheets. Getting customer feedback (and feedback from friends) is an excellent way to know what to improve in your application.

We're almost at the end of our journey. In the final chapter, we're going to add a couple of features we're skipped over so far. We'll learn about things including sending e-mail and making payment gateway purchases.

CHAPTER 16

Finishing Touches

In the previous chapter, we looked at the many tools we can use to deploy AdonisJs applications. We set up a hosting space, purchased a domain, and configured DNS servers. We provisioned a server (with Forge) and installed an SSL certificate.

As we enter the last chapter, it's time to add a little polish to our application. We skipped over some things that we're going to cover now. We're going to learn how to send e-mail so that we can complete the password-reset features. We're also going to learn how to connect to a payment gateway so money can change hands during checkout. By the end of this chapter, we should have a completely functional commerce application.

Sending E-mail

There are many ways to send e-mail in Node.js. We could even implement a new method from scratch, but they tend to be needlessly complicated and it's not an important skill to have to learn right now. Instead, we're going to use one that already works well with AdonisJs applications:

```
adonis install @adonisjs/mail
```

As with most AdonisJs packages, the `adonis install` command will pop up an instruction page upon completion. The one in Listing 16-1 tells us we need to add another provider.

Listing 16-1. This is from `threadbear/start/app.js`

```
const providers = [
  // ...rest of the providers
  "@adonisjs/mail/providers/MailProvider",
]
```

This code also creates a config file, which resembles Listing 16-2.

© Christopher Pitt 2018
C. Pitt, *The Definitive Guide to AdonisJs*, https://doi.org/10.1007/978-1-4842-3390-0_16

Listing 16-2. This is from `threadbear/config/mail.js`

```
"use strict"

const Env = use("Env")

module.exports = {
  /*
  |-----------------------------------------------------
  | Connection
  |-----------------------------------------------------
  |
  | Connection to be used for sending emails...
  |
  */
  connection: Env.get("MAIL_CONNECTION", "smtp"),

  /*
  |-----------------------------------------------------
  | SMTP
  |-----------------------------------------------------
  |
  | Here we define configuration for sending emails...
  |
  */
  smtp: {
    driver: "smtp",
    pool: true,
    port: 465,
    host: "smtp.gmail.com",
    secure: true,
    auth: {
      user: Env.get("MAIL_USERNAME"),
      pass: Env.get("MAIL_PASSWORD"),
    },
    maxConnections: 5,
    maxMessages: 100,
    rateLimit: 10,
  },

  // ...rest of the config
}
```

Before we can send e-mail, we need to add values for MAIL_USERNAME and MAIL_PASSWORD to our .env file. We also need to set a value for host in this config file (and any other settings, depending on our SMTP provider).

Note I usually use Gmail as a provider in these sorts of situations. You can use any SMTP provider or you can use SparkPost (www.sparkpost.com), which is also configured in this file.

When customers request to reset their passwords, we should generate a random and unique token to validate their subsequent password-reset request. We can add a new field to the customers' migration file, as demonstrated in Listing 16-3.

Listing 16-3. This is from threadbear/database/migrations/x_customers_schema.js

```
up() {
  this.create("customers", table => {
    table.string("token").unique()
    // ...rest of the fields
  })
}
```

Let's also remember to rerun the migrations and seeder:

```
adonis migration:refresh
adonis seed
```

Now we need to modify the CustomerController.doForgotPassword method so that it saves the token and sends an e-mail to the customer. Listing 16-4 shows how.

Listing 16-4. This is from threadbear/app/Controllers/Http/CustomerController.js

```
const Mail = use("Mail")
const Hash = use("Hash")
```

```
// ...later

async doForgotPassword({ request, response }) {
  // create new password-reset token and send email
  // this.showRequestParameters(context)

  const email = request.input("email")

  const customer = Customer.findBy(
    "email",
    email,
  )

  if (!customer) {
    // displaying an error shows that this
    // email address isn't registered
    // we should respond as we otherwise would

    const delay = Math.random() * 3000 + 1000

    await new Promise(resolve =>
      setTimeout(resolve, delay),
    )

    response.redirect("/")
  }

  const token = await Hash.make(
    customer.email + new Date(),
  )

  customer.token = token
  customer.save()

  const data = {
    ...customer.toJSON(),
    token,
  }

  await Mail.send(
    "email.customer.forgot-password",
      data,
      message => {
        message
          .to(customer.email)
          .from("your email address")
          .subject("Password Reset")
      },
    )

  response.redirect("/")
}
```

Let's look at each step:

1. We get the e-mail address by using the `request.input` method. Then we look for a customer with a matching e-mail address.

2. If we don't find a customer with a matching email address, we simulate a 1–4 second delay and redirect to the home page. You'll notice that we redirect there anyway, so, this hides the fact that no customer account with that e-mail address exists. Otherwise, malicious people could repeatedly post this form looking for valid e-mail addresses.

3. If we did find a customer, we create a new reset token. This is a hash made up of the customer's e-mail address and a timestamp. We save this to the customer's database record. Because we added a unique index to the `token` field, there's an unlikely possibility that what we generate isn't unique and that it triggers a warning. Given the unlikely nature of this, we're better off displaying an error message and asking the customer to try again.

4. We build a data object of information we might use in the e-mail template, which includes the customer data and the new token. We provide this to the `Mail.send` method, along with a closure to transform the message with. `email.customer.forgot-password` is the name of an e-mail template. After this e-mail is sent, we can redirect back to the home page. This might also be a good time to display a message to the customer.

The e-mail template can be as simple as Listing 16-5.

Listing 16-5. This is from `threadbear/resources/views/email/customer/forgot-password.edge`

```
Click on this link to
<a href="http://127.0.0.1:3333{{ route(" reset-password ", { token }) }}">
  reset your password
</a>
```

Instead of hard-coding the domain name to `127.0.0.1:3333`, we could store this in the `.env` file and change it for each environment we are testing in. We use the `route` function to build the reset-password route, based on what we've defined in the routes file. Listing 16-6 reminds us to name the reset-password route.

Listing 16-6. This is from `threadbear/start/routes.js`

```
Route.get(
  "/reset-password/:token",
  "CustomerController.showResetPassword",
).as("reset-password")
```

Now we need to allow the customer to reset their passwords by clicking the e-mail link. When they do, they're directed to the reset-password route. We need to make that `PATCH` request update the customer's password. Listing 16-7 shows how to achieve this.

Listing 16-7. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
async doResetPassword({
  params,
  request,
  response,
}) {
  // create new password reset token and send email
  // this.showRequestParameters(context)

  const token = decodeURIComponent(params.token)
  const customer = await Customer.query()
    .where("token", token)
    .first()

  if (!customer) {
    // display an error
    throw new ProfileMissingException()
  }

  const password = request.input("password")

  customer.token = ""
  customer.password = password
  await customer.save()

  response.redirect("/")
}
```

The e-mail client is likely to encode the URL (with the hash in it), so we begin by decoding that. Then we get the customer, as we did before.

If everything's OK, we save the new password to the customer's account and redirect back to home. We previously set up a hook in the `Customer` model to automatically hash the password. We need to update it to happen when the customer record is updated, as well as when it is created. Listing 16-8 demonstrates this.

Listing 16-8. This is from `threadbear/app/Models/Customer.js`

```
static boot() {
  super.boot()

  this.addHook("beforeSave", async customer => {
    customer.password = await Hash.make(
      customer.password,
    )
  })
}
```

We changed `beforeCreate` to `beforeSave`, so that it happens with creates *and* updates. We could also add validation to the `password` and `password_confirmation` fields, but I'd rather move on. Consider that another exercise for you!

Connecting to Stripe

We previously created the shopping cart without any money changing hands. In a real application, we need to have some sort of payment gateway integration. The trouble is that this immediately creates pressure to adhere to security standards.

We've already dealt with this, in setting up SSL certificates for our hosting. That's one requirement, but there are many more. Luckily, most of them apply only if we store or transmit credit card data to our servers. Stripe (`https://stripe.com`) provides a way for us not to do this. In other words, Stripe takes care of all the security around transmission and storage of credit card details so we can focus on the business of selling teddy bears. Let's go to Stripe's website, shown in Figure 16-1.

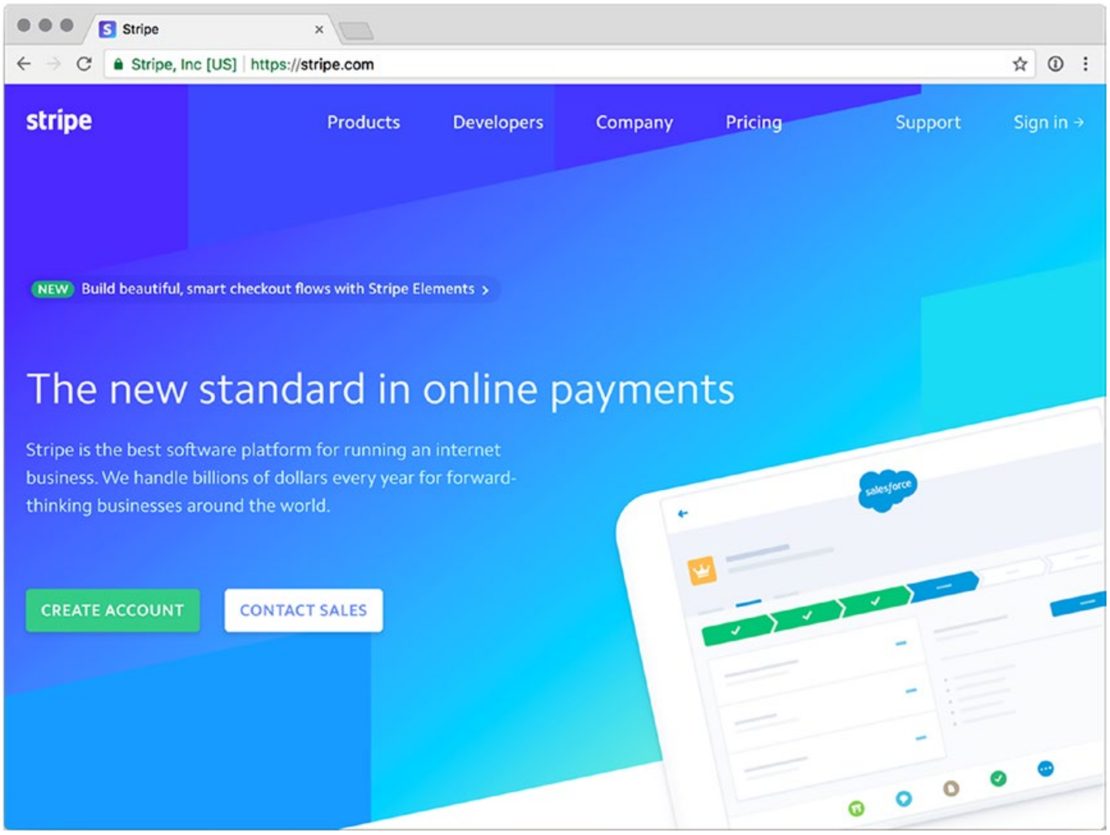

Stripe
Stripe, Inc [US] https://stripe.com
stripe
Products
Developers
Company
Pricing
Support
Sign in →
NEW Build beautiful, smart checkout flows with Stripe Elements ›
The new standard in online payments
Stripe is the best software platform for running an internet business. We handle billions of dollars every year for forward-thinking businesses around the world.
CREATE ACCOUNT
CONTACT SALES
salesforce

Figure 16-1. Registering a new Stripe account

Create a new account with Stripe, and let's get it integrated. After you're logged in, click API and you'll see a list of credentials. Let's add these to our `.env` file. Next, we need to expose the public key to the cart JavaScript, as shown in Listings 16-9 and 16-10.

Listing 16-9. This is from `threadbear/app/Controllers/Http/CustomerController.js`

```
const Env = use("Env")

// ...later

async showProfile({
  params,
  session,
  response,
  view,
}) {
  // ...rest of showProfile

  return view.render("customer/profile", {
    customer: customer.toJSON(),
    products: products.toJSON(),
    session: session.get("customer"),
    key: Env.get("STRIPE_KEY"),
  })
}
```

Listing 16-10. This is from `threadbear/resources/views/customer/profile.edge`

```
<div class="cart" data-seller={{ customer.id }} data-buyer={{ session }}
data-key={{ key }}>
  Shopping cart goes here
</div>
```

If we inspect the profile HTML in a browser, we should see whatever we stored in `.env` inside that `data-key` attribute. We can use that key to generate Stripe tokens, as shown in Listing 16-11.

Listing 16-11. This is from `threadbear/resources/js/app.js`

```
class Cart extends Component {
  onCheckout = async () => {
    const {
      buyer,
      seller,
      stripeKey,
      socket,
    } = this.props

    const items = Object.values(this.state.items)

    const result = await this.stripe.createToken(
      this.card,
    )

    const token = result.token

    console.log(
      "check out event",
      buyer,
      seller,
      token,
    )

    socket.emit(
      "check out",
      JSON.stringify({
        buyer,
        seller,
        token,
        items,
      }),
    )

    this.setState({ items: {} })
  }

  render() {
    const items = Object.values(this.state.items)

    return (
      <div className="items">
        {items.map(this.renderItem)}
        <div id="card-element" />
        <CartTotal
          items={items}
          onCheckout={this.onCheckout}
        />
      </div>
    )
  }

  // ...rest of Cart
}

// ...later

if (cartContainer) {
  const {
    buyer,
    seller,
    key,
  } = cartContainer.dataset

  render(
    <Cart
      buyer={buyer}
      seller={seller}
      stripeKey={key}
      socket={socket}
    />,
    cartContainer,
  )
}
```

We begin by creating a placeholder element, which we call `card-element`. When `Cart` mounts, we create a new `Stripe` object, and we mount the `card` object to the placeholder we created. This isn't how we would typically manage Stripe elements inside React (see `https://github.com/stripe/react-stripe-elements`), but it's to the point.

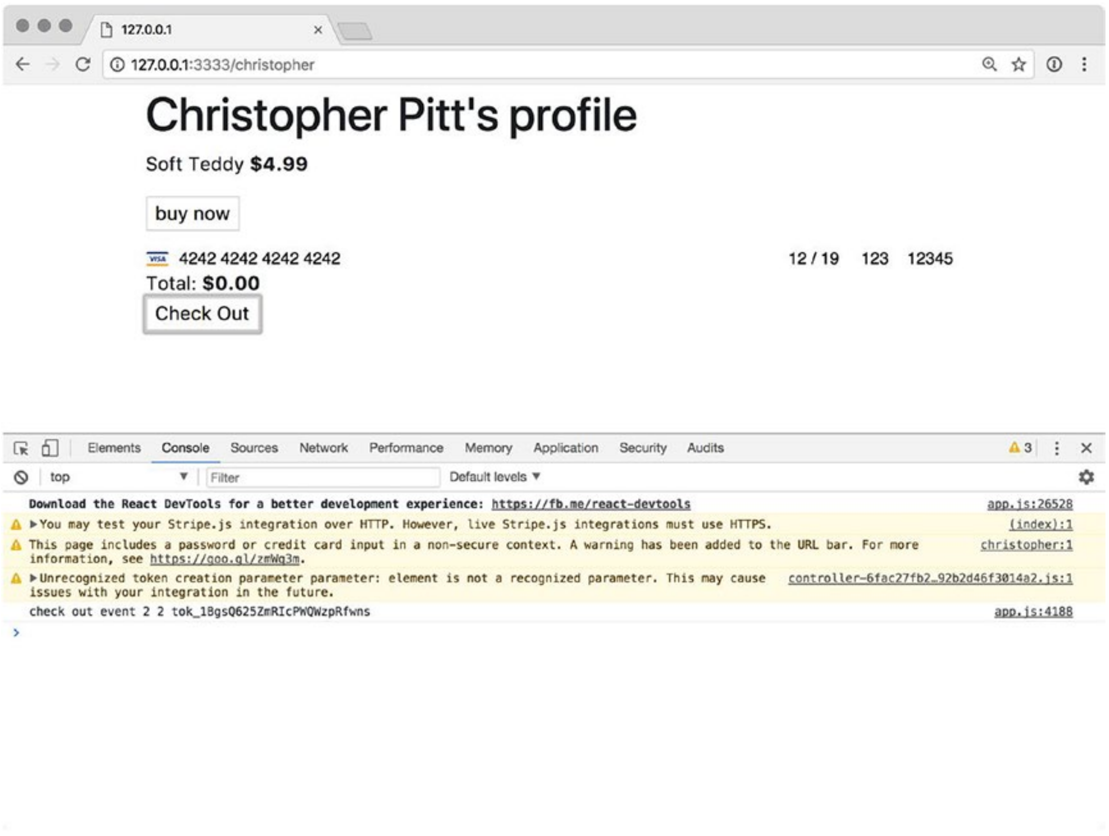
Note I was having some issues with the async/await conversion in my code. To resolve it, I installed babel-polyfill and imported that instead of the regenerator runtime we initially set up. Check out the example code if you're having the same issues. The result is the same, but the polyfill is different.

Instead of immediately sending the check out socket event, we first get a Stripe token and then send that along to the server. For this to work, we also need to include the Stripe JS SDK, as in Listing 16-12.

Listing 16-12. This is from resources/views/layout.edge

```
<!doctype html>
<html lang="en">
  <head>
    <!-- ...rest of head -->
  </head>
  <body>
    <!-- ...rest of body -->
    {{ script("/socket.io/socket.io.js") }}
    <script src="https://js.stripe.com/v3/"></script>
    {{ script("/js/app.js") }}
  </body>
</html>
```

We're not using the usual {{ script(...) }} tag because it messes with the Stripe SDK URL. Before we get into what needs to happen on the server, let's look at what this does in the browser; see Figure 16-2.



```
Download the React DevTools for a better development experience: https://fb.me/react-devtools    app.js:26528
▶You may test your Stripe.js integration over HTTP. However, live Stripe.js integrations must use HTTPS.    (index):1
This page includes a password or credit card input in a non-secure context. A warning has been added to the URL bar. For more    christopher:1
information, see https://goo.gl/zmWq3m.
▶Unrecognized token creation parameter parameter: element is not a recognized parameter. This may cause    controller-6fac27fb2…92b2d46f3014a2.js:1
issues with your integration in the future.
check out event 2 2 tok_1BgsQ625ZmRIcPWQWzpRfwns    app.js:4188
```

Figure 16-2. *Generating Stripe tokens*

Now that we have a Stripe token (which is like a placeholder that Stripe can swap out for credit card details), we can store this alongside the order record. We can execute payments by providing this token to Stripe on the server. This way, money can change hands without us ever seeing credit card details on the server.

Let's modify the `check out` event handler to store this token, and the order table migration to add a field for it. Listings 16-13 and 16-14 show what I mean.

Listing 16-3. This is from `threadbear/database/migrations/x_orders_schema.js`

```
up() {
  this.create("orders", table => {
    table.increments()
    table.integer("buyer_id")
    table.integer("seller_id")
    table.string("status")
    table.string("token")
    table.timestamps()
  })
}
```

Listing 16-14. This is from `threadbear/start/hooks.js`

```
connection.on("check out", async message => {
  const {
    seller,
    buyer,
    token,
    items,
  } = JSON.parse(message)

  const Database = use("Database")
  const moment = use("moment")

  const timestamps = {
    created_at: moment().format(
      "YYYY-MM-DD HH:mm:ss",
    ),
    updated_at: moment().format(
      "YYYY-MM-DD HH:mm:ss",
    ),
  }

  const orderId = await Database.table(
    "orders",
  ).insert({
    buyer_id: seller,
    seller_id: buyer,
    status: "pending",
    token,
    ...timestamps,
  })
  await Database.from("order_items").insert(
    items.map(item => ({
      order_id: orderId[0],
      product_id: item.id,
      quantity: item.quantity,
      price: item.price,
      ...timestamps,
    })),
  )
})
```

The only difference here is that we're extracting token from the message and storing it when we call Database.table("orders").insert. Otherwise, there's no difference. We can process the payments at any time, or we could do it in the same place. For the sake of simplicity, I'll show you what it looks like in the same place. Listing 16-15 shows a slightly modified version of this event handler.

Listing 16-15. This is from threadbear/start/hooks.js

```
connection.on("check out", async message => {
  const {
    seller,
    buyer,
    token,
    items,
  } = JSON.parse(message)

  // ...rest of the handler

  const Env = use("Env")
  const Stripe = use("stripe")
  const client = Stripe(Env.get("STRIPE_SECRET"))

  const total = items.reduce((acc, item) => {
    return acc + item.price * item.quantity
  }, 0)

  client.charges.create(
    {
      amount: total,
      currency: "usd",
      description: "Threadbear purchase",
      source: token,
    },
    (error, charge) => {
      // ...use this to check if
      // the charge was successful
    },
  )
})
```

We can create a new Stripe client object by providing it with the secret key we added to `.env`. Then we can reduce each order item to a total value and create a charge with that total. Keep the Stripe dashboard open and try to check out. You can use the card number `4242 4242 4242 4242` for testing purposes.

If everything is set up correctly, you should also start to see successful charges!

Note It's also possible to set up monthly recurring payments with Stripe. You can check out the documentation (`https://stripe.com/docs/subscriptions/quickstart`) for how to achieve this, but it's mostly the same as one-off charges. After you have the card token, you can do pretty much anything with Stripe.

Enabling CORS Requests

At times we'll need to make requests to the server from the browser, and we don't want to use WebSockets. Perhaps we'd like to make a quick post using the Fetch API (`https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API`). By default, such requests could be blocked.

AdonisJs provides a neat set of tools for managing these permissions:

```
adonis install @adonisjs/cors
```

Naturally, we will need to add a new provider and a new middleware to the application, as demonstrated in Listings 16-16 and 16-17.

Listing 16-16. This is from `threadbear/start/app.js`

```
const providers = [
  // ...rest of the providers
  "@adonisjs/cors/providers/CorsProvider",
]
```

Listing 16-17. This is from `threadbear/start/kernel.js`

```
const globalMiddleware = [
  // ...rest of the middleware
  "Adonis/Middleware/Cors",
]
```

The `adonis install` command also created a new configuration file for us. We need to modify one of its values, so that CORS (or Cross-Origin Resource Sharing) requests will still work. Listing 16-18 shows what to do.

Listing 16-18. This is from `threadbear/config/cors.js`

```
/*
|------------------------------------------------------------
| Origin
|------------------------------------------------------------
|
| Set a list of origins to be allowed. The value can be...
|
*/
origin: () => true,
```

As you can tell by the comment, there are many possible values for this option. We've added a closure that always returns `true`. That's like an allow-all for all cross-origin requests. We probably want to narrow it down to one of the other options or a finite list of allowed domains. For now, this will suffice.

Let's add a form to create new products, and endpoints to display and persist the form. Listings 16-19, 16-20, and 16-21 demonstrate this.

Listing 16-19. This is from `threadbear/resources/views/customer/partial/create-product.edge`

```
<h2>
  Create product
</h2>
<form method="post" class="ajax-form" action={{ route( "create-product", {
customer: customer.nickname }) }}>
  {{ csrfField() }}
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control" id="name" name="name">
  </div>
  <div class="form-group">
    <label for="price">Price</label>
    <input type="text" class="form-control" id="price" name="price">
  </div>
  <button type="submit" class="btn btn-primary">
    Create
  </button>
</form>
```

The only difference here, apart from the fact that this form isn't surrounded by the `@layout`/`@section` tags, is that we're specifying the action and a custom CSS class. We have to specify the action because this partial is being included on the dashboard page, and the default action would be `dashboard` instead of `/christopher/products`.

Listing 16-20. This is from `threadbear/resources/views/customer/dashboard.edge`

```
@layout("layout")
@section("header")
  <h1>
    Hello, {{ customer.displayName }}
  </h1>
@endsection
@section("content")
```

```
  <!-- ...rest of dashboard -->
  @include("customer.partial.create-product")
@endsection
```

Listing 16-21. This is from `threadbear/start/routes.js`

```
Route.group(() => {
  // ...rest of the protected routes

  Route.post(
    "/:customer/products",
    async ({ request, response, params }) => {
      const { name, price } = request.all()
      const Product = use("App/Models/Product")

      await Product.create({
        name,
        price,
        customer_id: request.customer.id,
      })

      response.route("dashboard")
    },
  ).as("create-product")
}).middleware(["auth"])
```

For the sake of simplicity, I've added the creation code in the routes file. If it's inside the auth middleware route group, we will have access to `request.customer` (thanks to how our `Auth` middleware works).

This means we can create a new product, along with the appropriate `customer_id` value. We should be able to submit this form, and it will make a `POST` request and redirect. Let's replace this default behavior with some Ajax, as demonstrated in Listing 16-22.

Listing 16-22. This is from `threadbear/resources/js/app.js`

```
const forms = document.querySelectorAll(
  ".ajax-form",
)
```

```
for (let form of forms) {
  const method = form
    .getAttribute("method")
    .toUpperCase()

  const action = form.getAttribute("action")

  form.addEventListener("submit", async event => {
    event.preventDefault()

    const data = []
    const inputs = form.querySelectorAll("input")

    for (let input of inputs) {
      data.push(
        input.name +
          "=" +
          encodeURIComponent(input.value),
      )
    }

    const response = await fetch(action, {
      mode: "cors",
      credentials: "include",
      redirect: "follow",
      headers: {
        "Content-Type":
          "application/x-www-form-urlencoded; charset=UTF-8",
      },
      method,
      body: data.join("&"),
    })

    alert("saved!")

    // we should do another ajax request
    // to reload the products list. for now,
    // we'll just refresh the whole page...
    location.reload()
  })
}
```

We begin by getting all the forms that have the custom CSS class. For each of them, we get their action and method so that we can re-create the intended request through the `fetch` function. Each input is used to construct the request body, including the hidden `_csrf` field.

After the request finishes, we can alert the customer that their new product was created. We could also do a second Ajax request to refresh the product list, but, a redirect will suffice.

Note I was having some CSRF failures, even though I was submitting the `_csrf` token field to AdonisJs. If you encounter 403 error responses, temporarily disable CSRF to the route and see if the error goes away. That should at least tell you whether the error you're seeing is related to CSRF or something else.

Summary

In this chapter, we tied up a few loose ends. We learned how to send e-mail and subsequently completed the password-reset functionality. We learned how to integrate a payment gateway so that customers could pay for goods and services. Finally, we configured our application to handle CORS requests, and we set up a product creation page.

We're at the end of our journey. Thank you for taking the time to read this book. I hope it has given you the tools you need to make your ideas into real applications. You might not be selling teddy bears, but I hope that you are able to thrive with AdonisJs.

Please reach out to me on Twitter (`https://twitter.com/assertchris`) if you have any questions or you are stuck with something.

Index